Seminario 2: C#

Integrantes:

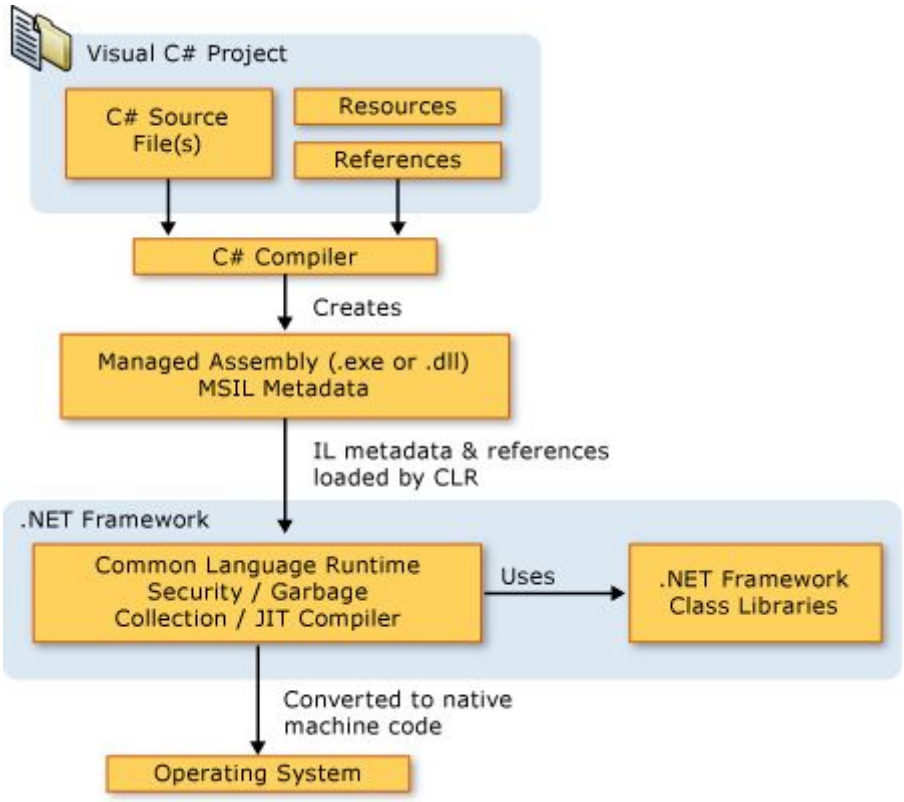
1. Joel David Hernández Cruz
2. Leandro González Montesino
3. Eliane Puerta Cabrera
4. Liset Alfaro González
5. Wendy Díaz Ramírez

---

1. ¿Qué se entiende por DLR y CLR en .NET?

Empecemos por CLR o ***Common Language Runtime***. CLR es el componente básico del ***Framework .Net***. Es el entorno de tiempo de ejecución de ***.Net***, que ejecuta los codigos y facilita el proceso de desarrollo al proporcionar diversos servicios.

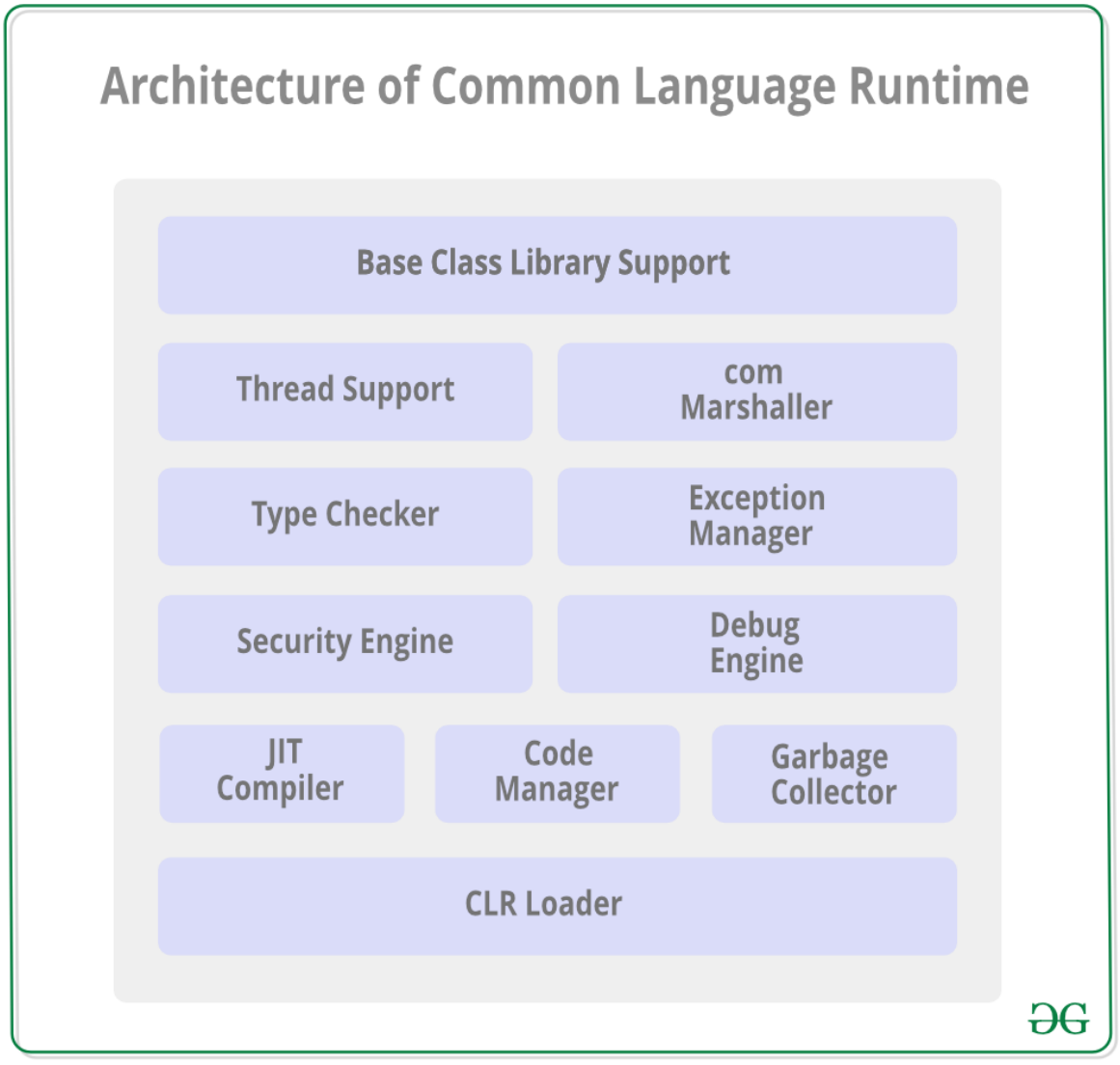
- Como parte del ***Microsoft .Net Framework***, el CLR es una **Maquina Virtual** que maneja la ejecución de los programas escritos en cualquier lenguaje que se use en el ***Framework .Net***, como son **C#**, **VB .Net**, **F#**, etc...
- Para cualquier código escrito en estos lenguajes el compilador especifico de ese lenguaje lo convierte en código MSIL(***Microsoft Intermediate Language***) o también conocido como CIL(***Common Intermediate Language***) o simplemente IL(***Intermediate Language***). Este es usado por el CLR el cual lo convierte a ***Machine Code*** el cual sera ejecutado por el procesador.
- La información sobre el entorno, el lenguaje de programación, su versión y las bibliotecas de clases que se utilizarán para este código se almacenan en forma de metadatos con el compilador luego le indica al CLR cómo manejar este código.



Arquitectura y Servicios que brinda CLR:

CLR brinda varios servicios para facilitar el desarrollo al programador, algunos de ellos se mención a continuación:

- ***Type Checker***: Proporciona seguridad de tipos, mediante el ***Common Type System*** y el ***Common Language Specification*** que se forman parte de CLR para verificar los tipos que se utilizan en la aplicación.
- ***Exception Manager***: El administrador de excepciones en el CLR maneja las excepciones independientemente del lenguaje de ***.Net*** que la provoco. Para una aplicación en particular, el bloque ***catch*** de las excepciones se ejecuta en caso de que ocurran y si no hay un bloque ***catch***, se finaliza la aplicación.
- ***Thread Support***: El CLT proporciona compatibilidad con subprocesos para administrar la ejecución paralela de múltiples subprocesos. La clase ***System.Threading*** se utiliza como clase base para esto.
- ***Security Engine***:El CLR maneja los permisos de seguridad en varios niveles, nivel de código, nivel de carpeta y a nivel de maquina. Esto se hace utilizando diversas herramientas que se proporcionan en el ***Framework .Net***.
- ***Degub Engine***: CLR permite que una aplicación se pueda depurar durante el tiempo de ejecución utilizando el ***Debug Engine***.
- ***JIT Compiler***: En CLR el ***JIT Compiler*** convierte el MSIL en el código de maquina que es especifico para el entorno en el que se esta ejecutando el mismo. Este es almacenado para que este disponible para llamadas posteriores si es necesario.
- ***Garbage Collector***: La gestión automática de la memoria es posible utilizando el recolector de basura en CLR. El recolector de basura libera automáticamente el espacio de memoria después de que ya no es necesario, para que este pueda reasignarse.



Ahora que ya sabemos que es el ***Common Language Runtime***, veamos el DLR(***Dynamic Language Runtime***):

DLR

El DLR es un ***runtime enviorment*** que agrega un conjunto de servicios para lenguajes dinámicos al CLR. Este facilita el desarrollo de lenguajes dinámicos en el ***Framework .Net*** y agrega características dinámicas a los lenguajes estaticamente tipados.

El DLR //también ayuda a crear bibliotecas que admiten operaciones dinámicas. Por ejemplo, si se tiene una biblioteca que usa objetos ***XML*** o ***JavaScript Object Notation*** (***JSON***), estos objetos pueden aparecer como objetos dinámicos para los lenguajes que usan DLR. Esto permite a los usuarios de la biblioteca escribir código sintácticamente más simple y más natural para operar con objetos y acceder a miembros de objetos. Por ejemplo//.

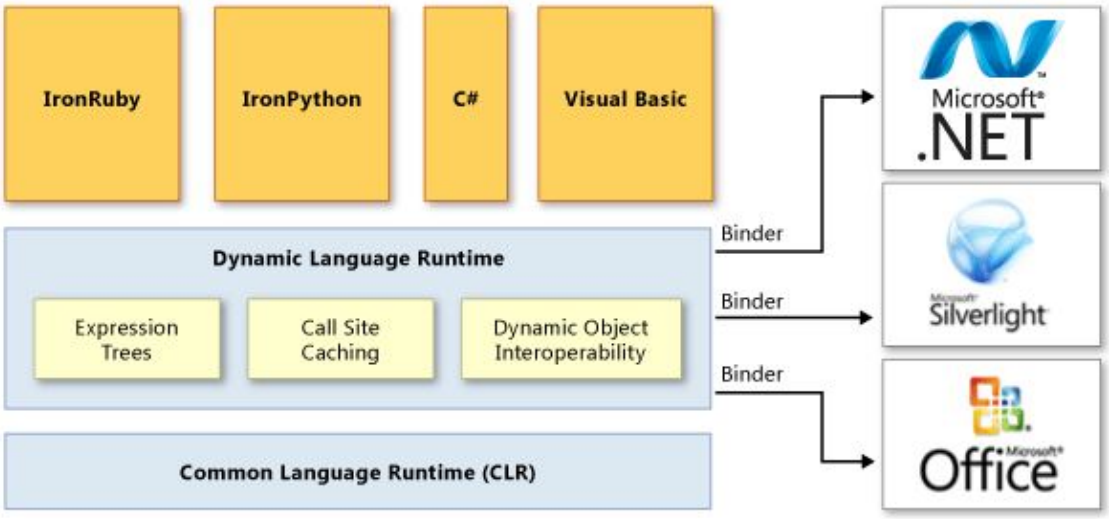
```
//For example, you might use the following code to increment a counter in XML in C#.

Scriptobj.SetProperty("Count", ((int)GetProperty("Count")) + 1);

//By using the DLR, you could use the following code instead for the same operation.

scriptobj.Count += 1;
```

Ventajas que ofrece DLR:

- Simplifica la creación de lenguajes dinámicos sobre el ***Framework .Net***
  - DLR permite a los desarrolladores evitar la implementación de ***lexical analyzers, parsers, semantic analyzers, code generators*** y otras herramientas que generalmente necesitarían implementar. Para poder usar la DLR un lenguaje necesita producir árboles de expresión, que representan código a nivel de lenguaje en una estructura en forma de árbol, rutinas auxiliares en tiempo de ejecución y objetos dinámicos opcionales que implementan la interfaz ***IDynamicMetaObjectProvider***. El DLR y .NET Framework automatizan muchos análisis de código y tareas de generación de código. Esto permite a los implementadores del lenguaje concentrarse en características únicas del lenguaje.
- Permite características dinámicas en lenguajes estaticamente tipados
  - Los lenguajes existentes del ***Framework .NET*** como ***C#*** y ***Visual Basic*** pueden crear objetos dinámicos y usarlos junto con objetos de tipo estático.
- Permite compartir librerías y objetos
  - Los objetos y bibliotecas implementados en un lenguaje pueden ser utilizados por otros lenguajes. El DLR también permite la interoperación entre lenguajes estáticamente tipados y dinámicos.
- Proporciona un rápido ***Dynamic Dispatch and Invocation***
  - El DLR proporciona una ejecución rápida de operaciones dinámicas al admitir almacenamiento en caché polimórfico avanzado.

Arquitectura y Servicios del DLR:

DLR architecture



DLR añade un conjunto de servicios que corren por encima del CLR para un mejor soporte de lenguajes dinámicos.

- ***Expression Trees***:
  - El DLR usa árboles de expresión para representar la semántica del lenguaje. Para este propósito, el DLR ha ampliado los árboles de expresión LINQ para incluir el flujo de control, la asignación y otros nodos de modelado de lenguaje. Si desea saber un poco mas sobre estos arboles de expresión los puede encontrar aquí Expression Trees C#
- ***Call site caching.***:
  - Una llamada dinámicas es un lugar en el código donde realiza una operación como `a + b` o `a.b ()` a objetos dinámicos. El DLR almacena en caché las características de `a` y `b` (generalmente los tipos de estos objetos) e información sobre la operación. Si dicha operación se ha realizado previamente, el DLR recupera toda la información necesaria de la memoria caché para un envío rápido.
- ***Dynamic object interoperability***:
  - El DLR proporciona un conjunto de clases e interfaces que representan objetos y operaciones dinámicos y pueden ser utilizados por desarrolladores de lenguajes y autores de bibliotecas dinámicas. Estas clases e interfaces incluyen IDynamicMetaObjectProvider, DynamicMetaObject, DynamicObject y ExpandoObject.

DLR utiliza ***binders*** en las llamadas para comunicarse no solo con el ***Framework .NET*** sino tambien con otras infraestructuras y servicios, incluidos ***Silverlight*** y ***COM***. Estos ***Binders*** encapsulan la semántica de un lenguaje y especifican cómo realizar operaciones en un llamado mediante el uso de árboles de expresión. Esto habilita lenguajes dinámicos y de tipo estático que utilizan el DLR para compartir bibliotecas y obtener acceso a todas las tecnologías que admite el DLR.

2. ¿Están al mismo nivel en la arquitectura de .NET?

Bueno como vimos anteriormente DLR es un conjunto de servicios, (basicamente es una libreria) por lo tanto no se encuentra en el mismo nivel que CLR. DLR no maneja ni ***JIT compilation***, ni ***Garbage Collection***, etc, es una librería que los lenguajes dinámicos y C# usan para ejecutar código dinámico.

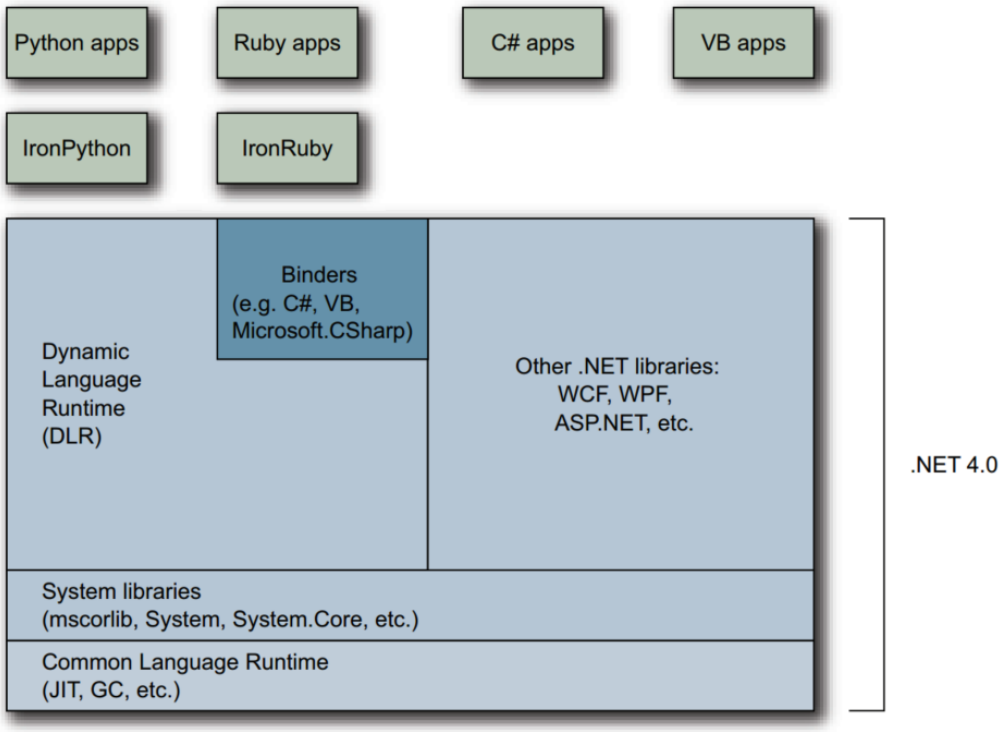


Figure 14.1 How the components of .NET 4 fit together, allowing static and dynamic languages to execute on the same underlying platform

Una de las ventajas que tiene esto es que DLR no fue necesario reimplementarlo para Mono, es decir que corre igual de bien en el ***Framework .NET*** como en ***.Net Core***.

Aunque el DLR no maneja el código nativo directamente, se puede pensar que hace un trabajo similar al CLR en un sentido: al igual que el CLR convierte IL (lenguaje intermedio) en código nativo, el DLR convierte el código representado mediante ***binders***, ***call sites***, metaobjetos y varios otros conceptos en árboles de expresión que luego pueden ser compilados en IL y eventualmente a código nativo por el CLR.

3. ¿Qué representan call site, receiver y binder?

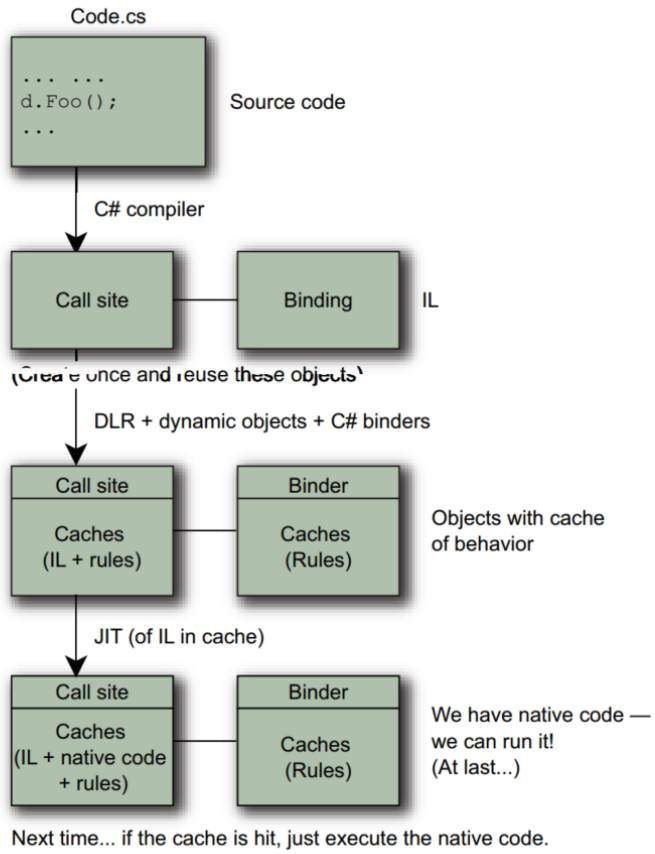


Figure 14.2 Lifecycle of a dynamic expression

- ***call sites***: Los ***call sites*** son el bloque de codigo mas pequeño que puede considerarse como una sola unidad ejecutable. Claro una expresión puede tener múltiples ***call sites*** pero el comportamiento se desarrolla de forma natural, evaluando uno solo a la vez. Ejemplo de un ***call site*** sencillo.

  ```
  d.Foo(10);
  ```

  Estos ***call sites*** se representan en codigo como ***System.Runtime.CompilerServices.CallSite***

- ***receivers & binders***: Dado los ***call sites*** ahora necesitamos algo para decidir que significan y como ejecutarlos. En DLR existen dos entidades que pueden decidir esto.
  - El ***receiver*** de un llamado y el ***binder***. El ***receiver*** de un llamado no es mas que el objeto sobre el cual se llama a un miembro, en el ejemplo anterior el ***receiver*** seria el objeto al cual `d` se refiera en tiempo de ejecucion.
  - El ***binder*** depende del lenguaje, y es parte del ***call site***. En el ejemplo anterior el compilador de ***C#*** emite codigo para crear un ***binder*** usando ***Binder.InvokeMember***. La clase ***Binder*** en este caso es ***Microsoft.CSharp.RuntimeBinder.Binder***.

Recomendamos leer como funciona DLR y el manejo la cache para optimizacion y mas a mayor profundidad, en el libro ***Manning, C# in Depth, SECOND EDITION, Chapter 14.4.2***

4. Explique que genera el compilador de C# para el siguiente código:

```
string text = "text to cut";
dynamic startIndex = 2;
string substring = text.Substring(startIndex);
```

El siguiente codigo es bastante sencillo, en este bloque de codigo tenemos dos operaciones dinamicas, una para llamar ***Substring***, y otra para convertir el resultado dinamicamente a ***string***(el cual es ***dynamic*** en tiempo de compilacion). Veamos que genera el compilador de ***C#*** para este ejemplo.

```
internal class Program
{
    // Nested Types
    // Bloque 1
    [CompilerGenerated]
    private static class <>o__0
    {
        // Fields
        public static CallSite<Func<CallSite, string, object, object>> <>p__0;
        public static CallSite<Func<CallSite, object, string>> <>p__1;
    }
    // Bloque 1

    // Methods
    private static void Main(string[] args)
    {
        // Bloque 2
        string str = "text to cut";
        object obj2 = 2;
        if (<>o__0.<>p__1 == null)
        {
            <>o__0.<>p__1 = CallSite<Func<CallSite, object, string>>.Create(
                Binder.Convert(CSharpBinderFlags.None, typeof(string),
typeof(Program)));
        }
        if (<>o__0.<>p__0 == null)
        {
            CSharpArgumentInfo[] argumentInfo = new CSharpArgumentInfo[]
            {
CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.UseCompileTimeType, null),
CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null) };
            <>o__0.<>p__0 = CallSite<Func<CallSite, string, object,
object>>.Create(
                Binder.InvokeMember(CSharpBinderFlags.None, "Substring", null,
typeof(Program), argumentInfo));
        }
        // Bloque 2

        // Bloque 3
        string str2 = <>o__0.<>p__1.Target(<>o__0.<>p__1, <>o__0.
<>p__0.Target(<>o__0.<>p__0, str, obj2));
        // Bloque 3
    }
}
```

En el `Bloque 1` vemos que se crea una clase estatica para guardar todos los ***call sites*** en el caso de nuestro ejemplo hay dos. El `Bloque 2` es donde se crean los ***call sites***, el primero es el de la conversion a ***string*** y el segundo el del llamado a ***Substring*** y luego de esto simplemente son invocados en el `Bloque 3`.

5. Muestre para qué otros lenguages además de C# la DLR brinda soporte. Compare la eficiencia con los intérpretes de los lenguajes originales.

- ***IronRuby***: ***IronRuby*** es una implementación del lenguaje de programación ***Ruby*** dirigido a ***Microsoft .NET Framework***. Se implemento sobre el ***Dynamic Language Runtime***(DLR), que proporciona tipeo dinámico y envío de métodos dinámicos, entre otras cosas, para lenguajes dinámicos.
- ***IronPython***: ***IronPython*** es una implementación del lenguaje de programación Python dirigido a .NET Framework y Mono. Está escrito completamente en C #, aunque parte de su código es generado automáticamente por un generador de código escrito en Python. Al igual que ***IronRuby***, ***IronPython*** se implemento sobre el ***Dynamic Language Runtime***(DLR).

Microsoft abandonó IronPython (y su proyecto hermano IronRuby) a fines de 2010, actualmente IronPython lo mantienen un grupo de voluntarios en GitHub. Es un software gratuito y de código abierto, y se puede implementar con Python Tools para Visual Studio. IronRuby esta inactivo desde entonces.

Comparar la eficiencia de IronPython y IronRuby sobre Python y Ruby en sus interpretes originales.

En general la comparación entre diferentes implementaciones de estos lenguajes es difícil. Ya que depende mucho de la tarea que se este realizando. Por ejemplo ***IronPython*** es mas eficiente que ***Python*** en tareas que usen varios hilos, o múltiples núcleos del ***CPU***, ya que tiene servicios como ***JIT*** entre otros, que son brindados por la CLR, si el codigo es compilado a ***ByteCode*** o ***MachineCode***, etc. En nuestro caso, nos limitaremos a correr varios algoritmos sobre cada implementacion 30 veces y promediar esos tiempos.

Recomendamos leer un poco mas sobre el tema en el siguiente enlace:

1. GeeksforGeeks

IronPython vs ***Python***:

```
# En ambos caso se usa el tiempo promedio de 30 pruebas.
# Usando la libreria time. La implementacion usada se encuentra en =>
code\test.py
```

Resultados:

1. Fibonacci(35)
2. BubbleSrot arr[10000]
3. QuickSort arr[1000000]

Implementacion	Fibonacci	QuickSort	BubbleSort
IronPython	1.53 segundos	0.26 segundos	3.66 segundos
Python	2.13 segundos	0.27 segundos	5.81 segundos

Como IronPython hay otras variantes, JPython, CPython (el mas usado), PyPy, Nukita, etc cada uno de estos compila Python de una manera distinta que y cada cual tiene sus ventajas en rendimiento. Si desean profundizar mas sobre esto, les sugerimos algunos links:

1. Medium

2. Blog
3. StackOverflow
4. StackOverflow

IronRuby vs ***Ruby***:

IronRuby fue abandonado en 2011, la ultima version de Ruby que tiene soporte fue la 1.9, asi que para comparar mejor ambas versiones usamos una comparación que encontramos en el siguiente enalce, donde usan el Ruby Benchmark Suit. En el enlace anterior pueden encontrar el codigo de las pruebas realizadas.

Resultados:

Benchmark	Params	IronRuby	Ruby 1.9.1
bm_mergesort.rb	1	3.906	3.219
bm_quicksort.rb	1	11.594	8.703
bm_so_ackermann.rb	9	14.938	9.281
bm_primes.rb	3000	9.594	0.031
bm_so_count_words.rb	100	60.688	42.250

6. Compare la estrategia de implementación de dynamic de C# con la propuesta e implementada para Java.

Dynamic en C#:

El tipo `dynamic` indica que el uso de la variable y las referencias a sus miembros omiten la comprobación en tiempo de compilación. En cambio, estas operaciones se resolverán en tiempo de ejecución. `dynamic` se comporta como el tipo `object` en la mayoría de las circunstancias. En particular, cualquier expresión no nula se puede convertir al tipo `dynamic`. El tipo `dynamic` se comporta diferente a `object` en que las operaciones que contengan expresiones de tipo `dynamic` el compilador no las resuelve, ni chequea su tipo. El compilador lo que hace es empaquetar información sobre esta operación y pasarla, para luego en tiempo de ejecución evaluar la operación usando esta información. Como parte de este proceso, las variables de tipo `dynamic` se compilan a variables de tipo `object`. Por lo tanto, el tipo `dynamic` solo existe en tiempo de compilación, no en tiempo de ejecución.

Veamos a que nos referimos con un ejemplo.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        Console.WriteLine(dyn.GetType());
        Console.WriteLine(obj.GetType());
```

```
    }

    // Salida:
    // System.Int32
    // System.Int32
}
```

Si este ejemplo lo escribimos en el Visual Studio(o cualquier Editor que tenga Intellisense), colocando el puntero encima de `dyn` y `obj` en las declaraciones de `WriteLine`, podemos ver como nos muestra que el tipo de `dyn` es efectivamente `dynamic` y el de `obj` es `object`. Las declaraciones `WriteLine` lo que imprimen seria el tipo en tiempo de ejecución el cual en ambos casos como se espera es `System.Int32`.

Para notar mejor la diferencia entre `dyn` y `obj` en tiempo de compilación veamos que pasa si agregamos estas dos lineas al ejemplo anterior.

```
dyn = dyn + 3;
obj = obj + 3;
```

Aquí hay un error en tiempo de compilación, ya que estamos intentando agregar un entero a un objeto en la expresion `obj + 3`. Sin embargo, no se informa de ningún error en `dyn + 3`. Esto sucede porque la expresión que contiene `dyn` no se verifica en tiempo de compilación, ya que su tipo es `dynamic`.

El tipo `dynamic` puede ser usado para:

1. Declarar variables
2. Argumento de funciones
3. Tipo de retorno de un método

Ejemplo:

```
using System;

namespace DynamicExamples{
    class Program{
        static void Main(string[] args){
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.exampleMethod(10));
            Console.WriteLine(ec.exampleMethod("value"));

            // Esta linea causa un error en tiempo de compilacion, ya que el
tipo de ec es ExampleClass, y el compilador
            // sabe en tiempo de compilacion que el metodo ExampleMethod solo
toma un argumento
            //Console.WriteLine(ec.exampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.exampleMethod(10));

            // Ahora como dynamic_ec es de tipo dynamic, el chequeo anterior no
se hace en tiempo de compilacion lo cual
            // implica que al intentar correr este codigo no se produce ningun
error en tiempo de compilaicon
            // Sin embargo, si lanza error en tiempo de ejecucion
            //Console.WriteLine(dynamic_ec.exampleMethod(10, 4));
```

```
        }
    }

    class ExampleClass{
        static dynamic field;
        dynamic prop { get; set; }

        public dynamic exampleMethod(dynamic d){
            dynamic local = "Local variable";
            int two = 2;

            if (d is int){
                return local;
            }
            else{
                return two;
            }
        }
    }
}

// Salida:
// Local variable
// 2
// Local variable
```

Si desean profundizar mas en el uso de `dynamic` recomendamos la documentación de ***Microsoft***:

1. Using of type dynamic
2. System.Dynamic.DynamicObject
3. Walkthrough: creating and using dynamic objects

Una de las características principales de lenguajes como Python, es poder agregar/eliminar miembros a una instancia de una clase en tiempo de ejecución. En C# para lograr algo parecido a esto tenemos los ExpandoObject, esta clase permite el ***dynamic binding***, lo que le permite utilizar sintaxis estándar como `sampleObject.sampleMember` en lugar de `sampleObject.GetAttribute("sampleMember")`. ***ExpandoObject*** representa un objecto cuyos miembros se pueden agregar o eliminar dinámicamente en tiempo de ejecución.

Ademas de ***ExpandoObject*** en C# también tenemos ***DynamicObject*** el cual permite definir tipos que tienen su propia semántica de despacho dinámico. En este caso ***DynamicObject*** es una clase base para especificar el comportamiento dinámico en tiempo de ejecución. Este clase debe ser heredada, no se puede instanciar directamente.

A continuación un ejemplo sencillo de ambas clases. Para profundizar un poco mas en ellas recomendamos los siguientes enlaces:

ExpandoObject:

```
class Program  {
    static void Main(string[] args)  {
        dynamic sampleObject = new ExpandoObject();

        // Agregando una nueva propiedad a una instancia de ExpandoObject
        sampleObject.test = "Dynamic Property";
        Console.WriteLine(sampleObject.test);
```

```
        Console.WriteLine(sampleObject.test.GetType());
        // Salida
        // Dynamic Property
        // System.String

        //Agregando un metodo
        // Los métodos representan expresiones lambda que se almacenan como
delegados
        sampleObject.number = 10;
        sampleObject.Increment = (Action)(() => { sampleObject.number++; });

        // Before calling the Increment method.
        Console.WriteLine(sampleObject.number);
        // Salida:
        // 10

        sampleObject.Increment();

        // After calling the Increment method.
        Console.WriteLine(sampleObject.number);
        // Salida:
        // 11

    }
}
```

DynamicObject:

```
// The class derived from DynamicObject.
public class DynamicDictionary : DynamicObject{
    // The inner dictionary.
    Dictionary<string, object> dictionary
        = new Dictionary<string, object>();

    // This property returns the number of elements
    // in the inner dictionary.
    public int Count{
        get{
            return dictionary.Count;
        }
    }

    // If you try to get a value of a property
    // not defined in the class, this method is called.
    public override bool TryGetMember(
        GetMemberBinder binder, out object result){
        // Converting the property name to lowercase
        // so that property names become case-insensitive.
        string name = binder.Name.ToLower();

        // If the property name is found in a dictionary,
        // set the result parameter to the property value and return true.
        // Otherwise, return false.
        return dictionary.TryGetValue(name, out result);
    }

    // If you try to set a value of a property that is
```

```
    // not defined in the class, this method is called.
    public override bool TrySetMember(
        SetMemberBinder binder, object value){
        // Converting the property name to lowercase
        // so that property names become case-insensitive.
        dictionary[binder.Name.ToLower()] = value;

        // You can always add a value to a dictionary,
        // so this method always returns true.
        return true;
    }
}

class Program{
    static void Main(string[] args){
        // Creating a dynamic dictionary.
        dynamic person = new DynamicDictionary();

        // Adding new dynamic properties.
        // The TrySetMember method is called.
        person.FirstName = "Ellen";
        person.LastName = "Adams";

        // Getting values of the dynamic properties.
        // The TryGetMember method is called.
        // Note that property names are case-insensitive.
        Console.WriteLine(person.firstname + " " + person.lastname);

        // Getting the value of the Count property.
        // The TryGetMember is not called,
        // because the property is defined in the class.
        Console.WriteLine(
            "Number of dynamic properties:" + person.Count);

        // The following statement throws an exception at run time.
        // There is no "address" property,
        // so the TryGetMember method returns false and this causes a
        // RuntimeBinderException.
        // Console.WriteLine(person.address);
    }
}

// This example has the following output:
// Ellen Adams
// Number of dynamic properties: 2
```

Dynamic in Java:

Java no trae por defecto un ***feature*** que permita algo similar al 'tipado dinámico' con ***dynamic*** de C#.Sin embargo tiene un ***feature*** conocido como ***dynamic proxy***, que se remplaza o situá como intermediario de una clase cualquiera para agregarle cierta funcionalidad.

Dynamic Proxy: En programación ***Proxy*** es un patrón de diseño. Los objetos ***Proxy*** se crean y son usados cuando se quiere añadir o modificar alguna funcionalidad de una clase que ya existe. El objeto ***Proxy*** pasa a ser un ***wrapper*** o el ***handler*** de la clase que maneja.

En Java ***Dynamic Proxy*** permite a una clase con un solo método, maneje llamadas de múltiples métodos a clases arbitrarias con un número arbitrario de métodos. Por dentro, esto lo que hace es enrutar todas las invocaciones de estos métodos a un único controlador, el método `invoke()`.

El uso del proxy `built-in` que trae Java es sencillo. Todo lo que se necesita hacer es implementar a `java.lang.InvocationHandler`, para que el objeto proxy pueda invocarlo. La interfaz `InvocationHandler` es extremadamente simple, contiene un único método: `invoke()`. Cuando `invoke()` es llamado, los argumentos contienen, al objeto original, el cual se esta proxificando, el método que fue invocado(como un objeto `Method` usando ***reflection***), y un array con los argumentos originales.

Veamos el siguiente ejemplo:

```
import java.lang.reflect.*;

interface Calculator {
  int add(int left, int right);
}

class Auditor {
  public void audit(String service, String extraData) {
    System.out.println(service + extraData);
  }
}

class CalculatorSum implements Calculator {
  public int add(int left, int right) {
    return left + right;
  }
}

class CalculatorMult implements Calculator {
  public int add(int left, int right) {
    return left * right;
  }
}

class AuditingInvocationHandler implements InvocationHandler {
  public Object target;
  public Auditor auditor;

  public AuditingInvocationHandler(Auditor auditor, Object target) {
    this.auditor = auditor;
    this.target = target;
  }

  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    auditor.audit(target.getClass().getName(), " before " + method.getName());
    Object returnObj = method.invoke(target, args);
    auditor.audit(target.getClass().getName(), " after " + method.getName());

    return returnObj;
  }
}
```

```
public class Main {
  public static void main(String[] args){
    Auditor auditor = new Auditor();
    Calculator real = new CalculatorSum();
    InvocationHandler handler = new AuditingInvocationHandler(auditor, real);
    Calculator proxy = (Calculator) Proxy.newProxyInstance(
      ClassLoader.getSystemClassLoader(), new Class[] { Calculator.class },
handler);
    real.add(2, 2); // Will not be audited
    proxy.add(2, 2); // Will be audited
  }
}
```

Este código es un poco mas complicado que algunos ejemplos anteriores, aquí el método ***AuditingInvocationHandler*** es capaz de manejar cualquier método para cualquier objeto que se llame, notar que en su constructor estamos pasando un ***Object***.

Para poder usar este `handler`, dinamicamente se generara un `proxy` que va a pretender implementar para nuestro caso la interfaz `Calculator`, pero en verdad lo que hará es invocar al `handler`. Esto se puede hacer con el método `newProxyInstance()` que se encuentra en `Proxy`.

Note que con esta implementación del `handler` podemos usarlo para diferentes proxys, cada uno implementando una interfaz diferente.