Transição para C#

Para programadores de JAVA

Programas simples

JAVA

```
//  Program.java

package MyProgram;

/**
 *
 * @author Jose Cordeiro
 */
public class Program
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

C#

```
// Program.cs

using System;

namespace MyProgram
{
    /// <summary>
    /// ola mundo
    /// </summary>
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Em C# usam-se *namespaces* com uma função equivalente aos *packages* do JAVA.
A diretiva **using** do C# é semelhante ao **import** do JAVA. Permite usar tipos de outro *namespace* sem ter de os especificar.
Em C# os comentários em XML, equivalentes ao Javadoc do Java, iniciam-se por **///** e originam ficheiros em XML
O método **main** do JAVA chama-se **Main** em C# seguindo a convenção de que todos os identificadores públicos começam por maiúsculas (em notação **Pascal Case**)

Tipos de dados

JAVA

Primitive types
byte
----
short
----
int
----
long
----
char
float
double
boolean

C#

Simple types		**Class**
sbyte	8-bit	SByte
byte	8-bit	Byte
short	16-bit	Int16
ushort	16-bit	UInt16
int	32-bit	Int32
uint	32-bit	UInt32
long	64-bit	Int64
ulong	64-bit	UInt64
char	16-bit	Char
float	32-bit	Single
double	64-bit	Double
bool	true ou false	Boolean
decimal	Até 28 casas decimais	Decimal

Em C# só existem objetos, os tipos simples são como nomes alternativos (*alias)* para as classes existentes.
Os tipos simples são *value types* significando que são guardados e passados para métodos por valor (e não por referência).

Tabelas

JAVA

```
String parameters[]
String[] parameters
```

C#

```
----
string[] parameters
```

Em C# os parenteses retos usados na declaração de *arrays* aparecem sempre a seguir ao nome do tipo

JAVA

```
int[][] values = new int[15][];
for(int i=0; i<15; i++)
    values[i] = new int[10];
```

C#

```
int[,] values = new int[10,15];
```

Em C# existem *arrays* multidimensionais, ao contrário do java, que apenas tem tabelas de tabelas (também possíveis em C#). Na criação destes *arrays* multidimensionais, as células são todas automaticamente criadas.

Seleção

JAVA	C#

`switch( j )`
`{`
`  case 1:`
`  x = “one”;`
`  break;`
`  case 2:`
`  x = “two”;`
`  case 3:`
`  x = “three”;`
`  break;`
`  default:`
`  x = “NA”;`
`  break;`
`}`
	`switch( j )`
	`{`
	`  default:`
	`  x = “NA”;`
	`  break;`
	`  case 1:`
	`  x = “one”;`
	`  break;`
	`  case 2:`
	`  x = “two”;`
	`  goto case 3;`
	`  break;`
	`  case 3:`
	`  x = “three”;`
	`  break;`
	`}`

Na instrução **`switch`** do C# o **`break`** é obrigatório para todos os **`case`** e o **`default`** pode aparecer em qualquer posição. Pode usar-se uma instrução “**`goto case 1`**”, por exemplo, para reproduzir o que se faz em JAVA quando se omite o **`break`**

Repetição

JAVA	C#
`----`	`Label: <instruction>;  goto Label;`

O C# permite a utilização de uma instrução **`goto <label>`** para alterar a sequência de instruções embora não se recomende o seu uso. Neste caso uma utilização aceitável é para a saída de ciclos encadeados.

JAVA	C#
`for(char c : str)` `  System.out.print(c);`	`foreach(char c in str)` `  Console.Write(c);`

O ciclo **`foreach`** em C# pode ser utilizado com qualquer classe que implemente a interface **`IEnumerable`**, incluindo *arrays*

Operadores e Instruções

JAVA	C#

	`using (type var = value)`
	`{`
	`...`
	`}`

A instrução **`using`** permite usar uma memória local para uma determinada variável que é válida dentro do bloco do *using*. A classe dessa variável deverá implementar a interface **`IDisposable`** onde está declarado o método **`Dispose()`**. Este método é **sempre** chamado (implicitamente) no fim do bloco.

JAVA	C#
`instanceof` `----`	`is` `as`

O operador **`as`** em C# permite fazer um cast de um objeto para um determinado tipo sem que seja gerada uma exceção se os tipos forem incompatíveis. Neste caso é devolvido o valor **`null.`** O operador **`is`** funciona como o **`instanceOf`** do Java.

JAVA	C#
`----`	`checked e unchecked`

`checked` e **`unchecked`** podem ser usados para controlar um pedaço de código (entre parenteses ou dentro de um bloco a seguir a estas instruções) gerando ou não exceções no caso de *overflows* em expressões aritméticas.

JAVA	C#

`string s1, s2;`
 	
`if(s1.equals(s2))`
`{`
`  ....`
`}`
	`string s1, s2;`
	
	`if(s1 == s2)`
	`{`
	`  ....`
	`}`

A comparação dos textos guardados em **`Strings`** em C# é feita usando o operador **`==`**. Para comparar referências deve-se usar o método **`ReferenceEquals`** (equivalente ao funcionamento do **`==`** do Java).

Redefinição de operadores

JAVA	C#

----	`public class Complex{`
	`private double im, re;`
	
	`public Complex(double real, double imaginary)`
	`{`
	`im = imaginary;`
	`re = real;`
	`}`
	
	`public static Complex operator + (Complex a, Complex b)`
	`{`
	`return new Complex(a.re+b.re, a.im+b.im);`
	`}`
	`}`

Em C# existe a possibilidade de redefinir operadores de uma forma semelhante ao que se faz em C++.
Os métodos de redefinição dos operadores devem ser sempre **`public`** e **`static`** e levar como argumentos os operandos.
Nem todos os operadores do C# podem ser redefinidos.

JAVA	C#

	`public class AccountBalance`
	`{`
	`private int balance;`
	
	`public static implicit operator int(AccountBalance a)`
	`{`
	`return balance;`
	`}`
	
	`public static explicit operator string(AccountBalance a)`
	`{`
	`return "$" + balance;`
	`}`
	`}`
	
	`AccountBalance account = new AccountBalance();`
	`int balance = account; // implicit conversion`
	`string str = string(account); // explicit conversion`

Tal como em C++ é possível redefinir os operadores de **cast** usados para converter uns valores noutros. Neste caso redefinem-se de uma forma semelhante aos operadores aparecendo o nome do tipo no lugar do operador. As conversões definidas deste modo pelo utilizador podem ou não necessitar de explicitar o **casting** através do uso, respetivamente, da palavra chave **`explicit`** ou **`implicit`**

Métodos

JAVA	C#

----	`public static void Add( ref int a, int b)`
	`{`
	`a += b;`
	`}`
	
	`...`
	`ClassName.Add( ref a, b);`
	`...`

Em C# os *reference types* (todos os objetos) são passados por valor (tal como os *value type*). Neste caso o que é passado é o "valor da referência". No entanto também é possível passar um *value type* ou *reference type* por referência desde que se assinale o facto com a palavra-chave **`ref`** antes do parâmetro na declaração do método e antes do valor passado como argumento na chamada do método. No caso do método ser usado para receber o valor que irá ser colocado na variável passada substitui-se a palavra **`ref`** pela palavra **`out`** tanto na assinatura do método como na chamada do mesmo, evitando-se assim ter de inicializar essa variável antes da chamada ao método.

Número variável de parâmetros

JAVA

```
class Drawing
{
    // …
    public void add(Figure... figs)
    {
        for(Figure fig : figs)
        {
            figures.add(fig);
        }
    }
}

// Example

Drawing draw = new Drawing();

draw.add(new Circle());
draw.add(new Circle(), new Square());
```

C#

```
class Drawing
{
    // …
    public void Add(params Figure[] figs)
    {
        foreach (Figure fig in figs)
        {
            figures.Add(fig);
        }
    }
}

// Example

Drawing draw = new Drawing();

draw.Add(new Circle());
draw.Add(new Circle(), new Square());
```

Quando se pretende passar para um método um número variável de valores do mesmo tipo é habitual usar um *array*. Para simplificar este processo, em C#, utiliza-se a palavra-chave **param**, antes do parâmetro que recebe os valores, para indicar que esse parâmetro recebe um número variável de valores. Depois basta passar o número de valores que se quiserem, separados por vírgulas, no lugar do argumento. É equivalente ao *varargs* do Java onde se utiliza as reticências (**…**) antes do parâmetro que recebe os vários valores. Em C#, o parâmetro é obrigatoriamente um *array* unidimensional e, tal como em Java, deve ser o último parâmetro do método.

Parâmetros opcionais e nomeados

JAVA

```
----
```

C#

```
class Example
{
    public void ExampleMethod(int required,
                              string optionalstr = "default",
                              int optionalint = 10)
    {
    // …
    }
}

// Call:

Example ex = new Example();

ex.ExampleMethod(10);  // optionalstr="default", optionalint=10
ex.ExampleMethod(17, "Other");  // optionalint=10
ex.ExampleMethod(25, "Other", 12);
```

Em C#, na definição de um método, é possível ter parâmetros opcionais. Para isso é apenas necessário fornecer o valor de omissão para esse parâmetro que é o valor com que fica se o valor não for fornecido. Os parâmetros opcionais devem ser os últimos do método, não podendo existir nenhum parâmetro não opcional entre parâmetros opcionais.

JAVA

C#

```
class NamedExample
{
    public void PrintDetails(string seller, int order, string product)
    {
        // …
    }
}

PrintOrderDetails("Gift Shop", 31, "Red Mug");
PrintOrderDetails(order:31, product:"Red Mug", seller: "Gift Shop");
PrintOrderDetails("Gift Shop", 31, product: "Red Mug");
```

Em C# é possível fornecer os argumentos de um método em qualquer ordem desde que sejam precedidos pelo nome do parâmetro seguido de dois pontos (**:**). No entanto, se se pretender usar também algum dos argumentos normais sem o nome, então deve-se obedecer à ordem em que aparecem na definição do método.

<table>
<tr><td rowspan="2">Estruturas</td><td>JAVA</td><td>C#</td></tr>
<tr><td>

```
----
```

</td><td>

```
public struct Point
{
    public int x;
    public int y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void Print()
    {
        Console.Write("x = {0}, y = {1}", x, y);
    }
}
```

</td></tr>
<tr><td colspan="2">À semelhança do C++ em C# existe um tipo de dados semelhantes às classes denominado struct. Este tipo de dados apresenta as seguintes diferenças em relação às classes:
- A herança não se aplica a estruturas, mas podem implementar interfaces.
- Os dados deste tipo são value types sendo, como tal, guardados e passados por valor.
- As estruturas não têm um construtor por omissão.</td></tr>
<tr><td rowspan="2">Tipos enumerados</td><td>JAVA</td><td>C#</td></tr>
<tr><td></td><td>

```
enum Gender : byte
{
    Male,
    Female
}
```

</td></tr>
<tr><td colspan="2">Em C# é possível usar tipos enumerados tal como em Java. Os tipos enumerados normalmente usam o tipo int na retaguarda para guardar o seu valor. Em C# pode-se especificar outro tipo inteiro qualquer (por exemplo byte)</td></tr>
<tr><td rowspan="2">Finalizar classes e variáveis</td><td>JAVA</td><td>C#</td></tr>
<tr><td>

```
public final class FinalCircle
{
    public static final double
                radius = 1.5;
    private final int id = 1;
}
```

</td><td>

```
public sealed class FinalCircle
{
    public const double radius = 1.5;

    private readonly int id = 1;
}
```

</td></tr>
<tr><td colspan="2">Em Java a palavra final tem diferentes utilizações que em C# são diferenciadas usando-se palavras-chave diferentes. Assim, sealed é usada para selar uma classe não permitindo a derivação de outras classes, readonly é usado para declarar uma variável apenas de leitura e const para declarar constantes. Neste último caso as constantes são por omissão variáveis estáticas.</td></tr>
</table>

Herança e Polimorfismo

JAVA

```
class Figure
{
int x,y;

public Figure(int x, int y)
{
   this.x = x;
   this.y = y;
}

public string getValues()
   { return "(" + x + "," + y + ")"; }
} // End Class Figure

class Circle extends Figure
{
int radius = 1;

public Circle(int x, int y, int radius)
  {
  super( x, y);
  this.radius = radius;
  }

public string getValues()
   { return super.getValues() + "r=" +
radius; }
} // End Circle
```

C#

```
class Figure
{
int x,y;

public Figure(int x, int y)
{
  this.x = x;
  this.y = y;
}

public virtual string GetValues()
  { return "(" + x + "," + y + ")"; }
} // End Class Figure

class Circle : Figure
{
int radius = 1;

public Circle(int x, int y, int radius)
                    : base( x, y)
  {
  this.radius = radius;
  }

public override string GetValues()
  { return base.GetValues() + "r=" + radius; }

} // End Circle
```

A herança em C# é feita usando o símbolo **:** em vez do **extends** do Java.
Em C#, a chamada ao construtor da classe base é feita a seguir à declaração do construtor e antes do bloco de código do mesmo. Neste caso, acrescenta-se dois pontos, a palavra **base** e os parênteses com o(s) argumento(s) a passar ao construtor da classe base. Esta palavra-chave corresponde ao **super** do Java.
Ao contrário do Java os métodos em C# não são virtuais sendo necessário declará-los como tal usando a palavra-chave **virtual**. Os métodos abstratos são virtuais por defeito.
Para redefinir um método virtual numa classe derivada em C# é necessário acrescentar a palavra-chave **override.**
A alternativa é usar a palavra **new** significando que se trata de um novo método e não de uma redefinição.

Interfaces

JAVA

```
public interface Drawable {
    public void draw();
}

public class Person implements Drawable {
…
}
```

C#

```
public interface IDrawable {
    public void Draw();
}

public class Person : IDrawable {
…
}
```

Em C#, os identificadores das interfaces começam, por convenção, pela letra **I** maiúscula. Uma classe que implemente uma ou mais interfaces coloca o(s) seu(s) identificador(es) depois do nome da classe e a seguir ao símbolo **:** tal como para a herança. Sendo assim, a seguir a **":"** vem o nome da classe base seguido dos identificadores das interfaces que a classe implementa, todos separados por vírgulas. Ao contrário do Java que usa a palavra **extends** antes do nome da classe base e a palavra **implements** antes do nome das interfaces, em C# não há distinção desses dois tipos, daí a utilização do prefixo **I** no nome das interfaces para as distinguir da classe.

Exceções

JAVA	C#
Exception and error java.lang.Throwable	Exception System.Exception

O C# têm uma hierarquia de exceções tal como em Java só que neste caso a classe base é **System.Exception**.
Também em C# não são feitas distinções entre exceções e erros.
A grande diferença em relação ao Java, é que não é necessário, nem possível, declarar a seguir à assinatura do método as exceções que são lançadas dentro desse método.
Existe também em C# uma forma especial de criar um bloco **catch** em que não aparecem os parenteses com a exceção gerada. Neste caso qualquer exceção que apareça leva à execução do bloco desse **catch.**

Propriedades

JAVA

```
class Circle
{
    private int radius;

    public void setRadius(int radius)
       { this.radius = radius; }

    public int getRadius()
       { return radius; }
}

Utilização:

Circle c = new Circle;

c.setRadius(10);

System.out.println(
          "Radius=" + c.getRadius()  );
```

C#

```
class Circle
{
    private int radius;

    public int Radius
    set
        { radius = value; }
    get
        { return radius; }
}

Utilização:

Circle c = new Circle;

c.Radius = 10;

Console.WriteLine( "Radius=" + c.Radius  );
```

Em C# é introduzido o conceito de propriedade usado nos componentes. Uma propriedade é usada como se fosse um atributo público mas o acesso é na realidade feito através de métodos **get** e **set**. Para se ter acesso ao valor recebido por uma propriedade dentro da sua classe usa-se a palavra-chave **value** como se fosse uma variável. Esta *variável* contém o valor recebido.
É possível omitir o **set** ou o **get** de uma propriedade tornando-a assim, respetivamente, só de leitura ou só de escrita.
Uma propriedade pode ser declarada como abstrata ou como virtual e pode ser usada em interfaces.

Indexers

JAVA

```
----
```

C#

```
class Ecra
{
    private char[,] ecra = new char[25, 80];

    public char this[int x, int y]
    {
        get { return array[x,y]; }
        set { ecra[x,y] = value;}
    }
}

Ecra e = new Ecra();
e[2,3] = 'c';
Console.Write(e[2, 3]);
```

Em C# é possível ter a notação usada com *arrays* em classes normais através de *indexers*. Neste caso definem-se os *indexers* de uma forma semelhante às propriedades com um método **set** usado para receber o elemento do *array* e um método **get** usado para colocar um elemento no *array*.
O parâmetro usado no índice dos *indexers* pode ser de qualquer tipo e pode, inclusive, fornecer-se mais que um parâmetro como é feito em *arrays* multidimensionais. No exemplo mostra-se uma classe **Ecra** com um *indexer* contendo 2 índices correspondendo a coordenadas dentro do ecrã.
Tal como nas propriedades é possível omitir o bloco **get** ou o bloco **set.**

Delegates e Eventos

JAVA	C#

	`public class Stock`
	`{`
	`  ...`
	`  public void ProcessPriceChanged(decimal price)`
	`  {`
	`  ...`
	`  }`
	`  ...`
	`}`
	
	`public delegate void Notify(decimal newPrice);`
	
	`Stock obs = new Stock();`
	`Notify n = new Notify(obs.ProcessPriceChanged);`
	
	`n( 123 ); // corre o método guardado`

Em C# existe o tipo **`delegate`** cujas variáveis guardam referências de métodos. A definição destas variáveis usa a assinatura do método que guarda. O nome da variável é fornecido no lugar do nome do método usado na assinatura. No exemplo acima as variáveis do tipo **`delegate`** com o nome **`Notify`** podem guardar métodos que não tenham tipo de retorno e que recebam um valor do tipo **`decimal`** como argumento.
Um **`delegate`** pode guardar mais do que um método, podendo ser acrescentados métodos usando o operador **`+=`** e retirados usando o operador **`-=.`**
Os métodos guardados nos *delegates* podem retornar um valor mas, neste caso, apenas o valor retornado pelo último método que foi guardado no *delegate* é devolvido. Também no caso de um dos métodos gerar uma exceção, esta exceção propaga-se aos outros métodos não sendo executado nenhum dos métodos seguintes.

JAVA	C#

	`public class MyEventNotifier`
	`{`
	`  public event Notify NotifyCallBack;`
	`}`

Para simplificar o uso de *delegates* o C# introduz o tipo **`event`** que cria um **`delegate`** na classe implicitamente. Neste caso ao expor-se o **`event`** permite-se uma inscrição fácil dos métodos a serem considerados através da utilização dos operadores **`+=`** e **`-=.`**

A utilização de eventos, que devem ser públicos, evita que se tenham de criar varáveis do tipo **`delegate`** públicas.