

Paradigmas de la Programación

Laboratorio 4

Ejercicio 1:

Para estos test de tipamiento estático y tipamiento dinámico nos basamos en el libro "Conceptos, Técnicas y modelo de programación" donde enuncia la siguiente definición:

Tipamiento estático: todos los tipos de las variables se conocen en tiempo de compilación.

Tipamiento dinámico: el tipo de una variable se conoce solo en el momento en que la variable se liga, es decir a lo largo de la ejecución del programa.

Siguiendo este criterio llegamos a la siguiente conclusión:

➤ *Lenguaje Python:*

En este test vamos a probar que Python es un lenguaje de programación de tipamiento dinámico.

Para ello vamos a ver que cuando Python compila no verifica los tipos de las variables en este tiempo de compilación, es decir, no nos tira error de tipo.

```
1 def dinamycTest():
2     st = "hola"
3     # Ahora trataremos de concatenar la variable st con otra variable p en el
4     # cual no tiene tipo.
5     concatenar = st + p
6     return concatenar
7
```

Para compilarlo usamos:

\$python -m compileall <nombre_de_archivo>

```
L> $ python -m compileall python_test.py
Compiling python_test.py ...
```

Ahora usamos un interprete como ipython para ejecutar la función.

```
>>> import python_test
>>> python_test.dinamycTest()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    File "python_test.py", line 5, in dinamycTest
      concatenar = st + p
NameError: global name 'p' is not defined
>>>
```

Al ejecutar esta función, Python ira linea a linea conociendo los tipos de las variables en tiempo de ejecución. Al llegar a la linea “concatenar = st + p”, recién ve el error de que la variable 'p' no tiene tipo, y nos lanza error refiriéndose a dicha variable ('p').

Con esto podemos concluir que Python es un lenguaje de tipamiento dinámico.

➤ *Lenguaje Haskell:*

En este test vamos a probar que el lenguaje Haskell es de tipamiento estático. Para ello vamos a ver como Haskell verifica en tiempo de compilación los tipos de las variables.

```
1 x :: Int
2 x = 0
3
4 y :: Char
5 y = x
6
7 z :: Int
8 z = y
9
10 main = print z
```

Al tratar de compilar este código nos muestra lo siguiente:

```
Compilation error time: 0 memory: 4712 signal:-1
[1 of 1] Compiling Main          ( prog.hs, prog.o )

prog.hs:5:5:
  Couldn't match expected type `Char' with actual type `Int'
  In the expression: x
  In an equation for `y': y = x
```

Al analizar este error, vemos que Haskell además de verificar que todas las variables tienen tipo, verifica que sean compatibles el resultado de las funciones.

Con este test llegamos a la conclusión de que Haskell es de tipamiento estático.

➤ *Lenguaje Java:*

En este test vamos a ver que Java es un lenguaje de programación de tipamiento estático.

Para ello vamos a ver que el siguiente código Java no compila, porque verifica que los métodos tengan su tipo.

```

1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5 /* Name of the class has to be "Main" only if the class is public. */
6 class Ideone
7 {
8     public static void main (String[] args) throws java.lang.Exception
9     {
10         otro();
11     }
12     otro() {
13         ;
14     }
15

```

Al compilar nos muestra lo siguiente:

```

input  Output
Compilation error time: 0.1 memory: 320256 signal:0
Main.java:14: error: invalid method declaration; return type required
    otro() {
        ^
1 error

```

Como se ve, Java requiere que los métodos declarados tengan su tipo. Con esto concluimos de que Java es de tipamiento estático.

➤ *Lenguaje Scala:*

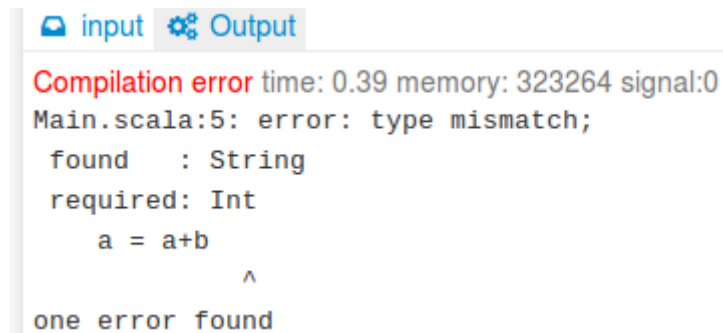
En este test vamos a ver que Scala es de tipamiento estático. Para ello escribimos este código:

```

1 object Main extends App {
2     def algo() : Int = {
3         var a:Int = 0
4         var b:String = "33"
5         a = a + b
6         return a
7     }
8

```

Al compilar este código nos muestra lo siguiente:




```
input Output
Compilation error time: 0.39 memory: 323264 signal:0
Main.scala:5: error: type mismatch;
  found   : String
  required: Int
    a = a+b
           ^
one error found
```

Aquí podemos ver que además de verificar los tipos de las variables, también verifica que sean compatibles los tipos para el operador “+”.
Y con esto podemos concluir que Scala es de tipamiento estático.

➤ Lenguaje C:

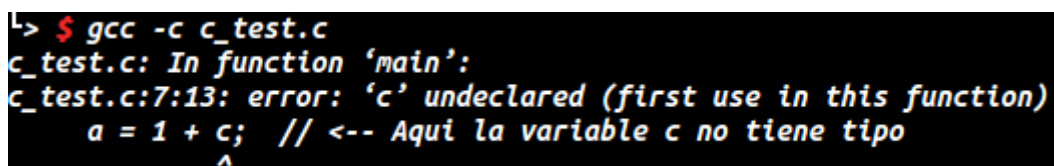
En este test vamos a ver que C es un lenguaje de programación de tipamiento estático.

Para ello vamos a ver como el siguiente código C no compila, porque al verificar los tipos de las variables, vera que hay una sin tipo y nos lanzara un error de tipo.



```
1 #include <stdio.h>
2 #include <stdbool.h>
3
4 int main(void) {
5     int a = -23;
6     char b[5];
7     a = 1 + c; // <-- Aqui la variable c no tiene tipo
8     return 0;
9 }
```

Al compilar este código nos muestra lo siguiente:



```
L> $ gcc -c c_test.c
c_test.c: In function 'main':
c_test.c:7:13: error: 'c' undeclared (first use in this function)
  a = 1 + c; // <-- Aqui la variable c no tiene tipo
             ^
```

Con esto concluimos que C es de tipamiento estático.

➤ Lenguaje JavaScript:

En este test vamos a ver que Ruby es de tipamiento dinámico.
Para ello escribimos este código:

```

1 <script>
2   var a = 'Hola';
3   var b;
4   var c = a + b;
5   document.write(c);
6 </script>

```

Lo cual al ejecutar el script no tira error de tipado, aun peor, no verifica los tipos del operador “+”. El resultado es el siguiente:



Este código lo corrimos de la pagina js.do. Con esto podemos concluir de que JavaScript es de tipamiento dinámico.

➤ *Lenguaje Ruby:*

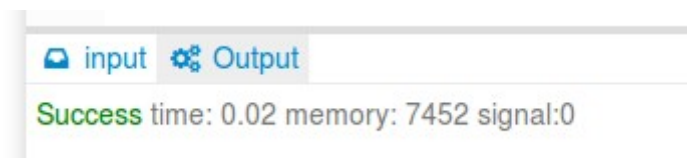
En este test vamos a ver que Ruby es de tipamiento dinámico. Para ello escribimos este código:

```

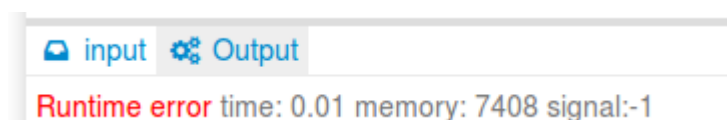
1 def algo()
2   a = 3
3   b = a + c
4   return b
5 end

```

Aquí se puede ver que la variable 'c' no esta ni definida, pero al “compilar” Ruby no nos advierte de este error.



Ahora si lo ejecutamos (algo()) vemos como nos tira error.



Con esto podemos concluir de que Ruby es de tipamiento dinámico.

Ejercicio 2:

Vamos a ver que tipo de tipado tienen los lenguajes.

Para ello nos basamos en el siguiente criterio:

Dado la declaración:

$a = b + c$

Vamos a decir que un lenguaje es de tipado fuerte, si al verificar el tipo de variables en la operación (+), y si ve que no son del mismo tipo, entonces salta con un error de tipo.

Vamos a decir que el lenguaje es de tipado débil, si al verificar el tipo de las variable en la operación (+), y ve que no son del mismo tipo, el lenguaje toma la decisión de castear a uno de ellos implícitamente y hacer que se lleve a cabo la operación.

➤ *Lenguaje Python:*

En este test, vamos a probar que Python es fuertemente tipado con respeto al operador +.

Para ello escribimos este código:

```
1 def algo():
2     a = 2
3     b = "hola"
4     c = a + b
5     return c
6
7 algo()
```

Al importar y ejecutar este código en ipython tenemos el siguiente error:

```
>>> import python_test
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    python_test.algo()
  File "python_test.py", line 4, in algo
    c = a + b
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Lo cual concluimos que Python es tipado fuerte con respeto al operador +.

➤ *Lenguaje Haskell:*

Vamos a ver que Haskell es un lenguaje de tipado fuerte.

Para ello nos basamos en el siguiente código:

```

1 import IO
2
3 f :: Double -> Double
4 f a = a + a
5
6 c :: Int
7 c = 3
8
9 b = f c
10
11 main = print b

```

Al ejecutar este código nos muestra:

```

input Output
Compilation error time: 0 memory: 0 signal:0
[1 of 1] Compiling Main          ( prog.hs, prog.o )

prog.hs:7:7:
    Couldn't match expected type `Double' with actual type `Int'
    In the first argument of `f', namely `c'
    In the expression: f c
    In an equation for `b': b = f c

```

Aquí vemos como se queja del tipo por lo tanto concluimos que Haskell es de tipado fuerte con respeto al operador +.

➤ *Lenguaje Java:*

Vamos a ver que Java es un lenguaje de tipado fuerte.
Para ello escribimos el siguiente código:

```

1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5 class Ideone
6 {
7     public static void main (String[] args) throws java.lang.Exception
8     {
9         double a = 3;
10        sumar(a);
11    }
12    public static int sumar(int a) {
13        return a+a;
14    }
15

```

Al compilar y ejecutar este código, nos muestra lo siguiente:

Compilation error time: 0 memory: 0 signal:0

Main.java:13: error: incompatible types: possible lossy conversion from double to int

```
int b = sumar(a);
      ^
```

Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
1 error

Como podemos ver, Java exige que la variable 'a' sea el mismo tipo del parámetro de la función. Esto lo hace en tiempo de compilación, porque Java verifica tanto los tipos de las variables, y que los tipos de las funciones sean compatibles. Por ello concluimos que Java es de tipado fuerte con respecto al operador +.

➤ *Lenguaje Scala:*

Vamos a ver que Scala es un lenguaje de tipado débil con respecto al operador +.

Para ello escribimos este código:

```
1 object Main extends App {
2   var a: Int = 3
3   var b: String = "Hola"
4   var c = a+b
5   print(c)
6 }
```

Y al compilarlo y ejecutarlo nos muestra lo siguiente:

```
input Output
Success time: 0.37 memory: 322240 signal:0
3Hola
```

Aquí vemos que Scala sigue el criterio de tipamiento débil. Por lo tanto concluimos que Scala es de tipado débil con respecto al operador +.

➤ *Lenguaje C:*

Vamos a ver que C es un lenguaje de tipado débil.

Para ello escribimos el siguiente código:

```
1 #include <stdio.h>
2
3 int main(void) {
4   int a = 3;
5   float b = 2.12;
6   int c = a + b;
7   printf ("c = %i\n", c);
8   return 0;
9 }
```


Al compilar y ejecutar este código, nos muestra lo siguiente:



```
input Output
Success time: 0 memory: 2008 signal:0
c = 5
```

Aquí podemos ver que C sumo (+) dos tipos diferentes, por lo cual sigue el criterio de tipado débil. Por lo tanto concluimos de que C es de tipado débil con respecto al operador +.

➤ *Lenguaje JavaScript:*

Vamos a ver que JavaScript es de tipado débil.
Para ello escribimos el siguiente código:

```
1 <script>
2   var a = 3;
3   var b = "hola";
4   var c = a + b;
5   document.write(c);
6 </script>
```

Al ejecutar este script nos muestra lo siguiente:



```
JavaScript 3hola Result
Load code samples
```

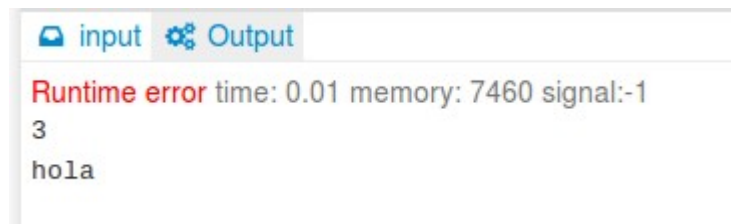
Al ver el resultado, vemos que JavaScript al tratar de sumar dos tipos diferentes lo tolera, por lo tanto cumple con el criterio de tipado débil. Concluimos que JavaScript es de tipado débil con respecto al operador +.

➤ *Lenguaje Ruby:*

Vamos a ver que Ruby es de tipado fuerte.
Para ello escribimos el siguiente código:

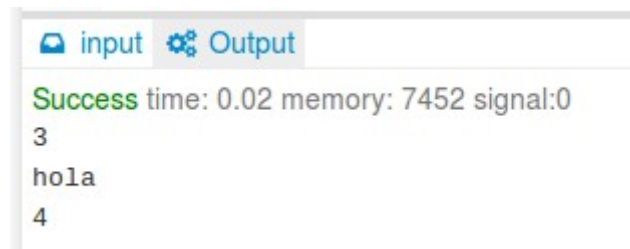
```
1 def algo()
2   a = 3
3   b = "hola"
4   puts a, b
5   c = a+b
6   puts c
7 end
8
9 algo()
```

Al ejecutar este código nos muestra lo siguiente:



A screenshot of a Ruby REPL window. The window has two tabs: 'input' and 'Output'. The 'Output' tab is active, showing a red 'Runtime error' message: 'time: 0.01 memory: 7460 signal:-1'. Below the error message, the output of the previous line is shown: '3' and 'hola'.

Aquí podemos ver que ejecuta línea a línea y cuando llega a “ $c = a+b$ ” devuelve un error. Con esto podemos ver que Ruby no suma dos tipos diferentes. Para asegurarnos, ya que no nos da mucha información del error, cambiamos reemplazamos la variable 'b' por 1, en “ $c=a+b$ ”, es decir, “ $c = a+1$ ”. Al ejecutarlo nuevamente obtuvimos:



A screenshot of a Ruby REPL window. The window has two tabs: 'input' and 'Output'. The 'Output' tab is active, showing a green 'Success' message: 'time: 0.02 memory: 7452 signal:0'. Below the success message, the output of the previous line is shown: '3', 'hola', and '4'.

Lo cual podemos concluir que Ruby es de tipado fuerte con respecto al operador +.

Ejercicio 3:

Para este ejercicio vamos a proponer el siguiente criterio para la asignación única.

Dado la siguiente sentencia:

```
int a = 3;
```

Se dice que un lenguaje de programación es de asignación única si la variable “a” permanece con el mismo valor a lo largo de toda la ejecución, dentro de su alcance; si la variable ya esta ligada, y los valores no son compatibles, señalara un error.

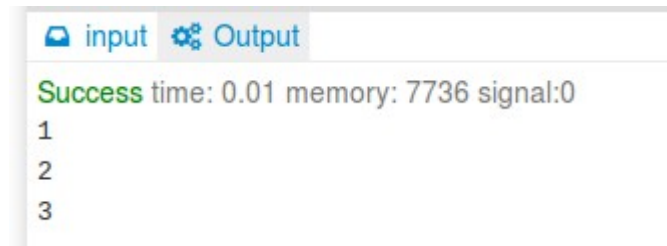
➤ *Lenguaje Python:*

Vamos a ver que tipo de asignación tiene este lenguaje.

Para ello escribimos el siguiente código:

```
1 x , y = 1 , 3
2 print x
3 x = 2
4 print x
5 print y
```

Al ejecutarlo nos muestra lo siguiente:



The screenshot shows a window with two tabs: 'input' and 'Output'. The 'Output' tab is active, displaying the following text: 'Success time: 0.01 memory: 7736 signal:0' followed by the numbers '1', '2', and '3' on separate lines, which are the outputs of the 'print' statements in the code.

Como se puede ver, al principio 'x' vale 1 y luego vale 2. Con esto concluimos de que Python no es de asignación única.

➤ *Lenguaje Haskell:*

Vamos a ver si Haskell es de asignación única.

Para ello escribimos el siguiente código:

```
1 x :: Int
2 x = 0
3
4 x = 2
5
6 main = print x
```

Al compilarlo nos muestra lo siguiente:

```
input Output
Compilation error time: 0 memory: 0 signal:0
[1 of 1] Compiling Main                ( prog.hs, prog.o )

prog.hs:4:1:
  Multiple declarations of `x'
  Declared at: prog.hs:2:1
               prog.hs:4:1
```

Como podemos ver, no llega a ejecutarlo ya que no pudo compilar porque para Haskell estamos tratando de declarar dos veces la misma variable. Esto nos dice que “x” mantiene su valor a lo largo de todo su alcance. Concluimos que Haskell es de asignación única.

➤ *Lenguaje Java:*

Vamos a ver si Java es de asignación única. Para ello escribimos el siguiente código:

```
1 public class java_test_3 {
2     public static void main (String[] args){
3         int x = 1;
4         x = 2;
5         System.out.println(x);
6     }
7 }
```

Al compilar y ejecutarlo nos muestra lo siguiente:

```
input Output
Success time: 0.1 memory: 320320 signal:0
2
```

Podemos ver como el valor de “x” cambio a lo largo de la ejecución. Con esto concluimos de que Java no es de asignación única.

➤ *Lenguaje Scala:*

Vamos a ver si Scala es de asignación única. Para ello escribimos el siguiente código:

```

1 object scala_test_3 {
2   var x = 1
3   x = 2
4   println(x)
5 }

```

Al compilar y ejecutar nos muestra lo siguiente:

```

input Output
Success time: 0.39 memory: 322240 signal:0
2

```

Vemos como la variable “x” se modifica en la ejecución. Concluimos que Scala no es de asignación única.

➤ *Lenguaje C:*

Vamos a ver si C es de asignación única.
Para ello escribimos el siguiente código:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5   int a = 1;
6   a = 2;
7   printf("a es: %i.\n", a);
8   return 0;
9 }

```

Al compilar y ejecutar nos muestra:

```

input Output
Success time: 0 memory: 2008 signal:0
a es: 2.

```

Podemos ver que la variable “x” no mantiene su valor a lo largo de la ejecución. Concluimos que C no es de asignación única.

➤ *Lenguaje JavaScript:*

Vamos a ver si JavaScript es de asignación única.
Para ello escribimos el siguiente código:

```

1 <script type="text/javascript">
2   var x = 1;
3   x = 2;
4   document.write("x: ", x, "<br>");
5 </script>

```

Al ejecutar este script nos muestra:



Podemos ver que la variable “x” no mantiene su valor a lo largo de la ejecución. Concluimos que JavaScript no es de asignación única.

➤ *Lenguaje Ruby:*

Vamos a ver si Ruby es de asignación única. Para ello escribimos el siguiente código:

```

1 x , y = 1 , 3
2 x = 2
3 print x , "\n"
4 print y , "\n"

```

Al compilar y ejecutar nos muestra lo siguiente:



Podemos ver que la variable “x” cambia a lo largo de la ejecución. Concluimos que Ruby no es de asignación única.

- El mecanismo para poder obtener un comportamiento de asignación única, solo se puede dar en algunos lenguajes: Scala, Java, JavaScript, Ruby y C.

En C le anteponemos “const” a la variable, lo cual deja sin efecto cualquier otra asignación de valor a la misma, produciendo un error de compilación.


```

1 #include <stdio.h>
2 #include <stdbool.h>
3
4 int main(void) {
5   const int a = 1;
6   a = 2;
7   printf("a es: %i.\n", a);
8   return 0;
9 }

```

Al compilar y ejecutar nos muestra lo siguiente:

Error de compilación

 stdin

Standard input is empty

información de compilación

prog.c: In function 'main':

prog.c:6:7: error: assignment of read-only variable 'a'

a = 2;


^

En Java agregamos antes de la declaración de la variable “final”, lo cual también establece que no se pueda modificar el valor de la misma, produciendo un error en la compilación.

```
1 class java_test_3_asig_force {  
2     public static void main (String[] args){  
3         final int x = 1;  
4         x = 2;  
5         System.out.println(x);  
6     }  
7 }
```

Al compilar y ejecutar nos muestra lo siguiente:

Error de compilación

 stdin

Standard input is empty


información de compilación

Main.java:4: error: cannot assign a value to final variable x

x = 2;

^

1 error

 stdout

Standard output is empty

En Scala le antepone al nombre de la variable un “CONSTANT_<nombre-variable>”, lo cual produce el efecto de no modificación del valor ligado, produciendo también un error en la compilación.

```

1 object scala_test_3_asig_force {
2   val CONSTANT_x = 1
3   CONSTANT_x = 2
4   println(CONSTANT_x)
5 }

```

Al compilar y ejecutar nos muestra lo siguiente:

```

Error de compilación

stdin
Standard input is empty

información de compilación
Main.scala:3: error: reassignment to val
    CONSTANT_x = 2
              ^

```

En JavaScript se declara una variable “const”, la cual no modificara su valor y en la ejecución del código generara un error.

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5 </head>
6 <body>
7 <script type="text/javascript">
8   const x = 1;
9   x = 2;
10  document.write("x: ", x, "<br>");
11 </script>
12 </html>

```

Al compilar y ejecutar nos muestra lo siguiente:

```

Inspe... Con... Depura... Editor de e... Rendimie... ...
Red CSS JS Seguridad Registro Limpiar Salida del filtro
SyntaxError: invalid assignment to const x javascript_... :9:4

```


En Ruby se define a la variable como “SOME_CONSTANT”, lo cual produce que se le ligue un valor, produciendo un error en tiempo de ejecución, cuando se trata de modificarla.

```
1 def asig_force()  
2   SOME_CONSTANT = 1  
3   SOME_CONSTANT = 0  
4 end  
5  
6 asig_force()
```

Al compilar y ejecutar nos muestra lo siguiente:

```
Error en tiempo de ejecución
```

```
stdin  
Standard input is empty
```

```
stdout  
Standard output is empty
```

```
stderr  
prog.rb:2: dynamic constant assignment  
    SOME_CONSTANT = 1  
                  ^  
prog.rb:3: dynamic constant assignment  
    SOME_CONSTANT = 0  
                  ^
```

En Python no hay un mecanismo para obtener un comportamiento similar a la asignación única.

Ejercicio 4:

En todos estos test nos basamos en el libro Concepts in Programming Language de la pagina 177 con el siguiente ejemplo:

```
int x = 1;
function g(z) = x+z;
function f(y) = {
    int x = y+1;
    return g(y*x);
};
f(3);
```

Alcance estático es aquel en el que la variable tiene ocurrencia dentro de la declaración textual mas interna del programa fuente.

Alcance Dinámico es aquel en el que la variable tiene la ocurrencia en la declaración más reciente vista durante la ejecución del programa.

➤ *Lenguaje Python:*

Con el siguiente código vamos a ver que nos devuelve Python luego de compilar.

```
1 x = "estatico"
2
3 def g():
4     print x
5
6 def f():
7     x = "dinamico"
8     g()
9
10 f()
```

Éxito

stdin

Standard input is empty

stdout

estatico

Al ser compilado y ejecutado este código, nos imprimió que Python es un lenguaje con alcance estático. Por lo tanto llegamos a esta conclusión.

➤ *Lenguaje Haskell:*

Con el siguiente código vamos a ver que nos devuelve Haskell luego de compilar.

```
1 import System.IO
2
3 x = 1
4
5 main =
6     do
7         result <- f 3
8         print(result)
9
10
11 g :: Integer -> IO Integer
12 g z =
13     do
14         let result = z + x
15         return result
16
17 f :: Integer -> IO Integer
18 f y =
19     do
20         let x = y + 1
21         g (x * y)
```

Éxito

stdin

Standard input is empty

stdout

13

Luego de compilar y ejecutar este código, Haskell nos imprime que su resultado ha sido 13, que concuerda con el resultado de alcance estático analizado en el ejemplo del libro (toma el valor de “x = 1” en la función “g”).

➤ *Lenguaje Java:*

```
1 class java_test_4 {
2     String x = "estatico";
3     void g(){
4         System.out.println(x);
5     }
6     void f(){
7         String x = "dinamico";
8         g();
9     }
10    public static void main (String[] args){
11        String x = "estatico";
12        java_test_4 a = new java_test_4();
13        a.f();
14    }
15 }
```

Éxito

stdin

Standard input is empty

stdout

estatico

Luego de compilar y ejecutar este código, Java nos imprime que es de alcance estático. Por lo tanto llegamos a esta conclusión.

➤ *Lenguaje C :*

Con el siguiente código vamos a ver que nos devuelve C luego de compilar.

```
1 #include <stdio.h>
2
3 char x[] = "estatico";
4
5 void g(void) {
6     printf("C tiene alcance: %s\n", x);
7 }
8
9 void f(void) {
10    char x[] = "dinamico";
11    g();
12 }
13
14 int main(void) {
15    f();
16    return 0;
17 }
```

```
Éxito

stdin
Standard input is empty

stdout
C tiene alcance: estatico
```

Al ser compilado y ejecutado este código, nos imprimió que C es un lenguaje con alcance estático. Por lo tanto llegamos a esta conclusión.

➤ *Lenguaje Scala :*

Con el siguiente código vamos a ver que nos devuelve Scala.

```
1 object Main extends App {
2     var x = "estatico"
3     def g() = {
4         println(s"Scala es de alcance: $x")
5     }
6
7     def f() = {
8         var x = "dinamico"
9         g()
10    }
11    f()
12 }
```

```
Éxito

stdin
Standard input is empty

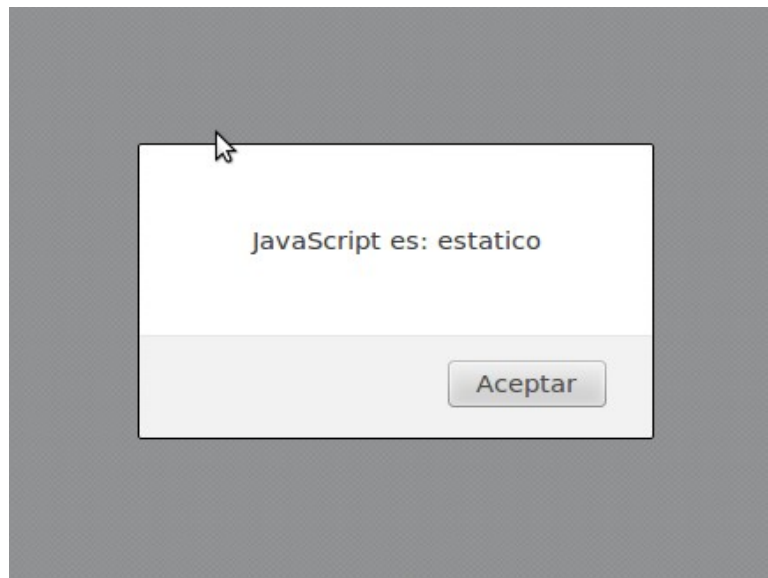
stdout
Scala es de alcance: estatico
```

Luego de compilar y ejecutar este código, Scala nos imprime que es de alcance estático. Por lo tanto llegamos a esta conclusión.

➤ *Lenguaje JavaScript :*

Con el siguiente código vamos a ver que nos devuelve JavaScript .

```
1 <script type="text/javascript">
2   var x = "estatico";
3   function g() {
4       alert("JavaScript es: " + x);
5   }
6   function f() {
7       var x = "dinamico";
8       g();
9   }
10  f();
11 </script>
```



Luego de ejecutar este código JavaScript nos dice que es de alcance estático. Por lo tanto llegamos a esta conclusión.

➤ *Lenguaje Ruby :*

En este lenguaje hay formas de declarar variables globales, entre ellas esta el "\$" y otro de alcance en class "@".

En el código de prueba está lo siguiente:

```
1 $x = "estatico"
2 def g()
3     return print $x , "\n"
4 end
5 def f()
6     $x = "dinamico"
7     return g()
8 end
9 f()
```

Éxito

stdin

Standard input is empty

stdout

dinamico

Lo cual el resultado es: "dinámico", esto tiene sentido ya que vamos modificando la variable global. Por lo tanto en este caso se podría concluir que Ruby tiene alcance dinámico.

Pero hay algunas curiosidades como en el siguiente código:

Ejemplo 1

```
var=2
```

```
def f()
```

```
  var = 6 <-- Modifico supuestamente la "variable" var.
```

```
end
```

```
f()
```

```
print var
```

Esto imprime: 2, y con esto se podría concluir que Ruby es de alcance estático. Pero hemos analizado un poco más e hicimos otro código:

Ejemplo 2

```
var=2
```

```
def g()
```

```
  print var
```

```
end
def f()
  var = 6
  g()
end
f()
```

Aquí al ejecutar f() lanza un error (no sabemos que tipo ya que compilamos y usamos la pagina web ideone.com y solo nos devuelve return -1). El error es producido cuando se ejecuta g(). Al entrar a este procedimiento, trata de imprimir var; para ruby el var=2 externo no tiene alcance en este procedimiento es decir, desconoce esa variable, por eso emite un error. Por lo tanto se puede decir que var=2 no es "global" a todos los procedimientos/funciones. Es por ello que hicimos uso del "\$" para definir las variables globales.

Conclusión de Ruby

Ruby tiene un modo de manejar los alcances de variables, los cuales son: de alcance local, alcance global, alcance de instancia y alcance de clase.

Estos dos últimos no lo probamos pero nos basamos en la pagina:

http://en.wikibooks.org/wiki/Ruby_Programming/Syntax/Operators#Default_scope

Por lo tanto, usando la variable global podemos decir que Ruby tiene alcance dinámico.

Además, según esta pagina (https://github.com/TTPS-ruby/capacitacion-ruby-ttps/blob/master/05-clases-objetos-variables/02_contenedores-bloques-iteradores.md), sobre el uso de lambda, tendrá alcance sobre el valor de una variable en forma estática si lambda esta definido de manera que la variable no sufre modificación en "<variable>.call", pero dinámica si lambda esta definida de forma que esta se modifique ante la llamada "<variable>.call(<valor>)".

Ejercicio 5:

En este ejercicio vamos a ver si los diferentes lenguajes de programación poseen optimización de llamada a la cola. Para eso establecemos un código en cada uno, que calcula el factorial de un numero, y luego tratamos de optimizarlo; si en los resultados la diferencia es significativa (imprimimos stack y llamadas de recursión), posee “tail call optimization”, sino no.

➤ Lenguaje Python:

Este código sin optimizar:

```
1 def recsum(x):
2     if x == 1:
3         return x
4     else:
5         print "print stack"
6         return x + recsum(x - 1)
7
8 print recsum(5)
```

Al compilar y ejecutar nos muestra lo siguiente:

Éxito

stdin

Standard input is empty

stdout

print stack

print stack

print stack

print stack

15

Al establecer el siguiente código optimizando:

```
1 def tailrecsum(x, running_total=0):
2     if x == 0:
3         return running_total
4     else:
5         print "print stack"
6         return tailrecsum(x - 1, running_total + x)
7
8 print tailrecsum (5, 0)
```

Obtenemos lo siguiente al compilarlo y ejecutarlo:

```
Éxito

stdin
Standard input is empty

stdout
print stack
print stack
print stack
print stack
print stack
15
```

Como conclusión vemos que Python no posee una optimización.

➤ *Lenguaje Haskell:*

Este código es el factorial:

```
1 module Main
2
3 where
4
5 import Debug.Trace
6
7 factorial :: Int -> Int
8 factorial 1 = traceStack ("fin stack") 1
9 factorial x = traceStack ("stack") x * (factorial $ x - 1)
10
11 main = do
12     putStrLn $ "factorial 5: " ++ show (factorial 5)
```

Al compilar y ejecutar nos muestra lo siguiente:

```
Éxito

stdin
Standard input is empty

stdout
factorial 5: 120

stderr
stack
stack
stack
stack
fin stack
```

Para poder optimizarlo lo compilamos con el prefijo “-O2” (`ghc -O2 haskell_test_5.hs`). Lo cual posee “tail call optimization”.

➤ *Lenguaje Java:*

Este código es el factorial sin optimizar:

```
1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5
6 class test {
7     static int factorial(int n) {
8         if (n == 0) {
9             for (StackTraceElement ste : Thread.currentThread().getStackTrace())
10                 System.out.println(ste);
11             return 1;
12         }
13         else {
14             return n * factorial(n-1);
15         }
16     }
17
18     public static void main(String[] args) {
19         System.out.println(factorial(5));
20     }
21 }
```

Al compilar y ejecutar nos muestra lo siguiente:

Éxito

 stdin

Standard input is empty

🔧 stdout

```
java.lang.Thread.getStackTrace(Thread.java:1552)
test.factorial(Main.java:9)
test.factorial(Main.java:16)
test.factorial(Main.java:16)
test.factorial(Main.java:16)
test.factorial(Main.java:16)
test.factorial(Main.java:16)
test.main(Main.java:21)
```

120

Al establecer el siguiente código optimizando la función factorial:

```
1 import java.util.*;
2 import java.lang.*;
3 import java.io.*;
4
5
6 class test2 {
7     static int factorial(int n, int acc) {
8         if (n == 0) {
9             for (StackTraceElement ste : Thread.currentThread().getStackTrace())
10                 {
11                     System.out.println(ste);
12                 }
13             return acc;
14         }
15         else {
16             return factorial(n-1, n*acc);
17         }
18     }
19
20 public static void main(String[] args) {
21     System.out.println(factorial(5, 1));
22 }
23 }
```

Obtenemos lo siguiente al compilarlo y ejecutarlo:

Éxito

stdin

Standard input is empty

stdout

```
java.lang.Thread.getStackTrace(Thread.java:1552)
test2.factorial(Main.java:9)
test2.factorial(Main.java:16)
test2.factorial(Main.java:16)
test2.factorial(Main.java:16)
test2.factorial(Main.java:16)
test2.factorial(Main.java:16)
test2.factorial(Main.java:16)
test2.main(Main.java:21)
120
```

Como conclusión vemos que Java no posee una optimización.

➤ *Lenguaje C :*

Este código es el factorial:

```
1 #include <stdio.h>
2 long factorial(int n) {
3     if (n == 0)
4         return 1;
5     else
6         return(n * factorial(n-1));
7 }
8
9 int main() {
10     int result = factorial(5);
11     printf("factorial result es: %d\n", result);
12     return 0;
13 }
14
```

Al compilar y ejecutar con gdb, para lograr ver su stack, nos muestra lo siguiente:

```
(gdb) b 4
Punto de interrupción 1 at 0x80483f0: file c_test_5.c, line 4.
(gdb) bt
No stack.
(gdb) run
Starting program:

Breakpoint 1, factorial (n=0) at c_test_5.c:4
4         return 1;
(gdb) bt
#0  factorial (n=0) at c_test_5.c:4
#1  0x08048405 in factorial (n=1) at c_test_5.c:6
#2  0x08048405 in factorial (n=2) at c_test_5.c:6
#3  0x08048405 in factorial (n=3) at c_test_5.c:6
#4  0x08048405 in factorial (n=4) at c_test_5.c:6
#5  0x08048405 in factorial (n=5) at c_test_5.c:6
#6  0x08048420 in main () at c_test_5.c:10
(gdb) □
```

Para poder optimizarlo lo compilamos con el prefijo “-O1”(gcc -g -O1 c_test_5.c -o c_test_5). Y obtenemos lo siguiente al compilarlo y correrlo con gdb.

```

(gdb) b 4
Punto de interrupción 1 at 0x804840c: file c_test_5.c, line 4.
(gdb) bt
No stack.
(gdb) run
Starting program:

Breakpoint 1, factorial (n=5) at c_test_5.c:4
4         return 1;
(gdb) bt
#0  factorial (n=5) at c_test_5.c:4
#1  0x0804843d in main () at c_test_5.c:10
(gdb) █

```

Concluimos que posee “tail call optimization”.

➤ *Lenguaje Scala :*

Este código es el factorial sin optimizar:

```

1 object scala_test_5 {
2
3     def factorial(n: Int): Int = {
4         if (n <= 1) return 1/0;
5         else return n*factorial(n - 1);
6     }
7
8     def main (args: Array[String]) {
9         try {
10            println (factorial(5));
11        } catch {
12            case e: Exception => e.printStackTrace;
13            System.exit(1);
14        }
15    }
16 }

```

Al compilar y ejecutar nos muestra lo siguiente:

```

Exception in thread "main" java.lang.ArithmeticException: / by zero
    at scala_test_5$.factorial(scala_test_5.scala:4)
    at scala_test_5$.factorial(scala_test_5.scala:5)
    at scala_test_5$.factorial(scala_test_5.scala:5)
    at scala_test_5$.factorial(scala_test_5.scala:5)
    at scala_test_5$.factorial(scala_test_5.scala:5)
    at scala_test_5$.main(scala_test_5.scala:10)
    at scala_test_5.main(scala_test_5.scala)

```

Al establecer el siguiente código optimizando la función factorial:

```
1 import scala.annotation.tailrec
2 object scala_test_5_opt {
3
4     @tailrec def factorialAcc(acc: Int, n: Int): Int = {
5         if (n <= 1) return 1/0;
6         else return factorialAcc(n * acc, n - 1);
7     }
8
9     def main (args: Array[String]) {
10         try {
11             println (factorialAcc(1, 5));
12         } catch {
13             case e: Exception => e.printStackTrace;
14             System.exit(1);
15         }
16     }
17 }
```

Obtenemos lo siguiente al compilarlo y ejecutarlo:

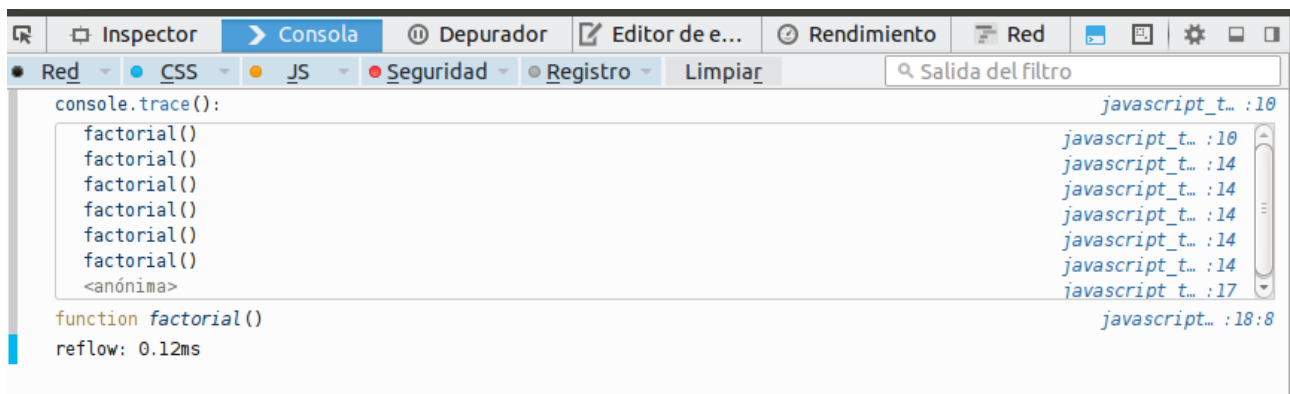
```
java.lang.ArithmeticException: / by zero
    at scala_test_5_opt$.factorialAcc(scala_test_5_opt.scala:5)
    at scala_test_5_opt$.main(scala_test_5_opt.scala:11)
    at scala_test_5_opt.main(scala_test_5_opt.scala)
```

Como conclusión vemos que Scala si posee una optimización.

➤ *Lenguaje JavaScript :*

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta charset="utf-8">
5 </head>
6 <body>
7     <script type="text/javascript">
8         function factorial (n) {
9             if (n == 0) {
10                 console.trace();
11                 return 1;
12             }
13             else {
14                 return n*factorial(n-1);
15             }
16         };
17         var result = factorial(5);
18         console.log(factorial);
19     </script>
20 </html>
```

La salida de pantalla al ejecutar el código es la siguiente:



Según nuestra conclusión JavaScript no posee optimización a la cola ya que al ser tan flexible para todos los navegadores existentes, no existe un método que permita tal optimización.

➤ *Lenguaje Ruby :*

Este código es el factorial sin optimizar:

```
1 def fact(n)
2   return 1 if n <= 1
3   puts "print stack"
4   n * fact(n-1)
5 end
6
7 puts fact(5)
```

Al compilar y ejecutar nos muestra lo siguiente:



Al establecer el siguiente código optimizando la función factorial:

```
1 RubyVM::InstructionSequence.compile_option = {  
2   tailcall_optimization: true,  
3   trace_instruction: false  
4 }  
5  
6 def fact(n, acc)  
7   puts caller  
8   return acc if n <= 1  
9   puts "print stack"  
10  fact(n-1, n*acc)  
11 end  
12  
13 puts fact(5,1)
```

Obtenemos lo siguiente al compilarlo y ejecutarlo:

```
Éxito  
  
stdin  
Standard input is empty  
  
stdout  
prog.rb:13:in `<main>'  
print stack  
prog.rb:10:in `fact'  
prog.rb:13:in `<main>'  
print stack  
prog.rb:10:in `fact'  
prog.rb:10:in `fact'  
prog.rb:13:in `<main>'  
print stack  
prog.rb:10:in `fact'  
prog.rb:10:in `fact'  
prog.rb:10:in `fact'  
prog.rb:13:in `<main>'  
print stack  
prog.rb:10:in `fact'  
prog.rb:10:in `fact'  
prog.rb:10:in `fact'  
prog.rb:10:in `fact'  
prog.rb:13:in `<main>'  
120
```

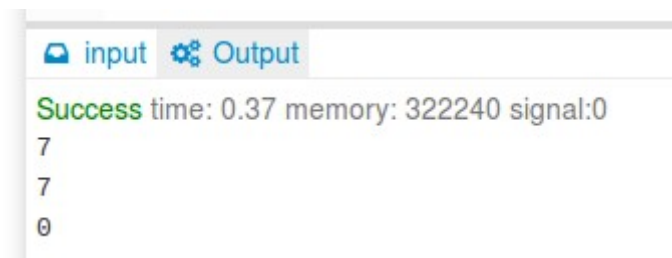
Como conclusión vemos que Ruby no posee una optimización ya que el stack es constante.

Ejercicio 6:

En este ejercicio vamos a ver como funciona el alto orden. Para ello utilizaremos el lenguaje de programación Scala. El código es el siguiente:

```
1 object ej6 {  
2     var y = 6  
3     lazy val x = 1 + y  
4     println(x)  
5     y = 0  
6     println(x)  
7     println(y)  
8 }
```

Al compilarlo y ejecutarlo nos muestra lo siguiente:



```
input  Output  
Success time: 0.37 memory: 322240 signal:0  
7  
7  
0
```

Aquí podemos ver que cuando declaramos una variable lazy 'x', y luego la invocamos en el primer println, el calculo lo hizo, es decir, el "1+y". Luego nuevamente en el println la invocamos, pero aquí ya no vuelve a calcular el "1+y" (de ser así 'x' valdría 1). Esto nos dice que la variable lazy solo se calculo una sola vez, es decir, su valor no sera recomputado en cada una de las veces que se lea de nuevo la variable.

Ahora veamos como seria en programación de alto orden. Para ello escribimos el siguiente código:

```
1 object ej6b {  
2     var y = 6  
3     var z = 1  
4     def sum (): Int =  
5         y + z  
6     lazy val lzy = sum _  
7     println (lzy)  
8     println (lzy())  
9     y = 7  
10    z = 2  
11    println (lzy())  
12 }
```

Al compilar y ejecutarlo nos muestra:

```
input Output
Success time: 0.37 memory: 322240 signal:0
<function0>
7
9
```

Al ver el resultado podemos observar que declaramos una variable lazy 'lzy'. Al invocarla en el primer println lzy se calcula y nos devuelve 7, luego modificamos las variables de "y" y "z", y al volver a invocarla en el println vemos que 'lzy' vuelve a calcular la suma devolviendo 9. Este comportamiento es porque estamos usando alto orden al pasar un argumento como función (la variable se recompute)

Ejercicio 7:

Vamos a ver cual es la diferencia entre estos dos tipos de pasajes por parámetro.

Para ello vamos a usar el lenguaje Scala.

Dado el siguiente código:

```
1 object Main extends App {  
2   var y = 3;  
3   lazy val x = { System.nanoTime }  
4   println("call-by-name")  
5   mostrar(prod(y))  
6   println("call-by-value usando lazy")  
7   mostrar(x)  
8   println("Vuelvo a llamar a ambos")  
9   mostrar(prod(y))  
10  mostrar(x)  
11  
12  def prod(a: =>Int) = { System.nanoTime }  
13  
14  def mostrar(t: =>Long) = { println("Parametro: " + t) }  
15 }
```

Al compilarlo y ejecutarlo nos muestra:

Éxito

stdin

Standard input is empty

stdout

call-by-name

Parametro: 12953567209685647

call-by-value usando lazy

Parametro: 12953567210304487

Vuelvo a llamar a ambos

Parametro: 12953567211205461

Parametro: 12953567210304487

Aquí se puede ver que cuando le pasamos una variable lazy (x) lo calcula una sola vez y no re-calcula cada vez que es invocada. Por otro lado vemos que cuando es pasada por valor, re-calcula cada vez que es invocada devolviendo diferentes resultados, en este caso el tiempo.

Ejercicio 8:

- Fragmento de C : funciona por defecto por *call-by-value*, en el cual se pasa un valor al procedimiento y se coloca en una celda (variable) local del mismo; así los datos pasados son modificados dentro del procedimiento (función). Se puede utilizar *call-by-reference*, que usa punteros, por la eficiencia en la memoria, pero trae complicaciones secundarias en el paso de parámetros de referencia y por falta de privacidad. Es recomendable usarla en el caso de querer cambios en los datos.

Código C (*call-by-value*):

```
1 #include <stdio.h>
2
3 void interchange(int x1, int y1);
4
5 void main(){
6     int x = 50, y = 70;
7     interchange(x,y); /* Pasaje por valor */
8     printf("x = %d, y = %d\n",x,y);
9 }
10
11 void interchange(int x1,int y1){
12     int z1;
13     z1 = x1;
14     x1 = y1;
15     y1 = z1;
16     printf("interchange: x1 = %d, y1 = %d\n",x1,y1);
17 }
```

Salida en pantalla obtenida al ejecutarlo:

```
interchange: x1 = 70, y1 = 50
x = 50, y = 70
```

Código C (*call-by-reference*):

```
1 #include <stdio.h>
2
3 void interchange(int *x1, int *y1);
4
5 void main(void){
6     int x = 50, y = 70;
7     interchange(&x,&y); /* Paso por referencia */
8     printf("x = %d, y = %d\n",x,y);
9 }
10
11 void interchange(int *x1,int *y1){
12     int z1 = *x1;
13     *x1 = *y1;
14     *y1 = z1;
15     printf("interchange: x1 = %d, y1 = %d\n",*x1,*y1);
16 }
```

Salida en pantalla obtenida al ejecutarlo:

```
interchange: x1 = 70, y1 = 50  
x = 70, y = 50
```

- Fragmento de Perl : funciona por *call-by-value*, ya que si utilizamos *call-by-reference* (que viene dado por defecto), puede ser difícil de leer y con una semántica, con resultados no deseables. Así que al utilizar *call-by-value* obtenemos las mismas conclusiones que en C.

Código Perl (*call-by-value*):

```
1 $x=50;  
2 $y=70;  
3 &interchange ($x, $y); # Pasaje por valor  
4 print "x:$x, y:$y\n";  
5  
6 sub interchange{  
7     ($x1, $y1) = @_  
8     $z=$x1;  
9     $x1=$y1;  
10    $y1=$z;  
11    print "interchange = x1:$x1, y1:$y1\n";  
12 }
```

Salida en pantalla obtenida al ejecutarlo:

Éxito

stdin

Standard input is empty

stdout

interchange = x1:70, y1:50

x:50, y:70

Código Perl (*call-by-reference*):

```
1 $x=50;
2 $y=70;
3 &interchange (*x, *y);      # Pasaje por referencia
4 print "x:$x, y:$y\n";
5
6 sub interchange{
7     (*x1, *y1) = @_;
8     $z=$x1;
9     $x1=$y1;
10    $y1=$z;
11    print "interchange = x1:$x1, y1:$y1\n";
12 }
```

Salida en pantalla obtenida al ejecutarlo:

Éxito

 stdin

Standard input is empty

 stdout

interchange = x1:70, y1:50

x:70, y:50

Ejercicio 9:

- En el siguiente código de Java, tenemos una clase Point, que posee un elemento 'x' e 'y', y funciones tricky1 (que cambia los valores de 'x' e 'y' entre dos elementos Point), y tricky2 (que pone los valores de 'x' e 'y' en null también de dos elementos Point).


Cuando se ejecuta el programa, crea dos elementos Point (con valores 'x' e 'y' en 0) y los imprime; luego ejecuta la función tricky1, pasándole como parámetros los elementos Point pnt1 y pnt2, y allí valúa al argumento uno (pnt1), en sus variables 'x' e 'y', en 100 cada una; crea un elemento Point temporal, para cambiar los valores de pnt1 a pnt2, y viceversa. Luego, imprime las variables de los elementos pnt1 y pnt2, donde 'pnt1.x' y 'pnt1.y' es igual a 100, y 'pnt2.x' y 'pnt2.y' es igual a 0, ya que al asignarle un valor a la estructura interna de Point se modifica, pero al tratar de intercambiar los valores entre uno y otro elemento Point, no surge efecto, porque los cambios realizados solo tienen efecto dentro la función; concluyendo así, que los argumentos se llaman por *call-by-value*. Esto también pasa cuando se ejecuta luego tricky2, que toma los elementos Point pnt1 y pnt2, y los valúa a null, ya que luego al imprimir los valores de las variables de pnt1 y pnt2, se mantienen los mismos anteriormente impresos, quedando demostrado que solo tiene efecto dentro de la función (*call-by-value*).

Código Java:

```
1 public class Point { //call by value
2     public int x;
3     public int y;
4     public Point(int x, int y){
5         this.x = x;
6         this.y = y;
7     }
8     public static void tricky1(Point arg1, Point arg2)
9     {
10         arg1.x = 100;
11         arg1.y = 100;
12         Point temp = arg1;
13         arg1 = arg2;
14         arg2 = temp;
15     }
16     public static void tricky2(Point arg1, Point arg2)
17     {
18         arg1 = null;
19         arg2 = null;
20     }
21     public static void main(String [] args)
22     {
23         Point pnt1 = new Point(0,0);
24         Point pnt2 = new Point(0,0);
25         System.out.println("\npnt1 X: " + pnt1.x + " pnt1 Y: " + pnt1.y);
26         System.out.println("pnt2 X: " + pnt2.x + " pnt2 Y: " + pnt2.y);
27         System.out.println("\ntriki1");
28         tricky1(pnt1,pnt2);
29         System.out.println("pnt1 X: " + pnt1.x + " pnt1 Y: " + pnt1.y);
30         System.out.println("pnt2 X: " + pnt2.x + " pnt2 Y: " + pnt2.y);
31         System.out.println("\ntriki2");
32         tricky2(pnt1,pnt2);
33         System.out.println("pnt1 X: " + pnt1.x + " pnt1 Y: " + pnt1.y);
34         System.out.println("pnt2 X: " + pnt2.x + " pnt2 Y: " + pnt2.y + "\n");
35     }
36 }
```


Salida en pantalla obtenida al ejecutarlo:

Éxito

 stdin

Standard input is empty

 stdout

pnt1 X: 0 pnt1 Y: 0

pnt2 X: 0 pnt2 Y: 0

triki1

pnt1 X: 100 pnt1 Y: 100

pnt2 X: 0 pnt2 Y: 0

triki2

pnt1 X: 100 pnt1 Y: 100

pnt2 X: 0 pnt2 Y: 0

Se puede concluir que en los métodos de Java se pueden pasar referencias a objetos por valor (pasa objetos como referencias y *las referencias* se pasan por valor).