Ejercicio 4 – celulares

a. Se pide representar con objetos a personas que hablan entre sí por celulares. Juliana tiene un Motorola U9, y Catalina tiene un iPhone. El Motorola U9 pierde 0,25 "puntos" de batería por cada llamada, y el iPhone pierde 0,1% de la duración de cada llamada en batería. Ambos celulares tienen 5 "puntos" de batería como máximo.

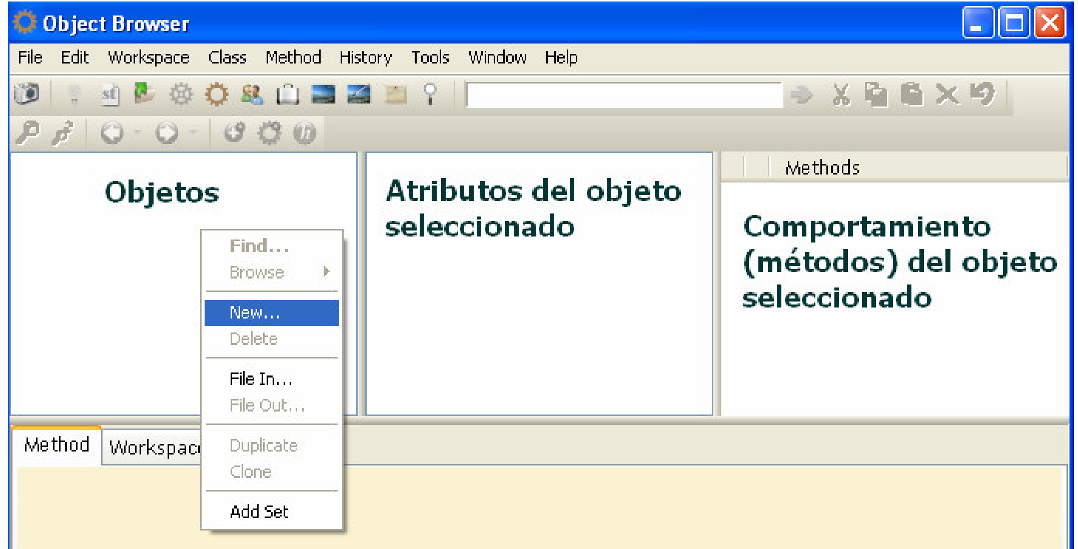
Implementar a Juliana, Catalina, el Motorola U9 de Juliana y el iPhone de Catalina en el ObjectBrowser y hacer un workspace en donde Juliana y Catalina se hagan llamadas telefónicas de distintas duraciones.

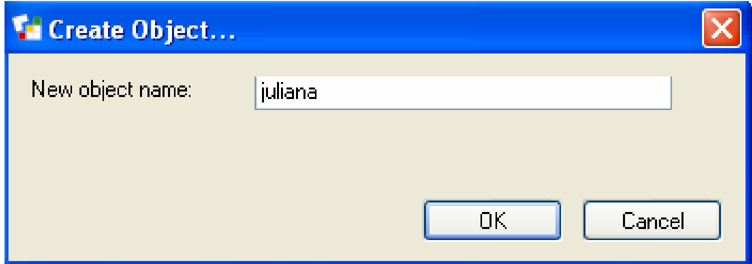
- Conocer la cantidad de batería de cada celular.
- Saber si un celular está apagado (si está sin batería).
- Recargar un celular (que vuelva a tener su batería completa).
- Saber si Juliana tiene el celular apagado; saber si Catalina tiene el celular apagado.

Solución

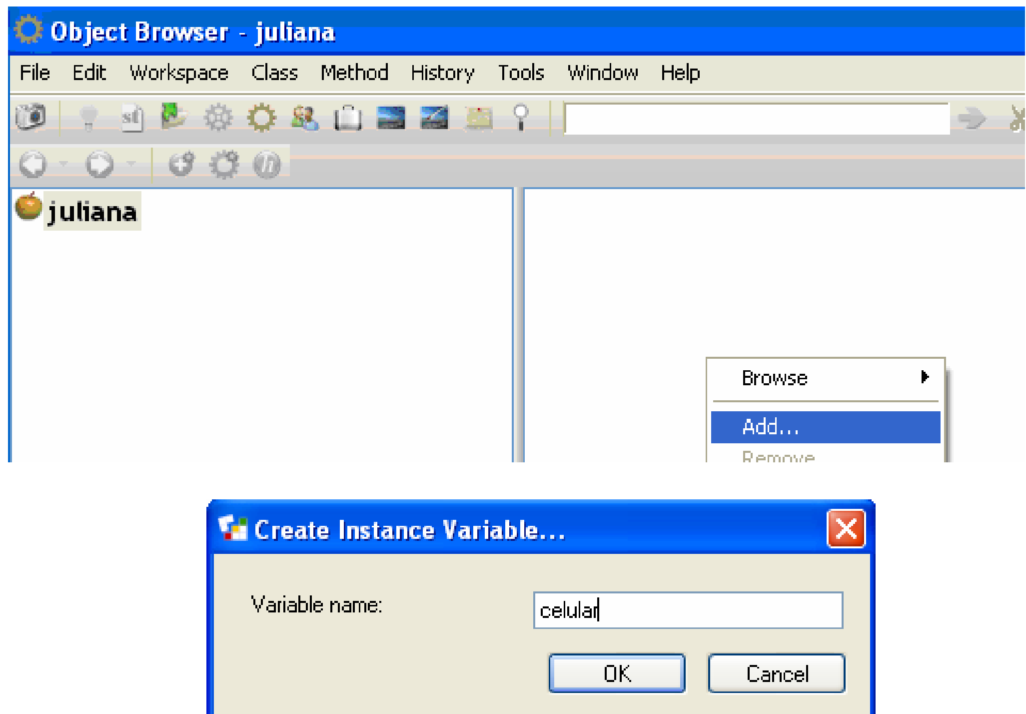
Creamos a juliana, que tiene un celular.

Botón derecho > New en el área donde definimos objetos:

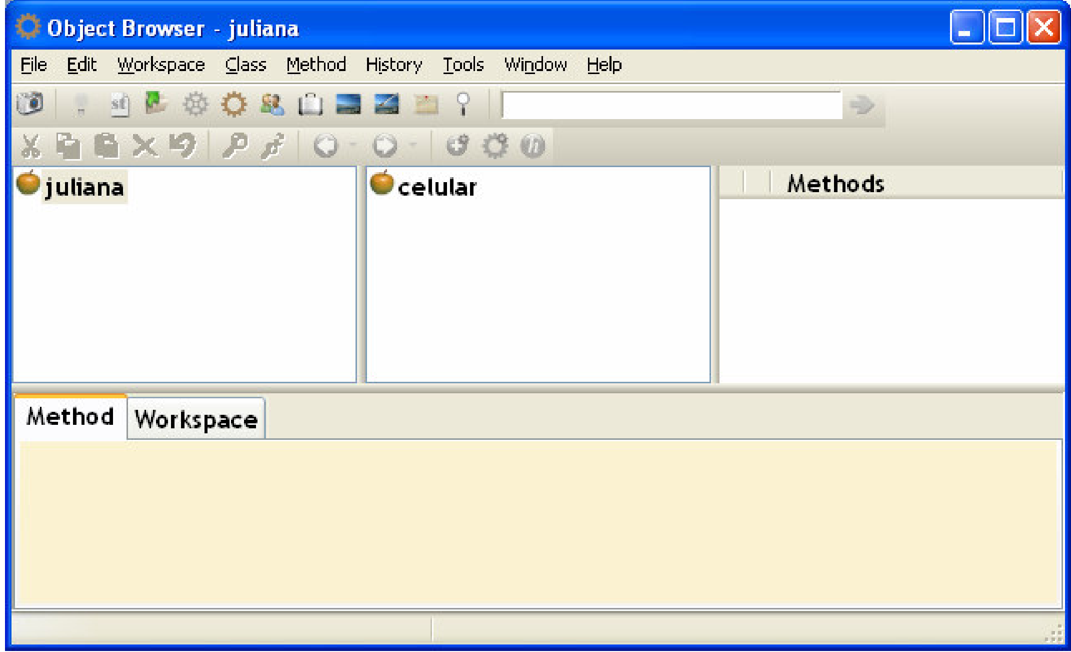




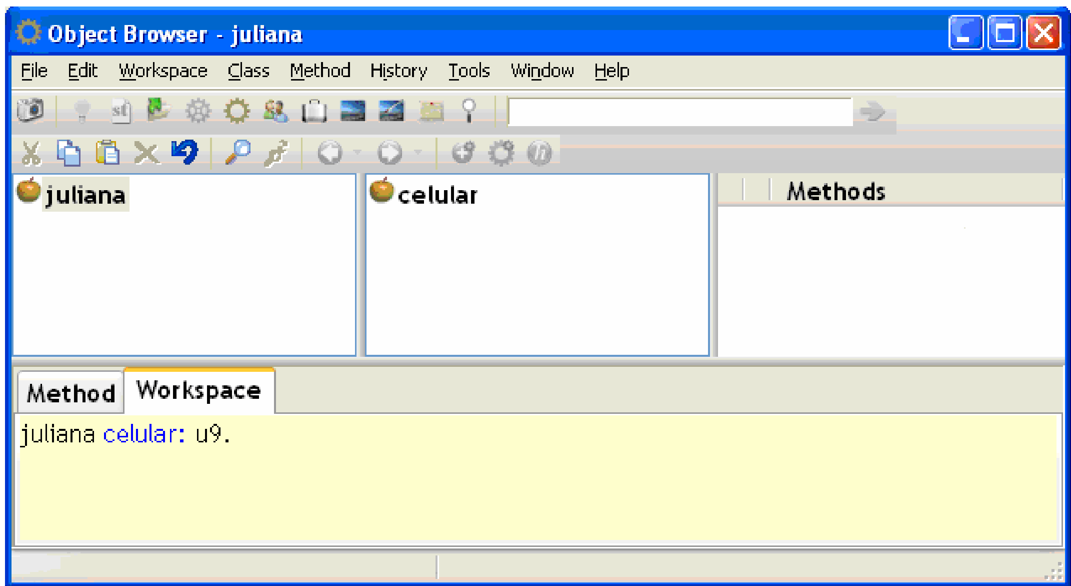
Con juliana seleccionada, en el área de atributos, botón derecho > Add…



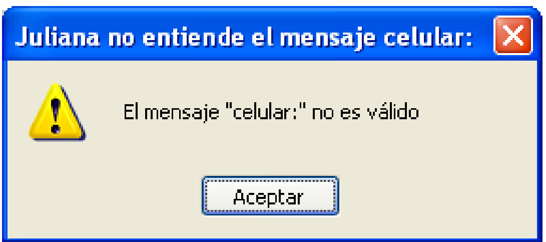
Nos queda:



Creamos el objeto u9 (de la misma manera).
Relacionamos a juliana y u9 escribiendo en la solapa workspace:
juliana celular: u9.

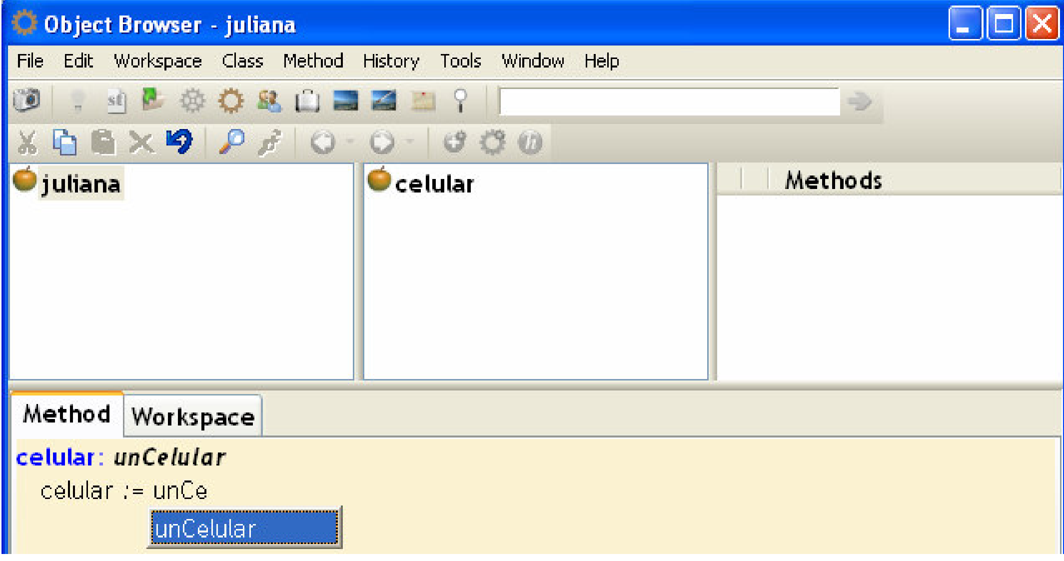


Para enviar el mensaje a juliana nos paramos en la misma línea, botón derecho > Evaluate It (Ctrl + E).

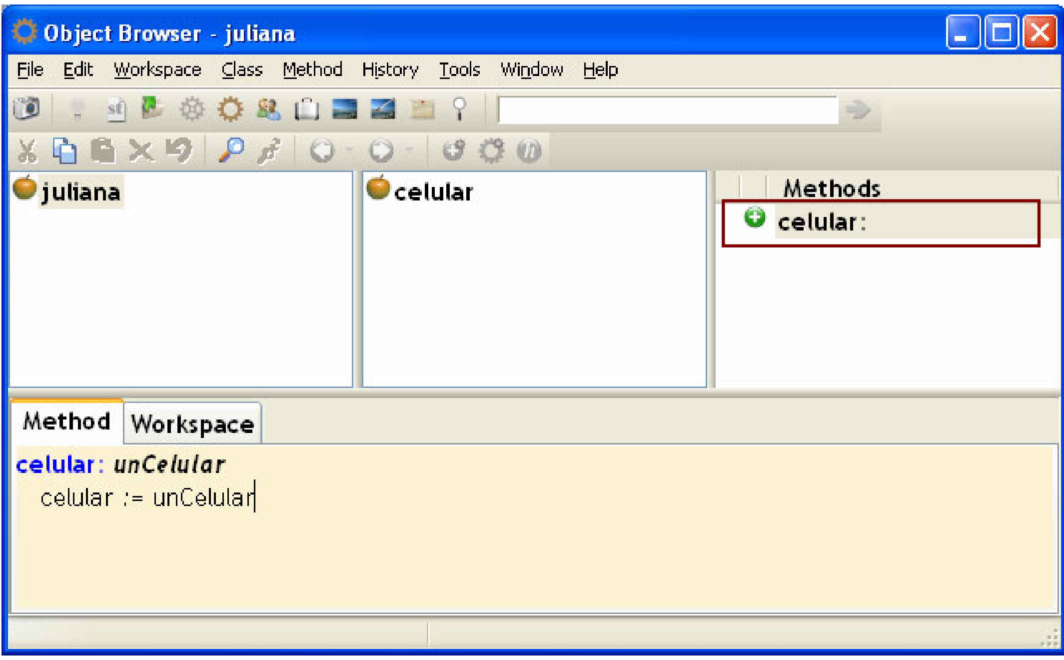


Nos da error, tenemos que crear el setter.

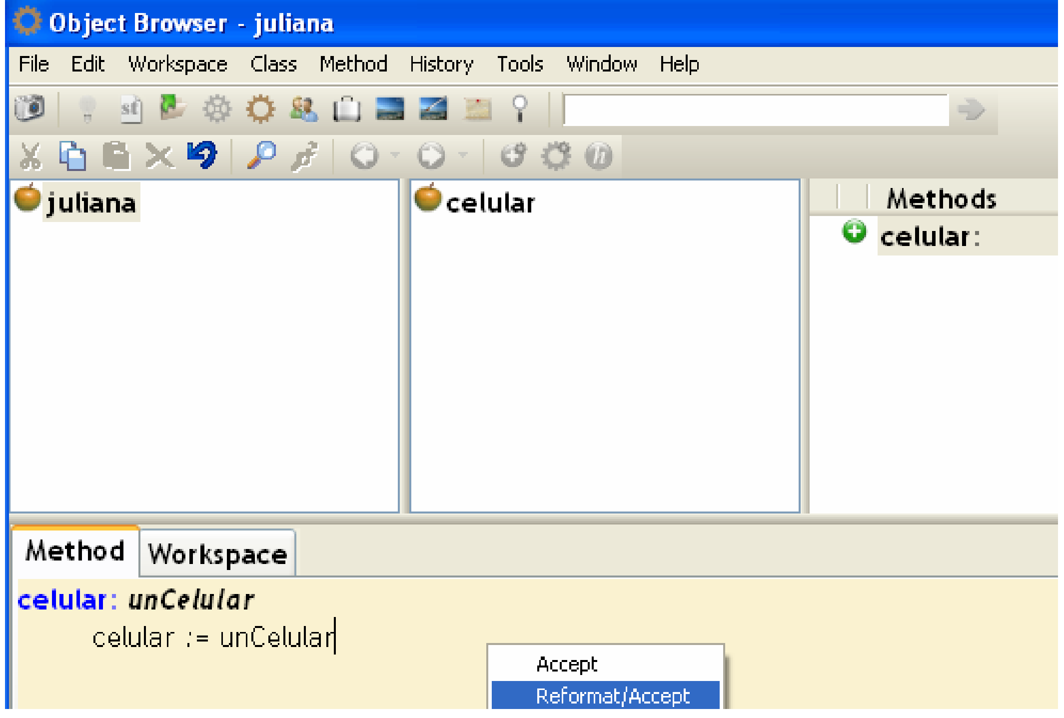
Para agregar comportamiento, en la solapa Method escribimos:



Cuando terminamos de escribir el método, lo grabamos y se compila automáticamente (botón derecho > Accept o bien Ctrl + S)



Tip: para formatear el método (sacando paréntesis innecesarios y poniendo los espacios donde corresponden) presionamos botón derecho > Reformat > Source / Reformat/Accept o Ctrl + W.



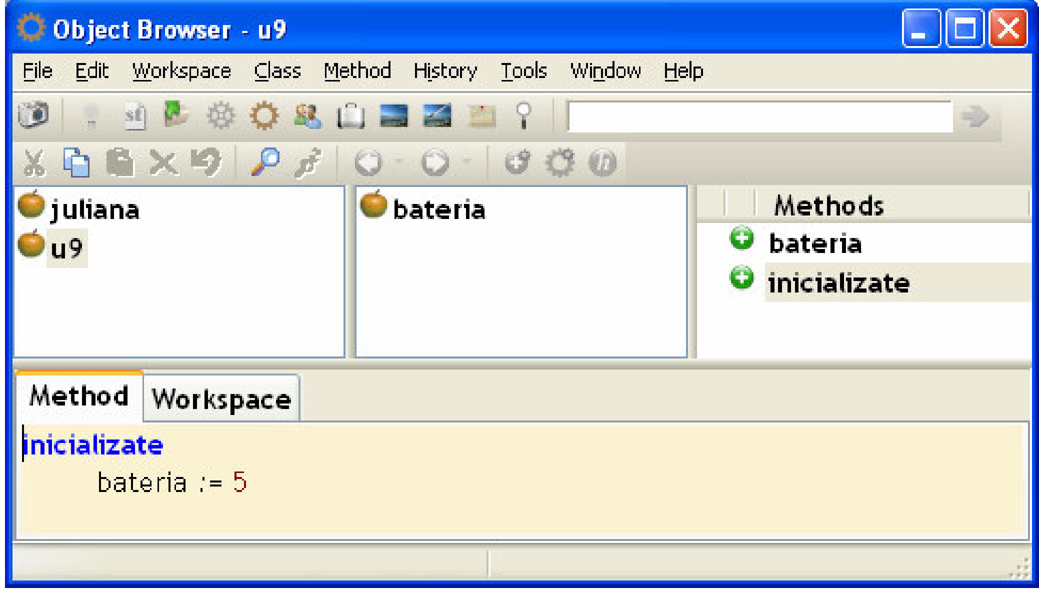
Escribimos en el Workspace:
"1 Conocer la cantidad de bateria en un celular"
u9 bateria.

Agregamos la variable bateria para el u9 con el getter
bateria
    ^bateria

Como ahora queremos ver el resultado, presionamos botón derecho > Display It (Ctrl + D). Nos devuelve nil, porque bateria nunca fue inicializado.

Vamos a hacer que el celular tenga una carga válida en la batería enviándole un mensaje inicializate.

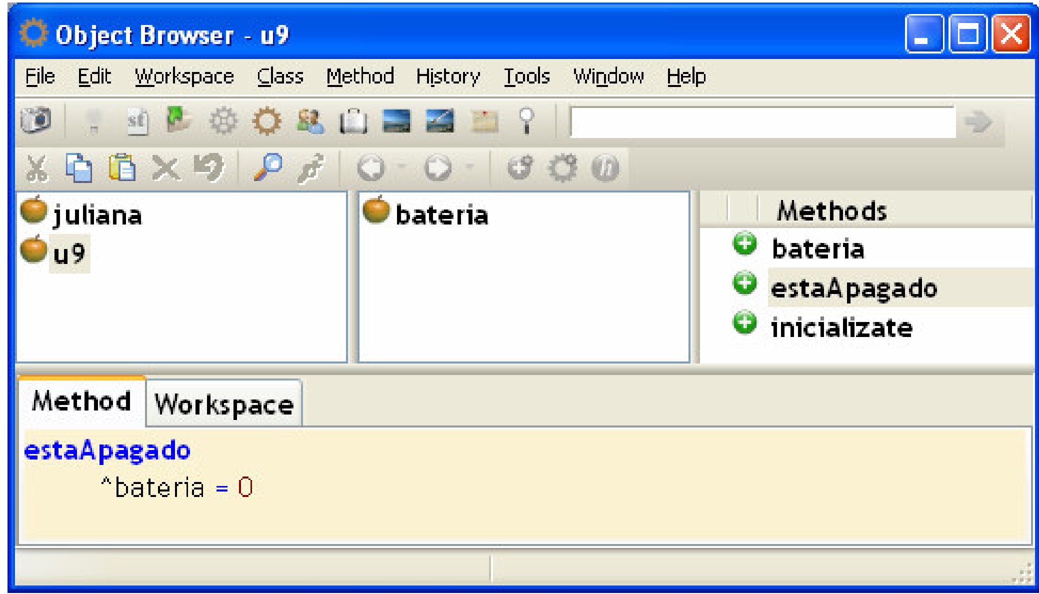
"1 Conocer la cantidad de bateria en un celular"
u9 inicializate.
u9 bateria.



Volvemos a enviar el mensaje u9 bateria y ahora sí recibimos un 5 como respuesta. Seguimos en el Workspace:

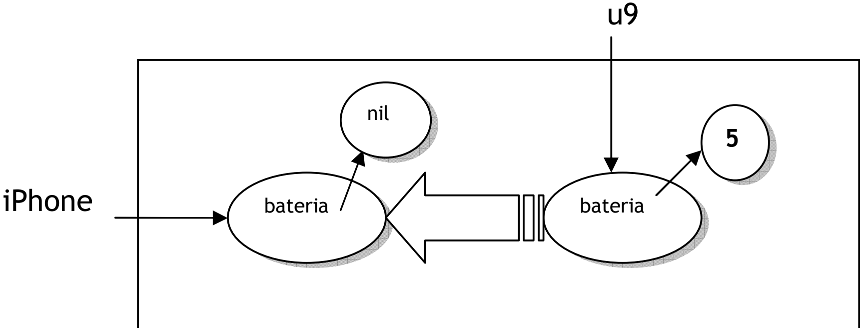
"2 Saber si un celular está apagado"
u9 estaApagado.

Escribimos el método en u9:

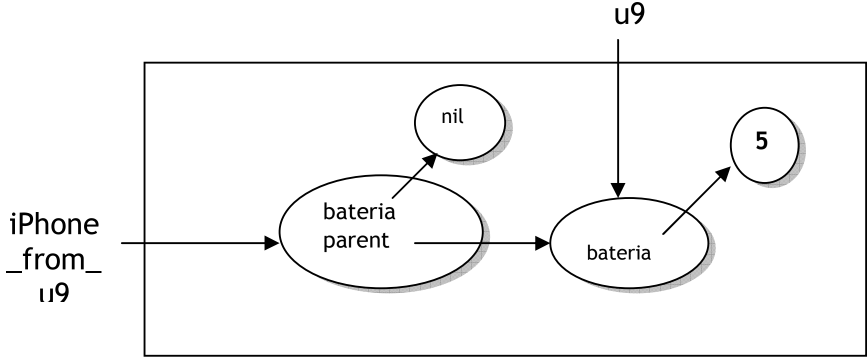


Para saber si el i-Phone está apagado queremos aprovechar el comportamiento que tiene u9:

- **Duplicar**: el i-Phone se crea como una copia de u9 que a partir de aquí tiene una nueva vida
- **Clonar**: hay una referencia entre el i-Phone y u9. u9 viene a ser el padre, *parent* o *prototipo*.



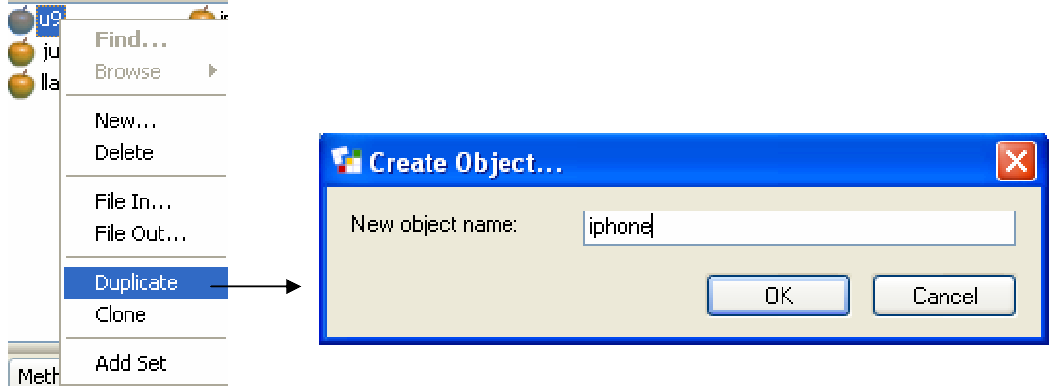
Duplicar: iPhone copia los atributos y métodos de u9 pero no tiene más relación con u9



Clonar: iPhone_from_u9 tiene una referencia a u9

- Al duplicar, podemos cambiar la definición del método inicializate de u9 y de iPhone por separado.
- Al clonar, no podemos modificar la definición del método inicializate de iPhone_from_u9, sólo desde u9. Es más: no podemos definir nuevos métodos para el objeto iPhone. El objeto u9 es el que maneja el comportamiento del iPhone (la ventaja: todo queda en un solo lugar).

Como el iPhone y el u9 son objetos que pueden tener distinto comportamiento (en principio sólo tienen bateria) elegimos crear el iPhone en base al u9 (nos paramos sobre u9 y hacemos botón derecho > Duplicar),



entonces el workspace nos queda:

```
"1 Conocer la cantidad de bateria en un celular"
u9 inicializate.
u9 bateria.
iphone inicializate.
iphone bateria.
"2 Saber si un celular está apagado"
u9 estaApagado.
iphone estaApagado.
"3 Recargar un celular (que vuelva a tener su batería completa)"
u9 recargate.
```

El método recargate, ¿qué hace?

```
recargate
	bateria := 5
```

¿Y el método inicializar?
Lo mismo, entonces vamos a hacer que inicializate llame a recargate, así:

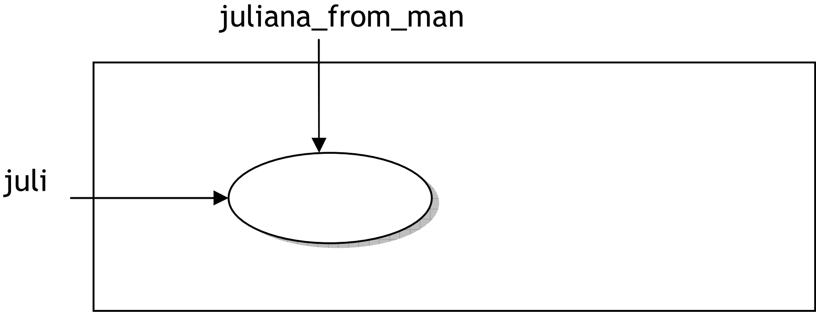
```
inicializate
	self recargate
```

Ahora que conocemos la diferencia entre clonar y duplicar, queremos que juliana y catalina tomen la misma definición de lo que es una persona. Es más, el prototipo de lo que es una persona lo vamos a representar con un objeto man…

Una vez creado man, hacemos botón derecho > Clone y creamos a juliana_from_man. Borramos al objeto juliana anterior. Copiamos la variable celular para man y podemos preguntarle a juliana_from_man si tiene celular…

Pero como escribir juliana_from_man es un poco molesto, podemos tener en el Workspace otra variable, juli. En el Workspace escribimos:

juli := juliana_from_man.
juli = juliana_from_man "Igualdad: juli y juliana_from_man son iguales”
juli == juliana_from_man "Identidad: juli y juliana_from_man apuntan al mismo objeto”



En el Workspace preguntamos:
juli tenesElCelularApagado.

Uh, cierto que no lo codifiqué. Pero igual ya se cómo enviar el mensaje. Me paro en man y escribo:

```
tenesElCelularApagado
	^celular estaApagado
```

Ah, entonces no le pregunto al celular si tiene batería, porque es responsabilidad del celular saber si está apagado o no.

Codificamos el método estaApagado en u9:

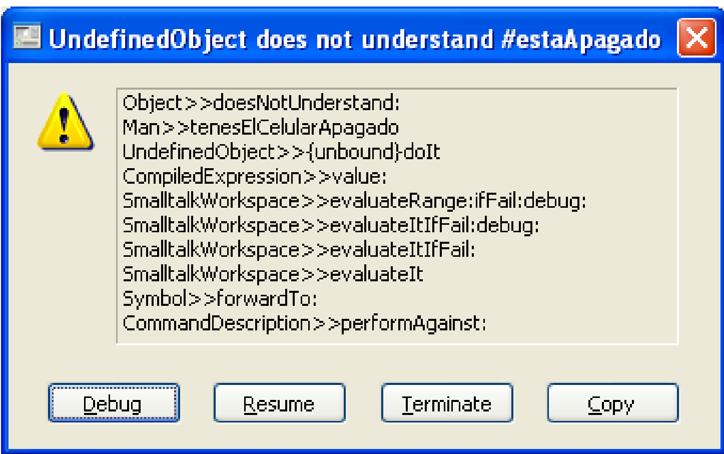
```
estaApagado
	^bateria = 0
```

Y en iPhone:

```
estaApagado
	^bateria < 1
```

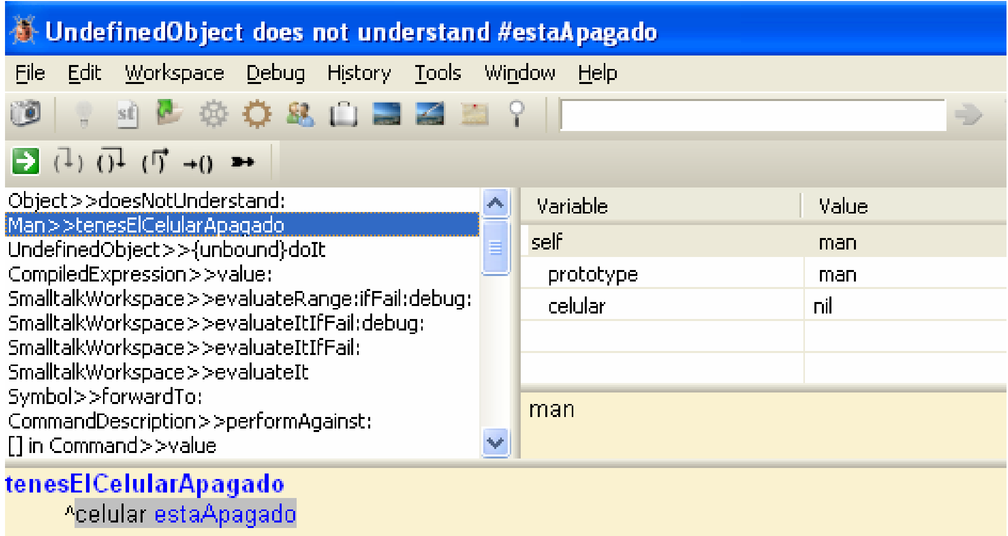
(sí, a propósito los hicimos distintos, pero en este caso no había diferencias)

Ahora lo probamos y ¡kaboom!



Dice que UndefinedObject (nil) no entiende estaApagado…

La lista de mensajes encadenada forma el Stack Trace, que nos brinda información más que útil. Presionamos el botón Debug:

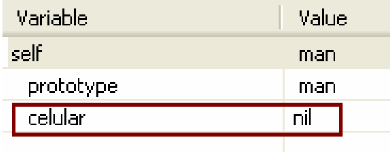


El Stack Trace guarda en una pila el encadenamiento de mensajes hasta el momento en que se produjo el error. La primera línea no nos brinda mucha ayuda, pero sí la segunda: es el objeto Man, en el método tenesElCelularApagado el que dio error.

A la derecha se visualiza el estado interno del objeto, demarcado por sus variables de instancia. Si se fijan, celular referencia a nil:

Variable	Value
self	man
prototype	man
celular	nil

Abajo se muestra resaltado en gris la línea donde ocurre el error:



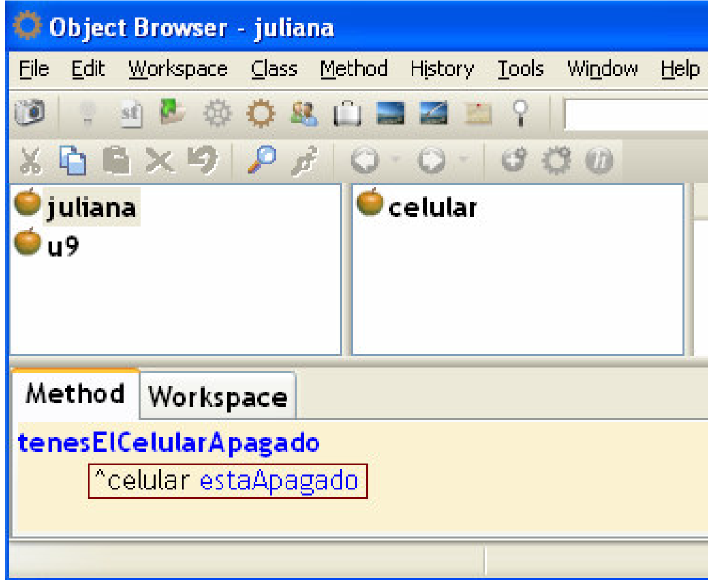
Ah, celular apunta a nil y quiero enviarle un mensaje, claramente la respuesta es: nil no entiende estaApagado…

UndefinedObject does not understand #estaApagado

Entonces nos falta asignar el celular a juli (porque al crear el clon de man no le asignamos el celular). En el workspace escribimos:
juli celular: u9.
juli tenesElCelularApagado.

Creamos a catalina y le asignamos el iPhone:
catalina_from_man celular: iphone.
catalina_from_man tenesElCelularApagado

Fíjense como hay polimorfismo cuando enviamos el mensaje celular estaApagado. El celular puede ser un iPhone o un u9, eso no nos importa, sólo importa que el objeto al que referencia la variable celular entienda el mensaje estaApagado:



Tampoco importa mucho si hablamos con juliana o catalina, a cualquiera de las dos podemos enviarle el mensaje tenesElCelularApagado.

Y recordemos lo que es la felicidad para PDP:

En el Workspace:

"Felicidad en PDP"
juli tenesElCelularApagado
"Infelicidad en la vida"
juli celular bateria = 0

Parte b

b. Ahora podemos también tener en cuenta el costo de las llamadas que se hacen entre Catalina y Juliana. Catalina tiene contratado como servicio de telefonía celular a Movistar, Juliana a Personal. Movistar cobra fijo $0,60 final el minuto, Claro cobra 0,50 el minuto + 21% de IVA y Personal $0,70 final los primeros 10 minutos que usaste el celu, y $0,40 el minuto el resto.

Solución:

Creamos un objeto movistar, otro personal. A Claro no lo creamos por ahora. También creamos un objeto llamada que sirve como prototipo (del cual clonamos un objeto llamadadejuli). Vamos a relacionar dichos objetos en el workspace:

```
llamadadejuli_from_llamada empresaPrestadora: movistar
llamadadejuli_from_llamada duracion: 3.
llamadadejuli_from_llamada costo
llamadadejuli_from_llamada empresaPrestadora: personal
```

Todavía no había pensado en variables, ahora se que necesito implementar algunos mensajes para la llamada:

- costo
- duración:
- empresaPrestadora:

Lo definimos, creando las variables empresaPrestadora y duracion.

Y escribimos los siguientes métodos para llamada:

empresaPrestadora: unaEmpresa
    empresaPrestadora := unaEmpresa

duracion: unaDuracion
    duracion := unaDuracion

costo
    ^empresaPrestadora costoDe: duracion

Mmm… Ahora tengo que pensar en movistar y personal para saber cómo calcular el costo:
Movistar va a hacer algo como:
costoDe: unaDuracion
    ^0.6 * unaDuracion

Y Personal:
costoDe: unaDuracion
  ^(self costoInicialDe: unaDuracion) + (self costoFinalDe: unaDuracion)

costoInicialDe: unaDuracion
 ^0.7 * (unaDuracion min: 10)

costoFinalDe: unaDuracion
 ^0.4 * (unaDuracion - 10 max: 0)

A la llamada no le importa si cambio de empresa prestadora: basta con que cada empresa respete la interfaz costoDe: unaDuracion:

```
costo
	^empresaPrestadora costoDe: duracion
```