**Informe**

**Seminario 5**

**Equipo 5 :**

* Luis Ernesto Ibarra Vazquez
* Luis Enrique Dalmau Coopat
* Jessy Gigato Izquierdo
* Laura Tamayo Blanco
* Alberto Helguera Fleitas

**C#**

¿Qué es la clausura?

La clausura es un tipo especial de función que permite usar variables definidas fuera de su entorno (scope). Cuando la función es creada, las variables externas se “traen” al scope, o sea, se asocian de forma tal que sean accesibles a la función

¿Cómo se ve la clausura en C#?

En C# la clausura se puede ver con el uso de expresiones lambda.

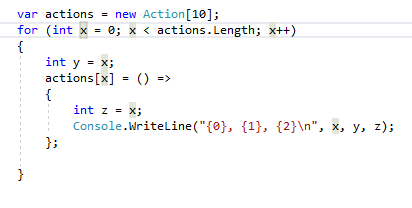
Nota: No todas las expresiones lambda usan clausura.

Ejemplo:

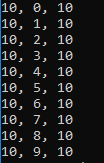


En este caso, la expresión lambda es independiente de otras variables que puedan existir en el programa y por tanto no tiene sentido usar clausura.

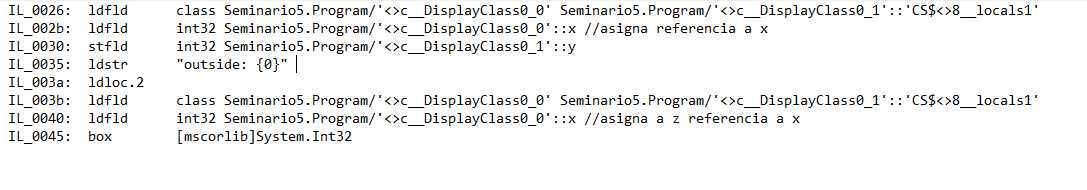
Veamos un ejemplo que si contiene clausura:



¿Qué imprime este código?



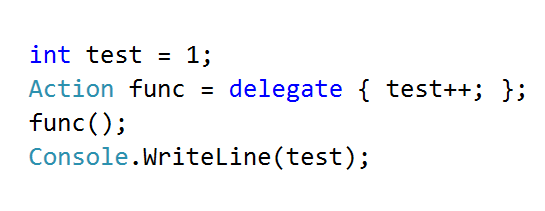
Veamos el código iL:



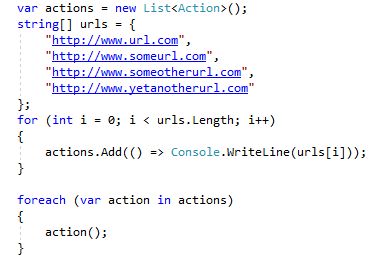
Contrario a lo esperado, el valor de x es el mismo en todas las funciones del array.

Esto se debe a la forma en la que C# implementa la clausura: genera una clase nueva con la expresión lambda y el entero x POR REFERENCIA.

Al ser copiado por referencia, si el valor se modifica en el scope de la expresión lambda el cambio se verá desde afuera. Ejemplo:



Veamos otro ejemplo de un código con clausura:

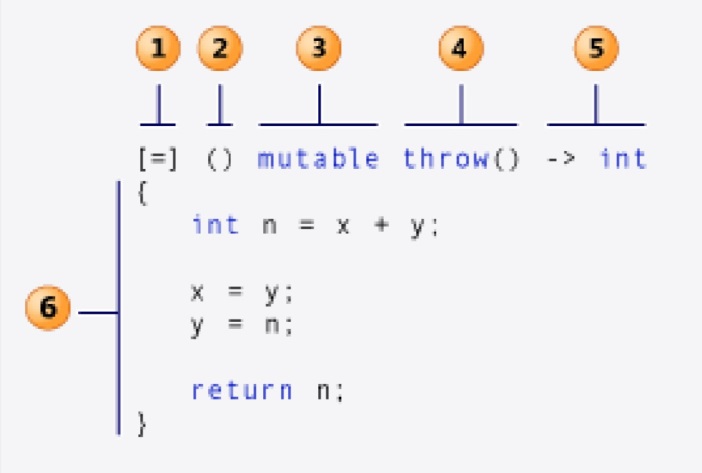


Este ejemplo produce una excepción porque el valor de i cambia en cada iteración hasta llegar a 4 y ese es el valor que se queda en las referencias de la lista.

**C++**

Las expresiones lambdas aparecen en C++11 como una manera conveniente de definir clausuras, en el lugar donde es invocada o pasada como un argumento.

1. Sintaxis;



1. [] ***capture clause***
2. () ***parameter list*** (opcional si no se pone ***mutable***  o ***throw***)
3. mutable ***mutable specification*** (opcional)
4. throw() ***exception specification*** (opcional)
5. -> return\_type ***trailing return*** (opcional)
6. {}; ***Body*** 
   1. Capture Clause:

La capture clause se puede decir que tiene dos formas: explícita e implícita.

Explícita: [<variables que se quieren capturar separadas por ,>]

Donde la variable se captura por defecto por copia a no ser que se especifique lo contrario poniendo un & delante del nombre de la variable. A partir de C++14 en la capture clause también se puede introducir variables al ámbito de la expresión lambda con el tipo de estas siendo inferido por el compilador, estas variables, a diferencia de los parámetros, son introducidas en el momento que se crea la expresión, no al momento en que se llama.

Ejemplos:

auto f = [] () {}; // No captura ninguna variable

int a = 10;

test b(10,20);

test c(30,40);

auto f2 = [a,b,&c](){return a + b.a1 + c.a2;}; /\* Captura a por valor, b por valor (llama al constructor por copia de test), c por referencia \*/

Implícita: [< & ó = >, <variables a capturar de forma opuesta al modo elegido>]

En el primer parámetro se pone & si se quiere que todas las variables sean capturadas por referencia, excepto las que están explicitas en la capture clause. Si se pone = entonces todas las variables son capturadas por valor excepto las que están explicitas en la capture clause. Una vez que se fije el tipo por defecto, no se pueden poner capturas del mismo tipo en la capture clause.

Ejemplos:

int a = 10;

test b(10,20);

test c(30,40);

auto f2 = [=,&c](){return a + b.a1 + c.a2;}; /\* Captura a por valor, b por valor (llama al constructor por copia de test), c por referencia \*/

auto f3 = [&,a,b](){return a + b.a1 + c.a2;}; /\* Captura a por valor, b por valor (llama al constructor por copia de test), c por referencia\*/

// la clausura de f2 y de f3 es la misma

auto f4 = [&,&a](){return a + b.a1 + c.a2;}; /\* Error de compilación 'explicit capture matches default' \*/

auto f5 = [=,a](){return a + b.a1 + c.a2;}; /\* Error de compilación 'explicit capture matches default' \*/

* 1. Parameter list:

La parameter list es la misma sintaxis y significado de los parámetros de una función

* 1. Mutable specification:

Si mutable está presente en la declaración de la expresión lambda significa que los llamados a la expresión lambda puede cambiar el valor de las variables capturadas por copia en su ámbito.

Ejemplo:

int a1 = 0;

auto func = [a1]()mutable {a1++; return a1; /\* a1 por valor pero mutable \*/};

cout <<func(); cout <<func() << a1 << endl;

El código anterior imprime: 1,2,0. Ya que como es mutable la expresión, los cambios al a1 de esta persisten a los llamados y como se capturó por valor estos cambios no afectan al a1 de afuera.

* 1. Throw specification:

Si throw() está presente en la declaración de la expresión lambda significa que esta no lanza excepciones.

* 1. Trailing return:

Le dice al compilador el tipo de retorno de la función, el tipo es inferido en caso de que se omita si no hay return, se infiere void.

* 1. Body:

El cuerpo de la expresión lambda.

1. ¿Cómo es representado una expresión lambda en C++?

Cada expresión lambda genera una clase en la cual se define el operador ***()***, para que esta sea llamada normalmente como una función.

Las variables capturadas por valor, se copian y se guardan en la clase como miembros de esta.

Las variables capturadas por referencias, son guardadas directamente en la clase, sin hacer ninguna copia.

Si la expresión lambda es marcada como ***mutable*** entonces las variables de la clase capturadas por copias se pueden modificar en el cuerpo de la expresión lambda. En otras palabras se le quita el modificador ***const*** al operador ***().***

La clausura no extiende el periodo de vida de las variables capturadas por referencia.

1. Ejercicio:

3) Explique y diga qué imprime el siguiente código en C++, el cual reﬂeja algunas características de las expresiones lambda:

1 auto funcs = vector<function<int()>>();  
2 int x = 1;

3 funcs.push\_back([=] { return x; });

4 x = 2;

5 funcs.push\_back([&] { return x; });

6 x++;

7 funcs.push\_back([x = 4] { return x; });

8 for (auto f : funcs)

{

9 int y = f();

10 cout << y << endl;

}

En las líneas 2-3, como en la ***capture clause*** se tiene ***=*** significa que los valores de la clausura se toman todos por valor, por lo tanto se trabaja a ***x*** por valor dentro del cuerpo de la expresión, sin que ambas se afecten mutuamente.

En las líneas 4-6, como en la ***capture clause*** se tiene ***&*** significa que los valores de la clausura se toman todos por referencia, por lo tanto se trabaja a ***x*** por referencia en el cuerpo de la expresión, afectándose ambas mutamente. Por lo tanto la línea 6 afectará el resultado de la expresión lambda.

En la línea 7, a partir de C++14 ***capture closure*** puede introducir variables en el ámbito de la expresión deduciendo el tipo de estas por el compilador. Por lo que esta ***x*** no tiene que ver con la definida afuera.

Las líneas 8-10 se utilizan para el llamado de las expresiones lambdas e imprimir el resutado de estas en la consola.

Resultados:

La primera expresión lambda imprime: 1, ya que esta captura por valor la variable x.

La segunda expresión lambda imprime: 3, ya que esta captura por referencia a la variable ***x*** y se le hace a esta un incremento en la línea 6 (y más nada en el resto del código).

La tercera expresión lambda imprime: 4, ya que en esta se define ***x*** como 4 dentro del ámbito de la expresión lambda.

3.a) ¿Cómo se pudiera acceder a los miembros de una clase desde el ámbito de una expresión lambda?

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. 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> o o this -> <member>*** dentro del ***body***. Otra forma sería encapsular a this en una variable en el ***capture clause***. Si se quiere pasar por valor el objeto se puede pasar \*this. Veamos ahora un ejemplo.

1 class test

2 {

3 public:

4 int a1; int a2;

5 test(int v\_a1, int v\_a2):a1(v\_a1), a2(v\_a2) {};

6 void print(); // Función de prueba

7 };

1 int main()

2 {

3 test c(1,2); // c.a1 = 1 c.a2 = 2

4 c.print();

5 return 0;

6 }

Ahora veremos diferentes implementaciones de print en la cual se podrá observar las diferentes vías en las cuales se pueden acceder a los miembros de una clase en la expresión lambda.

Capturando por referencia la instancia de la clase:

6 void print()

7 {

8 auto f = [this] (int a1)

9 {

10 cout << test::a1 << ','; // a1 de la clase

11 cout << a2 << ','; // a2 de la clase

12 cout << a1 << endl; // a1 de los parámetros

13 };

14 f(3); // a1 de la expresión lambda = 3

15 a1 += 10; a2 += 10;

16 f(3);

17 }

6 void print()

7 {

8 auto f = [x=this] (int a1)

9 {

10 cout << x->a1 << ','; // a1 de la clase

11 cout << x->a2 << ','; // a2 de la clase

12 cout << a1 << endl; // a1 de los parámetros

13 };

14 f(3); // a1 de la expresión lambda = 3

15 a1 += 10; a2 += 10;

16 f(3);

17 }

Ambas implementaciones de print imprimen: 1,2,3 y 11,12,3

Capturando por valor la instancia de la clase:

6 void print()

7 {

8 auto f = [\*this] (int a1)

9 {

10 cout << test::a1 << ','; // a1 de la clase

11 cout << a2 << ','; // a2 de la clase

12 cout << a1 << endl; // a1 de los parámetros

13 };

14 f(3); // a1 de la expresión lambda = 3

15 a1 += 10; a2 += 10;

16 f(3);

17 }

6 void print()

7 {

8 auto f = [x=\*this] (int a1)

9 {

10 cout << x.a1 << ','; // a1 de la clase

11 cout << x.a2 << ','; // a2 de la clase

12 cout << a1 << endl; // a1 de los parámetros

13 };

14 f(3); // a1 de la expresión lambda = 3

15 a1 += 10; a2 += 10;

16 f(3);

17 }

Ambas implementaciones de print imprimen: 1,2,3 y 1, 2,3

Java

4. ¿Existen delegados en Java? ¿Existen las expresiones lambda? ¿Qué es el Strategy Pattern? ¿Cómo se pudiera lograr en caso de no existir?

Realmente no existen delegados en Java tal como a los que estamos adaptados en C#, pero podemos ser capaces de lograr el mismo efecto usando *Reflection* para obtener objetos de tipo *Method* a los cuales podemos invocar y utilizar.

Otra variante sería crear una interfaz de un solo método (interfaz funcional) y luego instanciar dicha interfaz para llamar al método requerido utilizando Strategy Pattern.

Un ejemplo de implementación en este último caso sería utilizando clases internas anónimas, lo cual será visto más adelante.

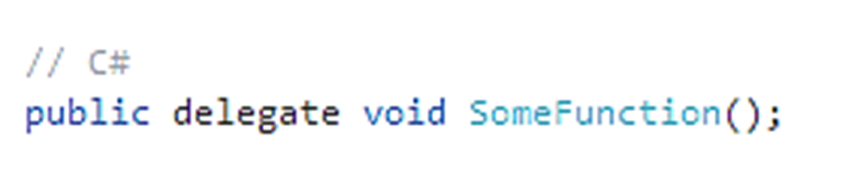
Veremos también que a partir de su versión 8 Java incluye soporte para el uso de expresiones lambdas.

Pero primero hablemos de en qué consiste el Strategy Pattern en Java.

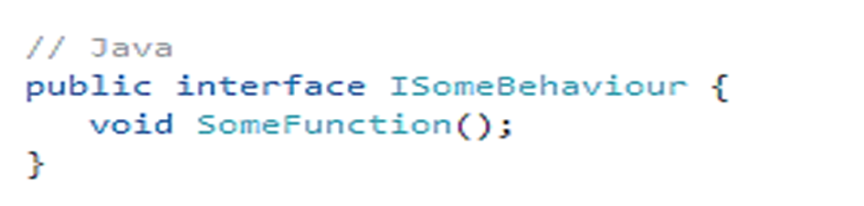
“STRATEGY PATTERN”

* **Strategy** es un patrón de diseño que convierte un conjunto de comportamientos en objetos y los hace intercambiables dentro de un objeto original llamado contexto
* El contexto guarda una referencia a un strategy object y lo delega ejecutando el comportamiento. Con tal de cambiar la manera en que el contexto hace su trabajo, otros objetos pueden reemplazar el, actualmente enlazado, strategy object.
* Es muy común en código Java. Usualmente es usado en varios frameworks para proveerle al usuario una forma de cambiar el comportamiento de una clase sin extenderla
* Java 8 trajo consigo soporte para expresiones lambdas las cuales sirven de una alternativa más simple y elegante para este patrón

Este sería el código C# a simular

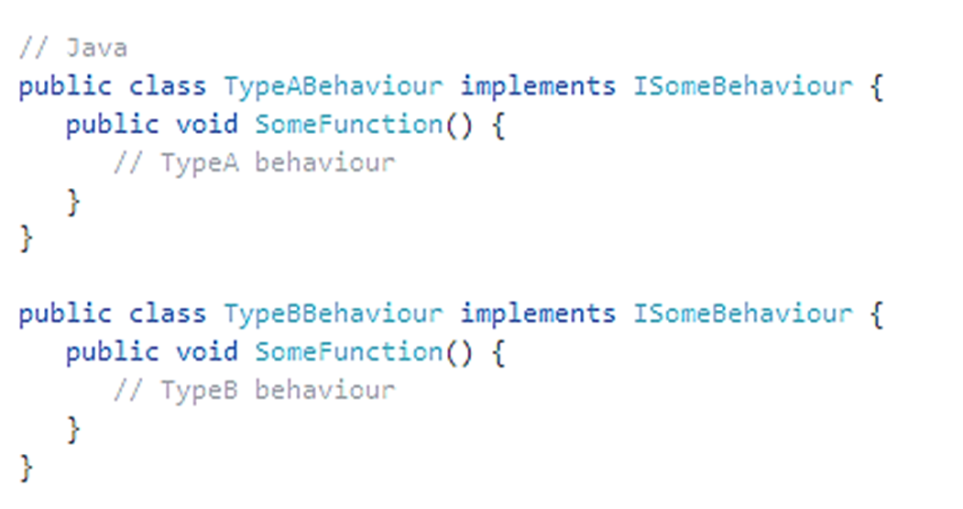
**

A partir de Java 8 se usa la anotación *@FunctionalInterface*

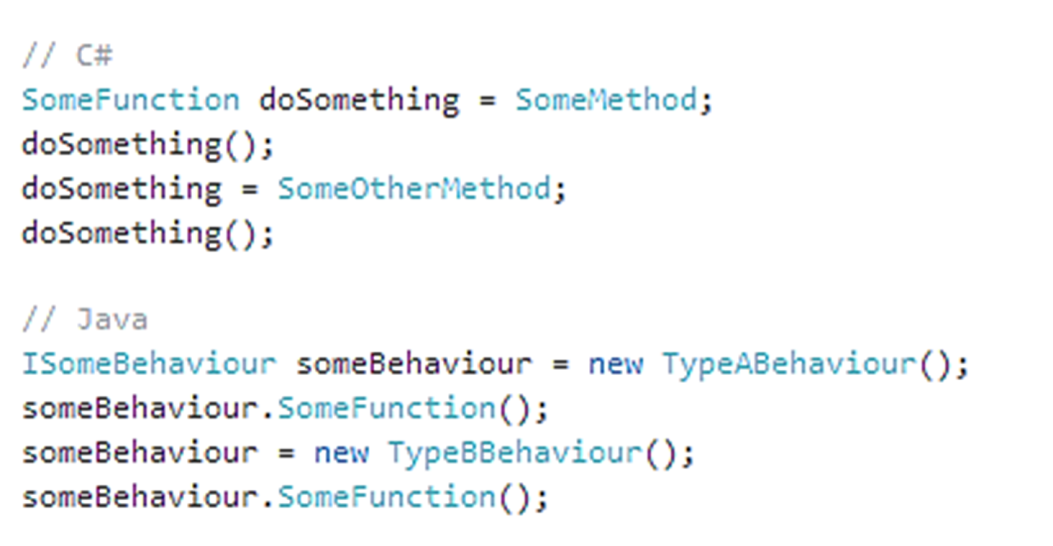


Meramente informativa para reforzar la intención de dichas interfaces y permite a los compiladores generar una excepción cuando se aplica sobre una interfaz que no cumpla con la definición

***Interfaz funcional: Interfaz que contiene un único método abstracto(Puede poseer, o no, métodos estáticos implementados por defecto)***

Para implementaciones concretas del método, definimos una clase que implemente el comportamiento deseado.

Entonces, donde sea que tengamos un delegado ***SomeFunction*** en el código C# lo sustituimos por su correspondiente método implementado en una de nuestras clases en Java



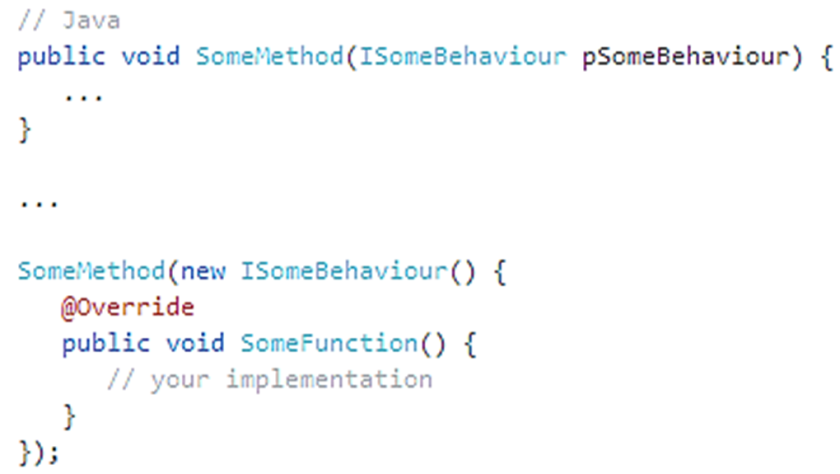
Esta implementación se ve bastante engorrosa, además de tener que estar declarando distintas clases por cada implementación del método ***SomeFunction*** de la Interfaz Funcional, por lo que uno pensaría que se puede esperar algo mejor…

ANONYMOUS INNER CLASSES

En efecto algo mejor es posible de hacer para simular el comportamiento de los delegados de C#

Las clases anónimas internas como su nombre lo indican son clases internas sin nombres y para las cuales un único objeto es creado y se pueden crear en clases (abstractas o concretas) o en interfaces

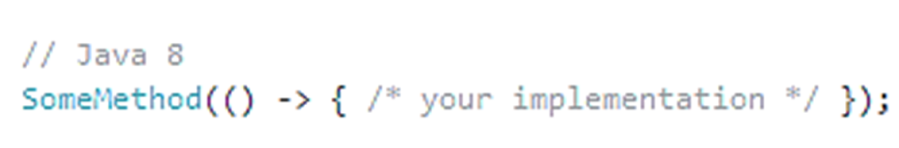
Veamos cómo usarlas para mejorar el código anterior:



Ahora podemos evitar declarar instancias de clases por separado y casi que simulamos un verdadero comportamiento de un delegado en C#... Pero aún se ve poco estilizado…

Probablemente su uso esté solo limitado para cuando la implementación es muy específica en cuanto al contexto en que se usa y sería poco provechoso reusarla.

¿JAVA 8 AL RESCATE?…LAMBDA EXPRESSIONS



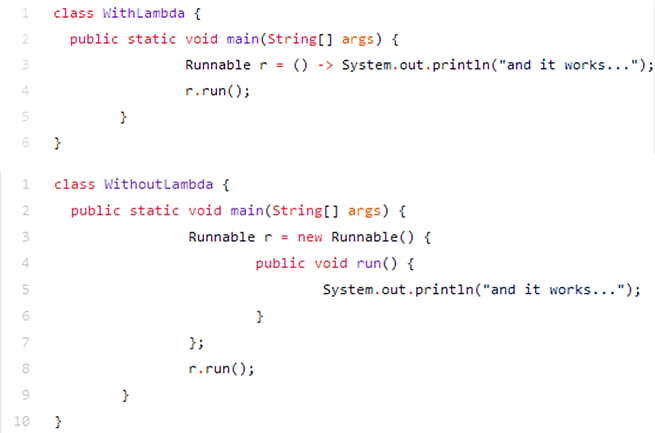
Luego por supuesto no podía faltar la mejora de la que hablábamos pues Java 8 agrega las expresiones lambdas a su repertorio y…

Aunque uno pensaría que esto simplemente lo lograron a modo de azúcar sintáctica sustituyendo la expresión lambda en tiempo de compilación por una clase interna anónima pues este no es el caso… Java hace algo un poco más complejo que eso…

Las expresiones lambda son funciones anónimas, es decir, funciones que no necesitan una clase.  
su sintaxis básica se detalla a continuación:

1. El operador lambda (->) separa la declaración de parámetros de la declaración del cuerpo de la función.
2. Parámetros:
   * Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
   * Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.
3. Cuerpo de lambda:
   * Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la clausula return en el caso de que deban devolver valores.
   * Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la cláusula return en el caso de que la función deba devolver un valor.

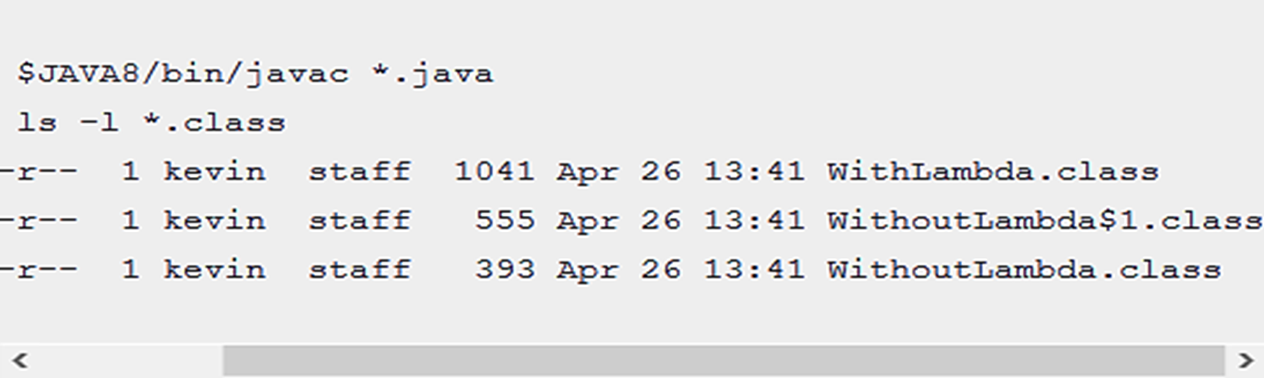
Veamos un ejemplo con estas dos clases de archivos distintos



Como se puede apreciar, en una simplemente usamos las expresiones lambdas y el la otra, una clase interna anónima.

Se esperaría que ambas fueran interpretadas por el compilador de la misma manera…pero no es así.

Resulta que un código hecho en Java 8 es incompatible con versiones anteriores de Java, si utiliza algo relacionado con expresiones lambdas, ya que en Java 8 se añade un nuevo método a la plataforma llamado ***lambdas metafactory,***  el cual dinámicamente crea una instancia del tipo correcto con el cuerpo de un método abstracto el cual consiste en un llamado al método de la expresión lambda

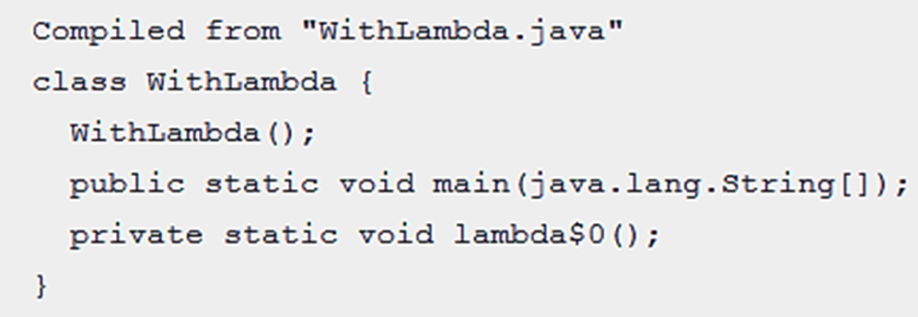


Aquí vemos el compilado de ambas y como se esperaba la que utiliza clase interna compiló como se esperaba, en dos clases, pero la versión lambda compiló en una sola clase… ¿a donde fue la instancia de Runnable?

Pues para asombro de munchos dicha instancia ni siquiera existe… al menos no a nivel de bytecode

Por eso ahora explicaremos que hace el compilador con estas “Lambdas Expressions”

Primero el cuerpo de la expresión lambda es convertido en un método privado interno, como se puede ver en este classfile:



Luego, en lugar de crear una instancia de la clase interna (la cual nunca se crea), se llama a la ***lambda metafactory***. Este llamado en particular a la ***lambda metafactory*** es una forma especial denominada ***lambda factory*** y utiliza la instrucción “invokedynamic” para hacer posible la especialización de la ***lambda metafactory***.

Específicamente, hay muchos casos en los cuales la JVM (Java Virtual Machine) no tendrá necesidad de generar una nueva clase implementando la interfaz, sino que resolverá, de manera más sencilla, retornando una estructura interna más simple.

Clausura en Java:

* Las expresiones lambdas( como también las clases anónimas internas ) en Java solo pueden acceder a variables finales
* Java solo salva el valor de las variables libres y las deja usar dentro de la expresión lambda por lo que si ha sido modifica el compilador para evitar escenarios incoherentes solo guarda el valor con el que fue instanciada dichas variables

1. void fn()
2. {
3. int var = 42;
4. Supplier<Integer> lambdaFun= ()-> var;
5. var++;
6. System.out.println(lambdaFun.get());
7. }

En este caso var no es una variable final ya que su valor ha ido modificado por tanto este código da error de compilación