

# Etudes des différences entre Java et C#

Introduction :	4
I. Les Similitudes	5
• Chaîne de caractères	5
• Classe mère	5
• Compilation	5
• Gestion de la mémoire	6
• Héritage	6
• Méthode globale et initialisations des variables	6
• Mots-clés	6
II. Les Pseudos similitudes	7
a. Syntaxique	7
• Accesseurs et Modificateurs	8
• Déclaration de tableau	9
• Exceptions	9
• Héritage	9
• Méthode main	10
• Mots-clés	11
• Namespaces et Package	12
• Synchronisation des méthodes	13
b. Concepts identiques mais géré différemment	14
• Accessibilité	14
• Appel du constructeur père	15
• Collection	16
• Constructeur et destructeur	17
• Génération de documentation	18
• Goto	18
• Méthode virtuelles	19
• Sérialisation	21
• Surcharge des opérateurs	22
• Switch	23
• Threads	23
• Type primitif	23
III. Les Différences	24
a. Fonctionnalités propres à C# :	24
• Alias	24
• Attributs (méta données) (présent dans le j2sdk1.5)	24
• AutoBoxing (Boxing/Unboxing) (présent dans le j2sdk1.5)	26
• Caractères spéciaux	26
• Délégués	27
• Détection débordement (checked/unchecked)	27
• Directives pré-processing	28
• Enumérations (présent dans le j2sdk1.5)	29
• Génération de code à l'exécution	29
• Implémentation explicite d'interface	30
• Instruction « for-each » (version similaire dans le j2sdk1.5)	31
• Indexeurs	31
• Libération explicite de mémoire	32

• Liste variables de paramètre (présent dans le j2sdk1.5).....	33
• Opérateur « as ».....	34
• Passage par références.....	34
• Pointeurs/code non protégé (unsafe).....	35
• Propriétés.....	35
• Structures (Types valeurs).....	36
c. Contre attaque de Java ->J2SDK 1.5.....	37
• Autoboxing.....	37
• Enumération.....	38
• "For" (améliorations).....	38
• Généricités.....	39
• Importation Statique.....	41
• Méta données.....	41
• Varargs (Liste variables d'arguments).....	43
IV. Critères de choix.....	44
a. Développement multi langage.....	44
b. Environnements de développement (IDE).....	45
c. Interfaces graphiques.....	46
d. Performances.....	49
e. Plates-formes.....	52
f. Portabilité.....	54
g. Processus de développement.....	55
h. Sécurité.....	56
Conclusion :.....	57

## Introduction :

Java est un langage de programmation qui est apparu au début des années 1990. Il a été créé par James Gosling et Sun.

Java est un langage orienté objet qui est un descendant de C++.

Il est devenu en quelques années le langage le plus utilisé pour le développement en particulier grâce à sa portabilité et car il est particulièrement adapté à internet.

C# quand à lui a été spécialement développé par Microsoft, et notamment par Anders Hejlsberg (qui a également conçu des langages comme Delphi, Turbo Pascal et Visual J++), pour l'environnement .Net afin de le doter d'un langage orienté objet qui a des fonctionnalités et une syntaxe agréable et moderne.

Certaines personnes peuvent penser que C# a été conçu par Microsoft pour contrer Sun, et que son seul but est d'ordre financier. Ce qui n'est pas vrai car C# apporte des innovations qui s'intègre bien aux besoins de l'environnement.

Sun et Microsoft ne sont plus en mauvais terme : les deux firmes ont passé un accord mettant définitivement fin à leur différend sur Java et .Net.

Microsoft versera plusieurs Millions de dollars pour mettre fin à leur contentieux pour pratiques anticoncurrentielles et pour solder un litige sur des brevets.

De plus, Sun et Microsoft se verseront des royalties lorsqu'ils utiliseront la technologie de l'autre.

C# se dit lui aussi un descendant de C++, or force est de constater que de nombreuses innovations introduites par Java ont été reprises par C# souvent en respectant la syntaxe d'origine.

Si bien sûr, C# a repris de nombreux éléments novateurs de Java à son compte, la volonté de Microsoft d'écrire plus qu'un simple clone amélioré de Java semble réellement sincère.

# I. Les Similitudes

Java et C# ont de nombreux concepts et caractéristiques en commun. Voici une énumération de leurs principales similitudes.

- **Chaîne de caractères**

*Les chaînes de caractères* en C# et en Java ne peuvent être modifiées afin de préserver le principe d'encapsulation, on ne peut donc pas modifier leurs valeurs. Java possède la classe `java.lang.String` et C# la classe `System.String`. Dans les deux langages pour créer une chaîne de caractères pouvant être modifiée on utilise la classe `java.lang.StringBuffer` en Java et son équivalent en C# `System.Text.StringBuilder`.

- **Classe mère**

C# et Java possèdent tout deux une super classe ou *classe mère*. En java toutes les classes sont des sous-classes de `java.lang.Object` et en C# les classes sont des sous-classes de `System.Object`. Les méthodes présentes dans les deux classes sont relativement similaires comme par exemple la méthode `toString()`.

- **Compilation**

En ce qui concerne la *compilation*, C# et Java fonctionnent de la même manière. Le bytecode Java trouve son équivalent avec le MSIL (Microsoft Intermediate Language), celui-ci est exécuté par le CLR (Common Language Runtime), équivalent de la machine virtuelle chez Microsoft. Le MSIL est alors traduit en langage natif via un compilateur JIT (Just in Time). Java peut lui aussi compiler le bytecode en langage natif via ce type de compilateur.

- **Gestion de la mémoire**

C# et Java emploie la *gestion automatique de la mémoire* qui libère les développeurs des tâches d'allocation et de libération manuelle de la mémoire occupée par les objets. Les stratégies de gestion automatique de la mémoire sont implémentées par un garbage collector (également appelé 'ramasse-miettes'). Il existe dans les deux langages la possibilité de "forcer" le processus, en sachant que la manière du C# est plus directe que celle de Java.

- **Héritage**

Dans les deux langages *l'héritage* multiple n'est pas permis, il existe seulement l'héritage simple. Mais en C# comme en Java on a la possibilité d'implémenter plusieurs interfaces. Les interfaces permettant de faire du polymorphisme.

- **Méthode globale et initialisations des variables**

Le code de C# et de Java contrairement à celui de C++ ne doit pas avoir de *méthode globale* il doit avoir seulement des méthodes intégrées dans des classes.

Dans les deux langages les variables d'instances et les variables statiques peuvent être initialisées dès leurs déclarations. De plus l'initialisation d'une variable membre sera effectuée avant l'appel du constructeur si cette variable est une variable d'instance.

- **Mots-clés**

Les *mots-clés* des deux langages sont eux aussi assez proches, la plupart des mots-clés de Java ont un équivalent en C#. (cf. le tableau dans la section II a.).

## II. Les Pseudos similitudes

### a. Syntaxique

Les langages C# et Java ont une syntaxe relativement similaire, héritée en grande partie de C et de C++.

Pour preuve l'exemple du classique « hello world ! ». Voici son implémentation, d'abord en java puis en c# :

Java Code

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

C# Code

```
using System;  
class HelloWorld {  
    public static void Main(String[] args) {  
        Console.WriteLine("Hello World");  
    }  
}
```

Nous constatons bien que la syntaxe présente des similitudes mais aussi, bien évidemment, des différences.

Java est un langage dans lequel le programmeur n'est pas libre pour la syntaxe, il est sensible à la casse, et il impose certaines règles : La majuscule en début de nom de classe qui sert à différencier les noms de classe et la minuscule au début des méthodes ne sont pas obligatoire mais sont recommandés.

C# est beaucoup plus souple au niveau de la syntaxe, il permet par exemples de nommer des classes Class1 et class1, et de nommer des méthodes Methode1 et méthode1. Classe1/classe1 et Methode1/methode1 sont des noms de classes et de méthodes différents.

Ainsi C# n'impose pas d'avoir au plus une classe publique par fichier contrairement à Java, qui de plus impose que le nom du fichier source doit être le même que celui de la classe publique.

- **Accesseurs et Modificateurs**

La syntaxe des *accesseurs et modificateurs* de Java a pour équivalent les propriétés en C#. Cela permet d'accéder à une classe en respectant les règles d'encapsulation.

Voici un exemple illustrant la syntaxe de ses méthodes :

Java Code

```
System.out.println(maVoiture.getImmatriculation());  
//...  
public int getImmatriculation(){  
    return immatriculation;  
}
```

C# Code

```
using System;  
    Console.WriteLine(maVoiture.Immatriculation);  
//...  
public int Immatriculation(){  
    get {  
        return immatriculation;  
    }  
}
```

- **Déclaration de tableau**

La syntaxe pour la *déclaration de tableau* est la même pour les 2 langages à la différence que Java permet la déclaration de tableau à la C.

Exemple :

C# Code

```
int [] monTableau = new int[10]
```

Java Code

```
int [] monTableau = new int[10]  
float monTableau2[] = new float[100]
```

- **Exceptions**

La syntaxe et l'utilisation des *exceptions* de C# et Java partagent énormément de caractéristiques. La principale différence vient du fait que le mot-clé `throws` n'existe pas en C#, par conséquent, on ne déclare pas les exceptions pouvant être levées dans la signature de la méthode. Les blocs `try/catch` et `finally` existent également dans les deux langages et **ont la même syntaxe**.

- **Héritage**

La syntaxe de l'*héritage* en C# diffère de celle du Java.

C# n'utilise pas les mots-clés `extends` ou `implements`, il utilise la syntaxe de C++ avec le symbole ":".

Voici un exemple simple illustrant la syntaxe de l'héritage :

Java Code

```
Public class B extends A implements Comparable{
    int compareTo(){}
    public static void main(String[] args){
        System.out.println("Hello World");
    }
}
```

C# Code

```
using System;
public class B:A,IComparable{

    int CompareTo(){}
    public static void Main(String[] args){
        Console.WriteLine("Hello World");
    }
}
```

Microsoft pour des raisons de lisibilité du code a défini (par convention) que les noms d'interfaces doivent commencer par I (ex : IMyInterface).

- **Méthode main**

Le profil de la *méthode main* présente une différence mineure dans la syntaxe :

En C# la signature est `public static void Main(String [] args)` et en java : `public static void main(String [] args)`.

- **Mots-clés**

Voici un tableau récapitulant les *mots-clés* C# ainsi que leurs équivalents en Java:

<b>C#</b>	<b>Java</b>	<b>C#</b>	<b>Java</b>	<b>C#</b>	<b>Java</b>
abstract	abstract	get	---	short	short
as	---	goto	goto	sizeof	---
base	super	if	if	stackalloc	---
bool	boolean	implicit	---	static	static
break	break	in	---	string	---
byte	---	int	int	struct	---
case	case	interface	interface	switch	switch
catch	catch	internal	protected	this	this
char	char	is	instanceof	throw	throw
checked	---	lock	synchronized	true	true
class	class	long	long	try	try
const	const	namespace	package	typeof	---
continue	continue	new	new	uint	---
decimal	---	null	null	ulong	---
default	default	object	---	unchecked	---
delegate	---	operator	---	unsafe	---
do	do	out	---	ushort	---
double	double	override	---	using	import
else	else	params	---	value	---
enum	---	private	private	virtual	---
event	---	protected	---	void	void
explicit	---	public	public	while	while
extern	native	readonly	---	:	extends
finally	finally	ref	---	:	implements
fixed	---	return	return	---	strictfp
float	float	sbyte	byte	---	throws
for	for	sealed	final	---	transient
foreach	---	set	---	---	volatile

- **Namespaces et Package**

La syntaxe des **namespaces**, qui est l'équivalent du **package** de Java, est plus proche de C++ que de Java. La structure arborescente des packages en Java impose que les noms des paquetages correspondent au nom relatif du répertoire contenant les fichiers .class, ce qu'il n'est pas le cas en C#.

Une autre différence est que la syntaxe de C# permet la déclaration de plusieurs **namespaces** à l'intérieur d'un même fichier.

C# Code

```
using System;

namespace Classe A {
    public class MaClasse {
        /* Classe A.MaClasse*/
        int x;
        void m();
    }

    namespace Classe B {
        public class MaClasseBis {
            /* Classe A.ClasseB.MaClasseBis*/
            ....
        } //classe MaClasseBis
    } //namespace Classe B
} //namespace Classe A
```

- **Synchronisation des méthodes**

C# et Java supportent les *méthodes synchronisés*. L'équivalent du **synchronized** de Java est le mot-clé **lock**. Les instructions **synchronized** et **lock** assurent le verrouillage par exclusion mutuelle d'un objet particulier, exécute une instruction puis annule le verrouillage. Les méthodes synchronisées vont servir lors de l'utilisation des **threads** (cf. II b.).

Voici un exemple montrant l'utilisation des méthodes synchronisées en C# puis en Java :

C# Code

```
lock (objet) { //pose du verrou
    //code
} //déverrouillage de l'objet
```

Java Code

```
synchronized(objet){ //pose du verrou
    //code
} //déverrouillage de l'objet
```

## b. Concepts identiques mais géré différemment

C# et Java possèdent de nombreux concepts en communs dont voici l'énumération.

- **Accessibilité**

Java et C# n'ont pas les mêmes *accessibilités*. Voici un tableau récapitulatif des possibilités de visibilité et d'accès aux classes :

C#	Java
private	private
public	public
internal	pas d'équivalent
protected	pas d'équivalent
internal protected	protected

Sun a décidé de ne pas garder la même *accessibilité* que Java. En effet l'accès **protected** n'est pas le même entre les deux langages. Un membre **protected** en C# ne peut être accédé que par d'autres méthodes membres situées dans la même classe ou dans une classe dérivée mais n'est en aucun cas visible de l'extérieur, alors qu'en Java les membres **protected** sont à la fois visible dans les classes fille et dans les classes du même paquetage. L'équivalent du **protected** de Java est **internal protected**.

En C# le mot clé **internal** rend accessible tout ce qui est *dans* la même **assembly** (plus similaires au **Jar** de Java).

On peut noter également que la visibilité par défaut d'un champ ou d'une méthode est **private** en C# et **package** (protection plus restrictive que **protected**) en Java.

On peut noter que C# impose des contraintes d'accessibilités. Par exemple les exemples suivant sont sources d'erreur à la compilation :

```
class A {...}
public class B: A {...}
```

La classe B est source d'erreur de compilation car A n'est pas au moins aussi accessible que B.

De même, dans l'exemple :

```
class A {...}
public class B {
    A F() {...}
    internal A G () {...}
    public A H() {...}
}
```

La méthode H dans B est source d'erreur de compilation car le type de retour A n'est pas au moins aussi accessible que la méthode.

- **Appel du constructeur père**

En cas de création d'un objet si la première instruction d'un constructeur n'est pas **super(...)** en Java et **base(...)** en C# les deux langages vont appelés implicitement le constructeur père. C# et Java permettent tout les deux la surcharge de constructeur ainsi que l'appel inter constructeur.

Voici un exemple simple illustrant l'appel au constructeur de la classe mère :Java

Code :

```
Public class Moto extends DeuxRoue {
    private float puissanceMoteur ;
    public Moto(int nbPassager) {
        this(nbPassager) ;
    }
    public Moto(int nbPassager, float puissance) {
        super(nbPassager) ;
        this.puissanceMoteur=puissance ;
    }
}
```

C# Code :

```
using System;
class Moto: DeuxRoue {
    private float puissanceMoteur ;
    public Moto(int nbPassager): this(nbPassager) {}
    public Moto(int nbPassager, float puissance): base(nbPassager) {
        this.puissanceMoteur=puissance ;
    }
}
```

- **Collection**

Il existe de nombreux points communs entre les *collections* en C# et de Java. Les collections de C# font partie du **namespace** System.Collections et pour Java du **package** java.util. Le package Java est plus complet que son homologue car il propose des caractéristiques supplémentaires comme notamment les classes concrètes TreeSet ou HashSet. C# se distingue de Java concernant les collections sur la façon dont il gère la synchronisation. A noter qu'en Java certaines classes ne sont pas synchronisées (Hashtable, ...) et d'autre le sont (ArrayList, ...).

- **Constructeur et destructeur**

Java et C# ont la même syntaxe et sémantique pour les *constructeurs*, mais C# propose en plus la notion de *destructeur*. Un destructeur est un membre qui implémente les actions requises pour détruire une instance d'une classe. Les destructeurs C#, sont déclarés à l'aide du « ~ ». La sémantique des destructeurs de C# est la même qu'un **finaliseur** en Java.

Exemple d'utilisation d'un constructeur et d'un destructeur :

C# Code :

```
using System;

public class MyClass {
    MyClass() {
        Console.WriteLine("Creation objet");
    }
    ~MyClass() {
        Console.WriteLine("Destruction objet");
    }
}
```

Java Code :

```
public class MyClass {
    MyClass() {
        System.out.println("Creation objet");
    }

    public void finalize() {
        System.out.println("Object finalisé");
    }
}
```

- **Génération de documentation**

La *génération de la documentation* à partir du code source est relativement similaire dans les deux langages. La différence majeure provient du code généré à partir des fichiers sources. Javadoc, qui est l'outil de Java pour générer la documentation, va créer du code HTML alors que C# va générer du code XML. Le code XML pourra facilement être converti d'en d'autres formats comme le HTML par exemple. Pour obtenir la documentation le code source doit être commenté suivant certaines règles.

Voici un exemple illustrant la façon de documenter le texte source :

Java Code

```
/**
 * Réalise la division de a par b
 * @param divisé: a divisant: b.
 * @return a/b.
 * @exception DivisionParZeroException si b=0
 */
public static int division(int a, int b) throws DivisionParZeroException {}
```

C# Code

```
///<summary>Réalise la division de a par b.</summary>
///<param name="num">divisé: a divisant: b.</param>
///<return>a/b.</return>
///<exception>DivisionParZeroException si b=0.</exception>
public static int division(int a, int b) throws DivisionParZeroException {}
```

- **Goto**

L'instruction **goto** est une instruction qui permet de réaliser un saut direct vers un label dans un programme. Cette instruction est considérée comme « dangereuse » dans la plupart des langages de programmation car elle est souvent source d'erreur. Certains programmeurs reprocheront à Microsoft de l'avoir maintenue.

Voici un exemple de code en C# utilisant le **goto** :

```

using System;
class Test{
    static void Main(string[] args){
        string[,] table = {
            {"Red","Blue","Green"} ,
            {"Monday","Wednesday","Friday"}
        };
        foreach(string str in args){
            int row,colm;
            for(row=0;row<=1;++row)
                for(colm=0;colm<=2;++colm)
                    if(str == table[row,colm])
                        goto done;
            Console.WriteLine("{0}not found",str);
        done :
            Console.WriteLine("Found {0} at [{1}][{2}]",str,row,colm);
        }
    }
}

```

- **Méthode virtuelles**

La notion de *méthode virtuelles* n'est pas la même en Java qu'en C#. Les méthodes virtuelles vont permettre de traiter un concept très important dans les langages de programmation objet qui est le polymorphisme. Par défaut toutes les méthodes de Java sont virtuelles alors qu'en C# *les méthodes virtuelles* doivent l'être explicitement à l'aide du mot-clé **virtual**.

Si on veut empêcher que des méthodes soient redéfinies on utilise les mots clés **final** en Java et **sealed** en C#.

Exemple de code utilisant le polymorphisme :

## C# Code

```
using System;

public class Parent
{
    public virtual void m(string str)
    {
        Console.WriteLine("Parent: " + str);
    }
}

public class Enfant: Parent
{
    public override void m(string str)
    {
        Console.WriteLine("Enfant: " + str);
    }
}
```

## Java Code

```
class Parent
{
    public void m(String str)
    {
        System.out.println("Parent: " + str);
    }
}

class Enfant extends Parent
{
    public void m(int n)
    {
        System.out.println("Enfant: " + n);
    }
}
```

En C# une méthode d'instance contenant le modificateur **override** est dite méthode **override**. Une méthode **override** se substitue à une méthode virtuelle héritée ayant la même signature. Alors qu'une déclaration de méthode virtuelle présente une nouvelle méthode, une déclaration de méthode **override** spécialise une méthode virtuelle héritée existante en fournissant une nouvelle implémentation de cette méthode.

- **Sérialisation**

La *sérialisation* en C# s'utilise en utilisant les attributs personnalisés **[Serializable]** et **[NonSerialized]**. L'attribut **[Serializable]** indique que les objets sont sérialisables et l'attribut **[NonSerialized]** indique qu'ils ne le sont pas.

En Java les objets sérialisables sont ceux qui implémentent l'interface `Serializable` et les champs non sérialisable sont précédé du mot-clés **transient**.

Exemple de sérialisation en C# :

```
using System;

[Serializable]
public class Animal
{
    public string Genre;
    public string Espece;
    public string Race;
    public bool int nbrMembres;

    public void ClassificationAnimal()
    {
        Console.WriteLine("{0} / {1} / {2}.", Genre, Espace, Race);
    }

    [NonSerialized]
    public void NombreDeMembres()
    {
        Console.WriteLine("Cette race compte {0} membres.", nbrMembres);
    }
}
```

Exemple d'utilisation des mots-clés en Java pour illustrer la sérialisation :

```
import java.io.*;
import java.util.*;
public MaClasse implements Serializable {
    private String monPremierAttribut;
    private transient int monSecondAttribut; // non sérialisé
    // ...éventuelles méthodes ...
}
```

- **Surcharge des opérateurs**

La *surcharge des opérateurs* permet en C# à une classe ou un **struct** de déclarer plusieurs opérateurs avec le même nom, sous réserve que leurs signatures soient uniques dans cette classe ou dans cette structure (**struct**).

La surcharge d'opérateur, qui a été introduite par C++, doit permettre au programmeur d'ajouter des significations différentes à presque tous les opérateurs. En Java la surcharge d'opérateur est seulement possible avec les **strings** (opérateur +). Cela s'utilise couramment par exemple pour concaténer 2 chaînes de caractères.

Exemple de code en C# utilisant la surcharge d'opérateur :

```
public static Complex operator +(Complex c1, Complex c2)
{
    return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
}
```

- **Switch**

L'instruction **switch** présente 2 différences en C# et en Java. La première est que C# autorise l'utilisation d'une string dans l'expression entre parenthèse qui suit le mot-clés **switch**, ce qui permet une plus grande souplesse par rapport à Java, et la deuxième différence est que C# interdit le « fall-through », c'est-à-dire qu'un label ne peut exécuter plusieurs ligne lorsque l'instruction break est absente. C'est un choix volontaire qu'a fait Microsoft de désactiver les « fall-through » car ils sont sources d'erreur, souvent difficiles à détecter, dans d'autres langages (notamment Java).

- **Threads**

Java et C# gèrent les **threads** différemment. En Java, deux possibilités existent pour créer un **thread**. On peut utiliser une classe dérivée de la classe `java.lang.Thread`, ou on peut utiliser l'interface `java.lang.Runnable`. On a donc le choix entre l'héritage ou la délégation. En C# il faut d'abord créer un objet de type `System.Threading.Thread` et le passer à un "delegate" `System.Threading.ThreadStart`. Ce delegate (comparable à un pointeur de fonction en C), est initialisé avec la méthode destinée à devenir un **thread**. Contrairement à Java qui impose comme point d'exécution d'un **thread** la fonction `run()`, il est possible en C# de spécifier n'importe quelle fonction. En Java, toutes les classes héritent des méthodes `wait()`, `notify()` and `notifyAll()` situées dans `java.lang.Object`, en C# les classes hérités se trouvent dans la classe `System.Threading.Monitor`.

- **Type primitif**

Pour chaque *type primitif* en Java, il existe un correspondant en C# avec le même nom, excepté **byte**. Le **byte** en Java est signé et est ainsi proche du type **sbyte** de C# (et non **byte**). De plus, C# possède des versions non signées pour la plupart des types : **ulong**, **uint**, **ushort** et **byte**. La différence majeure provient du type **decimal** qui n'existe pas en Java. Le type **decimal** n'effectue aucun arrondi au prix d'un surplus de place et d'une rapidité de traitement moindre.

## III. Les Différences

### a. Fonctionnalités propres à C# :

- **Alias**

Mécanisme similaire au typedef en C/C++.

Le mot clé "**using**" augmente la lisibilité du code en définissant des raccourcis :

```
using identifiant = namespace-or-type-name ;
```

Il permet d'assigner à un nom donné un type correspondant :

```
namespace unice.package1
{
    class A {}
}
namespace package2
{
    using A = unice.package1.A;
    class B : A {}
}
```

- **Attributs (méta données) (présent dans le j2sdk1.5)**

Le langage C# permet au programmeur de spécifier des informations déclaratives (appelées attributs) au sujet des entités définies dans le programme.

Les métas données sont un peu comme la javadoc qui génère de la documentation, sauf que eux génèrent du code.

Le développeur indique le type de code que le compilateur doit générer en lui passant des arguments sous la forme d'annotations insérées directement dans le code. Cette approche permet d'obtenir un code plus concis et plus visible.

Il existe des attributs prédéfinis comme par exemple l'attribut `AttributeUsage` qui permet de décrire le mode d'utilisation d'une classe d'attributs (voir exemple ci dessous).

Mais les développeurs peuvent créer leur propres attributs de Runtime en sous-classant `System.Attribute` : par exemple :

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class AideAttribute: Attribute {
    public AideAttribute(string url){
        ...
    }
    public string Sujet {
        get{...}
        set{...}
    }
    public string Url {
        get{...}
    }
}
```

La classe d'attributs pourrait être utilisée de la manière suivante :

```
[Aide("http://www.tutoriels.com/./meta-donnees.html")]
class classMetaDonnee {
    ...
}
[Aide(http://www.tutoriels.com/./sujets.html", Sujet = "TrucBidul")]
class classTrucBidul {
    ...
}
```

Cette innovation de Microsoft a pour but de faciliter la programmation par composants qui nécessitait avant C# des descripteurs séparés du code source, et va au delà en répondant à d'autres besoins.

- **AutoBoxing (Boxing/Unboxing) (présent dans le j2sdk1.5)**

Le concept du boxing et unboxing est au centre du système de types de C#. Il fournit un pont entre les types primitifs (valeur) et les objets (références) en permettant de convertir toute valeur d'un type primitif vers et depuis le type objet. Les conversions boxing et unboxing permettent une vue unifiée du système des types dans laquelle une valeur de n'importe quel type peut en définitive être traitée comme un objet.

En Java on utilise les classes Wrapper (Integer, Double,..) pour effectuer cette conversion et on accède aux valeurs par des méthodes..

En C# cela se passe de façon beaucoup plus simple :

```
Public static void boxingUnboxing(int I, double d,string s) {  
    object obj_entier = i; //en Java : Integer obj_entire = new Integer(i);  
    object obj_double = d;  
    object obj_string = s;  
    Console.WriteLine("i: "+(int)obj_entier+ " d: "+(double)obj_double+"...");  
    //en Java : "i "+obj_entier.intValue() ...  
}
```

Avec deux Casts (un en entrée et un en sortie) la conversion est effectuée.

- **Caractères spéciaux**

Les chaînes de caractères dites "verbatim" permettent de se passer du doublement des caractères spéciaux.

La seule contrainte est de préfixer la déclaration de la chaîne avec le symbole @ :

```
string filename = @"C:\My Documents\My Files\File.html";
```

- **Délégués**

Les délégués activent des scénarios que d'autres langages, tels que C et C++ (et d'autres : Pascal..) traitent à l'aide de pointeurs de fonction.

Contrairement aux pointeurs de fonction C++, les délégués sont complètement orientés objet et, à la différence également des pointeurs de fonction C++, les délégués encapsulent à la fois une instance d'objet et une méthode.

Effectivement nous avons la possibilité de passer l'adresse d'une fonction en paramètre d'une méthode.

Une déclaration delegate définit une classe dérivée de la classe System.Delegate et peut être faite dans un namespace ou dans une classe :

```
using System;
delegate void D(int i);
class C {
    public static void M1(int i){..}
    public static void M2(int i){..}
}
class Test {
    static void main(){
        D cd1 = new D(C.M1); //M1
        D cd2 = new D(C.M2); //M1
        D cd3 = cd1 + cd2; //M1+M2
    }
}
```

- **Détection débordement (checked/unchecked)**

C# fournit une option pour explicitement détecter ou ignorer les débordements dans le cas de conversion vers un type plus faible. Le débordement lève une exception du type `System.OverflowException` et comme la détection met en place des mécanismes pouvant affecter les performances, il est possible de l'activer ou de la désactiver en spécifiant les mots-clés `checked` et `unchecked` au début de blocs :

```

class Test {
    static int x = 1000000;
    static int y = 1000000;
    static int F(){
        return checked(x * y); //Throws OverflowException
    }
    static int G(){
        return unchecked(x * y); //Returns -727379968
    }
}

```

- **Directives pré-processing**

Le compilateur ne dispose pas de préprocesseur distinct, mais les directives décrites dans cette section sont traitées comme s'il en existait un.

Ces directives permettent de faciliter la compilation conditionnelle. Contrairement aux directives C et C++, ces directives ne permettent pas de créer des macros.

Une directive de préprocesseur doit être la seule instruction sur une ligne.

```

#define DEBUG
using system;
class Test {
    public static void Main(string[] args){
        #if DEBUG
            Console.WriteLine("Mode Debug")
        #else
            Console.WriteLine("Mode Normal");
        #endif
    }
}

```

- **Enumérations (présent dans le j2sdk1.5)**

Le mot clé "**enum**" permet de regrouper les types définis par l'utilisateur.

Il s'agit d'un raccourci syntaxique :

Ex :

```
enum Color{  
    Red,  
    Green,  
    Blue  
}
```

Déclare un type enum nommé Color avec les membres Red, Green et Blue.

Le code gagne alors en fiabilité puisque le compilateur C# est alors capable de vérifier automatiquement que les types employés le sont correctement (Le manque de type d'énumération dans Java (avant la j2sdk1.5) contraint à utiliser des valeurs entières ne garantissant pas la sécurité des types).

- **Génération de code à l'exécution**

Le package `Reflection.Emit` contient un ensemble de classes pouvant être utilisées afin de générer des instructions .NET à l'exécution. Ces instructions sont ensuite compilées en mémoire et peuvent être stockées physiquement sur disque sous la forme d'une assembly. Ce mécanisme existe dans Java mais n'est pas proposé en standard dans les APIs. Les moteurs de Servlets/JSP l'utilisent pour générer le source d'une servlet et pour la charger en mémoire. Les conteneurs EJB s'en servent pour générer l'implémentation des classes techniques (EjbObject) ou le code de persistance des Entity. Malheureusement, ces classes se trouvent dans les packages `sun.tools.*` (`tools.jar`), ce qui limite fortement leur utilisation.

- **Implémentation explicite d'interface**

Lors de l'implémentation de plusieurs interfaces, Java ne gère pas le cas de deux méthodes qui ont la même signature (2 méthodes homonymes peuvent signifier des choses différentes).

C# propose une alternative très simple d'utilisation permettant d'implémenter explicitement ces interfaces :

```
public interface VehiculeRoulant { void PreparerAAvancer(); }
public interface VehiculeVolant { void PreparerAAvancer(); }
public class Ovni : VehiculeRoulant, VehiculeVolant {
    void VehiculeRoulant.PreparerAAvancer() {
        RentrerElice();
        SortirRoues();
    }
    void VehiculeVolant.PreparerAAvancer() {
        RentrerRoues();
        SortirElice();
    }
    private void RentrerRoues() { Console.WriteLine("RentrerRoues"); }
    private void RentrerElice() { Console.WriteLine("RentrerElice"); }
    private void SortirRoues() { Console.WriteLine("SortirRoues"); }
    private void SortirElice() { Console.WriteLine("SortirElice"); }
}
// ...
private void PiloterVehiculeRoulant(VehiculeRoulant v) {
    v.PreparerAAvancer();
}
private void PiloterVehiculeVolant(VehiculeVolant v) {
    v.PreparerAAvancer();
}
// ...
Ovni ovni = new Ovni();
PiloterVehiculeRoulant(ovni);
Console.WriteLine("---");
PiloterVehiculeVolant(ovni);
```

- **Instruction « for-each » (version similaire dans le j2sdk1.5)**

L'instruction foreach énumère les éléments d'une collection, en exécutant une instruction incorporée pour chaque élément de la collection.

```
string[] color_drapeauFrance = {"bleu", "blanc", "rouge"};
foreach (string str in color_drapeauFrance)
    Console.WriteLine(str + "est une couleur du drapeau français");
```

- **Indexeurs**

Un indexeur est un membre qui permet à un objet d'être indexé de la même manière qu'un tableau. Ils peuvent aussi être surchargés.

La syntaxe est la suivante :

```
... type this [ liste parametres ] { declarations accesseurs }
```

En Java, les indexeurs sont implémentés à travers des accesseurs. Ainsi, en Java on écrirait : `myList.getElement(1)` et en C# : `myList[1]`.

L'exemple suivant montre une classe grille pourvue d'un indexeur avec deux paramètres :

```
using System;
class Grid {
    const int NumRows = 26;
    const int NumCols = 10;
    int[,] cells = new int[NumRows, NumCols];
    public int this[char c, int col] {
        get {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') { throw new ArgumentException(); }
            if (col < 0 || col >= NumCols) { throw new IndexOutOfRangeException(); }
            return cells[c - 'A', col];
        }
    }
}
```

```

set {
    c = Char.ToUpper(c);
    if (c < 'A' || c > 'Z') { throw new ArgumentException(); }
    if (col < 0 || col >= NumCols) {throw new IndexOutOfRangeException(); }
    cells[c - 'A', col] = value;
}
}
}

```

Pour modifier une valeur du tableau cells il suffit d'écrire :

```

Grid g = new Grid();
g[1,2] = 4;

```

- **Libération explicite de mémoire**

Si une application utilise une ressource externe coûteuse, il est recommandé de permettre de libérer explicitement la ressource avant que le garbage collector libère l'objet. Pour cela il faut implémenter une méthode *Dispose* (à partir de l'interface *System.IDisposable*) qui effectue le nettoyage nécessaire pour l'objet.

Effectivement l'interface *IDisposable* fournit la méthode *Dispose* pour des classes de ressource qui implémentent l'interface. Étant donné qu'elle est publique, les utilisateurs peuvent appeler la méthode *Dispose* directement pour libérer la mémoire utilisée par des ressources non managées.

Concrètement, le fait d'appeler la méthode *Dispose()* invoque la méthode *GC.SuppressFinalisation(this)* qui supprime l'objet de la file de finalisation.

A ne pas confondre avec la Finalisation proposée par Java qui n'assure pas une libération "déterministe" des objets.

- **Liste variables de paramètre (présent dans le j2sdk1.5)**

Il est possible de spécifier qu'une fonction prend un nombre variable de paramètres. Ce paramètre doit être déclaré avec un modificateur **params** ce qui correspond à un tableau de paramètres.

Hormis qu'il autorise un nombre variable d'arguments dans un appel, un tableau de paramètres est l'équivalent parfait d'un paramètre de valeur du même type.

```
using System;
class Test {
    static void F(params int[] args) {
        Console.WriteLine("F contient {0} arguments:", args.Length);
        foreach (int i in args)
            Console.WriteLine(" {0}", i);
    }
    static void Main() {
        int[] varargs = {1, 2, 3};
        F(varargs);
        F(1, 2, 3, 4);
        F();
    }
}
```

Donne le résultat suivant :

```
F contient 3 arguments: 1 2 3
F contient 4 arguments: 1 2 3 4
F contient 0 arguments:
```

- **Opérateur « as »**

L'opérateur **as** permet d'effectuer des conversions entre des types compatibles.

L'opérateur **as** est semblable à un cast si ce n'est qu'il fournit la valeur null en cas d'échec de conversion au lieu de lever une exception.

De manière plus formelle, une expression de la forme :

```
expression as type
```

Équivaut à :

```
expression is type ? (type)expression : (type)null
```

- **Passage par références**

En Java, les arguments sont passés par valeur.

En C#, en plus du passage par valeur, le passage par référence est autorisé via le mot clé "**ref**".

Contrairement à un paramètre de valeur, un paramètre de référence ne crée pas de nouvel emplacement de stockage. Il représente plutôt le même emplacement de stockage en tant que variable fournie comme argument dans l'appel de méthode :

```
class Test {  
    static void Swap(ref int x, ref int y) {  
        int temp = x;  
        x = y;  
        y = temp;  
    }  
    static void Main() {  
        int i = 1, j = 2;  
        Swap(ref i, ref j);  
        Console.WriteLine("i = {0}, j = {1}", i, j);  
    }  
}
```

- **Pointeurs/code non protégé (unsafe)**

Malgré que les pointeurs ne soient pas gérés de manière intégré, il existe un mode "non protégé" qui permet leur utilisation.

Ainsi le mot clé **unsafe** indique un contexte non sécurisé, qui est requis pour toute opération impliquant des pointeurs.

Le mot-clé **unsafe** permet au CLR de baisser la garde. Moins de contrôle sont effectués, les pointeurs sont alors autorisés.

```
unsafe static void FastCopy ( byte[] src, byte[] dst, int count )
{
    // unsafe context: possibilités d'utiliser des pointeurs ici
}
```

- **Propriétés**

Une propriété est un membre qui permet d'accéder à une caractéristique d'un objet ou d'une classe. En java ce sont les accesseurs que l'on utilise pour cela.

Mais l'avantage des propriétés en C# est de permettre à l'utilisateur d'accéder aux attributs d'un objet de la même manière que s'il effectuait directement l'opération `object.attribut` alors qu'en réalité il appelle une méthode de manière totalement transparente.

```
class A {
    private string name;
    private static int age;
    public string Name {           //read-write
        get { return name; }
        set { name = value; }
    }
    public static int MinimumAge { //read-only
        get { return minimum_age; }
    }
}
```

- **Structures (Types valeurs)**

Le type "**struct**" permet en C# d'optimiser la gestion de la mémoire.

C# réutilise le mot-clé "**struct**" avec un nouveau sens : une structure est une classe dont toutes les instanciations seront faites sur la pile à savoir que les allocations sur la pile sont beaucoup plus rapide que celles dans le tas.

Ca signifie que :

- \* le passage d'une instance de structure en paramètre se fait par valeur.
- \* pas d'allocation dynamique.
- \* interdiction d'affecter 'null' à une variable de type struct .

Ceci permet d'économiser de la mémoire et du temps.

Pour déclarer une structure, il suffit de d'utiliser le mot-clé **struct** au lieu de class .

Lorsque vous employez l'opérateur **new** pour créer un objet struct, celui-ci est créé et le constructeur approprié est appelé. Contrairement aux classes, les structs peuvent être instanciées sans recourir à l'opérateur **new**. Si vous n'utilisez pas cet opérateur, les champs ne seront pas assignés et l'objet ne pourra pas être utilisé tant que tous les champs n'auront pas été initialisés :

```
struct Point {  
    public int x;  
    public int y;  
    public Point( int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public override string ToString() {  
        return String.Format("{0}, {1}", x, y);  
    }  
    public static void Main(string[] args) {  
        Point debut = new Point(1,1);  
        Console.WriteLine("Début: " + debut);  
        Point fin = new Point(); // Ne compilerai pas si Point était une classe  
        Console.WriteLine("Fin: " + fin);  
    }  
}
```

### c. Contre attaque de Java ->J2SDK 1.5

La sortie de la Java2 Platform Standard Edition 1.5 est prévue pour l'été 2004 (déjà des versions bêta sont sorties).

Cette dernière constitue une étape significative pour le langage et la plateforme Java. Plusieurs des fonctionnalités de C# absente dans Java se retrouvent présente avec la version 1.5 de la J2SE.

Nous exposerons les principales nouveautés de cette dernière :

- **Autoboxing**

Java à repris le concept de C# qui permet d'éliminer les conversions manuelles entre les types primitifs et les Wrapper (Integer,...).

Java stocke les types primitifs dans la pile contrairement aux objets qui se trouvent dans le tas. Une collection d'objets ne permet pas d'accepter des types primitifs directement, il faut pour se faire passer les classes enveloppes (Wrapper).

On pourra dorénavant utiliser Integer ou int plus ou moins de la même façon, c'est à dire mettre un 1 dans une hashtable par exemple au lieu d'obligatoirement le transformer en objet.

Avec version 1.4.x et antérieur :

```
public class Freq {
    public static void main(String[] args) {
        Map m = new TreeMap();
        m.put(args[1], new Integer(1));
    }
}
```

Avec la version 1.5 :

```
public class Freq {
    public static void main(String[] args) {
        Map m = new TreeMap();
        m.put(args[1],1);
    }
}
```

- **Enumération**

Le J2sdk 1.5 introduit le concept d'énumération dans le langage. Il s'agit en fait d'un raccourci syntaxique qui fonctionne comme en C# comme nous le faisons ici pour la class Color (même syntaxe qu'en C#):

```
public enum Color{  
    RED,GREEN,BLUE  
}
```

L'intérêt de l'intégration syntaxique est multiple :

- La forme est + lisible
- Les performances sont comparable à l'utilisation de constante entière.
- Les enums sont Comparable et Serializable
- Elles sont utilisables comme clés (pour les Map)
- Elles sont utilisables dans l'instruction switch.
- Ce sont des classes (on peut donc ajouter des constructeurs, des champs, des méthodes...)

- **"For" (améliorations)**

Le j2sdk 1.5 propose une extension de l'instruction de boucle for très similaire au **foreach** de C#. Cette instruction a pour but de simplifier grandement le parcours des séquences :

Avec version 1.4.x et antérieur :

```
Collection collec : ... ;  
For (iterator i = collec.iterator();i.hasNext();){  
    String s = (String)i.next();  
    faireDesTrucsAvec(s);  
}
```

Avec la version 1.5 :

```
Collection collec : ... ;  
For( String s :collec){  
    faireDesTrucsAvec(s);  
}
```

Cela fonctionne aussi avec des énumérations :

```
Enum Jour {  
    Lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche  
}  
  
for(Jour j : Jour.VALUES){  
    System.out.println(j) ;  
}
```

- **Généricités**

Les génériques (ou gabarits) sont une facilité syntaxique de compilation très (vraiment très) attendue en Java.

Le j2sdk 1.5 nous les donne enfin. Un générique est une classe ou une fonction paramétrée par des types. Ce sont les fameux template de C++.

Un compilateur Java compatible avec les génériques effectuera un contrôle des types non plus au moment de l'exécution mais lors de la compilation ce qui permet un gain en terme de sécurité.

Avec version 1.4.x et antérieur :

```
List words = new ArrayList();
```

Avec la version 1.5 :

```
List<String> words = new ArrayList<String>();
```

Ce qui engendre une facilité (pas besoin de caster) lorsque l'on veut récupérer l'élément de la collection :

Avec version 1.4.x et antérieur :

```
String title = ((String) words.get(i)).toUpperCase();
```

Avec la version 1.5 :

```
String title = words.get(i).toUpperCase();
```

Ceci est simple à expliquer, puisque nous connaissons le type paramétré par notre classe générique, le transtypage s'avère inutile et le compilateur effectue donc la conversion automatiquement sans risque de générer une erreur à l'exécution.

Note : Les génériques sont des artifices de compilation (certes très pratiques !) : Ils permettent de vérifier plus proprement les types des objets à la compilation ; ils permettent aussi d'utiliser une seule définition pour plusieurs cas d'utilisation.

Voici un exemple qui implémente les génériques, la nouvelle boucle for et l'autoboxing (cf Article : *Conversation with Joshua Bloch a senior staff engineer at Sun Microsystems*) :

Avec version 1.4.x et antérieur :

```
public class Freq {
    private static final Integer ONE = new Integer(1);
    public static void main(String args[]) {
        // Maps word (String) to frequency (Integer)
        Map m = new TreeMap();
        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE :
                new Integer(freq.intValue() + 1)));
        }
        System.out.println(m);
    }
}
```

Avec la version 1.5 :

```
public class Freq {
    public static void main(String args[]) {
        Map<String, Integer> m = new TreeMap<String, Integer>();
        for (String word : args)
            m.put(word, m.get(word) + 1); //le deuxième paramètre est un type primitif
converti automatiquement en un objet
        System.out.println(m);
    }
}
```

- **Importation Statique**

Le j2sdk 1.5 introduit les importations statiques. Le mot clé **static** est adjoint à la déclaration d'importation et modifie celle-ci de façon suivante : tous les champs et toutes les méthodes static de la classe importée deviennent utilisables sans avoir besoin d'être qualifiées complètement.

Ex :

```
Import static java.lang.Math ;
Class Test{
    Public double calcul(double x,double y){
        Double c = acos(x/y) ; //au lieu de Math.acos(x) !
    }
}
```

- **Méta données**

Java comme C# intègre les méta données à sa plate-forme. (cf. III a. : Attributs. pour explications).

En résumé cela mène à un style de programmation "déclaratif" où le programmeur dit ce qui devrait être fait et les outils émettent le code pour le faire.

Actuellement, une classe implémentant une interface se doit de redéfinir les méthodes contenues dans celle-ci comme le montre cet exemple:

```

public interface CoffeeOrderIF extends java.rmi.Remote {
    public Coffee [] getPriceList()
        throws java.rmi.RemoteException;
    public String orderCoffee(String name, int quantity)
        throws java.rmi.RemoteException;
}
public class CoffeeOrderImpl implements CoffeeOrderIF {
    public Coffee [] getPriceList() {
        ...
    }
    public String orderCoffee(String name, int quantity) {
        ...
    }
}

```

Avec les métas données, il n'est plus nécessaire de passer par cette méthode d'implémentation. On doit simplement annoter au code source qu'un outil prendra connaissance des méthodes à implémenter comme le montre bien l'exemple ci-dessous:

```

import javax.xml.rpc.*;
public class CoffeeOrder {
    @Remote public Coffee [] getPriceList() {
        ...
    }
    @Remote public String orderCoffee(String name, int quantity) {
        ...
    }
}

```

- **Varargs (Liste variables d'arguments)**

Le J2sdk 1.5 introduit une nouveauté syntaxique : les varargs, ou liste variables d'arguments. Cette nouveauté permet de définir des fonctions avec un nombre d'arguments non défini. Le même concept est présent dans C#.

En Java il s'agit simplement de spécifier la liste des arguments comme un tableau spécial d'arguments : un type de base suivi de 3 points. Ce tableau sera géré comme tout autre tableau. La différence se voit donc surtout lors de l'appel de fonction :

```
class Varargs {
    Public static String concat(String... args){
        String res = "";
        For(String s : args){
            Res += s + " ";
        }
        return res;
    }
}
class Tableau{
    Public static String concat(String[] args){
        String res = "";
        For(String s : args){
            Res += s + " ";
        }
        return res;
    }
}
public class Test{
    public static void main(String[] args){
        System.out.println(Tableau.concat(new String[] { "c'est", "long", "!" }));
        System.out.println(Varargs.concat("c'est", "moins", "long", "!"));
    }
}
```

## IV. Critères de choix

Java et C# sont deux langages qui se ressemblent beaucoup (cf. I, II & III), alors qu'est ce qui peut pousser à programmer en C# plutôt qu'en Java et inversement ?

Java a l'avantage d'être plus ancien donc plus éprouvé et il est très attractif car il dispose de beaucoup d'outils de développement, il a une constante évolution (qui est gérée par la JCP) et a des codes sources disponibles librement.

Mais Java n'a pas de véritable plateforme dédiée en tant que poste de travail, contrairement à C# (et .Net en général) qui a la plateforme Windows.

Les solutions connexes à C# (base de donnée, messagerie...) sont développées par le même éditeur, cela offre plus de facilité et moins de compétences pour développer des solutions intégrées.

Voici quelques points pouvant inciter un programmeur à choisir entre C# ou Java.

### **a. Développement multi langage**

Le multi langage désigne la capacité du code à interagir avec du code écrit dans un autre langage de programmation. L'interopérabilité des langages peut permettre d'optimiser l'utilisation du code et, par conséquent, améliorer l'efficacité du processus de développement.

L'infrastructure .NET s'est appropriée la primeur de l'aspect multi langage. Le multi langage occupe une place prédominante dans .Net et elle intègre plus d'une vingtaine de langages différents. Par exemple lorsqu'on utilise des classes fournies par le Framework elles peuvent très bien avoir été développées en C#, VB.NET ou en J#.

Pour que le code managé soit accessible aux développeurs utilisant un autre langage de programmation, le .NET Framework propose la spécification CLS (Common Language Specification) qui décrit un ensemble fondamental de fonctionnalités de langage et définit les règles d'utilisation de ces fonctionnalités.

La position de Java concernant cet aspect est assez délicate. En effet, si jusqu'à présent le problème ne se posait pas réellement car l'architecture J2EE imposait un seul et même langage il se pourrait qu'à l'avenir cette situation soit amenée à devenir pénalisante pour

Sun dans le cas où les développeurs s'éparpillent au gré de leurs envies dans le vaste choix proposé aujourd'hui.

Actuellement en Java il est possible de rendre un langage compatible avec la JVM, c'est le cas par exemples des langages Jython qui est une implémentation en langage Java du célèbre langage de script Orienté Objet Python, Talks2 qui est un environnement de développement Smalltalk, et J-Eiffel (environnement Eiffel).

L'architecture .NET a créée un nouveau concepts, l'interopérabilité des langages, plus ou moins marketing destinés à séduire la communauté des développeurs, dans ce domaine .Net à largement l'avantage par rapport à la plateforme J2EE qui possède l'inconvénient majeur d'avoir centré son architecture autour d'un seul et même langage : Java.

## ***b. Environnements de développement (IDE)***

Java et C# possèdent tout deux de nombreux outils de développement. Voici une liste, non exhaustive de ses outils.

Les principaux IDE Java :

- **Eclipse** : Eclipse est une plateforme de développement écrite en Java, fruit du travail d'un consortium de grandes entreprises (Borland, IBM, ...). Il en résulte un IDE performant et OpenSource qui a su trouver sa place.
- **JBuilder** : Borland JBuilder X est l'un des meilleurs environnements professionnel pour le développement de solutions Java. Il intègre dans une interface agréable tous les concepts d'ingénieries modernes.
- **NetBeans** : NetBeans créée à l'initiative de Sun Microsystem, présentes toutes les caractéristiques indispensables à un RAD Java de qualité.  
De licence OpenSource, NetBeans permet de développer et déployer rapidement et gratuitement des applications graphiques Swing, des Applets, des JSP/Servlets, des architectures J2EE, dans un environnement fortement customisable.
- **IntelliJ IDEA** : IntelliJ IDEA s'inscrit comme une référence dans le domaine des IDE pour Java. JetBrains offre aux développeurs un outil de très bonne qualité.

Les principaux IDE C# :

- **Visual Studio** : Visual Studio .Net est le célèbre logiciel de Microsoft, très complet, très intuitif, pas grand chose à lui reprocher sauf le prix peut être...
- **C# Builder** : C# Builder est l'IDE .NET de Borland. L'édition personnelle de C# Builder est gratuite mais limitée à des développements non commerciaux.
- **WebMatrix** : Cet outil permet de développer des applications en ASP.NET, il supporte le C# et le VB.NET.
- **SharDevelop** : Cet outil permet de développer des applications Winform en .NET.
- **Eclipse** : Il existe un plugin permettant de programmer en C# avec le célèbre logiciel d'IBM.

### **c. Interfaces graphiques**

Au niveau interface graphique d'une application, quelles sont les avantages et inconvénients que nous offre Java et C#?

Avec Java il existe 3 APIs pour la création d'interfaces graphiques :

AWT, Swing et SWT de java apportent chacune autant de problèmes que de solutions :

- AWT utilise les librairies native de chaque OS, donc les fonctions se réduisent au plus petit dénominateur commun... c'est a dire, pas grand chose de fonctionnel. Le look est dépendant de la plate forme.
- Swing 100% java, est plus lente que des librairies natives. Le look est indépendant de l'OS.
- SWT le look est a moitié dépendant de l'OS, mais on perd le garbage collector -> retour des fuites de mémoire.

Tous les composants de AWT ont leur équivalent dans Swing en plus joli et avec plus de fonctionnalités en plus d'offrir de nombreux composant qui n'existent pas dans AWT.

Bref les utilisateurs de Java utilisent Swing plutôt que AWT.

Mais celle-ci est réputée pour sa lenteur... (par rapport entre autre à une application créée en Windows forms)

Effectivement, la lenteur de Swing vient surtout du fait que l'interface sera identique sur toutes les architectures, ce qui est bien plus que de la simple portabilité!

Il aurait peut être été intéressant que Sun soit moins radical sur cela ... (que le checkbox soit rond, carré ou autre n'a pas une grande importance, surtout entre différent poste client!).

La question est donc :

Vaut-il mieux avoir un programme avec une interface graphique plus rapide et pas portable(GDI+ de .NET(C#)) ou une interface graphique plus lente et entièrement portable (swing)?

Le sujet porte ici plutôt sur la portabilité (cf. IV d. Portabilité), mais il ne faut pas se voiler la face, il y a beaucoup plus de monde qui travaille sous Windows que sous Linux et les utilisateurs Windows auront du mal à se mettre à Linux (il faut changer ses habitudes et ce n'est pas facile).

Etant donné cela, l'intérêt de faire du java plutôt que du C# pour une portabilité qui ne sert pas (ou très rarement) mais pour avoir une interface graphique apparemment plus lente ne semble pas un bon compromis...

Si on part donc du fait que les utilisateurs finaux sont sur Windows, il est vrai qu'il est préférable d'utiliser C# qui s'intègre parfaitement dans l'ensemble des produits microsoft. (cf en Annexe le Benchmark de DotnetGuru qui compare une application graphique c# windows forms d'un coté et en Java – Swing de l'autre.. Les résultats parlent d'eux même).

Il faut aussi souligner que beaucoup d'utilisateurs utilise mal l'API Swing ce qui rajoute de la lenteur à leurs applications...

Mais optimisations du programme ou non, une interface graphique en swing sera toujours beaucoup fois plus lente qu'une interface native, c'est lié a l'architecture de swing, et au choix de Sun (que l'on peut trouver judicieux) de ne (presque) pas utiliser de composants natifs.

On a beau la critiquer, swing reste une excellent librairie destinée aux interfaces graphiques, aux performances tout a fait honnête, avec un 'look and feel' pluggable, une très bonne architecture (MVC entre autres), et 100% portable avec ça. On ne peut pas être bon partout.

Ce qui est sur, c'est que la portabilité n'est pas gênante en soit, d'autant qu'il parait que SWT marche plutôt bien. C'est plus simple que Swing et plus rapide. L'application aura un look dépendant de la machine, mais ça reste portable.

## d. Performances

Benchmarks récupérés sur le net :

- <http://www.tommti-systems.de/main-Dateien/reviews/languages/benchmarks.html>
- <http://www.dotnetguru.org/articles/Comparatifs/benchJ2EEDotNET/J2EEvsNETBench.html>

Ces test parlent d'eux même et sont très complet, ils vont du simple calcul du minimum d'un tableau de types primitifs à la comparaison entre 2 interfaces graphiques.

Ces benchmarks sont en annexe de ce rapport...

Pour confirmer les choses, nous avons fait le dernier test de tommti-systems.de (1<sup>er</sup> benchmark) avec java 1.4.2\_03 avec les valeurs maximales divisées par 10 :

résultat :

```
Total C# benchmark time: 7529 ms
Total Java benchmark time: 19265 ms
```

Java prends 2,5 fois plus de temps ...

Tests Personnel des différentes machine virtuel Java :

*1<sup>er</sup> test :*

```
public class Loop{
    public static void main(String[] args)
    {
        long time = System.currentTimeMillis();
        int REPEAT1 = 1000 * 1000;
        int REPEAT2 = 1000 * 100;
        for (int i = 0; i < REPEAT1; i++){
            for (int j = 0; j < REPEAT2; j++){
            }
        }
        time = (System.currentTimeMillis() - time)/1000;
        System.out.println("Time taken: (in seconds) " + time);
    }
}
```

*Résultats :*

```
92 seconds avec j2dk 1.4
93 seconds avec j2dk 1.4.2_04
120 seconds avec j2dk 1.5.0
```

*2ème test :*

```
public final class TestTableaux {
    private double[] valeurs;
    public TestTableaux(int nombreElements) {
        valeurs = new double[nombreElements];
        for (int i=0; i<valeurs.length; i++){
            valeurs[i] = (Math.random() - 0.5) * Double.MAX_VALUE;
        }
    }
    public double min(){
        double minimum = Double.MAX_VALUE;
        double longueur = valeurs.length;
        double valeurCourante = 0;

        for (int i=0; i<longueur; i++){
            valeurCourante = valeurs[i];
            if (valeurCourante < minimum){
                minimum = valeurCourante;
            }
        }
        return minimum;
    }

    public long testMin(int nbItérations){
        long start = System.currentTimeMillis();
        for (int i=0; i<nbltérations; i++){
            min();
        }
        long end = System.currentTimeMillis();
        return (end - start);
    }
    public static void main(String[] args) {
        TestTableaux tt = new TestTableaux(1000000);
        System.out.println("Temps écoulé : " +tt.testMin(1000));
    }
}
```

*Résultats :*

```
6813 milliseconds avec j2dk 1.4
6797 milliseconds avec j2dk 1.4.2_04
7140 milliseconds avec j2dk 1.5.0
```

### 3ème test :

```
import java.util.*;

public final class TestListes {
    private List valeurs;
    public TestListes(int nombreElements) {
        valeurs = new ArrayList();
        for (int i=0; i<nombreElements; i++){
            valeurs.add(new Double((Math.random() - 0.5) * Double.MAX_VALUE));
        }
    }
    public double min(){
        double minimum = Double.MAX_VALUE;
        double valeurCourante = 0;

        Iterator i = valeurs.iterator();
        while (i.hasNext()){
            valeurCourante = ((Double) i.next()).doubleValue();
            if (valeurCourante < minimum){
                minimum = valeurCourante;
            }
        }
        return minimum;
    }

    public long testMin(int nbItérations){
        long start = System.currentTimeMillis();
        for (int i=0; i<nbItérations; i++){
            min();
        }
        long end = System.currentTimeMillis();
        return (end - start);
    }
    public static void main(String[] args) {
        TestListes tl = new TestListes(1000000);
        System.out.println("Temps écoulé : " + tl.testMin(1000));
    }
}
```

### Résultats :

```
52938 milliseconds avec j2dk 1.4
61259 milliseconds avec j2dk 1.4.2_04
42969 milliseconds avec j2dk 1.5.0
```

A savoir que le matériel utilisé est un PentiumIII 1.8ghz avec 256Mo de RAM. La dernière version de la JVM à l'air plus rapide lorsqu'il s'agit de manipuler les Collections (List ici) par contre pour les types primitifs les anciennes versions de la JVM étaient plus rapide que la version 1.5... Encore une fois ces tests sont exhaustifs...

## e. Plates-formes

	Implémentation conforme J2EE	Microsoft .NET
Contrôle d'infrastructure	Sun	Microsoft
Site Web	<a href="http://java.sun.com/j2ee">http://java.sun.com/j2ee</a>	<a href="http://www.microsoft.com/net">http://www.microsoft.com/net</a>
Axiome	Un langage, n'importe quelle plate-forme (indépendance de plate-forme) «Write once, run anywhere» (S'écrit une fois, s'exécute n'importe où)	Une plate-forme, n'importe quel langage (optimisation de plate-forme)
Plates-formes	Unix, Linux, Windows, MacOS, IBM iSeries, IBM zSeries (S/390)	Windows
Éléments compris	Java Development Kit (JDK) Enterprise Java Beans (EJB) Servlet API Java Server Pages (JSP) Java Transaction API (JTA) Java Naming and Directory Interface (JNDI) Java Database Connectivity (JDBC) Java Message Service (JMS) Java XML Pack	.NET Framework .NET Common Language Runtime (CLR) .NET Common Language Specification .NET Languages (VB, C++, C#, JScript, J#) .NET My Services .NET Web Services Framework ADO.NET, ASP.NET Windows Forms, Web Forms Mobile Internet Toolkit
Langages de programmation	Java	APL, C++, C#, COBOL, Component Pascal, Curriculum, Eiffel, Fortran, Haskell, J# (Java Language), Microsoft JScript®, Mercury Mondrian, Oberon, Oz, Pascal, Perl, Python, RPG, Scheme, SmallTalk, Standard ML, Microsoft Visual Basic
Serveurs d'applications	Conformité J2EE 1.3 vérifiée [2002-05]: BEA WebLogic, Borland Enterprise Server, Computer Associates Advantage Joe, IBM WebSphere, Macromedia JRun Server, Pramati Server, SilverStream eXtend App Server, Sybase EAServer, Trifork EA Server, Fujitsu Interstage  De nombreux autres, y compris Sun iPlanet, JBoss, Oracle 9i Application Server et Apache Jakarta Tomcat.	ASP .NET avec COM+
Environnements de développement interactifs (IDE)	Options multi-fournisseur : Borland JBuilder, IBM Visual Age for Java, Sun Forte for Java, WebGain VisualCafé™, TogetherSoft ControlCenter	Visual Studio .NET
Environnement d'exploitation	Java Runtime Environment (JRE)	Common Language Runtime (CLR)
File d'attente de messages	Java Message Service (JMS)	MSMQ

Code intermédiaire	Java Virtual Machine et CORBA IDL et ORB Les spécifications de Java Virtual Machine (JVM) permettent d'exécuter le pseudo-code Java sur n'importe quelle plate-forme dotée d'une JVM conforme.	Intermediary Language (IL); Common Language Runtime .NET CLR permet au code de multiples langages d'utiliser un ensemble commun de composants, sur Windows.
Stratégie de composants	Enterprise JavaBeans (EJB)	Gestion des composants .NET Le service des composants .NET est assuré au moyen des services COM+.
Interface utilisateur graphique/Formulaires	Java Swing : ensemble de composants d'interface graphique, qui fait partie des Java Foundation Classes (JFC), intégré à J2SE.	.NET Windows Forms et Web Forms
Normalisation	Un consortium d'entreprises contribue aux spécifications et normes Java; il est dirigé par Sun, par l'entremise de son comité Java Community Process (JCP). <sup>14</sup> La majeure partie du code source J2EE est disponible.	Microsoft a soumis les spécifications du langage C# et de l'infrastructure CLI (Common Language Infrastructure) à l'ECMA à des fins de normalisation. <sup>15</sup>  La majeure partie de .NET est une propriété exclusive.
Pages Web de serveur dynamique	Java Server Pages (JSP) et Servlets Les JSP utilisent le code Java (snippets ou références JavaBean), compilé en pseudo-code Java.	Active Server Pages (ASP .NET) Le code ASP .NET (n'importe quel langage) est compilé en code natif par l'intermédiaire du CLR.
Connectivité de bases de données	Java Database Connectivity (JDBC), SQL/J	SQL Server, ADO .NET
Connectivité de systèmes existants	J2EE Connector Architecture (JCA)	Host Integration Server
API de nommage/d'annuaire	Java Naming And Directory Interface (JNDI)	Active Directory Services Interface (ADSI)
Prise en charge XML	API et architectures Java pour le langage XML :  Java XML Pack Java API for XML Processing (JAXP) Demandes de spécifications émergentes :  Java Architecture for XML Binding (JAXB), Java API for XML Messaging (JAXM), for XML Registries (JAXR), for XML-based RPC (JAX-RPC)	Inhérente à l'architecture  L'espace de noms <i>System.Xml</i> de la bibliothèque .NET Framework Class Library assure la prise en charge du langage XML en fonction des normes établies.
Services Web	Non inhérents à l'architecture. J2EE 1.3 assure la prise en charge de certains services Web avec la préversion du Java Web Services Developer Pack (Java WSDP). J2EE 1.4 (1 <sup>er</sup> trim. 2003) offrira la prise en charge intégrale des services Web.	Inhérents à l'architecture.  L'espace de noms <i>System.Web.Services</i> de la bibliothèque .NET Framework Class Library permet la création de services Web XML à l'aide d'ASP.NET et de clients de services Web XML.  Prise en charge des protocoles XML, SOAP, WDSL et UDDI.

Les deux environnements disposent de mécanismes très similaires pour gérer chaque couche de l'application. L'important est de savoir utiliser ces services de manière à découpler au maximum les couches entre elles. Finalement, Microsoft ou J2EE, les questions relevant de l'architecture demeurent les plus importantes, ensuite ce n'est qu'une question de critères (portabilité serveur, multi-plateformes, scalabilité, ...).

## **f. Portabilité**

Une notion importante qui a contribué à la réussite du langage Java est sa capacité à s'exécuter sur n'importe quelle plate-forme sans avoir à ré-écrire le code. Sun officiellement supporte Linux, Windows, Solaris et d'autres éditeurs tiers assurent le portage de la machine virtuelle sur OS/2, AIX et MacOS.

A l'heure actuelle, la plate-forme .NET n'est vraiment utilisable que sur les systèmes Windows. Plusieurs efforts de portage ont été entrepris à travers le monde. Citons dans ce domaine, les projets Mono (<http://www.go-mono.org>).

Perdre la portabilité est lourde de conséquence ... pour l'avenir. Si c'est pour développer un soft spécifique pour une entreprise et puis stop d'accord, mais si on compte faire une application évolutive dans le temps et distribuer à plusieurs futurs clients, là il est préférable de conserver un maximum de portabilité.

Java a acquis la portabilité grâce aux développements de Sun et d'autres sociétés et communautés qui souhaitent utiliser les nouveaux apports de java dans leurs environnements. (cf. processus de développements)

Pour Microsoft, c'est différent et voilà quelques points exposant le fait que le portage aura beaucoup de mal à se faire :

### *Raison culturelle*

Microsoft a la main mise sur la plupart des pôles intégrables autour du langage (serveur IIS, base de donnée SQL server, Exchange, MSQM....) tous ces produits sont très riches en fonctionnalités et parfaitement intégrables. Du coup l'aspect communautaire est pas le même: par tradition on développe plus des solutions intégrées que des pièces maîtresses destinées à être intégrées. Il y a donc moins de chance qu'une communauté se mette au travail pour porter .Net sur d'autres plates-formes (le projet Mono est une exception).

### *Raison Economique*

Les grandes entreprises, pour migrer ou créer des solutions utilisant .Net, prennent moins de risques si elles achètent du 'tout-intégré' plutôt que tenter de porter .Net dans leur environnement actuel (les webservices, tellement mis en avant ces derniers temps, sont l'argument technique pour dire: "vous pouvez garder une partie de l'existant")

Microsoft n'a aucun intérêt à porter .Net, car cela permet de vendre l'OS maison intermédiaire (W2003) et le suivant (Longhorn) et tous les produits satellites (Cela se vérifie dans la volonté de Microsoft à faire passer de plus en plus de clients en licences OPEN, plus profitable)

#### *Raison technique*

Les innovations du framework vont être totalement propriétaires (WinFS, Indigo...) au système d'exploitation à venir (Longhorn) et ne pourraient être portées qu'au prix de pénibles efforts sur d'autres OS

.Net sera donc difficilement portable à terme, il sera plutôt, peut-être "émulable".

### **g. Processus de développement**

Le processus de développement de Java est contrôlé par le JCP (<http://www.jcp.org>) qui est une organisation ouverte formée par des programmeurs et utilisateurs Java du monde entier. Sun tient une place prépondérante dans cette organisation. Le but de la JCP est de développer et de d'améliorer les technologies relatives à la plate-forme Java ainsi que ses spécifications, ses implémentations de référence et les kits de compatibilité avec d'autres technologies.

À la différence de Sun, Microsoft a accepté de standardiser C# (processus de normalisation d'ECMA) ce qui le libère de son emprise directe. Il restera en pratique lié aux bibliothèques de .Net et permet aux concurrents d'écrire des compilateurs sans verser de royalties à Microsoft, comme c'est le cas actuellement pour IBM ou Amiga avec leur Java Development Kit.

L'évolution de C# se fera certainement grâce au projet go-mono (<http://www.go-mono.org>). Le but du projet Mono, est de créer une plate-forme Open Source facilitant la création et le déploiement d'applications .NET sur Linux et UNIX. Mono est encore à l'état de projet bien que l'on puisse déjà télécharger des versions bêtas.

La rivalité entre Sun et Microsoft a pour avantage de « booster » l'évolution de ses langages. Ainsi par exemple Java dans sa nouvelle version (1.5) a du créer de nouvelles fonctionnalités qui étaient déjà présente en C# (cf. III b.)

## ***h. Sécurité***

Java et .NET sont deux plates-formes très sécurisées puisqu'elles garantissent que les ressources critiques sont protégées par défaut.

Le Common Language Runtime de C#, tout comme la machine virtuelle Java, dispose d'un vérificateur de code (Class Verifier). Cet organe prend la main dès le chargement d'une classe en mémoire, et avant même que la moindre instruction de cette classe ne soit exécutée. Son objectif est simple : vérifier tout d'abord que les instructions contenues dans le code sont "correctes" (ou "valides") selon la spécification du code intermédiaire (Bytecode Java ou Intermediary Language .NET). Le vérificateur de code permet donc d'empêcher que l'on programme directement au niveau du code intermédiaire.

En Java la sécurité fait partie intégrante du système d'exécution et du compilateur. Un programme Java planté ne menace pas le système d'exploitation. Il ne peut pas y avoir d'accès direct à la mémoire et l'accès au disque dur est réglementé les applets.

En C# les règles de conception qui imposent d'avoir un modèle de mémoire permettant d'accéder à tout, à tout moment. Or cela peut être à l'origine de problème de sécurité, et de fiabilité.

On peut dire que C# un langage beaucoup plus sécurisé que C ou C++ cela est dû en partie à l'omission de pointeurs en tant que type de données et la possibilité de créer des objets qui sont managés par un « garbage collector », mais il reste tout de même moins sécurisé que Java.

## Conclusion :

En conclusion, C# est séduisant dans sa promesse de concilier le meilleur de C++ et de Java en rendant au programmeur la liberté d'écrire un code plus optimisé et élégant que ne le permettrait Java dans certaines occasions. James Gosling, le concepteur de Java, pourra très probablement le détester au motif de paradigme objet pur tel qu'il le conçoit est violé par C#; il avait d'ailleurs déjà qualifié Hejlsberg (fondateur de C#) en 1998 du sobriquet de "Mr method pointers".

Il est vrai que la sélection dynamique des méthodes est une propriété essentielle d'un langage pour être qualifié de "objet". Devoir être obligé de le spécifier (avec le mot clef 'virtual') fait, dans un sens, que l'on est obligé de dire au langage "d'être objet"...

Aussi, pouvoir, dans un langage, spécifier des contraintes d'implémentation (allocation pile/tas) est très dangereux, compliqué, sujet à bug, et va à l'encontre d'un langage objet moderne (toujours + d'abstraction pour gérer de plus gros projets/applications, laisser le compilateur faire le boulot du développeur et certainement mieux que lui).

On pourrait quand même admettre que les possibilités "proches du système" ne sont pas forcément une mauvaise chose, puisqu'elles permettent éventuellement une plus grande optimisation.

Avec Java, le parti pris est clairement de tout laisser gérer par le système. C'est un parti pris excellent si le système gère correctement les choses, ce qu'il fait dans 99% des cas. Mais le 1% restant ?

Mais il ne faut pas oublier que C# va tourner sur une machine virtuelle, alors va t'on vraiment gagner beaucoup en performance en bidouillant avec la machine virtuelle? cela paraît difficile, ou alors au prix de tellement d'effort que cela ne vaudra pas le coup. Les bugs coûtent biens plus chers que quelques cycles d'horloge. Mais c'est vrai qu'il ne faut pas ce satisfaire de la "lenteur" des langages...

Malgré tout, il semble que C# reste un bon langage objet, un seul désavantage réside peut être dans une plus grande difficulté d'apprentissage que Java car son expressivité peut être source de confusion. A cet égard, Java semble être une très bonne école avant de s'engager vers C#.

En tout les cas, la venue de ce langage est sans aucun doute bénéfique en ce qui concerne les futurs évolutions de Java et C# : En effet, une telle concurrence incite quelque part les

personnes responsables de l'évolution de ces deux langages à regarder, voir incorporer à leur propre langage les concepts de l'autre si il s'avérerait être une bonne idée...

On peut voir d'ailleurs toutes les nouveautés que le j2sdk 1.5 apporte à Java et qui étaient déjà présentent dans C#. (les metadonnées, autoboxing, liste variables de paramètres, amélioration de la boucle for , ...).

Et pareillement dans l'autre sens, la généricité présente dans la version 1.5 du j2sdk sera supportée dans la nouvelle version de visual studio (Whidbey) et donc aussi dans les prochaines versions du .net framework...

Malgré toutes ces caractéristiques qui leurs sont propres, rien ne peut nous prédire de la réussite de tel ou tel langage...