TUTORIAL DE POOI

Contents

1 Introducción

1.1 Programación orientada a objetos

La programación orientada a objetos es un paradigma de programación que maneja el concepto de **objeto** para representar entidades. Cada objeto tiene una lista de características que lo describen(definen su estado) llamados **atributos** y una serie de comportamientos asociados conocidos como **métodos**.

Por ejemplo, imaginemos un objeto gato que tiene un atributo color y un método maullar.

Actualmente existen dos estilos principales de programación a la hora de trabajar con objetos: lenguajes orientadados a objetos basados en **clases** como *C++* o *Java* y lenguajes orientados a objetos basados en **prototipos** como *Javascript*, *Python* o *Lua.*

1.2 Concepto de clase

Una clase es una construcción que se utiliza para crear instancias u objetos de si misma que tienen un comportamiento similar pero difieren en el valor de sus atributos, es decir, en su estado. En el mundo real una clase se podría decir que equivale a por ejemplo una máquina de hacer plastilina, todo lo que salga de la máquina será plastilina pero dependiendo de los ingredientes o del molde tendrá un color o una forma distinta a las demás.

1.3 Concepto de prototipo

Un prototipo no es más que un objeto a partir del cual se pueden crear otros . A diferencia de las clases aquí la creación de nuevos objetos no se basa en construir objetos nuevos si no en copiar los ya existentes. Por seguir con las analogías, quizás podríamos imaginarnos a un prototipo como una escultura de un cuerpo sin rasgos que un escultor utiliza de referencia. Cada vez que esculpe una estatua nueva, copia la referencia para luego añadir rasgos específicos. Ésta última a su vez también puede que sea utilizada como prototipo en un futuro.

1.4 Herencia

La herencia en un lenguaje basado en prototipos suele resolverse mediante delegación. Cada objeto mantiene uno o más atributos que apuntan al objeto a partir del cual han sido creados. Cuando un objeto recibe un **mensaje** para el cual no tiene respuesta escala a través de su árbol de padres buscando uno que sí sepa responder a ese mensaje.

Vamos a ilustrar esto con un ejemplo en *Javascript*, *Python* y *POOI*.

A partir de este punto distinguiremos lo que introduce el usuario de lo que muestra el intérprete con el símbolo '>'.

Ejemplo en *Javascript*:

Declaramos una variable que va a hacer de padre.

```
> var padre = {capital_chile:"Santiago"}
```

Creamos una variable que va a hacer de **hijo**.

```
> var hijo = {}
```

Establecemos el parentesco. Modificamos el atributo al objeto padre de *hijo* a ***padre***.

```
> hijo.__proto__ = padre
```

El objeto **hijo** no sabe responder al mensaje, sin embargo tiene la capacidad de delegar en su padre y dar una respuesta.

```
> hijo.capital_chile
  "Santiago"
```

Ejemplo en *Python*:

Python esconde su pertenencia al modelo basado en prototipos, utiliza la palabra reservada *class* pero en realidad es un objeto. Por lo tanto estamos creando dos prototipos nuevos, ***Padre*** e ***Hijo***.

```
> class Padre: pass
```

```
> class Hijo: pass
```

```
> hijo  = Hijo()
```

```
> Padre.capital_chile = "Santiago"
```

A través del atributo *__class__* accedemos su prototipo.

```
> hijo.__class__ = Padre
```

```
> hijo.capital_chile
"Santiago"
```

Ejemplo en *POOI*:

Sin profundizar en muchos detalles en este punto acerca de *POOI* podemos ver que el planteamiento es similar a los anteriores.

```
> (anObject copy) setName "Padre"
```

```
> Padre.capital_chile = "Santiago"
```

```
> (anObject copy) setName "Hijo"
```

```
> Hijo.parent = Padre
```

```
> Hijo.capital_chile
"Santiago"
```

Como se puede deducir a partir de estos ejemplos una de las claves de la versatilidad de los lenguajes pertenecientes a este modelo es la capacidad de manipular el atributo objeto padre de casi cualquier elemento en tiempo de ejecución.

En el fondo, el modelo basado en prototipos engloba al de clases. Es posible trabajar de forma idéntica con prototipos a como lo haríamos con clases sin embargo la relación no se cumple a la inversa.

2 POOI

La herramienta *POOI* es un intérprete de un lenguaje de objetos basado en prototipos con fines didácticos.

Enlaces de la aplicación:

- http://trevinca.ei.uvigo.es/~jgarcia/TO/pooi/
- http://trevinca.ei.uvigo.es/~jgarcia/software/

2.1 Dependencias

POOI necesita la máquina virtual de java para funcionar. Se puede obtener en el siguiente enlace: http://www.java.com/. La instalación corre a cargo de un asistente que guía al usuario a través de todo el proceso.

No se requiere configurar la máquina virtual de ningún modo específico.

2.2 Iniciar *POOI*

Una vez esté instalada la máquina virtual de java no hay más que hacer doble clic sobre *"Pooi.jar"*.

2.3 Interfaz de *POOI*

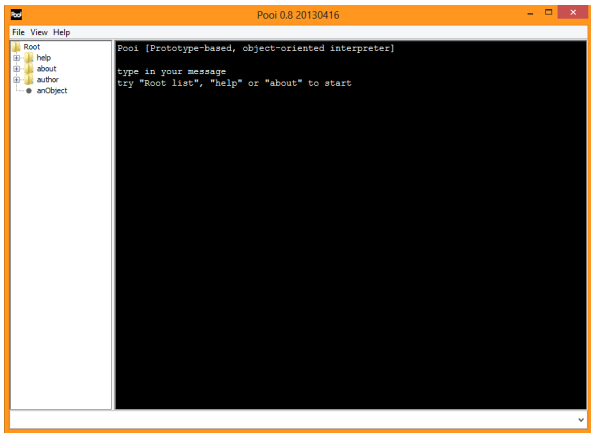


Imagen 1 – Ventana principal de *POOI*

Podemos dividir la interfaz en cuatro secciones:

- Barra de menú
- Árbol de objetos
- Entrada del usuario
- Salida del intérprete

2.3.1 Barra de menú

La barra superior contiene las siguientes opciones:

- File
  - Load session transcript
  - Save session transcript
  - Export object
  - Reset
  - Quit
- View
  - Font
- Help
  - Help content
  - About

La funcionalidad de alguna de estas opciones se tratará más adelante.

2.3.2 Árbol de objetos

El árbol de objetos contiene una lista de los objetos actuales dentro de la sesión. Haciendo clic sobre alguno de ellos se escribirá automáticamente en la sección de entrada del usuario la instrucción necesario para que el intérprete muestre el valor de los atributos de ese objeto.

2.3.3 Entrada del usuario

El usuario puede escribir instrucciones en el intérprete a través de la barra inferior.

2.3.4 Salida del intérprete

En este apartado el intérprete muestra lo que ha introducido el usuario y su respuesta.

2.4 Trabajando con *POOI*

Como hemos dicho antes *POOI* es un intérprete de un lenguaje basado en prototipos. Existe un prototipo padre (**Object**) que contiene una serie de métodos que serán comunes a todos los objetos. Esto es, cualquier objeto que creemos en *POOI* **hereda por delegación** una serie de operaciones básicas. Esto es posible ya que todos los prototipos guardan un enlace a su padre.

En pos de mantener separada la lógica común de esos métodos y evitar repetirlos en cada uno de los objetos no está permitido clonar este prototipo. Para solucinar esto existe un prototipo inicial **anObject** vacío cuyo padre es **Object**. Éste es el prototipo con el cual podremos empezar a trabajar.

2.4.1 Enviar un mensaje a un objeto

Todo en POOI es un objeto, la manera de interactuar con cada objeto es la siguiente:

```
<objeto> <método> <param1> <param2>
```

Ejemplo:

(Como antes, el símbolo '>' indica el comando que introduce el usuario)

```
> 5 + 5
10
```

Partimos de la base de que todo es un objeto por lo tanto podemos verlo de la siguiente forma, el objeto '5' llama a su método '+' y le pasa como parámetro otro objeto '5'.

Object (y por lo tanto el resto de objetos en *POOI*) tiene un método llamado *list* que devuelve todos los miembros de ese objeto.

Veámoslo:

Creamos un prototipo Persona.

```
> (anObject copy) setName "Persona"
```

A partir del prototipo persona construimos una persona en particular.

```
> (Persona copy) setName "p1"
```

Le damos un nombre.

```
> p1.nombre = "Pablo"
```

El objeto **p1** llama al método *list* y el intérprete nos muestra sus miembros, en este caso el nombre que le hemos dado y el atributo **parent** que apunta a su padre, el mismo que satisface la llamada a *list* ya que como también podéis ver, **p1** no tiene a *list* entre sus miembros.

```
> p1 list
Object p1 = {
    parent = Object Object : {}
    nombre = Str nombre : {}
        "Pablo"

}
```

Además de *list,* existe otro método que muestra el interior de cada objeto pero en este caso no nos enseña sus atributos si no el valor de esos atributos. Este método se llama *str*.

Probemos con el mismo objeto **p1** de antes.

```
> p1 str
p1 = { nombre = "Pablo" }
```

2.4.2 Tipos de datos soportados

Los tipos de datos que están definidos en POOI son los siguientes:

- Bool
- Int
- Real
- Str

Como no podía ser menos, los tipos de datos también son objetos. Por lo tanto es posible ver que operaciones implementan llamando a su método *list*. Introduzcámonos en cada uno de ellos.

2.4.2.1 Int

```
> Int list
Object Int = {
    - = {: }
    == = {: }
    + = {: }
    * = {: }
    > = {: }
    < = {: }
    neg? = {: }
    / = {: }
    abs = {: }
    parent = Object Object : {}

}
```

Veámos que hace cada uno de ellos:

El operador '-' resta y devuelve un nuevo objeto con el valor de la operación.

```
> 10 - 10
Root.bin.lit34 substracted from Root.bin.lit35, giving 0
```

El operador '==' devuelve 'true' si se cumple la igualdad, en caso contrario devuelve 'false'.

```
> 3 == 3
true
```

El operador '+' suma y devuelve un nuevo objeto con el valor de la operación.

```
> 4 + 6
Root.bin.lit40 and Root.bin.lit41 added, giving 10
```

El operador '*' multiplica y devuelve un nuevo objeto con el valor de la operación.

```
> 9 * 9
Root.bin.lit43 multiplied by Root.bin.lit44, giving 81
```

El operador '>' devuelve 'true' si el primer operando es mayor al segundo, en caso contrario devuelve 'false'.

```
> 9 > 1
true
```

El operador '<' devuelve 'true' si el primer operando en menor al segundo, en caso contrario devuelve 'false'.

```
> 5 < 8
true
```

El método *neg?* devuelve 'true' si el operando es un número negativo, en caso contrario devuelve 'false'.

```
> 5 neg?
false
```

El operador '/' divide y devuelve un nuevo objeto con el valor de la operación. Nótese que el resultado de la división siempre es un resultado entero, si la operación no es exacta se desprecia la parte decimal. [*Ver división en Real*]

```
> 4 / 3
Root.bin.lit54 divided by Root.bin.lit55, giving 1
```

El operador 'abs' hace el valor absoluto y devuelve un nuevo objeto con el valor de la operación.

```
> (3 - 5) abs
Root.bin.lit68 substracted from Root.bin.lit69, giving -2
2
```

2.4.2.2 Real

```
> Real list
Object Real = {
    - = {: }
    == = {: }
    + = {: }
    * = {: }
    > = {: }
    < = {: }
    neg? = {: }
    / = {: }
    abs = {: }
    parent = Object Object : {}

}
```

Veamos que hace cada uno de sus métodos:

El operador '-' resta y devuelve un nuevo objeto con el valor de la operación.

```
> 10.5 - 10.3
Root.bin.lit34 substracted from Root.bin.lit35, giving 0.2
```

El operador '==' devuelve 'true' si se cumple la igualdad, en caso contrario devuelve 'false'.

```
> 3.0 == 3.0
true
```

El operador '+' suma y devuelve un nuevo objeto con el valor de la operación.

```
> 4.3 + 6.7
Root.bin.lit40 and Root.bin.lit41 added, giving 11.0
```

El operador '*' multiplica y devuelve un nuevo objeto con el valor de la operación.

```
> 7.5 * 5.4
Root.bin.lit43 multiplied by Root.bin.lit44, giving 40.5
```

El operador '>' devuelve 'true' si el primer operando es mayor al segundo, en caso contrario devuelve 'false'.

```
> 9.5 > 9.0
true
```

El operador '<' devuelve 'true' si el primer operando es menor al segundo, en caso contrario devuelve 'false'.

```
> 5.2 < 7.8
true
```

El método *neg?* devuelve 'true' si el operando es un número negativo, en caso contrario devuelve 'false'.

```
> 7.9 neg?
false
```

El operador '/' divide y devuelve un nuevo objeto con el valor de la operación. En este caso el resultado sí es un número real.

```
> 4.0 / 3.0
Root.bin.lit54 divided by Root.bin.lit55, giving 1.333333
```

El operador 'abs' hace el valor absoluto y devuelve un nuevo objeto con el valor de la operación.

```
> (3 - 5) abs
Root.bin.lit68 substracted from Root.bin.lit69, giving -2
2
```

2.4.2.3 Str

```
> Str list
Object Str = {
    sub = {: }
    num? = {: }
    == = {: }
    + = {: }
    at = {: }
    left = {: }
    trim = {: }
    > = {: }
    < = {: }
    lower = {: }
    int = {: }
    len = {: }
    empty? = {: }
    real = {: }
    right = {: }
    upper = {: }
    parent = Object Object : {}

}
```

El método *sub* devuelve el trozo de cadena contenido entre las posiciones que recibe como argumento. La posición dos no se incluye.

```
> "hola mundo" sub 0 2
ho
```

num? Devuelve ‘true’ si la cadena es un número, en caso contrario devuelve ‘false’.

```
> "5" num?
true
```

El operador ‘==’ devuelve ‘true’ si se cumple la igualdad, en caso contrario devuelve ‘false’.

```
> "j" == "j"
true
```

El operador ‘+’ concatena dos cadenas y devuelve un nuevo objeto con la cadena resultante.

```
> "hola" + " mundo"
Root.bin.lit114 and Root.bin.lit115 added, giving '"hola mundo"'
```

El método *at* devuelve el carácter en la posición que recibe como argumento.

```
> "hola" at 3
a
```

El método *left* devuelve la cadena contenido desde la posición hasta la posición que recibe como argumento menos uno.

```
> "hola" left 2
ho
```

El método *trim* elimina los espacios al principio de la cadena.

```
> " hola" trim
Hola
```

El operador ‘>’ devuelve ‘true’ si la codificación del primer operando es mayor a la codificación del segundo, en caso contrario devuelve ‘false’.

```
> "a" > "b"
false
```

El operador ‘<’ devuelve ‘true’ si la codificación del primer operando es menor a la codificación del segundo, en caso contrario devuelve ‘false’.

```
> "a" < "b"
true
```

El método *lower* transforma la cadena a su versión en letras minúsculas.

```
> "A" lower
A
```

El método int devuelve el entero correspondiente a la cadena, si la cadena no es un numero devuelve menos uno.

```
> "5" int
5
```

El método len devuelve la longitud de la cadena.

```
> "hola" len
4
```

El método *empty?* devuelve 'true' si la cadena está vacía, en caso contrario devuelve 'false'.

```
> "" empty?
True
```

El método *right* devuelve el trozo de cadena desde la última posición a la última menos el número que recibe como argumento.

```
> "hola" right 2
la
```

El método *real* devuelve el número real correspondiente a la cadena, si la cadena no es equivale a un número devuelve -1.0.

```
> "1.0" real
1.0
```

El método upper convierte la cadena en su versión en letras mayúsculas.

```
> "a" upper
A
```

2.4.3 Métodos de Object

Podemos ver la lista de miembros a través del método *list.*

```
> Object list
Object Object = {
    is? = {: }
    copy = {: }
    name = {: }
    erase = {: }
    setName = {: }
    list = {: }
    str = {: }
    set = {: }
    path = {: }
    createChild = {: }

}
```

Breve descripción de cada uno de ellos.

- is?: Devuelve 'true' si el objeto que llama al método y el que recibe como argumento son iguales o si deriva de éste.
- copy: Crea un objeto nuevo a partir del objeto que invoca a la función copiando todos sus atributos y métodos. El atributo *parent* del nuevo objeto a punta al objeto a partir del cual ha sido creado.
- name: Devuelve el nombre del objeto.
- erase: Elimina el atributo o método que recibe como argumento del objeto que llama a la función.
- setName: Modifica el nombre del objeto.

- list: Devuelve una lista con todos los miembros del objeto.
- str: Devuelve el valor de todos los atributos contenidos en el objeto.
- set: Modifica el valor del atributo del objeto que invoca el método que recibe como primer argumento al valor del segundo argumento.
- path: Devuelve el identificador completo del objeto que lo indentifica de forma unívoca.
- createChild: Crea un objeto nuevo a partir del objeto que llama a la función. Los miembros no se copian en el objeto hijo. El atributo *parent* del nuevo objeto a punta al objeto a partir del cual ha sido creado.

Veámos como funcionan estos métodos a través de un ejemplo:

copy devuelve una copia de anObject, un clón exacto con todos sus miembros.

```
> anObject copy
anObject was copied into Root as 'Root.anObject1'
```

setName modifica el nombre de anObject a **Padre**.

```
> anObject1 setName "Padre"
Root.anObject renamed to Root.Padre
```

name devuelve el nombre de **Padre**.

```
> Padre name
Padre
```

De esta manera añadimos un miembro nuevo a **Padre**.

```
> Padre.capital_chile = "Santiago"
'capital_chile' set in Root.Padre
```

erase borra el miembro que se le pasa como parámetro.

```
> Padre erase "capital_chile"
'capital_chile' erased from 'Padre'.
```

str muestra el contenido de los miembros de Padre, al haber borrado capital_chile ahora está vacío.

```
> Padre str
Padre = { }
```

Volvemos a añadir el miembro anterior.

```
> Padre.capital_chile = "Santiago"
'capital_chile' set in Root.Padre
```

Ahora *str* sí que nos muestra el contenido de capital_chile.

```
> Padre str
Padre = { capital_chile = "Santiago" }
```

set modifica un miembro al valor que le pasemos como segundo parámetro.

```
> Padre set "capital_chile" "Santiago de chile"
'capital_chile' set in Root.Padre

> Padre str
Padre = { capital_chile = "Santiago de chile" }
```

path devuelve el identificador de objeto Padre.

```
> Padre path
Root.Padre
```

createChild crea un un nuevo objeto a partir de **Padre** pero sin copiar sus métodos.

```
> (Padre createChild) setName "Hijo"
'Root.Padre2' was created as a child of 'Root.Padre'
Root.Padre2 renamed to Root.Hijo
```

Tambien podemos crear un **Hijo** copia de **anObject** y a posteriori modificar el valor del atributo parent a **Padre**.

```
(anObject copy) setName "Hijo"
Hijo.parent = Padre
'parent' set in Root.Hijo
```

Devuelve 'true' si los objetos son iguales.

```
> Padre is? Hijo.parent
true
```

También devuelve 'true' si el objeto que invoca al método es derivado del objeto que recibe como argumento.

```
> Hijo is? Padre
true
```

Al listar los miembros de **Hijo** vemos que no tiene capital_chile.

```
> Hijo list
Padre Hijo = {
    parent = Object Padre : {}

}

> Hijo.capital_chile str
"Santiago de chile"
```

2.4.4 Añadir miembros

Para añadir un atributo nuevo se hace lo siguiente:

```
objeto.nombre_nuevo_atributo = valor_atributo
```

Un ejemplo concreto:

```
> anObject.nuevo_miembro = 10
'nuevo_miembro' set in Root.anObject
```

Para añadir un método la sintaxis es la siguiente:

```
objeto.nuevo_metodo  = { param1 param2 : param1 + param2 }
```

El método va entre corchetes, los parámetros se separan por un espacio y el operador ':' delimita la separación entre parámetros y lógica de la función. El resultado de la expresión que va a la derecha del operador ':' es el objeto que devuelve el método.

Por ejemplo:

```
> anObject.suma = {a b: a + b}
'suma' set in Root.anObject

> anObject suma 10 5
Root.bin.lit4 and Root.bin.lit5 added, giving 15
```

2.4.5 Creación de objetos

Existen dos métodos que nos permiten copiar un objeto y obtener una copia idéntica.

2.4.5.1 Método copy

El método *copy* clona y devuelve un objeto completamente idéntico a su padre. Hace una copia de **atributos** y **métodos**.

```
> (anObject copy) setName "Padre"

> Padre.color_ojos = "azul"

> Padre.get_color_ojos = {: self.color_ojos str}

> (Padre copy) setName "Hijo"
```

Hijo contiene el atributo *color_ojos* y el método *get_color_ojos*, exactamente igual que su padre.

```
> Hijo list
Object Hijo = {
    get_color_ojos = {: ( self.color_ojos str );  }
    parent = Object Object : {}
    color_ojos = Str color_ojos : {}
        "azul"

}
```

2.4.5.2 Método createChild

Veamos la diferencia si ahora creamos el objeto **Hijo** con el método *createChild*.

```
> (Padre createChild) setName "Hijo"
```

Ahora **Hijo** no presenta entre sus miembros los atributos y métodos de **Padre**.

```
> Hijo list
Padre Hijo = {
    parent = Object Padre : {}

}
```

Sin embargo puede acceder a ellos delegando en él.

```
> Hijo.color_ojos
```

```
"azul"
```

Esto es fundamental para cumplir una de las bases de la programación basada en prototipos: la separación de elementos entre un prototipo padre y su hijo. Esto evita la copia innecesaria de miembros. ¿Es necesario que todos los objetos de *POOI* tengan entre sus miembros el método *list*? Todo saben responder a mensaje gracias a la herencia por delegación y además hemos evitado su copia en cada uno de ellos lo cual supondría un problema de eficiencia.

2.4.6 Sobreescribir un método ya existente

Podemos redifinir un método heredado por delegación. De esta forma el objeto responde a ese mensaje por el mismo. Esto puede ser últi si queremos darle un comportamiento específico a algún elemento.

Por ejemplo:

```
> (anObject copy) setName "Persona"

> Persona.nombre = "Pepe"
'nombre' set in Root.Persona

> Persona.apellido = "Garcia"
'apellido' set in Root.Persona

> Persona str
Persona = { apellido = "Garcia" nombre = "Pepe" }

> Persona.str = {: ((self.nombre + " ") + self.apellido) str}
'str' set in Root.Persona

> anObject str
"Pepe Garcia"
```

2.4.7 ¡Hola mundo! En POOI

El siempre presente hola mundo:

```
> (anObject copy) setName "HolaMundo"

> HolaMundo.saludo = {: "¡Hola mundo!"}

> HolaMundo saludo
"¡Hola mundo!"
```

2.4.8 Otro ejemplo más, el prototipo Punto

Vamos a calcular la distancia Manhattan entre dos puntos. De la misma forma que ***Object*** contiene métodos generales que no van a ser copiados en sus descendientes podemos plantear un escenario similar con **Punto**.

Creamos un objeto **TraitsPunto** y le añadimos el método distancia, método que no estará presente en cada punto, sin embargo podrán recurrir a él delegando en su padre.

```
> (anObject copy) setName "TraitsPunto"

> TraitsPunto.distancia = { x y : ((self.x - x) abs) + ((self.y - y)
abs) }
```

Creamos un prototipo **Punto** a partir de **TraitsPunto**.

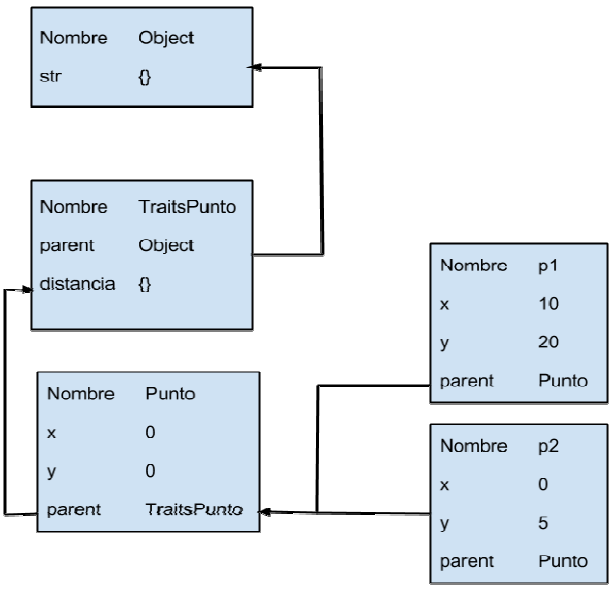
```
> (TraitsPunto createChild) setName "Punto"
```

Ahora, a partir del prototipo **Punto**, construimos puntos particulares.

```
> (Punto copy) setName "p1"

> (Punto copy) setName "p2"

> p1.x = 10

> p1.y = 20

> p2.x = 0

> p2.y = 5
```

Ahora, **p1** puede calcular la distancia con **p2** y hemos mantenido separada la lógica de esa función de cada punto particular.

```
> p1 distancia p2.x p2.y
25
```



El diagrama muestra como **p1** y **p2** escalan a través de **Punto** hasta **TraitsPunto** para responder al mensaje *distancia*.

2.4.9 Otras opciones de *POOI*

2.4.9.1 Load/Save session transcript

Esta opción permite salvar el trabajo de una sesión para continuar más tarde con él o como ejemplo didáctico para otro usuario. Cargar una sesión implíca recuperar una por una cada una de las instrucción de la sesión que hemos guardado que además serán ejecutadas por lo que también recuperaremos todos los objetos.

2.4.9.2 Export objects

POOI ofrece la posibilidad de exportar el estado de cada uno de los objetos a un formato *JSON*. *JSON* es una notación de objetos utilizada en *Javascript* que sigue el siguiente formato.

```
objeto = { "atributo1" : "valoratributo1",
           "atributo2" : "valoratributo2" }
```

Veámos como funciona en profundidad :

Creamos un objeto **Humano** y le añadimos un atributo.

```
> (anObject copy) setName "Humano"

> Humano.especie = "homo sapiens"
'especie' set in Root.Humano
```

Ahora nos vamos a *file -> export objects*, se nos pedirá que escojamos los objetos que queremos exportar y a continuación nos pedirá una ruta para guardar el archivo. Una vez hecho esto *POOI* genera un archivo con la representación en *JSON* del objeto.

En este caso tenemos lo siguiente:

```
Humano={"__name__":"Humano",
        "especie":"homo sapiens",
        "parent":"Object"}
```

Véamos la representación de nuestro archivo JSON utilizando la herramienta online *jsonviewer*. Página del proyecto http://jsonviewer.stack.hu/.

Cargamos nuestros archivo *JSON* y le damos a “Viewer”.

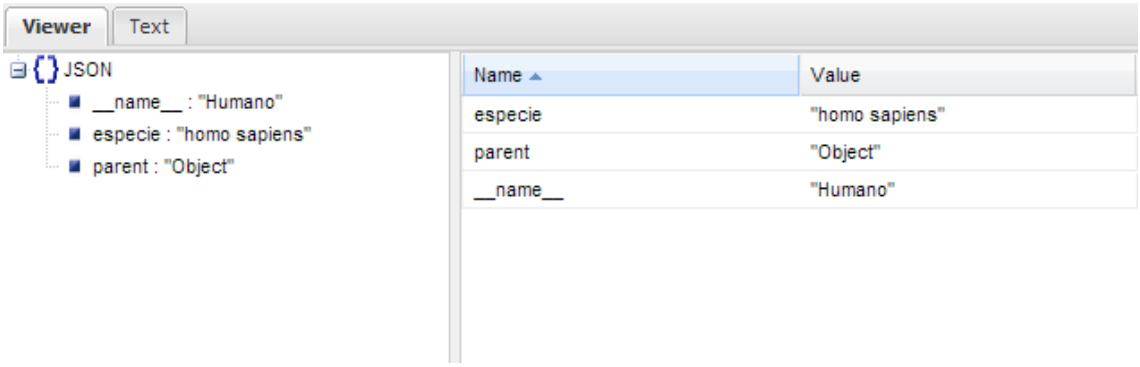


Imagen 2 – Representación de un objeto escrito en representación JSON

Se puede ver como la notación *JSON* describe el estado de un objeto.