Guía 3: Referencias
Programación con Objetos 1
Versión del 10/08/2018

En estos ejercicios vamos a concentrarnos en cómo quedan las *referencias* entre objetos, o sea, cómo se conocen los objetos que interactúan para resolver un enunciado.

En esta guía, además de programar, vamos a dibujar en papel cómo quedan las referencias entre objetos, y cómo cambian al hacer modificaciones en un REPL. También vamos a hacer cálculos que nos van a ayudar a entender cómo están interactuando los objetos entre ellos cuando enviamos cierto mensaje inicial desde el REPL.

Ejercicio 1: Entrenamiento de Pepita y otros (continuación)

Retomamos el ejercicio sobre entrenamiento de la guía 2.

Parte 1: *Susana, otra entrenadora*
Primero agreguemos un segundo entrenador, en rigor una entrenadora, Susana. La rutina de Susana consiste en:

1. comer 100 gramos de alpiste
2. volar 20 km

y nada más.

Susana también tiene que acordarse qué ave está entrenando, y entender el mensaje `entrenar()`.

Agregar en Roque y en Susana el mensaje `aQuienEntrenas()`, que devuelva el ave que se le pasó a cada uno para que entrene.

Parte 2: *Primer prueba con referencias*
Levantar un REPL nuevito y hacer lo siguiente

1. darle 200 gramos de comer a Pepita y a Pepón,
2. decirle a Roque que entrene a Pepita, y a Susana que entrene a Pepón,
3. fijarse la energía de Pepita y de Pepón,
4. dibujar el grafo de referencias entre objetos en este momento,
5. hacer la cuenta de cómo debería quedar la energía de Pepita y de Pepón después de decirles a Roque y a Susana que entrenen,
6. ejecutar `roque.entrenar()` y luego `susana.entrenar()`

7. ver si la energía de Pepita y de Pepón quedó como pensaban.

¡OJO! **no cerrar** el REPL, vamos a seguir usándolo en la parte que sigue.

Parte 3: *Todos con Pepita*
Sobre el mismo REPL de la parte anterior, que por supuesto no cerramos, hacer lo siguiente

1. decirle a Susana que ahora tiene que entrenar a Pepita,
2. dibujar cómo quedó el grafo de referencias entre objetos después de este cambio,
3. hacer la cuenta de cómo debería quedar la energía de Pepita y de Pepón después de decirles a Roque y a Susana que entrenen,
4. ejecutar `roque.entrenar()` y luego `susana.entrenar()`
5. ver si la energía de Pepita y de Pepón quedó como pensaban. ¿Quién cambió, quién quedó igual, por qué?
6. pensar cómo puedo darme cuenta, desde el REPL, que están entrenando a la misma ave. Ayuda: usar el mensaje `aQuienEntrenas()` de la parte 1.

¡OJO! otra vez te pedimos **no cerrar** el REPL.

Parte 4: *Accedemos a las aves entrenadas*
Siempre sobre el mismo REPL, hacer lo siguiente

1. pensar qué resultado se obtiene si ponemos `roque.aQuienEntrenas().energia()` en el REPL. Probarlo, y ver si da lo que pensabas. Hacer lo mismo con `susana.aQuienEntrenas().energia()`.
2. cambiar a Susana, para que vuelva a entrenar a Pepón. Repetir el punto anterior, ¿cambió algún resultado, por qué?
3. volver a cambiar a Susana, para que ahora entrene a Pipa. Pensar qué va a pasar ahora si ponemos en el REPL `susana.aQuienEntrenas().energia()`. Ejecutarlo, y ver si pasa lo que pensaste. Si pasa algo distinto, explicate qué pasó.

Ejercicio 2: Cuentas bancarias (continuación)

Retomamos el ejercicio sobre cuentas bancarias de la guía 2.

Parte 1: *Agregar las cosas que se compran*
Agregar al modelo distintas cosas que la casa de Julián y Pepe puede comprar. De cada cosa nos interesa el precio, si es comida o no, y si es un electrodoméstico o no. Incluir: una heladera que vale 20000 pesos, una cama que sale 8000, una tira de asado que sale 350 pesos, un paquete de fideos que sale 50 pesos, y una plancha que vale 1200 pesos. Por las dudas aclaramos: la cama no es un electrodoméstico, la plancha sí.

En el objeto que representa a la casa, separar `gastar(importe)` que simplemente retira de la cuenta el importe indicado, y `comprar(cosa)`. Obviamente, comprar una cosa implica gastar su precio. Lograr que este objeto entienda

- `tieneComida()` si compró algo que es comida al menos una vez,
- `vieneDeEquiparse()` si la última cosa que se compró es un electrodoméstico, o bien vale más de 5000 pesos. OJO acá, no hace falta que la casa se acuerde de dos cosas, alcanza que se acuerde de una sola, ¿de qué?
- `puedeComprar(cosa)`, si puede afrontar la compra de la cosa, de acuerdo a su precio.
- `cuentaParaGastos()`, la cuenta usada para compras y otros gastos.

Parte 2: *Jugamos con los objetos construidos*
Levantar un REPL nuevito, y hacer lo siguiente

1. Depositar 8000 pesos en la cuenta de Pepe, 15000 en la de Julián, y 45300 (que equivalen a 3000 dólares) en la del padre.
2. Asignar como cuenta primaria y secundaria de la cuenta combinada, las cuentas de Pepe y de Julián respectivamente.
3. Configurar la casa de Pepe y Julián para que use la cuenta combinada.
4. Hacer que la casa compre, en este orden, la plancha y la cama.
5. Dibujar el grafo de referencias entre objetos a esta altura. Incluir todos los objetos definidos, incluso los que quedan "sueltos" en el grafo.
6. Calcular cuánto debería ser el saldo de la cuenta combinada después de todo esto, y verificar que el saldo es el calculado.
7. Preguntarle a la casa si viene de equiparse, y si puede comprar la heladera, ¿qué responde? ¿Podés explicar por qué?

¡OJO! como en el ejercicio anterior, te pedimos **no cerrar** el REPL, vamos a seguir usándolo en la parte que sigue.

Parte 3: *Cambios en la configuración*
Con el mismo REPL de antes, hacer lo que sigue

1. Preguntar `casaDePepeYJulian.cuentaParaGastos().saldo()`. Este es el saldo de ¿cuál de las cuentas, por qué? Anotar el valor que da. Marcar en el grafo los objetos que intervienen para obtener la respuesta a esta consulta.
2. Cambiar la configuración de la casa de Pepe y Julián, para que use la cuenta del padre.

3. Dibujar el grafo de referencias como queda después de este cambio. Incluir todos los objetos definidos.

4. Si ahora le pregunto a la casa si puede comprar la heladera, ¿qué me responde, lo mismo que antes o cambió? Explicar el cambio a partir de mirar el grafo de referencias.

5. Volver a cambiar la cuenta que usa la casa, hacer que use la de Julián. Comprar el paquete de fideos.

6. Volver a dibujar el grafo de referencias. Mirándolo, pensar las respuestas que deberían obtenerse si preguntamos el saldo de la cuenta combinada, y si la casa viene de equiparse. Probar a ver si las respuestas que se obtienen son las esperadas.

Ejercicio 3: Sueldo de pepe (continuación)

Retomamos el ejercicio sobre el sueldo de Pepe, de la guía 2.

Parte 1: *Armamos una configuración*
Levantar un REPL nuevito, configurar a Pepe para que sea gerente, tenga descuento por sindicato como "comprometido", y bono por presentismo normal. Hacer que falte una vez.

Dibujar el grafo de referencias entre objetos como quedó.

Calcular el sueldo que debería tener Pepe, preguntar, ver que dio lo que se esperaba.

Después, hacer un test Wollok que pruebe esto, el assert es sobre el sueldo.

Parte 2: *Más gente*
Agregar al mismo modelo a las siguientes personas

- Roque, que tiene 28000 fijo de neto. El sueldo se calcula como neto - descuento por sindicato + 9500 pesos. Para los descuentos por sindicato, se usan las mismas opciones que para Pepe.
- Ernesto, que trabaja junto con un compañero, que puede cambiar. El neto de Ernesto es igual al de su compañero. Su sueldo se calcula como neto + bono por presentismo. Para los bonos, usar las mismas opciones que para Pepe. Se sabe que Ernesto no falta nunca.

Después de agregar estos objetos, armar un test Wollok en el que Pepe esté configurado como en el test anterior, Roque tenga descuento por sindicato como "comprometido", Ernesto tenga como compañero a Pepe, y como bono el de "época de ajuste". El assert sobre este test son los sueldos de Roque y de Ernesto.

Dibujar el grafo de referencias entre objetos. Marcar los objetos que intervienen en el cálculo del sueldo de Ernesto.

Ejercicio 4: Tuberías

Hacer un modelo sencillo de una tubería que tiene varios tramos. Modelar, al menos, estos tramos:

- un tramo recto largo, de 2 metros,
- un tramo recto corto, de 40 cm,
- un codo, cuya longitud es 70 cm,
- una bifurcación en Y, cuya longitud es 50 cm. Pensar que tiene una entrada y dos salidas,
- un tramo en subida de 1 metro de longitud,
- un tramo en bajada de 2 metros de longitud, y
- un tope cuya longitud es 0.
- una canilla cuya longitud es 10 cm.

Cada uno de los tramos, salvo el tope, la canilla y la bifurcación, deben conocer al tramo que le sigue. Para esto, lograr que entiendan el mensaje `siguiente()`. P.ej. si ensamblo tramo recto largo, codo, tramo en subida y tramo recto corto, en ese orden, entonces:

- si le pregunto al tramo recto largo cuál es el siguiente, debe responder "el codo",
- si le pregunto al codo, me tiene que decir "el tramo en subida",
- si le pregunto al tramo en subida, me tiene que decir "el tramo recto corto".

La bifurcación debe conocer a dos siguientes, el `siguienteIzquierda()` y el `siguienteDerecha()`.

El tope y la canilla no tienen siguiente, y se usan para terminar una tubería.

Todos estos objetos deben entender el mensaje `longitudMaximaHastaElFinal()`, que tenga en cuenta la longitud del tramo al que se le pregunta, más todos los tramos siguientes. Manejar las longitudes en centímetros.

En el ejemplo de tramo recto largo, codo, tramo en subida y tramo recto corto, si pongo
`>>>tramoRectoLargo.longitudMaximaHastaElFinal()`
debería responder 410, y si pregunto
`>>>tramoEnSubida.longitudMaximaHastaElFinal()`
debería responder 140.
Atención, si pregunto `>>>tope.longitudMaximaHastaElFinal()` debe responder 0 siempre, porque el tope no tiene siguiente.

En el caso de la bifurcación, es su longitud propia que es 50 cm, más el máximo de las longitudes hasta el final a su izquierda y a su derecha. Recordar los mensajes `max` y `min` que entienden los números.

Ayuda: para p.ej. el tramo recto largo, la longitud máxima hasta el final es: 2 metros más la longitud máxima hasta el final del siguiente. O sea, el tramo recto largo le puede mandar *el mismo* mensaje a su siguiente, y a lo que da, sumarle 200. Para la bifurcación es parecido, pero hay que obtener el mensaje a los dos siguientes, de esos el máximo, y a eso sumarle 50.

Estos objetos también deben entender `velocidadMaximaDeSalida(velocidadEntrada)`, que responden a qué velocidad sale de la tubería un objeto que entra al tramo al que se le pregunta a la velocidad que se dice. El codo le resta 20 a la velocidad, la canilla resta 5, la subida la baja a la mitad, y la bajada la duplica. El tope deja la velocidad en 0. Los tramos rectos y la bifurcación no modifican la velocidad. En el caso de la bifurcación, tomar el máximo entre las velocidades que se obtienen preguntándole a cada salida.

En el ejemplo de tramo recto largo, codo, tramo en subida y tramo recto corto, si pregunto
`>>>tramoRectoLargo.velocidadMaximaDeSalida(70)`
debería responder 25 (del codo salen 50 porque resta 20, de la subida salen 25, que es la mitad de los 50 que entraron, y del tramo recto corto salen los 25 que entraron ahí).

Probar el circuito de los ejemplos en un test Wollok. También probar uno así:

- conectar la bifurcación en Y a la salida de la subida,
- a la izquierda de la Y, poner tramo recto largo, codo y tope,
- a la derecha de la Y, poner bajada y tramo recto corto

En ambos casos, dibujar el grafo de referencias entre objetos, y entender cómo interactúan los objetos para resolver cada consulta.

Nota
sí, los objetos que hay que definir tienen muchas cosas en común. Por ahora vamos a copiar y pegar mucho. P.ej. muchos de los objetos van a tener algo así:

```
var siguiente

method siguiente() {
 return siguiente
}

method siguiente(elSiguiente) {
 siguiente = elSiguiente
}
```

Más adelante en la materia veremos formas de no tener esta repetición de código ... que no son perfectas. Se van a ver alternativas más sutiles ... en Objetos 3.

Ejercicio 5: Juego con personajes

Nos piden modelar un juego en el que cada jugador maneja un conjunto de personajes (p.ej. guerreros, trabajadores, sacerdotes). En el juego hay distintos elementos (p.ej. casas, animales, ríos). Cuando un personaje se encuentra con un elemento, hace cosas que pueden afectar a ambos.

Como recién estamos empezando a aprender programación con objetos, vamos a modelar una situación muy reducida, en la que aparecen solamente estos objetos:

- `luisa`, una jugadora,
- `floki`, un personaje guerrero,
- `mario`, un personaje trabajador,
- `ballesta` y `jabalina`, dos armas. `floki` tiene una de estas armas.
- `castillo`, `aurora` (que es una vaca) y `tipa` (que es un árbol), tres elementos. De cada elemento nos va a interesar el alto. El `castillo` mide 20 metros, `aurora` mide 1 metro, la `tipa` arranca en 8 metros pero puede crecer (ya veremos cómo).

El programa debe resolver el encuentro entre un personaje y un elemento.

Cuando `floki` encuentra un elemento pasan dos cosas.
Primero, el elemento recibe un ataque. La fuerza del ataque es la potencia del arma que tenga encima `floki` en ese momento. Segundo, se registra que el arma fue usada.
Pero ¡atención!, todo esto pasa solamente si el arma está cargada. Si no, no se hace nada.

Para resolver todo esto, `floki` tiene este método:

```
method encontrar(unElemento) {
    if (arma.estaCargada()) {
        unElemento.recibirAtaque(arma.potencia())
        arma.registrarUso()
    }
}
```

Obviamente, `floki` tiene un atributo que es su arma.

Qué pasa con cada elemento cuando recibe un ataque:

- El `castillo` disminuye su nivel de defensa en la potencia del ataque (p.ej. si recibe un ataque de 30, disminuye el nivel de defensa en 30). El nivel de defensa del castillo nace en 150.
- `aurora` muere si la potencia del ataque es 10 o más; si no, no le pasa nada. `aurora` sabe en todo momento si está viva o no. Obviamente, nace viva.
- a `tipa` no le pasa nada.

Cómo funcionan las armas

- La `ballesta` nace con 10 flechas. Cada vez que se usa, pierde una flecha. Está cargada si tiene flechas. Su potencia es 4.
- La `jabalina` nace cargada. Se puede usar solamente una vez, o sea, con el primer uso deja de estar cargada. Su potencia es 30.

Entonces, p.ej., si `floki` se encuentra al `castillo` que tiene 130 de defensas (porque ya recibió antes otros ataques), y su arma es la `ballesta` que tiene 7 flechas, lo que pasa es que: la defensa del `castillo` pasa a 126, y la `ballesta` queda con 6 flechas.

Ahora veamos qué pasa cuando `mario` encuentra un elemento. Son tres cosas:

- El valor recolectado por `mario` aumenta en un valor que depende del elemento.
- El elemento recibe el trabajo que hace `mario` sobre él.
- ... la tercer cosa que tiene que pasar la tenés que descubrir vos

El método en `mario` queda así:

```
method encontrar(unElemento) {
    valorRecolectado += elemento.valorQueOtorga()
    elemento.recibirTrabajo()
    // ... acá hay que agregar una línea ...
}
```

`mario` tiene un atributo `valorRecolectado`, que nace en 0. Veamos cómo es cada elemento respecto del valor y del trabajo:

- El `castillo` otorga como valor el 20 % de su defensa (o sea, su defensa / 5). Al recibir un trabajo, aumenta su defensa en 20, hasta un tope de 200. O sea, si tiene 192 no pasa a 212, queda en 200 (ayuda: usar `min`).
- `aurora` otorga como valor 15 unidades. Al recibir un trabajo, no le pasa nada.
- La `tipa` otorga como valor el doble de su altura. Al recibir un trabajo, su altura crece en un metro (porque se supone que la riegan y le dan nitratos, ponele).

Ah, otra cosa. Se le tiene que poder preguntar a `mario` si es feliz o no. `mario` es feliz si: o bien recolectó en total al menos 50 unidades, o bien el último elemento con el que se encontró mide al menos 10 metros de alto (atenti: la acción que falta en el método que pusimos de `mario` tiene que ver con esto).

Finalmente, se supone que en cada momento del juego, cada jugador (en nuestro modelo reducido tenemos a `luisa`) está manejando uno de sus personajes, es el *personaje activo*. En este modelo reducido, el personaje activo de `luisa` puede ser `floki` o `mario`.

El objeto `luisa` debe entender el mensaje `aparece(unElemento)`. Cuando le llega este mensaje, `luisa` le dice a su personaje activo que se encuentre al elemento. Si no tiene ningún personaje activo, debe lanzar un error. Luisa no nace con ningún personaje activo, hay que asignarle uno.

Se pide programar los 8 objetos indicados, y armar los siguientes tests.

1. `luisa` tiene como personaje activo a `mario`. A `luisa` le aparece primero `aurora`, y después el `castillo`. Después de esto, verificar que: el valor total recolectado por `mario` es 45, `mario` es feliz (porque lo último que se encontró, el `castillo`, tiene 20 metros de alto), y el `castillo` tiene 170 de defensa.

2. `floki` tiene como arma la jabalina. `luisa` tiene como personaje activo a `mario`. A `luisa` le aparece la `tipa`. Después el personaje activo cambia a `floki`, y después de este cambio, a `luisa` le aparecen primero `aurora` y después el `castillo`. Verificar que: el valor total recolectado por `mario` es 16, la altura de la `tipa` es 9 metros, `mario` no es feliz, `aurora` está muerta, la defensa del `castillo` es 150, y la `jabalina` no está cargada.

3. `floki` tiene como arma la ballesta, `luisa` tiene como personaje activo a `floki`. A `luisa` le aparecen primero `aurora` y después el `castillo`. Verificar que: el valor total recolectado por `mario` es 0, `aurora` está viva, la defensa del `castillo` es 146, la `ballesta` está cargada, y le quedan 8 flechas.

4. A `luisa` le aparece `aurora`, sin que se le haya cargado un personaje activo. Verificar que ocurre un error.

Para los primeros dos tests, dibujar el grafo de referencias como queda al final, después de ejecutar todo el test.

Ayudines

Este ejercicio tiene mucho polimorfismo. Veamos:

- Los personajes, `floki` y `mario`, son polimórficos para `luisa`. ¿Por qué? Porque cuando le dicen a `luisa` que aparece un elemento, `luisa` tiene que decirle a su personaje activo (que puede ser cualquiera de los dos) que se lo encuentre.

- Los elementos son polimórficos para `floki`. Esto se ve en el método de `floki` que detallamos, todos los elementos deben entender el mensaje `recibirAtaque(potencia)`. En forma similar, los elementos son polimórficos para `mario`, obviamente no respecto de `recibirAtaque(potencia)`. ¿Qué mensajes les `mario` a un elemento, que obliga a los elementos a ser polimórficos para `mario`?

- Las armas también son polimórficas para `floki`, ya que les pide la `potencia()`, les pregunta si `estaCargada()`, y les avisa que deben `registrarUso()`.

Si se usa bien el polimorfismo, los únicos `if` que debería haber son: el que mostramos en el método de `floki`, y el que necesita `luisa` para lanzar un error. La condición de error es que el personaje activo sea `== null`.

¡Ah! una aclaración final. Los métodos `encontrar(unElemento)` en `floki` y `mario`, van exactamente como los pusimos. Y los objetos que hay que implementar son exactamente los 8 que decimos, ni más, ni menos. En cada uno de los gráficos debe haber al menos 8 globitos, se pueden agregar algunos que corresponden a números o booleanos. No se olviden de poner **todos** los atributos de un objeto en el globito, los que apuntan a otros "objetos que hicimos nosotros", y *también* los que apuntan a números o booleanos.