

Tipado Dinámico y Metaprogramación

Tema 5

Ejemplos de Código

- Todos los ejemplos de **código** mostrados en estas transparencias están **disponibles para el alumno**
 - Siguen apareciendo en etiquetas como la que se muestra:

Consulta el código en:
`generics/inference`
- Para comprender los conceptos explicados, el alumno deberá **abrir** el código, **analizarlo**, **modificarlo**, **ejecutarlo** y **asegurarse** de que lo **entiende**

Contenido

- Tipado dinámico
- Duck Typing
- Multiple Dispatch
- Reflexión computacional
- Anotaciones o atributos
- Generación dinámica de código
- Metaprogramación

Tipado Dinámico

- El **tipado dinámico** es el proceso de posponer la comprobación e inferencia de tipos en **tiempo de ejecución**
- Hay lenguajes
 - Con tipado estático (C, Fortran, Pascal)
 - Con tipado dinámico (Python, Ruby, Smalltalk)
 - Híbridos (C#, VB, Boo, Objective-C)
- El tipado dinámico tiene **ventajas** e **inconvenientes** comunes
 - Detección de **errores** de tipo en tiempo de compilación
 - **Optimización** de código (rendimiento)
 - + Mayor **adaptabilidad** y **flexibilidad** del código
 - + En general, mayor nivel de **abstracción**

Tipado Dinámico en C#

- C# 4.0 incluye un nuevo tipo **dynamic** que **pospone** la **comprobación e inferencia** de tipos al **tiempo** de **ejecución**
- El compilador no da error, contemplando que
 - Cualquier tipo se puede convertir a **dynamic**
 - **dynamic** se puede convertir a cualquier tipo
 - Cualquier operación del lenguaje se puede aplicar al **dynamic**
- Se comprueba en tiempo de ejecución, pudiendo dar error (se lanza una excepción)

```
static dynamic Max(dynamic a, dynamic b) {  
    return a > b ? a : b; }  
...
```

```
Max(3, 4);           Max(4.4, 3.3);  
Max(4, 3.3); Max(3.4, "no compiler error");
```

Consulta el código en:

dynamic.typing/dynamic

Duck Typing

- Es una propiedad de la mayoría de los lenguajes con tipado dinámico
- Proviene de la frase *if it walks like a duck and quacks like a duck, it must be a duck*
- Significa que el estado dinámico de un objeto determina qué operaciones pueden realizarse con él
- Si en el momento de pasarle el mensaje **m** el objeto posee un método o propiedad público **m** apropiado, éste podrá ejecutarse
- El método **Max** anterior utiliza *duck typing* (admite como parámetro cualquier objeto que acepte la operación **>**)
- **No es necesario**, por tanto,
 - que el objeto sea instancia de una clase derivada de otra que declare el método virtual **m**
 - que implemente un interfaz con un método **m**
- Simplificando así las jerarquías en el código del programa

Duck Typing

- *Duck typing* permite tratar de forma homogénea a `Circumference` y `Rectangle`, incluso aunque no tengan un tipo más general común que defina `X` e `Y`

```
public class Rectangle {  
    private int width, height;  
    public int X { get; set; }  
    public int Y { get; set; }  
    ...  
}
```

...

```
static void Move(dynamic figure, int x, int y) {  
    figure.X += x;  
    figure.Y += y;  
}
```

```
public class Circumference {  
    private int radius;  
    public int X { get; set; }  
    public int Y { get; set; }  
    ...  
}
```

Consulta el código en:

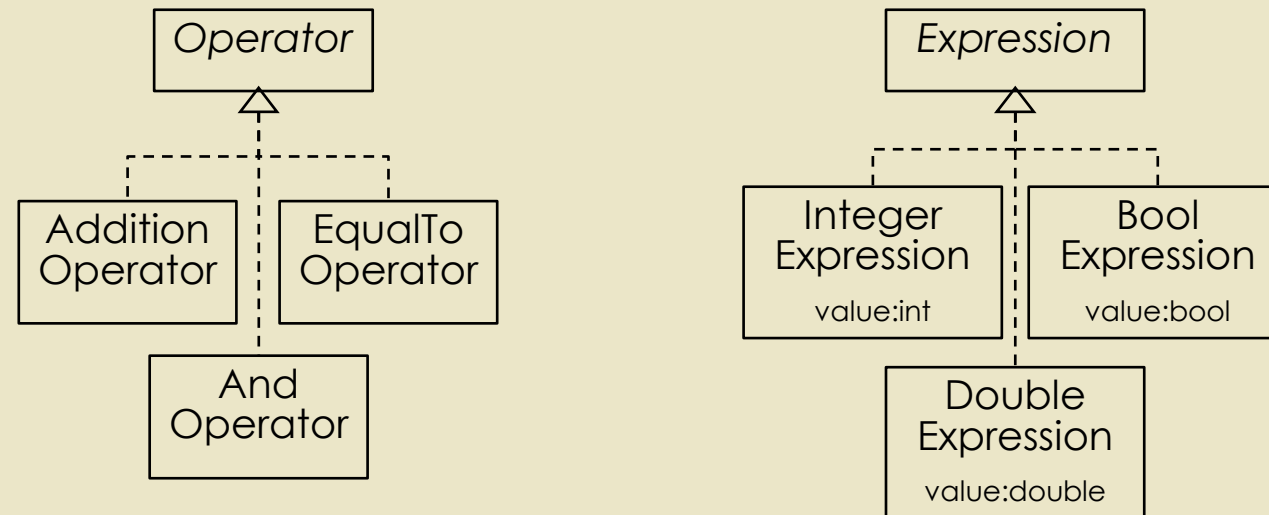
dynamic.typing/duck.typing

Multiple Dispatch

- *Duck typing* es a veces denominado **late binding** (enlace tardío), puesto que posponen a tiempo de ejecución (hasta que se va a pasar el mensaje) la resolución del método (u operación) a ejecutar
- Una de las limitaciones del **enlace dinámico** es que supone usar **single dispatch**:
 - Sólo permite resolver un método dependiendo del tipo de un único objeto (**this**)
- Cuando queremos que la resolución del método **dependa de varios tipos**, necesitamos **multiple dispatch**
- Un mecanismo de obtener *multiple dispatch* son los **multimétodos**
- C# no soporta multimétodos (CLOS, Dylan y Cecil sí)

Multiple Dispatch

- Supongamos la siguiente jerarquía:



- ¿Cómo podemos evaluar cualquier par de expresiones con cualquier operador?

Multiple Dispatch en C#

Consulta el código en:

dynamic.typing/multimethods

- Las diferentes implementaciones serían:

```
private static IntegerExpression Evaluate(IntegerExpression op1, AdditionOperator op,
    IntegerExpression op2)
```

```
{ return new IntegerExpression(op1.Value + op2.Value); }
```

```
private static DoubleExpression Evaluate(DoubleExpression op1, AdditionOperator op,
    IntegerExpression op2)
```

```
{ return new DoubleExpression(op1.Value + op2.Value); }
```

...

```
private static BoolExpression Evaluate(IntegerExpression op1, EqualToOperator op,
    IntegerExpression op2)
```

```
{ return new BoolExpression(op1.Value == op2.Value); }
```

...

```
private static BoolExpression Evaluate(BoolExpression op1, AndOperator op, BoolExpression op2)
```

```
{ return new BoolExpression(op1.Value && op2.Value); }
```

...

Multiple Dispatch en C#

- Un método que evalúe cualquier par de expresiones con cualquier operador binario sería:

```
private static Expression Evaluate(Expression op1,  
                                   Operator op,  
                                   Expression op2) {  
  
    return ?  
}
```

Consulta el código en:

dynamic.typing/multimethods

- No obstante, como implementaríamos este método?
- Como se implementaría para una operación unaria?

Multiple Dispatch en C#

- El requisito sería resolver la **sobrecarga de métodos en tiempo de ejecución** (tipado dinámico)
- Hay alguna forma de posponer la resolución de la sobrecarga de métodos hasta la ejecución?

```
public static Expression MultimethodEvaluate(dynamic op1,  
                                             dynamic op,  
                                             dynamic op2) {  
  
    return Evaluate(op1, op, op2);  
}  
...  
private static Expression Evaluate(Expression op1,  
                                   Operator op,  
                                   Expression op2) {  
  
    throw new ArgumentException(  
        String.Format("La operación ({0} {1} {2}) no está soportada.",  
            op1, op, op2));  
}
```

Consulta el código en:

dynamic.typing/multimethods

Reflexión

- **La reflexión** es la capacidad de un Sistema computacional de **razonar** and **actuar** sobre **si mismo**, adaptándose a si mismo a condiciones cambiantes
- El dominio computacional se extiende con la **representación del propio sistema**, ofreciendo en tiempo de ejecución su estructura y semántica como datos computables
 - Tipos de los objetos, estructura de los tipos, invocación a un método de un tipo...
- **Introspección:** Cuando se puede consultar la representación de un programa pero no modificarla
 - Ej.: obtener la estructura de una clase
- **Intercesión:** Cuando se permite además modificar dicha representación, desencadenando un cambio en lo que éstos representan o significan (reflexión)
 - Ej.: borrar o añadir un atributo a una clase

Ejemplo de Introspección

- C# y Java ofrecen **introspección**
- El siguiente código utiliza introspección para obtener *duck typing* (en Java es la única forma; no hay **dynamic**)

```
static void Move(object figure, int x, int y) {  
    Type type = figure.GetType();  
    PropertyInfo xProperty = type.GetProperty("X");  
    PropertyInfo yProperty = type.GetProperty("Y");  
    int currentX = (int)xProperty.GetValue(figure, null);  
    int currentY = (int)yProperty.GetValue(figure, null);  
    xProperty.SetValue(figure, currentX + x, null);  
    yProperty.SetValue(figure, currentY + y, null);  
}
```

Consulta el código en:
[reflection/introspection](#)

Estructural y de Comportamiento

- Según a la información reflectiva a la que se pueda acceder, la reflexión puede clasificarse en:
 - **Reflexión estructural**, si la información es la estructura de un programa
 - Tipo de un objeto, atributos y métodos del tipo, cuerpo del método...
 - **Reflexión de comportamiento**, si la información de la semántica (significado) del lenguaje
 - Cambiar el significado del paso de mensajes, creación de objetos, acceso a campos...
 - Ejemplo: <https://www.hedonisticlearning.com/posts/behavioral-reflection.html>

Reflexión Estructural en C#

- Java y C# ofrecen introspección estructural
- Java no ofrece intercesión estructural
- Python, Ruby, SmallTalk y CLOS ofrecen intercesión estructural
- C# no ofrece intercesión estructural sobre cualquier tipo, sólo de **ExpandoObject**
 - Permite añadir atributos y métodos a este tipo de objeto
 - Los métodos se representan como miembros de tipo delegado (se pueden usar expresiones lambda)
- ExpandoObject deriva de **IDictionary<string, object>** y por tanto puede ser tratado como un diccionario
- Gracias a **dynamic**, se puede aplicar **duck typing** sobre los **ExpandoObject**

Reflexión Estructural en C#

```
dynamic person = new ExpandoObject();  
person.FirstName = "John"; // Añadimos un atributo  
Console.WriteLine(person.FirstName );
```

Consulta el código en:

[reflection/structural.intercession](#)

```
IDictionary<string, object> dict =  
    GetDictionaryFromFile("person.txt");  
AssignProperties(person, dict);  
Console.WriteLine("{0} {1}, born on {2} {3}, {4}, in {5}.",  
    person.FirstName, person.Surname, person.MonthBirth,  
    person.DayBirth, person.YearBirth, person.PlaceBirth);  
  
static void AssignProperties(dynamic myObj,  
    IDictionary<string, object> dict) {  
    foreach (var item in dict) // Añadimos un atributo programáticamente  
        ((IDictionary<string, object>)myObj)[item.Key] = item.Value;  
}
```

Reflexión Estructural en C#

```
Func<int> getAge = () =>
    (int) (DateTime.Now -
        new DateTime(person.YearBirth, person.MonthBirth,
            person.DayBirth)
        ).TotalDays / 365;
person.GetAge = getAge; // Añadimos un método
Console.WriteLine("{0} {1} tiene {2} años.",
    person.FirstName, person.Surname, person.GetAge());
```

Consulta el código en:

[reflection/structural.intercession](#)

Reflexión de Comportamiento en C#

- C# ofrece un mecanismo limitado de reflexión de comportamiento
- Permite al usuario crear objetos derivados de la clase `DynamicMetaObject` y poder modificar su comportamiento relativo a:
 - Invocación a métodos
 - Obtención de un miembro
 - Modificación de un miembro
 - Creación de una nueva instancia
 - Aplicar operadores unarios
 - Aplicar operadores binarios
 - Utilizar el operador de indexación (`[]`)
 - ...

Anotaciones o Atributos

- Los **atributos** (.Net) o **anotaciones** (Java) son **metadatos** para dar información adicional de elementos del lenguaje (clases, métodos, campos...)
- Añaden información **declarativa** que puede ser utilizada por otra aplicación o sistema para obtener un comportamiento distinto
 - Para el resto de aplicaciones, esta información es inocua
- Se basa en el principio de **Separation of Concerns**
 - Dividir el código en módulos cuya funcionalidad se solape lo menor posible, evitando el **entremezclado de código** de distintas funcionalidades
- Los atributos o anotaciones están escritos en el propio lenguaje
- Ejemplos de atributos son: **Serializable**, **Obsolete**, **DLLImport**, **WebService**, **Test...**

Atributos en C#

- Los atributos en .Net se pueden aplicar a
 - Ensamblados, módulos, clases, structs, interfaces, métodos, propiedades, eventos, campos, parámetros y valores de retorno
- Se utilizan los **corchetes** antes del elemento a etiquetar

```
[Obsolete("Use m_v2 en su lugar.")]  
public static void m() { ... }
```

- En Java se utiliza @

```
@Override  
public void m() { ... }
```

Atributos en C#

- Lo siguiente es un ejemplo de uso de atributos en C#

```
// Hace que VS muestre un warning
[Obsolete("Use m_v2 en su lugar (m está obsoleto).")]
public static void m() { }
public static void m_v2() { }
[DllImport("kernel32.dll")] // Librería nativa
public static extern bool Beep(int frequency, int duration);
...
[Serializable()]
public class Person {
    public uint IDNumber;
    public string firstName, surname;
    [NonSerialized()] // Age no se serializa
    public byte age;
    ...
}
```

Consulta el código en:

[annotations.attributes/use.of.attributes](https://docs.microsoft.com/es-es/visualstudio/ide/annotations.attributes/use.of.attributes)

Custom Attributes

- Los atributos se crean **en el mismo lenguaje de programación** (*custom attributes*)
 - Ofrecen una *pseudoextensión* del lenguaje
- Para crear un atributo por el programador:
 1. Definimos el uso del atributo (clase, método...)
 2. Derivamos de la clase **Attribute**
 3. Implementamos la clase (el atributo)

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class AuthorAttribute : Attribute {
    public AuthorAttribute(string name) {
        this.name = name; this.version = 0; this.tested = false;
        this.date = default(DateTime);
    }
    public AuthorAttribute(string name, uint version, bool tested, string date) {
        this.name = name; this.version = 0;
        this.tested = tested; this.date = DateTime.Parse(date);    }
}
```

...

Consulta el código en:

[annotations.attributes/authors](#)

Custom Attributes

4. Aplicamos el atributo a un programa

```
[Author("John")]  
public class MyClass {  
}
```

```
[Author("Mary", 2, true, "07/10/2012")]  
public class Program {  
    static void Main(string[] args) {  
    }  
}
```

Consulta el código en:

[annotations.attributes/application.with.attributes](#)

5. Se crea otra aplicación que procesa los atributos (anotaciones) de ésta

- Pregunta: ¿Cuál es el mejor mecanismo para tratar esta información?

Custom Attributes

```
class Introspector {  
    static void Main(string[] args) {  
        Assembly assembly = Assembly.Load("application.with.attributes");  
        Type[] types = assembly.GetTypes();  
        foreach (Type type in types) {  
            Console.WriteLine("Information of the {0} class:", type);  
            foreach (Object attribute in  
                type.GetCustomAttributes(typeof(AuthorAttribute), false))  
                Console.WriteLine("\t{0}.", attribute);  
        }  
    }  
}
```

Consulta el código en:

[annotations.attributes/inspection.of.attributes](#)

Generación Dinámica de Código

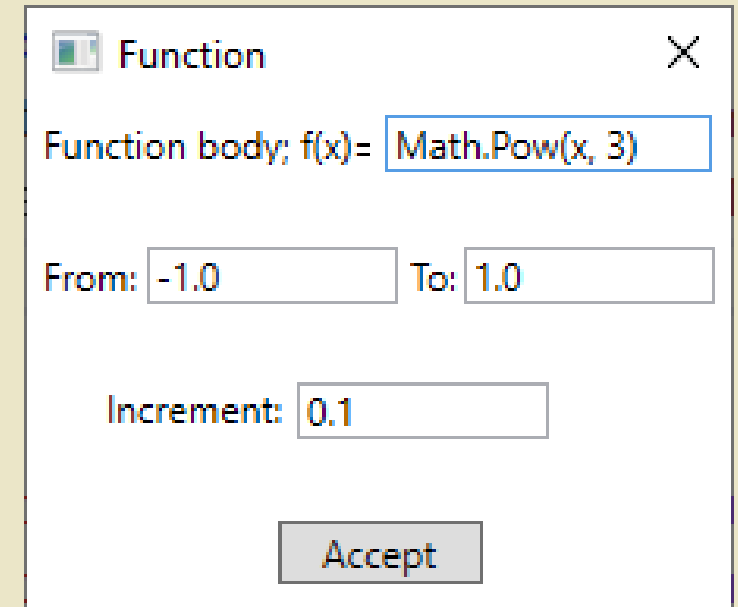
- Es la capacidad de generar **programas** (parte de los mismos) **en tiempo de ejecución**
 - Es decir, programas que generan programas
 - El código generado de esta forma puede ser parte de la aplicación que lo generó
- Se puede utilizar para generar dinámicamente el código de nuevos métodos o clases (**intercesión estructural**)
- El nuevo código se puede crear a partir de:
 - El propio programa: **Adaptiveness**
 - El usuario (directa o indirectamente): **Adaptability**
- Es muy utilizado en los lenguajes dinámicos para generar código automático de acceso a bases de datos y creación automática de vistas (*Ruby on Rails* o *Django*)
- También llamada **programación generativa**

Dynamic Code Generation in C#

- .NET proporciona la característica de Compilador como Servicio (*Compiler-as-a-Service*) a través de **Roslyn**, permitiéndonos usar los servicios del compilador desde nuestro código
- Esto permite construir programas que modifican, interpretan y realizan razonamientos sobre otros programas
- También permiten la generación dinámica de código y su ejecución a partir de:
 - Strings
 - Ficheros de texto
 - Árboles de sintaxis abstracta (*Abstract syntax trees* - AST)
- El código se puede **compilar**, **cargar** e **invocar** en tiempo de ejecución
 - Aprenderás más en la asignatura **Diseño de Lenguajes de Programación** (próximo curso)
- Pregunta: ¿Cómo podemos hacer en C# una aplicación que muestre una función **f(x)** gráficamente, permitiendo al usuario escribir su cuerpo?

Dynamic Code Generation in C#

- Pedimos el cuerpo de $f(x)$ al usuario
 - Puede escribirse cualquier expresión C# que utilice x
 - Se almacenará en una variable `string` llamada `function`
- Invocamos el método `WorkOutValues`. Este evalúa la función introducida por el usuario en un bucle (de valores del **eje X**)
 - Incrementa el valor de X en cada iteración
 - A través de la llamada asíncrona a la función `CSharpScript.EvaluateAsync`
- De este modo obtenemos los diferentes valores del **eje Y** para así poder representar la gráfica



Function

Function body; $f(x) =$

From: To:

Increment:

Accept

Consulta el código en:

[reflection/generative.programming](https://github.com/Reflection/GenerativeProgramming)

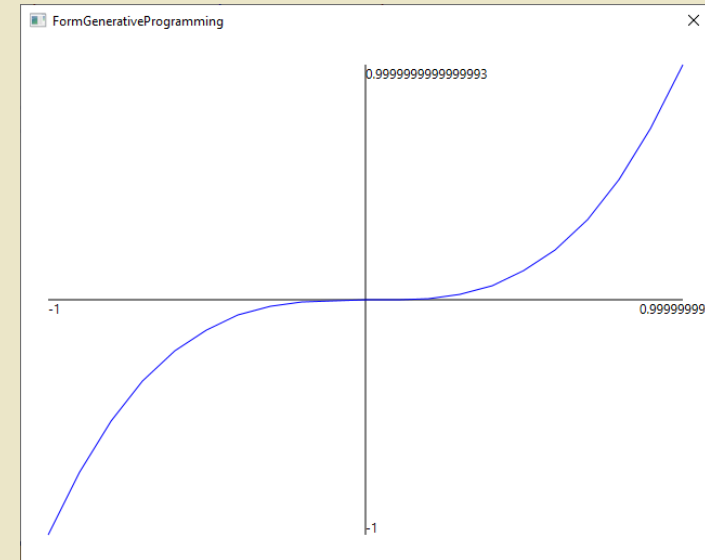
Dynamic Code Generation in C#

- Parte de la aplicación es generada dinámicamente por el usuario
- Se puede aprovechar toda la expresividad de C# para implementar el cuerpo de la función

```
...  
for (double localX = fromX; localX <= toX; localX = localX + incrementX, i++) {  
    xs[i] = localX;  
    using (var task = CSharpScript.EvaluateAsync<double>(function,  
        options: ScriptOptions.Default.AddImports("System"),  
        globals: new ScriptGlobals() { x = localX }))  
    {  
        ys[i] = task.Result;  
    }  
}
```

Consulta el código en:

[reflection/generative.programming](https://github.com/Orfín/reflection/generative.programming)



Meta-Programación

- Es la capacidad de poder escribir **programas** que **escriban o manipulen otros programas**
 - Es un concepto muy general que puede ser utilizado en diversos escenarios
- Los siguientes elementos de lenguajes de programación están relacionados con la meta-programación
 - Reflexión
 - Generación dinámica de código
 - Anotaciones o Atributos
 - Tipado dinámico