CUSTOM ATRIBUTES

[System.AttributeUsage(

System.AttributeTargets.Class |

System.AttributeTargets.Struct |

System.AttributeTargets.Method,

AllowMultiple = true

)

]

class BugFixedAttribute : System.Attribute

{

private int type;

private String err;

public BugFixedAttribute(int type, String err) { this.type = type; this.err = err; }

public BugFixedAttribute(String err) { this.type = -1; this.err = err; }

public override String ToString()

{

return type > 0 ? ("type=" + type + " and error=" + err) : ("type below zero and error=" + err);

}

}

VERIFICAR SE É UM DELEGATE

public static bool IsDelegate(this MemberInfo mi)

{

return mi.DeclaringType.Equals(typeof(Delegate));

}

retorna true se o tipo representado por t1 for derivado do tipo representado por t2

public static bool IsSubclassOf(Type t1, Type t2)

{

return t1.IsSubclassOf(t2);

}

retorna true se uma variável do tipo representado por tvar pode ser afectada com um valor do tipo representado por tval

public static bool IsAssignableFrom(Type tvar, Type tval)

{

return tvar.IsAssignableFrom(tval);

}

MANAGED VS UNMANAGED

Modelo “unmanaged” – Contrato físico

- Informação de tipo (declarações) presentes no ficheiro header

- Informação de tipo específica da linguagem

- Implementação (instruções) presentes no ficheiro biblioteca

- Problema: sincronização entre header e biblioteca

- Instruções na linguagem nativa

- Normalmente associado à construção de aplicações “monolíticas” – ligação estática

Modelo “managed” - Contrato lógico

- Informação de tipo (metadata) e implementação (linguagem intermédia) presentes no mesmo ficheiro

- A ligação entre componentes é sempre dinâmica

AS vs IS

O operador as verifica se um objecto é compatível com um dado tipo. Se for, faz a conversão, caso contrário retorna null.

ClassA a = o as ClassA;

O operador is verifica se um objecto é compatível com um dado tipo.

if (o is ClassA){ … }

Conversão entre tipos valor (coerção)

- widening

- narrowing

Conversão -> tipos referencia

EQUALS PATTERN

public override bool Equals(object obj) {

if (obj.GetType() != this.GetType()) return false;

....

return ...;

}

OVERRIDE DE OPERADORES

public struct Ponto {

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

public override bool Equals(object obj) {

if (obj == null) return false;

if (!(obj is Ponto)) return false;

return Equals( (Ponto) obj);

}

public bool Equals(Ponto p) { return x== p.x && y == p.y; }

public override int GetHashCode() { return x^y; }

public override string ToString() { return String.Format("({0},{1})", x, y);}

public static bool operator ==(Ponto p1, Ponto p2) { return Object.Equals(p1, p2); }

public static bool operator !=(Ponto p1, Ponto p2) { return !Object.Equals(p1, p2); }

}

COMMON TYPE SYSTEM

- O Sistema de tipos especifica a definição, comportamento, declaração, uso e gestão de tipos.

- Suporta o paradigma da Programação Orientada por Objectos.

- Desenhado por forma a acomodar a semântica expressável na maioria das linguagens modernas.

Define:

- Hierarquia de tipos

- Conjunto de tipos “built-in”

- Construção de tipos e definição dos seus membros

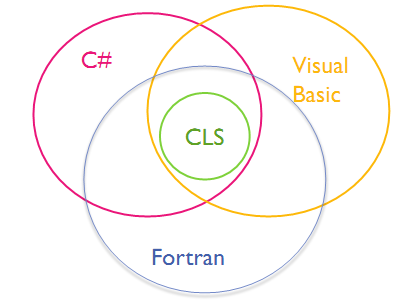
- Utilização e comportamento dos tipos

Common Language Specification

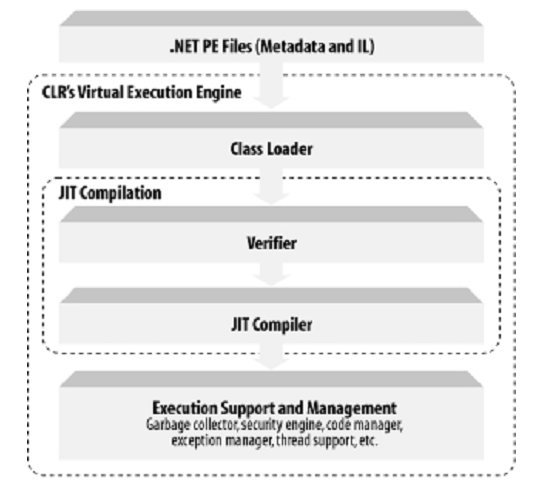
- Conjunto de restrições ao CTS para garantir a interoperabilidade entre linguagens

- Define um sub-conjunto do CTS

- Contém as regras que os tipos devem respeitar por forma a serem utilizados por qualquer linguagem “CLS-compliant”



VIRTUAL EXECUTION SYSTEM



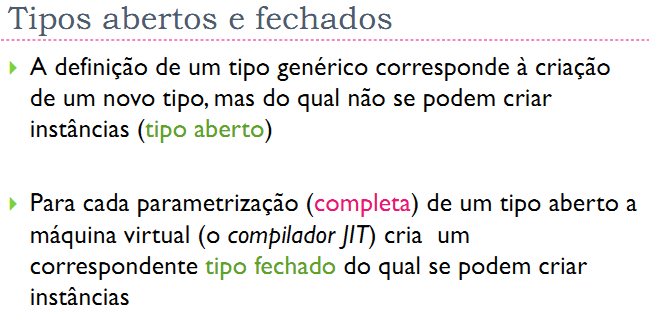
Políticas de iniciação de tipo

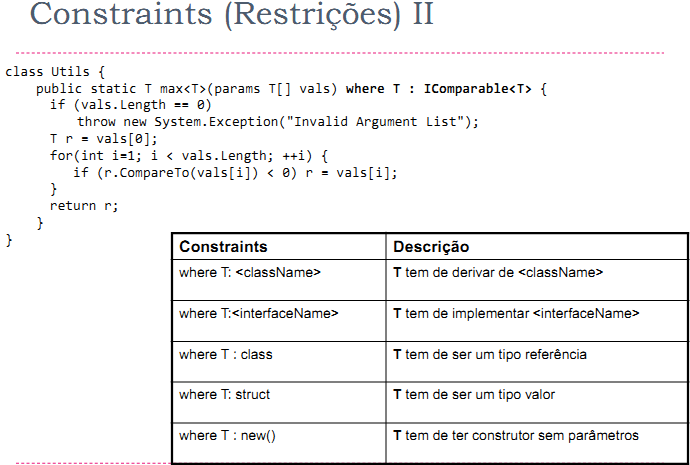
O CLR garante a chamada ao construtor de tipo antes de qualquer acesso a um campo de tipo;

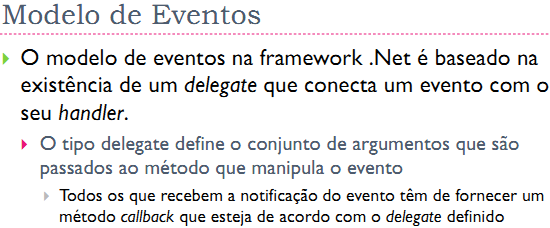
Políticas de iniciação de tipos:

- Imediatamente antes do primeiro acesso a qualquer membro – usada em C# quando há construtores de tipo;

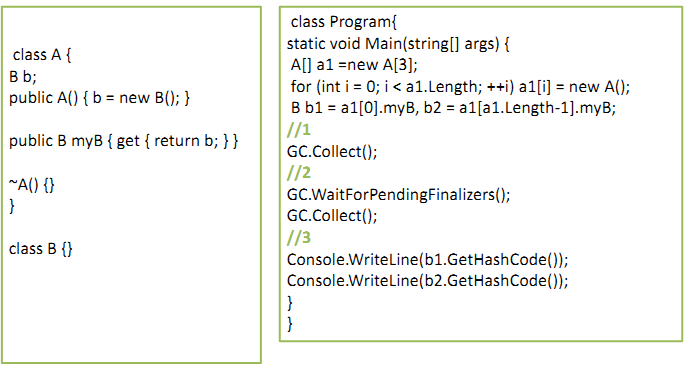
- Em qualquer altura desde que antes do primeiro acesso a um campo de tipo (atributo de Metadata beforefieldinit em C# quando não for definido explicitamente um constructor de tipo).

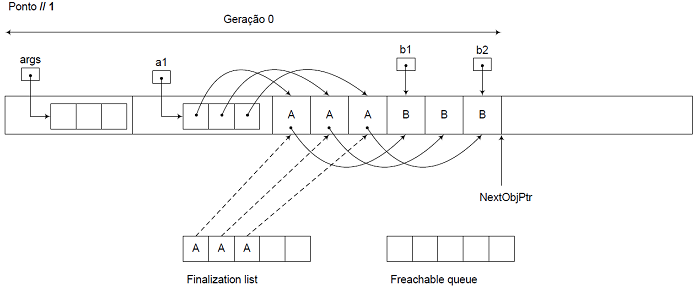


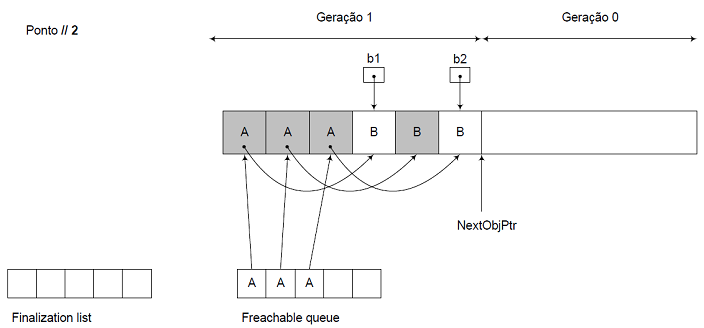


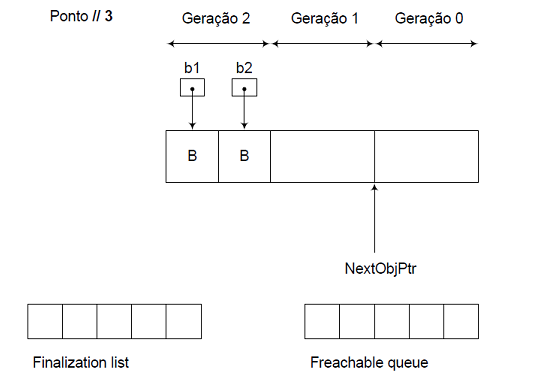


GARBAGE COLLECTOR









EXTENSION METHODS

public static class Ex3\_extension

{

public static IEnumerable<T> BiggerThan2<T>(this IEnumerable<T> enumer, T element)

where T : IComparable<T>

{

LinkedList<T> res = new LinkedList<T>();

foreach (T t in enumer)

{

if (t.CompareTo(element) > 0) { yield return t; }

}

}

}

DELEGATES

class Ex5\_GrupoB

{

public delegate void Action(int x);

public delegate void OtherAction(int x);

public static void DoAction(Action a, int i) { a(i); }

public static void SomeAction(int i) { Console.WriteLine(i); }

public static void Test()

{

OtherAction o = SomeAction;

DoAction(SomeAction, 10);

//DoAction(o, 10); // DoAction -> recebe um delegate do tipo Action e nao do tipo OtherAction

}

}

FUNCTIONS

static IEnumerable<T> Accumulate<T>( IEnumerable<T> source, T seed, **Func<T, T, T> acum** ) {

T acumulated = seed;

foreach (T t in source) {

acumulated = **acum(t, acumulated );**

yield return acumulated;

}

} **// acum é uma função que tem um parâmetro de retorno T, e tem como input dois T’s**

LINQ

static class Ex3\_GrupoC

{

public static IEnumerable<KeyValuePair<K, U>> Mapper<T, K, U>(

this IEnumerable<T> input,

Func<T, IEnumerable<KeyValuePair<K, U>>> map )

{

return input.SelectMany(map);

}

public static IEnumerable<IGrouping<K, KeyValuePair<K, U>>> Joiner<K, U>(

this IEnumerable<KeyValuePair<K, U>> input

) where K : IComparable<K>

{

return input.GroupBy(x => x.Key); // group / join by key

}

public static IEnumerable<KeyValuePair<K, V>> Reducer<K, U, V>(

this IEnumerable<IGrouping<K, KeyValuePair<K, U>>> groups,

Func<IGrouping<K, KeyValuePair<K, U>>, KeyValuePair<K, V>> reduce )

{

return groups.Select(reduce); // run function for every group, returnig just a touple

}

public static IEnumerable<KeyValuePair<string, int>> CountWords(IEnumerable<string> lines)

{

Dictionary<string, int> dicto;

KeyValuePair<string, int> par;// = new KeyValuePair<string, int>("",0);

return lines.Mapper(

delegate(string str)

{

dicto = new Dictionary<string, int>();

foreach (string s in str.Split(' '))

{

if (dicto.ContainsKey(s))

{

dicto[s]++;

}

else

{

dicto.Add(s, 1);

}

}

return dicto;

}

).Joiner().Reducer(

(x =>

{

par = new KeyValuePair<string, int>("", 0);

foreach (KeyValuePair<string, int> p in x)

{

par = new KeyValuePair<string, int>(p.Key, p.Value + par.Value);

}

return par;

}

)

);

}

public static void Main()

{

string[] lines = { "ola mundo ola ave", "ave ola ave" };

foreach (KeyValuePair<string, int> p in CountWords(lines))

Console.WriteLine(p);

}

}

REFLEXÃO – IMPRIMIR CLASSES E METODOS COM ATRIBUTOS

public static void ListAllFixedBugs(Type type) {

Console.WriteLine("class atributes:");

printAttributes(type.GetCustomAttributes(typeof(BugFixedAttribute), false));

Console.WriteLine("methods atributes:");

MemberInfo[] mInfos = type.GetMethods(

BindingFlags.Public |

//BindingFlags.NonPublic |

BindingFlags.Instance |

BindingFlags.Static

);

foreach (MemberInfo m in mInfos)

{

printAttributes(m.GetCustomAttributes(typeof(BugFixedAttribute), false));

}

}

private static void printAttributes(object[] atts)

{

foreach (object o in atts)

{

Console.WriteLine(o.ToString());

}

}

REFLEXÃO - INVOCAÇÃO DE METODOS

public class ReflectionInvoker{

public static object Invoker(string args)

{ string[] argsArray = args.Split(' ');

if (argsArray.Length < 2) return null;

Type tipo = Type.GetType(argsArray[0]); //tipo do objecto

MethodInfo mis = tipo.GetMethod(argsArray[1],BindingFlags.InvokeMethod); //metodo

object objToInvoke = Activator.CreateInstance(tipo, BindingFlags.Instance, null, null, null);

//objecto a invocar

return mis.Invoke(objToInvoke,**null**); // valor a null são os argumentos do método a invokar

}

}