Universidad de La Habana

Seminario 5

Adrián Rodríguez Portales
Andy Castañeda Guerra
Richard Gracía de la Osa
Adrián Hernández
Eduardo Moreira

March 24, 2020

1 Clausura

En programación, la clausura es una técnica relacionada con la programación funcional, mediante la cual se asocia una función con el contexto que la encapsula. Esto permite a la función acceder a variables que están fuera de su cuerpo. Este concepto puede ser extendido al paradigma orientado a objetos, generalmente mediante el uso de algunos trucos por parte del compilador

1.1 Clausura en C#

En C# las clausuras se pueden crear usando delegados y funciones lambda. Cuando uno de estos objetos es creado, todas aquellas variables que sean usadas en el body del mismo y que además sean declaradas fuera de este, son "copiadas" y guardadas dentro de la clausura, de manera que puedan luego ser usadas sin problema; en la literatura se refieren a estas variables (intuitivamente) como variables libres. Es importante saber que este proceso ocurre justo cuando (y donde) se define la función en cuestión. La pregunta entonces sería: cómo logra C# esto? Mencionamos que, idealmente, los valores deberían ser guardados. Analicemos el sgte código:

```
var actions = new Action[10];
for (int x = 0; x < actions.Length; x++)
{
   int y = x;
   actions[x] = () =>
   {
      int z = x;
      Console.WriteLine("{0}, {1}, {2}\n", x, y, z);
```

```
   };
}
```

La respuesta esperada sería: Pero lo q ocurre es:

Table 1.1:

Respuesta esperada	**Respuesta**
0,0,0	10,0,10
1,1,1	10,1,10
...	...
9,9,9	10,9,10

Resulta que C# modela las clausuras como clases fantasmas, creadas cuando se compila el código a CIL, veamos dos ejemplos de códigos en C#, y sus respectivos códigos IL generados:

```
Using System;

namespace Seminario5LPNetFramework
{
    public class Program
    {
        public static void Main(string[] args)
        {
            int x = 2;
            Action actions = () => { int y = 4; };
        }
    }
}
```

```
        .locals init ([0] int32 x,  //Declaracion de las variables locales, int
            x, Action action
           [1] class [mscorlib]System.Action actions)
  IL_0000: nop         // no hace nada, se usa para breakpoints
  IL_0001: ldc.i4.2      //pushear un int32 con valor 2 a la pila
  IL_0002: stloc.0       //pop a la pila y guardar en la variable local 0 (int x)
  IL_0003: ldsfld    class [mscorlib]System.Action
      Seminario5LP.Program/'<>c'::'<>9__0_0' //.according to Wikipedia, carga el
      valor de la derecha en el stack, whatever that means
  IL_0008: dup      // pushea al tope de la pila una copia de lo que este en el
      tope
  IL_0009: brtrue.s  IL_0022     // hace pop, y si lo que salga vale true, salta
      a IL_0022
  IL_000b: pop      //duh
  IL_000c: ldsfld    class Seminario5LP.Program/'<>c'
      Seminario5LP.Program/'<>c'::'<>9'  //mismo enredo de ahorita
  IL_0011: ldftn     instance void
      Seminario5LPNetFramework.Program/'<>c'::'<Main>b__0_0'()
//pushea un puntero que apunte al mtodo de la derecha(<Main>b__0_0)
  IL_0017: newobj    instance void [mscorlib]System.Action::.ctor(object,
                                                              native int)
  IL_001c: dup
  IL_001d: stsfld    class [mscorlib]System.Action
```

```
      Seminario5LP.Program/'<>c'::'<>9__0_0' //llama al ctor de lo de la derecha
  IL_0022: stloc.1 //hace pop y lo que salga lo guarda en la variable local
      1(Action action)
  IL_0023: ret     //return
} // end of method Program::Main

.method assembly hidebysig instance void
       '<Main>b__0_0'() cil managed
{
  // Code size     4 (0x4)
  .maxstack 1
  .locals init ([0] int32 y)
  IL_0000: nop     //nada
  IL_0001: ldc.i4.4   //pushea un int32 con valor 4 a la pila
  IL_0002: stloc.0    //saca de la pila y asigna lo que sale a la variable local
      0(int y)
  IL_0003: ret        //return
} // end of method '<>c'::'<Main>b__0_0'
```

```
Using System;

namespace Seminario5LPNetFramework
{
    public class Program
    {
        public static void Main(string[] args)
        {
            int x = 2;
            Action actions = () => { x = 4; };
        }
    }
}
```

```
.locals init ([0] class Seminario5LP.Program/'<>c__DisplayClass0_0'
    'CS$<>8__locals0',
          [1] class [mscorlib]System.Action actions) //variables locales, notar
              que lo que antes era un int x, ahora se sustituy por una clase
IL_0000: newobj    instance void
    Seminario5LP.Program/'<>c__DisplayClass0_0'::.ctor() //llamar al ctor de
    c__DisplayClass0_0
  IL_0005: stloc.0 //pop y guardar en la variable local 0
  IL_0006: nop  //nada
  IL_0007: ldloc.0 //cargar variable local 0 al stack
  IL_0008: ldc.i4.2 //cargar al stack un int32 que valga 2
  IL_0009: stfld     int32 Seminario5LP.Program/'<>c__DisplayClass0_0'::x
      //equivalete ac __DisplayClass0_0.x = 2
  IL_000e: ldloc.0 //cargar nuevamente variable local 0 al stack
  IL_000f: ldftn     instance void
      Seminario5LP.Program/'<>c__DisplayClass0_0'::'<Main>b__0'()         //push
      puntero a metodo b__0 de c__DisplayClass0_0
  IL_0015: newobj    instance void [mscorlib]System.Action::.ctor(object,
                                                                  native int)
                                                                      //pushear
                                                                      instancia de
                                                                      Action
```

```
  IL_001a: stloc.1 //pop y cargar a variable local 1
  IL_001b: ret     //FIN
} // end of method Program::Main

<Main>b__0 de la clase c__DisplayClass0_0
IL_0000: nop
  IL_0001: ldarg.0 //cargar argumento 0(idealmente la variable int x)
  IL_0002: ldc.i4.4 //pushear un int32 = 4
  IL_0003: stfld     int32
      Seminario5LPNetFramework.Program/'<>c__DisplayClass0_0'::x //setear this.x
      = 4
  IL_0008: ret     //FIN 2.0
} // end of method '<>c__DisplayClass0_0'::'<Main>b__0'
```

En C#, para permitir las clausuras, se crea una Clase auxiliar, la cual va a poseer como único método (aparte del constructor) uno que contenga exactamente el código declarado en el body de la expresión lambda. Además, esta clase va a tener como propiedades cada una de las variables libres que deban ser "capturadas" del ámbito donde se define. De ahí surge una pregunta, que pasaría si tengo en el mismo scope otra declaración de función lambda, que también utilice variables libres, y por tanto, deba crearse una clausura? Pues resulta que en realidad la clase auxiliar no tiene por que poseer un solo método, en efecto, tiene un método por cada función lambda que comparta dicho ámbito, y cada uno de estos métodos contiene como código el body correspondiente a la función que representa. Y si llevamos la curiosidad un paso más allá, y en el mismo ámbito declaramos una función lambda que utilice variables libres de distintos scopes, quedaría algo asi:

```
public static void Main(string[] args)
       {
           int x = 5;
           Action[] actions = new Action[10];
           for (int i = 0; i < actions.Length; i++)
           {
               int j = 1;
               actions[i] = () => { int k = 2; j = 4; };
               Action actionAux = () => { j = 3; x = 6; };
           }
       }
```

Bueno, analizando el código IL generado (el cual nos tomaremos la libertad de no poner ahora por lo enredado que se vuelve), sucede que se crean dos clausuras, o mejor dicho, dos clases auxiliares, una que estará "ligada" al scope donde se define la variable x, y que poseera como propiedad a la x y no tendrá ningún método excepto el constructor, y la segunda clase auxiliar tendrá como propiedades al int j, y una referencia a la clase auxiliar antes mencionada, de manera que pueda accederse al valor de x; además, será a esta clase a la que se le asignarán los dos métodos asociados a las funciones lambdas que se declaran. Para mayor comprensión, brindamos un código en C# que simula este comportamiento:

```
class AuxiliarClass1
    {
        public int x;
        public AuxiliarClass1(int x)
        {
            this.x = x;
```

```
    }
}
class AuxiliarClass2
{
    public int j;
    public AuxiliarClass1 AuxC1;
    public AuxiliarClass2(int j, AuxiliarClass1 ac)
    {
        this.j = j;
        this.AuxC1 = ac;
    }
    public void Method1() //asociado a la expr lambda () => { int k = 2;j =
        4; }
    {
        int k = 2;
        this.j = 4;
    }


public void Method2() //asociado a la expr lambda () => { j = 3; x = 6; };
    {
        this.j = 3;
        this.AuxC1.x = 6;
    }
}
```

---

2 Expresiones lambda en C++

Una expresión lambda es una forma conveniente de definir un anonymous function object(clausura). A partir de C++11 es que se introducen las expresiones lambdas. Una expresion lambda en c++ tiene la siguiente sintaxis:

$$[\,]\ ()\ \text{mutable throw()} \rightarrow \text{int}\ \{\}$$

Está dividida en 6 partes:

- Capture clause : [ ]
  El capture clause es el que define que variables van a ser accesibles en el cuerpo de la expresion lambda, y tambien especifica si las variables capturadas se van a tomar por valor o por referencia.Primeramente si dejamos el capture clause vacio,el cuerpo no tendra acceso a ninguna variable externa. A partir de C++14 se pueden introducir variables en la expresion lambda y su tipo lo infiere el compilador. Se puede usar el modo por defecto [] que toma todas las variables externas por referencia, o [=] que toma todas las variables externas por valor. Ahora, si queremos tomar alguna variable en específico simplemente la ponemos en el capture clause,y será tratada por su tipo por defecto, a no ser que se lo cambiemos con el modificador o *. Ojo, si el default de la variable era por valor y se le puso el modificador * delante, va a dar error de compilación, igual que si es por referencia y se le pone el modificador &. Igual si usamos [,var1,var2,...] todas las variables explícitas deben ser capturadas por valor, si no va a dar error de compilación ya que estás tomando la variable por referencia dos veces y habría redundancia. También si usamos [=,var1,var2,..] todas las variables explícitas deben ser capturadas por referencia, porque también da error de compilación. Estos son unos ejemplos de los errores que puede dar:

```
struct S { void f(int i); };
void S::f(int i){
  [&, i]{};      //OK
  [&, &i]{};     //ERROR: i preceded by & when & is the default
  [=, this]{};   //ERROR: this when = is the default
  [=, *this]{ }; //OK: captures this by value.See below.
  [i, i]{};      //ERROR: i repeated
}
```

El operador *this fue incluido a partir de C++ 17 que el parametro this es para acceder a los miembros de una clase,y el operador por valor fue introducido a partir de esa version de C++.

- Lista de parámetros :
  Esta es una parte opcional,pues una expresión lambda puede recibir parámetros o no. A partir de C++14 los parámetros pueden ser genéricos usando la palabra clave auto,e incluso pueden recibir otra expresión lambda como parámetro.

```
auto y = [] (auto first, auto second)
{
   return first + second;
};
```

- Mutable specification: mutable
  Esta es una parte opcional,y define si las variables capturadas por valor en la expresion lambda pueden ser modificadas,y sus cambios se verán afectados en el ámbito de la expresión, no en la variable, ya que no se le pasó la referencia. Es como si creara una variable nueva para su scope. Lo que haría es quitarle el modificador const al operador ().
- Exception-specification : throw()
  Indica si la función puede lanzar excepciones, si no se pone, por defecto simplemente puede lanzar excepciones. Se puede poner noexcept para que no lo haga.
- Trailing-return-type : type
  El tipo de retorno de le expresión lambda es automáticamente deducido. Aunque puede ser especificado manualmente con la palabra clave ->. Si no devuelve nada, el compilador asume que es void.
- Body : {}
  Toma las variables capturadas en la clausura. Puede declarar variables locales. En fin, es un bloque igual que el de un método cualquiera (con la particularidad de la clausura).

Las expresiones lambdas en C++ al ser creadas lo que hacen es generar una clase nueva e implementarle el operador(). Las variables por valor se copian y se guardan en la clase como campos, y las variables tomadas por referencia se toman directamente, no se crea una copia. Un detalle importante que debemos tener en cuenta es que cuando tomamos una variable por referencia hay que ser muy cuidadoso, ya que no extiende su tiempo de vida,o sea, supongamos que tenemos un método que recibe una variable por referencia,y dentro del método creamos una expresión lambda y le pasamos esa variable por referencia, y el valor de retorno del método es esa expresión lambda que creamos. Note que lo que tomó el método de la variable por referencia fue una copia de la misma, y que esa referencia esta guardada en la pila, y cuando el método retorne la función, esta tendrá una referencia apuntando a un lugar en la pila dispuesto a ser usado nuevamente, y esto puede generar un comportamiento inesperado.

Veamos algunas características de las expresiones lambdas analizando el siguiente código:

```
auto funcs = vector<function<int()>>();
int x = 1;
funcs.push_back([=] { return x; });
x = 2;
funcs.push_back([&] { return x; });
x++;
funcs.push_back([x = 4] { return x; });
for (auto f : funcs)
{
   int y = f();
   cout << y << endl;
}
```

La primera expresión lambda imprime 1, ya que se toma x por valor y se le da `return` directamente, y el valor de x sigue siendo el mismo. La segunda imprime 3 porque x se toma por referencia y como en la línea después de la declaración se le hace x++, entonces ese cambio también se ve reflejado en la x que tomó la función por referencia, por lo tanto imprime 3. La tercera imprime 4 ya que lo que se aprovecha es la facilidad que ofrece C++14 de introducir variables en el scope de la función,o sea, esa x que se crea no tiene nada que ver con la x que hay en el scope externo.
Para acceder a los miembros de una clase se le pasa la palabra clave this en el capture clause de la expresión lambda, pasado por valor por defecto,que en este caso es por referencia, aunque también se puede usar *this y se crea una copia de la clase y es lo que se le pasa. Dentro del body se utilizan los miembros llamándolos normalmente. En caso de haber alguna colisión con los nombres se puede expecificar que miembro se está llamando de la siguiente forma: `<class_name>::<member>` dentro del body. Otra forma sería encapsular a this en una variable en el capture clause.

3 Delegados y expresiones lambdas en Java

3.1 Existen los delegados en Java?

Los delegados realmente no existen en Java. Sin embargo, se puede lograr el mismo efecto utilizando `reflection`,el cual permite obtener métodos de objetos que se pueden invocar. Esto no es recomendable en la mayoría de los casos porque disminuye el `perfomance` del programa y el código es díficil de entender y debuguear.
En el siguiente ejemplo se aprecia como se usa `refelction` para pasar el método `method1` como parámetro al método `method2` :

```
import Java.lang.reflect.Method;

public class Demo {

    public static void main(String[] args) throws Exception{
        Class[] parameterTypes = new Class[1];
        parameterTypes[0] = String.class;
        Method method1 = Demo.class.getMethod("method1", parameterTypes);

        Demo demo = new Demo();
        demo.method2(demo, method1, "Hello World");
    }
```

```
    public void method1(String message) {
        System.out.println(message);
    }

    public void method2(Object object, Method method, String message) throws
        Exception {
        Object[] parameters = new Object[1];
        parameters[0] = message;
        method.invoke(object, parameters);
    }
}
```

Otra forma es usar Strategy Pattern.

3.1.1 Strategy Pattern

Este patrón facilita la implementación de distintas estrategias o comportamientos específicos en clases hijas a través de una clase común. Así, en tiempo de ejecución y en función de algún parámetro como el tipo de instancia, se ejecutará la estrategia concreta para esa situación.
Se recomienda usar este patrón cuando en un mismo programa debamos proporcionar distintas alternativas de comportamiento, permitiendo a través de clases independientes, encapsular estrategias.
Los distintos componentes de este patrón son:

- Interfaz Strategy:
  Será aquella interfaz que define el nombre del método o métodos que conformarán la estrategia.
- Clases Strategy concretas:
  Todas aquellas clases que implementan la interfaz Strategy dando forma al algoritmo.
- Contexto:
  Elemento donde se desarrollará la estrategia.

A continuación se muestra cómo se pueden simular los `delegados` usando este patrón.
En C# declarar un delegado sería algo así:

```
    public delegate void SomeFunction();
```

En Java se haría lo siguiente:

```
    public interface ISomeBehaviour{
       void SomeFunction();
    }
```

Para las implementaciones concretas del método, se definen clases que implementen la interfaz

```
    public class TypeABehaviour implements ISomeBehaviour{
       public void SomeFucntion(){
          //...
       }
    }

    public class TypeBBehaviour implements ISomeBehaviour{
       public void SomeFucntion(){
          //...
```

```
        }
    }
```

Luego, donde se tendría un delegado SomeFunction en C#, se usa una referencia a ISomeBehaviour:

```
    //C#
    SomeFunction doSomething = SomeMethod();
    doSomething();
    doSomething = SomeOtherMethod;
    doSomething();

    //Java
    ISomeBehaviour someBehaviour = new TypeABehaviour();
    someBehaviour.SomeFunction();
    someBehaviour = TypeBBehaviour();
    someBehaviour.SomeFunction();
```

3.2 Expresiones lambda en Java

3.2.1 Antes de Java 8

Antes de Java 8 no existían las expresiones lambdas. Sin embargo, haciendo uso de las clases anónimas se puede lograr un comportamiento similar.
Las clases anónimas en Java son una solución rápida para implementar una clase que se va utilizar una vez y de forma inmediata. De la definición anterior concluimos dos cosas la primera es que para crear una clase anónima es necesario haber definido una interfaz, una clase o una clase abstracta. La clase anónima lo que hará sera implementar la interfaz definida o sobrescribir los métodos definidos. La sintaxis para la definición de una clase de este tipo es:

```
Superclase var = new Superclase(){
//Definicin de la clase annima

}
```

En el siguiente ejemplo se ve como se crea una clase anónima:

```
public class Operaciones {

    public void imprimir(){
        System.out.println("imprimir original");
    }
}

public class Externa {
    Operaciones op = new Operaciones(){
        // definicin de la clase anonima
      public void imprimir(){
          System.out.println("Imprimir anonimma");
      }
    };// creacion del objeto termina con ;

    void proceso(){
```

```
        op.imprimir();
}
```

3.2.2 Después de Java 8

Entre las múltiples novedades que brinda Java 8 se encuentran las expresiones lambda. su sintáxis básica se detalla a continuación:

(parámetros) -> {cuerpo de la expresión}

Con la llegada de las expresiones lambdas se agrega un nuevo concepto: interfaces funcionales. Se le conoce como interface funcional a toda aquella interface que tenga solamente un método abstracto, es decir puede implementar uno o más métodos default, pero deberá tener forzosamente un único método abstracto.
Ejemplo de interfaces funcionales:

```
package com.osb.functionalinterface;

public interface IStrategy {
    public String sayHelloTo(String name);
}
```

```
package com.osb.functionalinterface;
public interface IStrategy {

    public String sayHelloTo(String name);

    public default String sayHelloWord(){
        return "Hello word";
    }
}
```

Otra forma de asegurar que se está definiendo correctamente una interface funcional, es anotarla con `@FunctionalInterface` , ya que al anotarla el IDE automáticamente arrojará un error si no se cumplen con las reglas de una interface funcional. Sin embargo, es importante resaltar que en tiempo de ejecución no dará un comportamiento diferente, puesto que es utilizada para prevenir errores al momento de definir las interfaces.

```
package com.osb.functionalinterface;
@FunctionalInterface
public interface IStrategy {

    public String sayHelloTo(String name);

    public default String sayHelloWord(){
        return "Hello word";
    }
}
```

Interfaces funcionales con expresiones lambda:

```
package com.osb.functionalinterface;
public class Main {
    public static void main(String[] args) {
        IStrategy strategy = (name) -> "Hello " + name;
```

```
        System.out.println(strategy.sayHelloTo("Jon Doe"));
        System.out.println(strategy.sayHelloWord());
    }
}
```

La expresión lambda siempre se asigna al método abstracto, por este motivo el método sayHelloTo es implementado con el cuerpo de la expresión lamba, mientras que el método sayHelloWord continua con la misma implementación que viene desde la interface.