Eventos

- um evento têm de derivar da classe System.EventArgs
- um delegate que conecta um evento com o seu handler
- EventHandler é um delegate pré-definido
  - para quando o evento não gera dados

Delegates

- Um delegate indica a assinatura de um método callback
- delegate define um novo tipo
- Quando o compilador de C# processa um tipo delegate, gera automaticamente uma sealed class que deriva de System.MulticastDelegate (que por sua vez derivada de System.Delegate).
- A classe gerada define 3 métodos:
  - Invoke()
  - BeginInvoke()
  - EndInvoke()
- instâncias de delegates são imutáveis
  - Duas instâncias podem ser combinadas, dado origem a uma terceira instância
  - O operador += invoca o método Delegate.Combine()
  - Método Delegate.Remove() (operador -= em C#) realiza a remoção
- Métodos extensão (têm de estar presentes em classes estáticas)
  - IEnumerable<T> f1(this IEnumerable<T>, P1 p1)
- System.Linq.Enumerable define um conjunto de métodos de extensão sobre IEnumerable<T>
  - Exemplos
  - Restrição: Where
  - Projecção: Select, SelectMany
  - Ordenação: OrderBy, ThenBy
  - Agrupamento: GroupBy
  - Quantificadores: Any, All
  - Partição: Take, Skip, TakeWhile, SkipWhile
  - Conjuntos: Distinct, Union, Intersect, Except
  - Elementos: First, FirstOrDefault, ElementAt
  - Agregação: Count, Sum, Min, Max, Average
  - Conversão: ToArray, ToList, ToDictionary

Tipos Nullable

- Instâncias de tipos-referência podem não ter objecto associado (valor nulo)
- Instâncias de tipos-valor têm sempre valor não nulo

```
public struct Nullable<T> where T: struct {
    private bool hasValue;
    internal T value;
    public Nullable(T value);
    public bool HasValue { get; }
    public T Value { get; }
    public T GetValueOrDefault();
    public T GetValueOrDefault(T defaultValue);
    public override bool Equals(object other);
    public override int GetHashCode();
    public override string ToString();
}
```

- Modificador ? para declarar um tipo como anulável.
  - typeof(int?) == typeof(Nullable<int>)
- Operador ?? para indicar o valor pré-definido numa atribuição de uma instância de um tipo anulável a um não-anulável.
  - (a ?? 0) <=> (a.HasValue ? a.Value : 0)

Generics

- Type Safety
- Aumento da legibilidade: O código não necessita de cast´s explícitos
- Aumento de performance: Instâncias de tipo valor não precisam de ser boxed para serem guardadas em colecções genéricas.
- O código genérico é compilado para IL, que fica com informação genérica de tipos
- O compilador não conhece a interface dos tipos que vão ser usados na instanciação do genérico
- Quando um método que usa parâmetros do tipo genérico é compilado pelo JIT, o CLR :
  - substitui o tipo genérico dos argumentos de cada método por cada tipo especificado, criando código nativo específico para operar para cada tipo de dados especificado
  - CLR fica com código nativo gerado para cada combinação método/tipo.Isto designa-se por Code Explosion.
- Versões genéricas de IEnumerable e de IEnumerator estendem as versões anteriores (não genéricas)
- A definição de um tipo genérico corresponde à criação de um novo tipo, mas do qual não se podem criar instâncias (tipo aberto)
- Para cada parametrização (completa) de um tipo aberto a máquina virtual (o compilador JIT) cria um correspondente tipo fechado do qual se podem criar instâncias
- Constraints Descrição
  - where T: <className> T tem de derivar de <className>
  - where T:<interfaceName> T tem de implementar <interfaceName>
  - where T : class T tem de ser um tipo referência
  - where T: struct T tem de ser um tipo valor
  - where T : new() T tem de ter construtor sem parâmetros

TypeSystem

- Os tipos valor contêm directamente os seus dados, e as suas instâncias estão ou alocadas na stack ou alocadas inline numa estrutura.
  - São passados por valor .
  - O Tipo Valor definido pela construção não admite tipos derivados
- Os tipos refêrencia representam referências para objectos armazenados no heap.
  - São passados por referência.
- O operador *is* verifica se um objecto é compatível com um dado tipo.
- O operador *as* verifica se um objecto é compatível com um dado tipo. Se for, faz a conversão, caso contrário retorna null.
- The **out** keyword causes arguments to be passed by reference. This is similar to the ref keyword, except that ref requires that the variable be initialized before being passed. To use an **out** parameter, both the method definition and the calling method must explicitly use the **out** keyword. Although variables passed as an **out** arguments need not be initialized prior to being passed, the calling method is required to assign a value before the method returns.
- The **ref** keyword causes arguments to be passed by reference. The effect is that any changes made to the parameter in the method will be reflected in that variable when control passes back to the calling method. To use a **ref** parameter, both the method definition and the calling method must explicitly use the **ref** keyword. An argument passed to a ref parameter must first be initialized. This differs from out, whose argument need not be explicitly initialized before being passed.

Interfaces

- As interfaces suportam herança múltipla de outras interfaces.
- Não podem conter campos de instância nem métodos de instância com implementação.
- Todos os métodos de instância têm, implicitamente,os atributos public e virtual.
- Em C# a implementação de uma interface resulta, por omissão, em métodos sealed.
- O que não acontece se o método na classe for declarado como virtual.

Common Language Infrastructure(CLI)

- Common Type System (CTS)
  - O conjunto dos tipos de dados e operações que são partilhados por todas as linguagens de programação que podem ser executadas por uma implementação de CLI, tal como a CLR.
- Metadata
  - Informação sobre os tipos presentes na representação intermédia
- Common Language Specification (CLS)
  - Um conjunto de regras básicas que as linguagens compatíveis com a CLI devem respeitar para serem interoperáveis entre si. As regras CLS definem um subconjunto do Common Type System.
- Virtual Execution System (VES)
  - O VES carrega e executa programas CLI-compatíveis, utilizando a metadata para combinar separadamente as partes em tempo de execução.

CLI

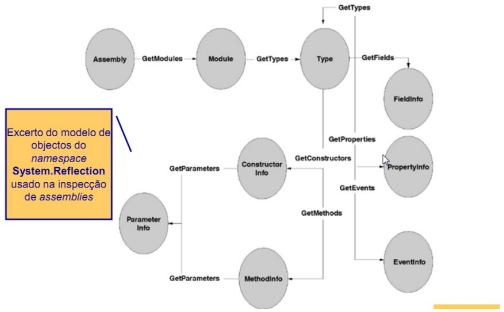
- Metadata
  - Conjunto de tabelas com:
    - os tipos definidos no módulo – DefTables;
    - os tipos referênciados (importados) – RefTables;
  - Informação sobre tipos
    - Sempre incluída no módulo pelo compilador
      - Inseparável do módulo
      - gravada em formato binário
  - Descreve tudo o que existe no módulo
    - Descreve tudo o que existe no módulo :
  - Pelo facto dos compiladores emitirem ao mesmo tempo a metadata e o código dos módulos, estas duas entidades nunca podem deixar de estar em sintonia.
    - Dispondo da metadata não são necessários os ficheiros header e os ficheiros biblioteca para compilar e ligar o código das aplicações. Os compiladores e os linkers poderão obter a mesma informação consultando a metadata dos managed modules.
    - O processo de verificação do código do CLR usa a metadata para garantir que o código realiza apenas operações seguras.
    - Usada para suportar os serviços de runtime. São exemplos:
      - A metadata suporta a serialização/desserialização automática do estado dos objectos
      - Para qualquer objecto, o GC pode determinar o tipo do objecto e, a partir da metadata, saber quais os campos desse objecto que são referências para outros objectos.
- Assembly
  - O manifest do assembly gerado contém a indicação de que o assembly consiste apenas de um ficheiro.
  - Um dos módulos constituintes do assembly contém obrigatoriamente um manifesto que define os módulos constituintes como um todo inseparável e contém o identificador universal do assembly.
  - Um dos módulos constituintes do Assembly, contém também a manifest metadata table.
    - Nome
    - Versão
    - Cultura
    - StrongName
    - Lista de Modulos que pertencem ao Assembly
    - Tipos Exportados
    - Assemblies referenciados
- just in time
  - As invocações de métodos são realizadas indirectamente através de "stubs"
    - O "stub" de cada método é apontado pela tabela de métodos
  - Inicialmente o "stub" aponta para o JIT
  - Na primeira chamada do método é invocado o "JIT compiler".
    - Usa o código IL do método e a informação de tipo para gerar o código nativo
    - Altera o "stub" para apontar para o código nativo gerado
    - Salta para o código nativo
  - As restantes chamadas usam o código nativo

```
public static bool IsMethodToTest(MethodInfo mi)
{
Type t =
Type.GetType("MethodTestingAttributes.MethodTestAttribute");
object[] atrib = mi.GetCustomAttributes(t, false);
if (atrib.Length != 1) return false;
return (mi.GetParameters().Length == 0 &&
(! mi.ReturnType.Equals(Type.GetType("System.Void")))
&&(! mi.ReturnType.IsArray) &&(!
mi.ReturnType.BaseType.Equals(Type.GetType("System.MulticastDelegate")))
        );
}
```

```
public class ValueType {

    public override bool Equals(object obj) {
          if(obj == null) return false;
          Type thisType = this.GetType();
          if(thisType != obj.GetType()) return false;
          FieldInfo[] fields = thisType.GetFields(BindingFlags.Public
                  | BindingFlags.NonPublic | BindingFlags.Instance);
          for(int i=0; i<fields.Length; ++i) {
                   object thisFieldValue = fields[i].GetValue(this);
                   object objFieldValue = fields[i].GetValue(obj);
                   if(!object.Equals(thisFieldValue,objFieldValue))
                            return false;
          }
          return true;
    }

}
```



Excerto do modelo de objectos do *namespace* **System.Reflection** usado na inspecção de *assemblies*

```
class Assembly {
  public  Type[] GetExportedTypes();
  public  Module GetModule(String name);
  public  AssemblyName[] GetReferencedAssemblies();
  public  AssemblyName GetName();
  public  AssemblyName GetName(Boolean copiedName);
  public static Assembly GetAssembly(Type type);
  public  Type GetType(String name);
  public  Type GetType(String name, Boolean
throwOnError);
  public  Type[] GetTypes();
  public static Assembly Load(String assemblyString);
  public static Assembly LoadFile(String path);
  public  Object CreateInstance(String typeName);
  public  Module[] GetLoadedModules();
  public  Module[] GetModules();
  public  Module[] GetModules(Boolean
getResourceModules);
  public static Assembly GetExecutingAssembly();
  public static Assembly GetCallingAssembly();
  public static Assembly GetEntryAssembly();
  String CodeBase{get;}
  String FullName{get;}
  MethodInfo EntryPoint{get;}
  Module ManifestModule{get;}
}
class Module {
  public  System.Type
  GetType(System.String className, System.Boolean
ignoreCase);
  public  System.Type GetType(System.String
className);
  public  System.Type[] GetTypes();
  System.String Name{get;}
  System.Reflection.Assembly Assembly{get;}
}
using Reflection;
public static void ListAllMembersInEntryAssembly() {
  Assembly assembly = Assembly.GetEntryAssembly();
  foreach (Module module in assembly.GetModules())
  foreach (Type type in module.GetTypes())
  foreach (MemberInfo member in
type.GetMembers(flags))
  Console.WriteLine("{0}.{1}", type, member.Name ); }
```

```
class Type : MemberInfo  {
 public static Type GetType(String typeName);
 public  Boolean IsSubclassOf(Type c);
 public  Boolean IsAssignableFrom(Type c);
 public  ConstructorInfo GetConstructor(Type[] types);
 public  ConstructorInfo[] GetConstructors();
 public  ConstructorInfo[] GetConstructors(BindingFlags bindingAttr);
 public  MethodInfo GetMethod(String name, BindingFlags bindingAttr);
 public  MethodInfo GetMethod(String name);
 public  MethodInfo[] GetMethods();
 public  MethodInfo[] GetMethods(BindingFlags bindingAttr);
 public  FieldInfo GetField(String name, BindingFlags bindingAttr);
 public  FieldInfo[] GetFields();
 public  FieldInfo[] GetFields(BindingFlags bindingAttr);
 public  Type GetInterface(String name);
 public  Type[] GetInterfaces();
 public  EventInfo GetEvent(String name);
 public  EventInfo GetEvent(String name, BindingFlags bindingAttr);
 public  EventInfo[] GetEvents();
 public  EventInfo[] GetEvents(BindingFlags bindingAttr);
 public  PropertyInfo GetProperty(String name, BindingFlags bindingAttr);
 public  PropertyInfo GetProperty(String name, Type returnType, Type[] types);
 public  PropertyInfo GetProperty(String name, Type[] types);
 public  PropertyInfo GetProperty(String name, Type returnType);
 public  PropertyInfo GetProperty(String name);
 public  PropertyInfo[] GetProperties(BindingFlags bindingAttr);
 public  PropertyInfo[] GetProperties();
 public  Type[] GetNestedTypes();
 public  Type GetNestedType(String name);
 public  Type[] GetGenericArguments();
}

 class Type : MemberInfo  {

          Type DeclaringType{ get; }
          Type ReflectedType{ get; }
          Module Module{ get;}
          Assembly Assembly{ get; }
          String FullName{ get; }
          String Namespace{ get; }
          Type BaseType{ get; }
          Boolean IsNested{ get; }
          Boolean IsNotPublic{ get; }
          Boolean IsPublic{ get; }
          Boolean IsLayoutSequential{ get; }
          Boolean IsClass{ get;}
          Boolean IsInterface{ get;}
          Boolean IsValueType{ get;}
          Boolean IsAbstract{ get; }
          Boolean IsSealed{ get; }
          Boolean IsEnum{ get; }
          Boolean IsSerializable{ get; }
          Boolean IsArray{ get; }
          Boolean IsGenericType{ get; }
          Boolean IsPrimitive{ get; }
 }

 class MethodBase : MemberInfo {
   public  ParameterInfo[] GetParameters();
   public  Type[] GetGenericArguments();
   public  Object Invoke(Object obj, Object[] parameters);
   Boolean IsOverloaded{ get;}
   Boolean IsGenericMethod{ get; }
   Boolean IsPublic{ get; }
   Boolean IsPrivate{ get; }
   Boolean IsFamily{ get; }
   Boolean IsAssembly{ get; }
   Boolean IsFamilyAndAssembly{ get; }
   Boolean IsFamilyOrAssembly{ get; }
   Boolean IsStatic{ get; }
   Boolean IsFinal{ get; }
   Boolean IsVirtual{ get; }
   Boolean IsHideBySig{ get; }
   Boolean IsAbstract{ get; }
   Boolean IsSpecialName{ get; }
   Boolean IsConstructor{ get; }
   String Name{ get; }
   Type DeclaringType{ get; }
   Type ReflectedType{ get; }
   Module Module{ get; }
 }

 class PropertyInfo : MemberInfo  {
         public  ParameterInfo[] GetIndexParameters();
         public  Object GetValue(Object obj, Object[] index);
         public  Void SetValue(Object obj, Object value, Object[] index)
         public  MethodInfo[] GetAccessors();
         public  MethodInfo GetGetMethod();
         public  MethodInfo GetSetMethod();
         Type PropertyType{ get; }
         Boolean CanRead{ get; }
         Boolean CanWrite{ get; }

 }

class EventInfo : MemberInfo {
         public  MethodInfo GetAddMethod();
         public  MethodInfo GetRemoveMethod();
         public  MethodInfo GetRaiseMethod();
         public  Void AddEventHandler(Object target, Delegate handler);
         public  Void RemoveEventHandler(Object target, Delegate handler);
         Type EventHandlerType{ get; }
         Boolean IsSpecialName{ get; }
         Boolean IsMulticast{ get; }
}
```

```
 class ConstructorInfo : MethodBase  {
         public  Object Invoke(Object[] parameters);
         public  ParameterInfo[] GetParameters();
         public  Type[] GetGenericArguments();
         public  Object Invoke(Object obj, Object[] parameters);

 }


 class MethodInfo : MethodBase {
         public  Type[] GetGenericArguments();
         public  ParameterInfo[] GetParameters();
         public  MethodImplAttributes GetMethodImplementationFlags();
         public  Object Invoke(Object obj, Object[] parameters);
          Type ReturnType{ get; }
          ParameterInfo ReturnParameter{ get; }

 }
class ParameterInfo    {
         Type ParameterType{ get; }
         String Name{ get; }

         Int32 Position{ get; }

         Boolean IsIn{ get; }
         Boolean IsOut{ get; }


}
class FieldInfo : MemberInfo  {
         public  Object GetValue(Object obj);
         public  Void SetValue(Object obj, Object value);
         public  Type GetType();

         Type FieldType{ get; }
         Boolean IsLiteral{ get; }
         Boolean IsNotSerialized{ get; }
         Boolean IsSpecialName{ get; }
}
public static void GenerateSQL(Object obj) {
  Type type = obj.GetType();
  Console.Write("create table {0} (", type.Name);
  bool needsComma = false;
  foreach (FieldInfo field in type.GetFields()) {
    if (needsComma)
       Console.Write(", ");
    else
       needsComma = true;
    Console.Write("{0} {1}", field.Name, field.FieldType);
  }
  Console.WriteLine(")");
}

public static void SetX(Object target, int value) {
   // Write to field: int x
   Type type = target.GetType();
   FieldInfo field = type.GetField("x");
   field.SetValue(target, value);
}
double CallDoIt(Object target, double x, double y)  {
  // Call method: double Add( double, double )
  Type type = target.GetType();
  MethodInfo method = type.GetMethod("DoIt");
  if( method != null ) {
    object[] args = { x, y };
    object result = method.Invoke(target, args);
    return (double)result;
  }
  return(0);
}
public sealed class Utils {
  static void DisplayMethodStatus(Type type) {
    foreach (MethodInfo m in type.GetMethods()) {
      Console.Write("{0} : ", m.Name);
      // check the doc'ed attribute
      if (m.IsDefined(typeof(DocumentedAttribute),true))
        Console.Write("Documented");
      else
        Console.Write("Undocumented");
      // check the tested attribute
      if (m.IsDefined(typeof(TestedAttribute),true))
        Console.WriteLine(" - OK");
      else
        Console.WriteLine(" - Broken");
    }
  }
}
```

```
[AttributeUsage(AttributeTargets.Class|...,
AllowMultiple=true)]
class FriendTypeAttribute:Attribute
```

Nome	Descrição
All	Attribute can be applied to any application element.
Assembly	Attribute can be applied to an assembly.
Class	Attribute can be applied to a class.
Constructor	Attribute can be applied to a constructor.
Delegate	Attribute can be applied to a delegate.
Enum	Attribute can be applied to an enumeration.
Event	Attribute can be applied to an event.
Field	Attribute can be applied to a field.
Interface	Attribute can be applied to an interface.
Method	Attribute can be applied to a method.
Module	Attribute can be applied to a module.
Parameter	Attribute can be applied to a parameter.
Property	Attribute can be applied to a property.
ReturnValue	Attribute can be applied to a return value.
Struct	Attribute can be applied to a value type.

```
private static void DumpGenericObject(int level, object
o) {
 Type t = o.GetType();
  BindingFlags bf = BindingFlags.NonPublic|
BindingFlags.Public |BindingFlags.Instance;
 while (t != typeof(System.Object)) {
  FieldInfo[] fields = t.GetFields(bf);
  foreach (FieldInfo f in fields) {
    object v = f.GetValue(o);
    indent(level); Console.Write(f.Name + ": ");
    if (v==null) {
     Console.WriteLine("null");
 }else if (v.GetType().IsPrimitive || v.GetType() ==
typeof(System.String))
          {Console.WriteLine(v.ToString());}
          else{
            Console.WriteLine();
            DumpObject(level + 1, v);
          }}
  t= t.BaseType;
 }}
  private static void showArray(int level, Array a)
  {
    bool first = true;
    indent(level); Console.WriteLine('[');
    foreach (object o in a)
    {
      if (first) first = false; else { indent(level+1);
Console.WriteLine(", "); }
      DumpObject(level + 1, o);
        }
    indent(level); Console.WriteLine(']');
  }

  private static void showDelegate(int level, Delegate d)
  {
    indent(level); Console.WriteLine("[[ {0} Delegate:
{1} listeners ]]", d.GetType().Name,
d.GetInvocationList().Length);
  }
  public static void DumpObject(int level, object o) {
    Type t = o.GetType();
    Array a = o as Array;
    if (a != null) showArray(level,a);
    else {
      Delegate d = o as Delegate;
      if (d != null) showDelegate(level, d);
      else DumpGenericObject(level, o);
    }
  }
```