Ruby: Un Tour Fotográfico

Este “Tour Fotográfico” por el lenguaje de programación **Ruby** nació durante un reposo que me vi obligado a tomar debido a la dolorosa inflamación de un nervio en la espalda. Durante esos días no podía sentarme y la mayor parte del tiempo lo pasé acostado de espaldas, con una tabla y una libreta de apuntes apoyadas en mis piernas mientras escribía estas ideas. Que yo recuerde no tuve fiebre, aunque esto parezca el resultado de un delirio febril **;-)**

Esto no pretende ser un tutorial, ni una referencia, sino un recorrido por las principales características que hacen de **Ruby** un lenguaje de programación que siente como una extensión de nuestras habilidades para programar. Tampoco intento “vender” aquí a Ruby como una especie de panacea universal, sino que como una excelente opción para que la incorporen en su “caja de herramientas”.

Esto es trabajo en progreso, ha medida que el tiempo y la creatividad hagan conjunción en mi vida, cuenten con que este irá creciendo y mejorando. Cualquier sugerencia, corrección o comentario será ampliamente apreciado pueden ir a la sección Sobre mí, si quieren contactarme.

Actualizado 2006-05-10

Objetos: Aprendamos de los niños

Los niños pequeños descubren el mundo a cada instante y a medida que crecen, y aprenden, pierden mucha de esa capacidad de explorar el mundo con genuino sentimiento de maravilla. En los peores casos, cuando llegan a ser grandes se convierten en programadores **;-)**

La suma es algo complejo para los niños pequeños, quienes sin embargo son capaces de contar sin ningún problema, a veces pienso que educamos a nuestros niños de formas extrañas. En **Ruby**, dado un número podemos obtener el siguiente en la secuencia, así que hagamos como un niño pequeño en este momento:

```
irb(main):001:0> 1.next
=> 2
irb(main):002:0> 2.next
=> 3
```

No sólo es **Ruby** un lenguaje de programación *orientado a objetos* sino que, además y en forma consistente con este paradigma, prácticamente todo en **Ruby** es un objeto.

En el ejemplo anterior vimos como en **Ruby** los número enteros son capaces de responder al mensaje `next`, retornando el siguiente número en la secuencia **:-)** Lo que no es mucho, pero es un comienzo.

`irb` son las siglas del **I**nteractive **R**u**B**y que es una librería que nos permite jugar con **Ruby** en forma interactiva a través de una línea de comando.

Cada vez que `irb` evalúa una expresión retorna el objeto resultante `=>`

La mayoría de los ejemplos en este “tour” pueden ser ejecutados con el `irb`.

http://www.lacaraoscura.com/2006/05/10/ruby-un-tour-fotografico

Clases, pero no sociales

Donde hay objetos, no es raro que haya clases, aunque no es indispensable. Y donde hay objetos y clases, es muy sospechoso que vamos a encontrar *herencia*. Cuando hablamos de herencia no nos referimos a toda esa cantidad de dinero que los sobrinos de **Rico Mc Pato** esperaban heredar, sino de comportamientos y características que se “pasan” de una clase a otra.

```
irb(main):001:0> 1.class
=> Fixnum
irb(main):002:0> Fixnum.superclass
=> Integer
irb(main):003:0> Integer.superclass
=> Numeric
irb(main):004:0> Numeric.superclass
=> Object
irb(main):005:0> Object.superclass
=> nil
```

`Object` es la cabeza de la jerarquía (bastante achatada por cierto) de clases de **Ruby**. En este punto es bueno insistir en que en **Ruby** *todo* es un objeto, y que en el ejemplo anterior las clases `Fixnum`, `Integer`, `Numeric` y `Object` no hacen más que responder al mensaje `superclass` retornando la clase de la que ellas extienden.

La clase `Object` por supuesto exhibe comportamientos muy generales, mientras que `Fixnum` está especializado en manejar números enteros. Veamos a que responde `Object`:

```
irb(main):003:0> Object.instance_methods
=> ["clone", "protected_methods", "freeze", "display", "send",
"instance_variable_set", "is_a?", "type", "methods", "=~",
"instance_of?", "__id__", "instance_variables", "to_s", "eql?",
"dup", "hash", "private_methods", "instance_eval", "extend", "nil?",
"__send__", "tainted?", "class", "singleton_methods", "untaint",
"instance_variable_get", "object_id", "kind_of?", "inspect",
"respond_to?", "taint", "frozen?", "==", "public_methods", "id",
"===", "equal?", "to_a", "method"]
```

NIL, NULL, NUL, Nulo

En muchos lenguajes de programación, `null` (o cualquier cosa similar) representa la ausencia de un objeto al que hacer referencia, y suele tener asociado un error cuando se intenta referenciar a este objeto inexistente.

En el ejemplo anterior como `Object` no extiende ninguna clase, **Ruby** nos retornó `nil`. En **Ruby** `nil` es un objeto *singleton* (único) de la clase `NilClass`.

```
irb(main):001:0> nil.class
=> NilClass
irb(main):002:0> nil.nil?
=> true
```

El único objeto que responde con `true` cuando se le pasa el mensaje `nil?` es `nil`, por supuesto. Y ya que nos asomamos a los dominios del **Sr. George *Boole***, es conveniente decir, que con el fin de hacernos la vida más práctica a los programadores, el valor de verdad de `nil` es `false`. Ustedes son personas inteligentes y a estas alturas del partido ya se podrán imaginar que `true` y `false` son a su vez singletons de las clases `TrueClass` y `FalseClass`.

No sólo el caracter ? (interrogación) es válido en el nombre de un método sino que además es un convención común para identificar los predicados, es decir métodos que devuelven `True` o `False`).

Mi tipo de sangre es O RH Negativo

Lo que no tiene nada que ver con los patos de la foto, y en realidad no viene al caso, pero es bueno que lo sepan a la hora de que *yo* necesite una transfusión. Generalmente en los lenguajes de programación orientados a objetos se asume que la *clase* de un objeto es su *tipo*, lo que además suele dar lugar a situaciones de incompatibilidad. Por ejemplo, cuando se tiene un objeto de una clase y un método espera que ese objeto sea de *otra* clase.

Ruby es mucho más flexible que mi sistema inmune (que sólo acepta transfusiones de sangre de tipo O RH Negativo) y que otros lenguajes de programación orientados a objeto, flexibilidad que se expresa en el principio del *Duck Typing*:

> *Si se mueve como un pato y suena como un pato, **entonces** debe ser un pato.*

Si usted tiene experiencia con **Python**, esto le será natural; si por el contrario se aproxima a **Ruby** desde **Java** o **C#** esto le parecerá *anatema* y estará considerando seriamente utilizar estas hojas para encender los carbones en su próxima parrillada, si es que las llegó a imprimir. Por favor respire hondo, espere un momento y siga adelante, el *chequeo estricto de tipo* tiene su sitio en la caja de herramientas de programación y el *Duck Typing* también.

En general, es comúnmente aceptado que en **Ruby** el tipo de un objeto está determinado por el conjunto de mensajes a los que responde:

```
irb(main):001:0> 5.respond_to? :next
=> true
irb(main):002:0> 5.respond_to?(:upcase)
=> false
```

El objeto `5` responde a los mensajes `next` y `-`, pero no a `upcase`. Si bien esta característica de **Ruby** es extremadamente interesante y poderosa, es conveniente que antes revisemos algunos de los “tipos” básicos de **Ruby ;-)** para así tener más elementos con los que jugar.

En **Ruby** generalmente los paréntesis pueden ser omitidos siempre que no haya ambigüedad, sin embargo en caso de duda *úsenlos*.

Una secuencia de caracteres que se inicia con `:` es un *símbolo*, una representación de un nombre en **Ruby**.

http://www.lacaraoscura.com/2006/05/10/ruby-un-tour-fotografico

Números, más allá de contar con los dedos

Ruby permite manipular número enteros de tamaño arbitrario así como números de punto flotante de doble precisión, en la forma usual que nuestras maestras de primaria se empeñaron en enseñarnos.

```
irb(main):001:0> 1 + 2
=> 3
irb(main):002:0> 1 + (-3 + 1)
=> -1
irb(main):003:0> 3.14159 * 2
=> 6.28318
irb(main):004:0> 3 + "3"
TypeError: String can't be coerced into Fixnum
        from (irb):4:in `+'
        from (irb):4
```

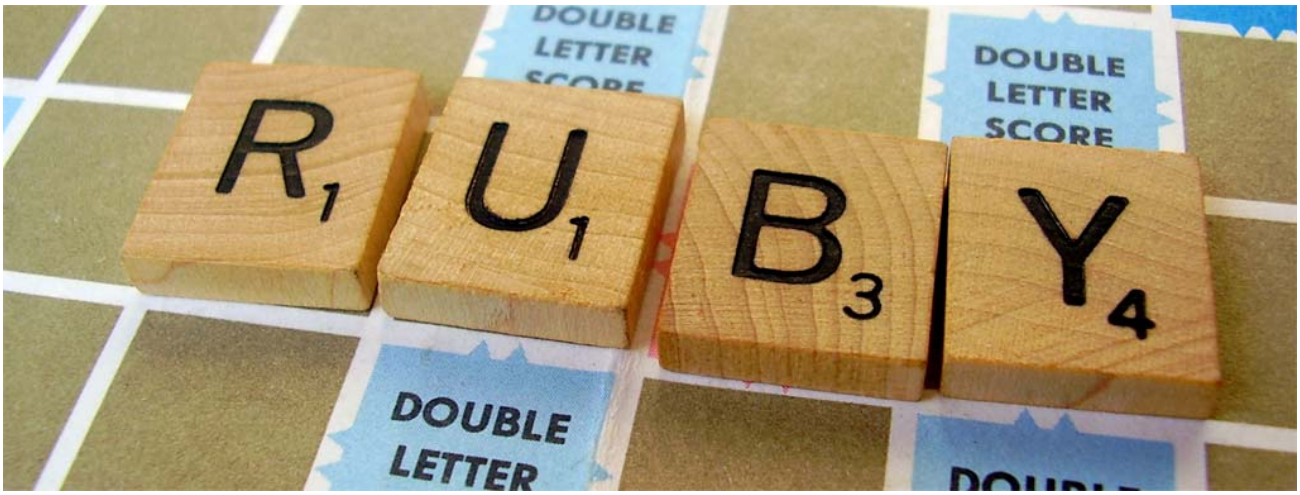
La mayor parte de estas operaciones (como la suma y la resta) están implementadas como métodos que usted podría redefinir si quisiera, aunque seguramente no sería buena idea. En **Ruby** no existen vacas sagradas y, si bien se proveen mecanismos para asegurar el código, nada le impide trastear con cualquiera de las *clases predefinidas* o parte de las *librerías estándares*.

```
irb(main):001:0> 123.class
=> Fixnum
irb(main):002:0> 1234567890.class
=> Bignum
irb(main):003:0> 1.2.class
=> Float
```

Si observan bien el ejemplo anterior, se darán cuenta que para representar número enteros se usaron dos clases diferentes, `Fixnum` y `Bignum`. Los primeros permiten representar `Integer`s del tamaño de una palabra nativa de la máquina (menos 1 bit) mientras que el intérprete de **Ruby** apela al segundo cuando la expresión evaluada sobrepasa esta medida.

A diferencia de lenguajes como **Perl** o **PHP**, **Ruby** no convertirá en forma automática un `String` al número equivalente para realizar una operación matemática, comportamiento que por otra parte es muy sencillo de cambiar si así lo quisiéramos. Ahora que lo nombramos, hablemos de los…

String, el infaltable



Los `String`s permiten almacenar y manipular secuencias de caracteres. La forma más común de crear objetos de esta clase es mediante literales delimitados con comillas dobles o simples, cosa que no creo que sorprenda a nadie hoy en día.

```
irb(main):001:0> s = 'Scrabble'
=> "Scrabble"
irb(main):002:0> s1 = 'Me gusta' + 'a' * 3 + ' el ' + s
=> "Me gustaaaa el Scrabble"
irb(main):003:0> s2 = "Me gusta#{'a'*3} el #{s}"
=> "Me gustaaaa el Scrabble"
```

Cuando el mensaje `+` es pasado a un `String`, sencillamente se realiza la concatenación, y en el caso del método `*`, lo que se retorna es un nuevo `String` repetido tantas veces como se haya especificado.

Los literales delimitados por comillas dobles son objeto de un mayor número de reemplazos que los delimitados por comillas sencillas, particularmente de la *interpolación de expresiones*. Las expresiones encerradas entre #{ y } son evaluadas, como cualquier otra expresión en **Ruby**, y reemplazadas por su valor en el `String` resultante.

Ahora tomemos un cuchillo, y saquemos algunas tajadas a un `String`:

```
irb(main):001:0> x = 'Esta es una prueba'
=> "Esta es una prueba"
irb(main):002:0> x[5,2]
=> "es"
irb(main):003:0> x[5..7]
=> "es "
irb(main):004:0> x[0]
=> 69
irb(main):005:0> x[0,1]
=> "E"
```

Cuando se especifica un solo índice se obtiene el código del carácter, no un substring.. La expresión `5..7` no es una construcción del lenguaje sino que origina un objeto de la clase `Range`…

http://www.lacaraoscura.com/2006/05/10/ruby-un-tour-fotografico

Rangos, aunque no militares

En el ejemplo anterior se seleccionó una parte de un String especificando un *rango* de posiciones del mismo. En Ruby, los rangos son objetos de la clase `Range`.

```
irb(main):001:0> r1 = 5..15
=> 5..15
irb(main):002:0> r1.class
=> Range
irb(main):003:0> r1.min
=> 5
irb(main):004:0> r1.include?(15)
=> true
irb(main):005:0> r2 = 5...15
=> 5...15
irb(main):006:0> r2.include?(15)
=> false
irb(main):007:0> r3 = Range.new(5,15)
=> 5..15
irb(main):008:0> r3 === 10
=> true
```

En la línea 7 construimos el rango `r3` equivalente al `r1` de la forma “tradicional” (es decir invocando el constructor sobre la clase con `new`), y además probamos el método `===` que convenientemente nos permite verificar si el parámetro suministrado está incluido en el rango.

Cualquier par de de objetos que puedan ser comparados usando el método `<=>`, y que soporten el método `succ` (que sencillamente debe retornar el siguiente en la secuencia) sirven para construir un rango, sin necesidad de construir nuestra propia clase podemos ver un ejemplo con los `Strings`.

```
irb(main):001:0> r4 = 'm'..'q'
=> "m".."q"
irb(main):002:0> r4 === 'a'
=> false
```

Sí, `===` es un método, y de hecho la expresión:

`r3 === 10`

Es equivalente a:

`r3.===(10)`

Sólo que mucho más conveniente de usar.

http://www.lacaraoscura.com/2006/05/10/ruby-un-tour-fotografico

Bloques, mejores que los de construcción

Una de las mejores cosas que tienen los rangos (y cualquier cosa que se comporte como un `Enumerable`) en **Ruby**, es que podemos recorrerlos. Y aquí es donde empieza la verdadera diversión así que ajusten sus cinturones y respiren hondo.

```
irb(main):001:0> r = 0..9
=> 0..9
irb(main):004:0> r.each { |i| print( "#{i+1} " ) }
1 2 3 4 5 6 7 8 9 10 => 0..9
```

En **Ruby** los bloques son porciones de código ejecutable delimitados por paréntesis o `do` .. `end`. En este caso particular el método `each` acepta o reconoce que se le ha suministrado un *bloque*, por lo que recorrerá el rango `r`, colocando en la variable `i` cada uno de los valores del rango.

¿Que qué tienen de especial los bloques? Bueno, que nos brindan una gran flexibilidad y claridad a la hora de traducir nuestras ideas en código en ejecución. El ejemplo que me hizo darme cuenta de esto tiene que ver con la manipulación de archivos:

```
File.readlines('prueba.txt').each do |linea|
  print("> #{linea}")
end
```

El código anterior sencillamente abre el archivo de nombre `prueba.txt` e imprime cada línea. Lo *bonito* de este código es la clase `File` se encarga de todas la tareas de *housekeeping*, como lo son abrir y cerrar el archivo. De esta forma nos pudimos concentrar en la tarea que queríamos realizar, sin perder el tiempo escribiendo código de soporte.

En Ruby encontrarán los bloques convenientemente soportados en muchos sitios, y con esto:

```
irb(main):001:0> 5.downto(1) { |numero| print "#{numero}.. " }
5.. 4.. 3.. 2.. 1.. => 5
```

¡Despegue! Si quieren incorporar soporte para bloques en sus propios métodos deben leer sobre `yield`, sin embargo es pronto para eso, antes mejor saquémosle punta a otros aspectos de **Ruby** …

Expresiones Regulares, cuidado con el filo

Todo el mundo me dice que no se entiende la foto que encabeza este artículo, no me importa. Es un acercamiento a una navaja suiza que me acompaña desde hace muchos años, y la idea era reflejar la utilidad de las Epresiones Regulares en **Ruby**. Las expresiones regulares son la adoración de los programadores **Perl**, y es una característica heredada que realmente se aprecia. Empecemos con algo sencillo para afilar esta navaja:

```
irb(main):001:0> "Esta es una prueba" =~ /es/
=> 5
```

Bien, revisemos este código y empecemos por lo sencillo, `"Esta es una prueba"` es una `String`, eso no tiene confusión. La expresión entre *slashes*, es una *expresión regular*, que no es más que un patrón contra el que se compara el `String` en búsqueda de coincidencias *¿Por qué se buscan coincidencias?* En este caso, porque se usó el operador `=~` que retorna la posición donde se encuentra la coincidencia con el patrón (o `nil` si no hay ninguna coincidencia)

> **Ruby** sigue la convención de que las posiciones (por ejemplo de los caracteres en los `String`s) se cuentan desde **cero**.

Ahora, hagamos algunas cosas más *interesantes*…

```
irb(main):001:0> correo = "juan.perez@dominiodementira.com"
=> "juan.perez@dominiodementira.com"
irb(main):002:0> correo.gsub!( /([aeiou])/ ) { |match| match.upcase }
=> "jUAn.pErEz@dOmInIOdEmEntIrA.cOm"
irb(main):003:0> correo
=> "jUAn.pErEz@dOmInIOdEmEntIrA.cOm"
```

El método `gsub!` (**g**lobal **sub**stitution) acepta como parámetro una expresión regular y (en la forma en que lo usamos) un *bloque* al que le pasa cada ocurrencia de la expresión regular que se le suministró. La expresión dentro del bloque es evaluada, convirtiendo cada ocurrencia a mayúsculas y utilizando el valor resultante para el reemplazo.

> En **Ruby** existe la convención de que los métodos terminados en exclamación `!` tienen efectos laterales. Es común encontrar pares como `gsub` y `gsub!`

Si no están muy claros con esto de las expresiones regulares, háganse un favor y estúdienlas para incorporarlas en su “caja de herramientas”

http://www.lacaraoscura.com/2006/05/10/ruby-un-tour-fotografico

Arreglos, pero no del hogar

Los arreglos son colecciones ordenadas de objetos que pueden ser referenciados por la posición que ocupan en el mismo. Dicho así, es como decir que una hamburguesa es carne de bovino muerto, cocinada y colocada entre dos capas de trigo procesado. Pero la realidad es un poco más interesante:

```
irb(main):001:0> a = Array.new
=> []
irb(main):002:0> a << 1
=> [1]
irb(main):003:0> a.concat [ 'a', 'b' ]
=> [1, "a", "b"]
irb(main):004:0> a[1]
=> "a"
irb(main):005:0> a[1..2]
=> ["a", "b"]
```

En la primera línea creamos un nuevo arreglo vacío, al que le agrega (`<<`) el número `1` (operación que devuelve el propio arreglo por lo que puede ser llamada en cadena) después concatenamos un arreglo ya existente mediante el método `concat` y finalmente referenciados un elemento por su posición y después tomamos un porción del arreglo con el rango `1..2`. Veamos una sintaxis alternativa:

```
irb(main):001:0> b = []
=> []
irb(main):002:0> b[0] = 1
=> 1
irb(main):003:0> b += ["a","b"]
=> [1, "a", "b"]
```

`x += y` en Ruby es convertido por el intérprete en `x = x + y`, y el operador `+` cuando se aplica a un `Array` se traduce en la concatenación. Y para los fanáticos de la serie de televisión **LOST**:

```
irb(main):001:0> [3, 7, 14, 15, 22, 41].collect! { |n| n + 1 }
=> [4, 8, 15, 16, 23, 42]
```

Ahí les dejo la secuencia generada a partir de un bloque que el método `collect!` ejecuta para cada elemento del arreglo “coleccionando” los valores generados en un nuevo arreglo.

http://www.lacaraoscura.com/2006/05/10/ruby-un-tour-fotografico

Esto es todo por el momento, consideren este documento una versión Beta temprana, abierta a sus comentarios, críticas y sugerencias. Como ya asomé al inicio la idea ha sido realizar un recorrido por los aspectos más relevantes de **Ruby** con el fin de brindar a las personas que se asoman a este lenguaje de programación un panorama de las bondades del mismo.

El formato elegido es obviamente limitado, y ha sido seleccionado así a propósito con el fin de mantener el foco en lo fundamental, y la longitud en lo mínimo posible. Si bien considero que Ruby es un lenguaje en el que lograr la maestría es un proceso de años, la comprensión básica del mismo debe ser alcanzada rápidamente.

Las fotos las tomé todas yo (bueno, en realidad la cámara), y el proceso de imaginación y composición fue realmente divertido, particularmente el que permitió llegar al “concepto” de `nil`. De lo más “cerebro derecho” que he hecho últimamente.

Una vez terminado, me gustaría traducirlo al inglés, aunque he de reconocer que mi inglés coloquial está lejos de ser suficiente para la tarea. Si saben de alguien interesado, sería cuestión de ponerse de acuerdo.

En las siguientes páginas tienen alguna información sobre mí, y la licencia que seleccioné. Si saben de alguien interesado en publicar un libro sobre Ruby en español avísenme, tengo algunas ideas sobre como “escalar” esto a un libro ¡Ah, y me encanta dar cursos y charlas!

Gracias por este tiempo de lectura que me han concedido.

Aníbal Rojas
`anibalrojas at gmail punto com`

Sobre su alucinado guía por el mundo de Ruby

Mi nombre es **Aníbal Rojas**, estudié computación en la **Universidad Central de Venezuela** y desde hace más de diez años trabajo desarrollando aplicaciones, principalmente bajo tecnologías Web. Cuando empecé en esto no había muchas más opciones más allá de **CGI** con **Perl**, y desarrollé con **Java** para la Web antes de que existiera el estándar **J2EE**, paraguas tecnológico que hoy en día me permite pagar las cuentas.

Desde hace un tiempo quedé prendado por el lenguaje de programación objeto de este recorrido por el que les acabo de guiar, al que llegué por el espectacular framework MVC para la Web **Ruby on Rails**.

Cuando el trabajo me lo permite, me pueden encontrar escribiendo en "**La Cara Oscura del Desarrollo de Software**"(http://www.lacaraoscura.com), un blog grupal dedicado a descargar nuestras experiencias y opiniones sobre el desarrollo de software, así como seguir temas de actualidad con una visión bien "ecléctica" de las tecnologías involucradas.

Por otra parte, cuando el trabajo me reclama estoy lidiando con clientes, requerimientos, programadores y códigos en **VALHALLA project. s.a.** Empresa establecida en **Caracas**, **Venezuela** hace más de 11 años, donde hemos convertido las necesidades de nuestros clientes en soluciones que se ejecutan 24 horas al día los 7 días de la semana. Si quiere conocer más sobre nuestra oferta de servicios profesionales, y nuestra experiencia, puede conseguirnos en:

VALHALLA project. s.a.

info@valhallaproject.com

Universidad Metropolitana,
La Colina Creativa,
Módulo 1, Planta Alta.
La Urbina Norte.
Caracas 1070 - Venezuela.

+58 (212) 242-4379 / 6662
+58 (212) 243-4055 / 6479

http://www.lacaraoscura.com/2006/05/10/ruby-un-tour-fotografico

Licencia, pero no de conducir

Creative Commons

Commons Deed

Reconocimiento-NoComercial-CompartirIgual 2.5

Usted es libre de:

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:

Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador.

No comercial. No puede utilizar esta obra para fines comerciales.

Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor

Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.