**Capítulo 3: Programación con Objetos**

**1-Objetos y mensajes:**

El paradigma de objetos, a veces también conocido como orientado a objetos nos propone solucionar problemas y modelar nuestra realidad empleando objetos que se comunican entre ellos intercambiando mensajes. Este nos permite representar al mundo como un **conjunto de objetos** que interactúan entre ellos.

Un objeto es cualquier entidad que pueda hacer algo por nosotros para resolver un problema, capaz de hacer alguna tarea, la cual conocemos como responsabilidad del objeto.

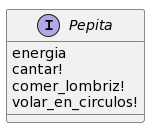
Un objeto tiene identidad: cada objeto sabe quién es y gracias a esto sabe también que es diferente de los demás.

Para comunicarnos o interactuar con los objetos, debemos enviarles **mensajes**. Cuando un objeto recibe un mensaje, este responde haciendo algo

La sintaxis del envío de mensajes:

1. Pepita.energia es un envío de mensaje, también llamado colaboración;
2. energía es el mensaje;
3. energía es el nombre del mensaje (en este caso es igual, pero ya veremos otros en los que no);
4. Pepita es el objeto receptor del mensaje.

un objeto puede entender múltiples mensajes; a este conjunto de mensajes que podemos enviarle y el objeto entiende lo denominamos **interfaz**

****

Cuando se envía un mensaje a un objeto, y este lo entiende, puede reaccionar de dos formas diferentes:

* Podría producir un efecto, es decir hacer algo. Por ejemplo, el mensaje cantar! reproduce el sonido del canto de Pepita.
* O también podría devolver otro objeto. Por ejemplo, el mensaje energía devuelve siempre un número.

los mensajes **con efecto** (es decir, que hacen algo) les pondremos un signo de exclamación ! al final.

Un programa es simplemente una secuencia de envío de mensajes

Puede haber más de un objeto que entienda el mismo mensaje. Dos o más objetos pueden entender un mismo mensaje, pero pueden **comportarse** de formas diferente

**un mensaje queda identificado no sólo por su nombre sino también por la cantidad de parámetros (argumentos) que tiene**

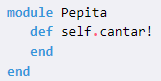
en Pepita.volar\_hacia! Barreal el objeto receptor es Pepita, el mensaje volar\_hacia! y el argumento [Barreal](https://es.wikipedia.org/wiki/Barreal)

**2-Definiendo objetos: métodos y estado:**

La **definición** de un objeto se inicia con la palabra reservada module, luego el nombre del objeto (con la primera letra en mayúscula) y su fin se indica con un end.

****

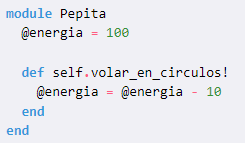
Para que un objeto entienda un mensaje debemos "enseñarle" cómo hacerlo, y para ello es necesario definir un **método** dentro de ese objeto:



Un método es, entonces, la descripción de **qué hacer cuando se recibe un mensaje del mismo nombre**:

* Todos los métodos comienzan con **def** y terminan con **end,**
* Todos los métodos que pertenezcan al mismo objeto van **dentro del mismo module**.

Creación de métodos para modificar el estado de energía de un objeto:



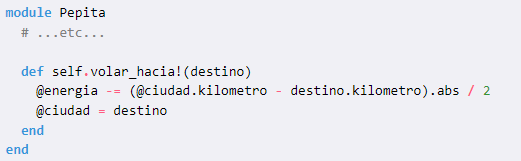
La energía es un **atributo** de nuestro objeto, y la forma de **asignarle** un valor es escribiendo @energia = 100.

los **atributos** (los cuales escribiremos anteponiendo @), son objetos que nos permiten representar una característica de otro objeto. Un objeto conoce a todos sus atributos por lo que puede enviarles mensajes

**los atributos NO son mensajes**, para mostrar un atributo en necesario definir un nuevo método. Estos métodos que sirven para conocer el valor de un atributo los llamamos **métodos de acceso** o simplemente **accessors**,.

Los objetos pueden tener múltiples atributos y al conjunto de estos atributos se lo denomina **estado.** El estado es siempre **privado**, es decir, solo el objeto puede utilizar sus atributos.

Los objetos pueden enviarse mensajes entre sí, por ejemplo:



Esto se conoce como delegar una responsabilidad, o delegar: la responsabilidad de saber en que kilometro se encuentra es responsabilidad del objeto ciudad, no de pepita.

A veces nos va a pasar que un objeto tiene un método muy complejo, y nos gustaría subdividirlo en problemas más chicos que **el mismo objeto** puede resolver. Un objeto puede enviarse un mensaje a sí mismo fácilmente usando self como receptor del mensaje.

La **delegación** es la forma que tenemos en objetos de **dividir en subtareas**: separar un problema grande en problemas más chicos para que nos resulte más sencillo resolverlo.

**3-Polimorfismo y encapsulamiento:**

En Ruby, es una **convención** que los mensajes que devuelven booleanos (o sea, verdadero o falso) terminen con un ?.

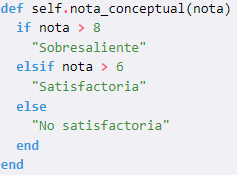
En Ruby, como en muchos otros lenguajes, esto se escribe con la palabra reservada if.

Hay veces que con un if alcanza, pero otras queremos hacer algo si no se cumple una condición. Como ya te podrás imaginar, donde hay un if ¡cerca anda un else!



Times: Es un mensaje que entienden los números que sirve para ejecutar una porción de código varias veces. En este caso regaríamos 3 veces la planta recibida como argumento.

En Ruby, podemos simplicar la manera de escribir un if dentro un else con elsif



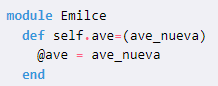
Para que un objeto pueda mandarle mensajes a otro debe **conocerlo**. Por ejemplo, llamándolo por su nombre como ya hemos hecho:



Decimos entonces que dos objetos son **polimórficos** cuando pueden responder a un mismo conjunto de mensajes y hay un tercer objeto que los usa indistintamente. Dicho de otra forma, dos objetos son **polimórficos para un tercer objeto** cuando este puede enviarles los mismos **mensajes**, sin importar cómo respondan o qué otros mensajes entiendan.

En los ejercicios anteriores, le habíamos incluido a Pachorra y Emilce un mensaje firmar\_contrato!(ave) que modificaba su **estado**, es decir, alguno de sus **atributos**. A estos mensajes que solo modifican un atributo los conocemos con el nombre de **setters**, porque vienen del inglés set que significa establecer, ajustar, fijar.

Forma para modificar los atributos utilizando un método.



estos métodos que solo sirven para acceder o modificar un atributo los llamamos **métodos de acceso** o **accessors**. Repasando, los **setters** son aquellos métodos que *establecen* el valor del atributo. Mientras que los **getters** son aquellos que *devuelven* el valor del atributo.

La convención en Ruby para estos métodos es:

* Los **setters** deben llevar el mismo nombre del atributo al que están asociados, agregando un = al final.
* Los **getters** usan exactamente el mismo nombre que el atributo del cual devuelven el valor pero sin el @.
* Aquellos **getters** que devuelven el valor de un atributo booleano llevan ? al final.

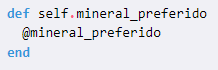
Estos métodos que solo sirven para acceder o modificar un atributo los llamamos **métodos de acceso** o **accessors**. Repasando, los **setters** son aquellos métodos que *establecen* el valor del atributo. Mientras que los **getters** son aquellos que *devuelven* el valor del atributo.

La convención en Ruby para estos métodos es:

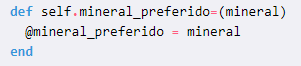
* Los **setters** deben llevar el mismo nombre del atributo al que están asociados, agregando un = al final.
* Los **getters** usan exactamente el mismo nombre que el atributo del cual devuelven el valor pero sin el @.
* Aquellos **getters** que devuelven el valor de un atributo booleano llevan ? al final.

Recordá la convención para nombrar los métodos *de acceso* que mencionamos antes:

* Para los **getters**, que sirven para **obtener** el valor de un atributo, usamos el mismo nombre que este.



* Para los **setters**, que sirven para **fijar** el valor de un atributo, usamos el mismo nombre que este pero con un = al final



Quien use nuestros objetos sólo verá lo que necesite para poder interactuar con ellos. A esta idea la conocemos como **encapsulamiento**, y es esencial para la separación de **responsabilidades**

Resumen final: En esta lección le dimos nombre al **polimorfismo** una idea con la que ya venías trabajando, pero sobre la que todavía no habíamos reflexionado. Este principio fundamental del paradigma de objetos nos permite que podamos interactuar de igual manera con diferentes objetos, con el único requisito de que todos ellos entiendan el o los mensajes que necesitamos enviarles.

Relacionado a esto, hablamos del **encapsulamiento** que nos permite el paradigma, haciendo que cada objeto solo **exponga** lo necesario para interactuar con él y se reserve para su ámbito privado lo que no sea necesario compartir.

En el caso de los atributos, esta exposición se logra implementando un **getter** (método que nos permite ver su valor) o un **setter** (método que nos permite modificar su valor). Y que nuestro código sea entendido fácilmente por otras personas, elegimos utilizar una **convención** para darle nombre a estos métodos

**4-Referencias:**

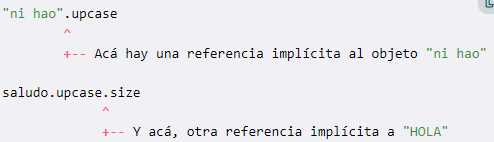
Declarar variables: Por ejemplo, podemos declarar una variable saludo, inicializarla con "hola", enviarle mensajes...



Sucede que en realidad las cosas son un poco más complejas: no conocemos a los objetos directamente, sino a través de etiquetas llamadas referencias. Entonces cuando tenemos una declaración de variable como ésta, lo que estamos haciendo es crear una referencia saludo que apunta al objeto "hola", que representamos mediante una flechita:

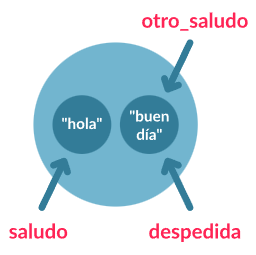


Cuando enviamos mensajes a objetos literales como el 2, el true u "hola", o expresiones, estamos conociendo a esos objetos a través de referencias implícitas, que son **temporales** (sólo existen durante ese envío de mensajes) y **anónimas** (no tienen un nombre asociado).



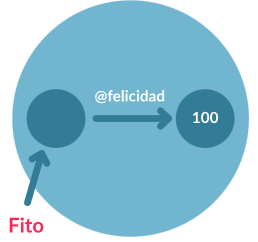
Las referencias explícitas son las que vimos hasta ahora. Por ejemplo:





Ambas referencias, otro\_saludo y despedida, apuntan al mismo objeto. La moraleja es que declarar una variable significa agregar una nueva referencia al objeto existente, en lugar de copiarlo.

Objetos bien conocidos: cuando los definimos no sólo describimos su comportamiento (comer!(calorias) y felicidad) y estado (@felicidad), sino que además les damos un nombre o etiqueta a través de la cual podemos conocerlos. Esas etiquetas también son referencias . Y son globales, es decir que cualquier objeto o **programa** puede utilizarla.



*siempre* estamos enviando el mensaje al objeto a través de una referencia.

En un ambiente hay muchos objetos, pero en realidad no interactuamos con ellos directamente, sino a través de referencias, que son nombres o etiquetas que les damos a los objetos.

Para un objeto pueden existir múltiples nombres: cuando le damos uno nuevo, no estamos creando una copia del objeto ni modificándolo realmente, sino que estamos creando una nueva referencia que apunta al objeto. Así que ¡ojo!, si compartís un objeto con otros, y lo mutás, ¡todos los que tengan una referencia al mismo verán los cambios!

Finalmente, en objetos, todo lo que se *parezca* a una variable es una referencia, y hay de muchos tipos:

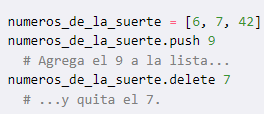
* variables de un programa
* variables locales de un método
* parámetros de un método
* atributos de un objeto
* y el nombre global de un objeto bien conocido.

**5-Colecciones:**

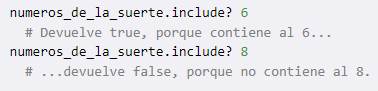
lista de objetos: es un tipo de colección en la cual los elementos pueden repetirse. Es decir, el mismo objeto puede aparecer más de una vez.

Todas las colecciones entienden una serie de mensajes que representan operaciones o consultas básicas sobre la colección.

Por ejemplo, podemos agregar un elemento enviándole push a la colección o quitarlo enviándole delete



También podemos saber saber si un elemento está en la colección usando include?:



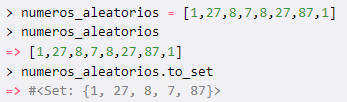
Finalmente, podemos saber la cantidad de elementos que tiene enviando size:



Otro tipo muy común de colecciones son los *sets* (*conjuntos*), los cuales tienen algunas diferencias con las listas:

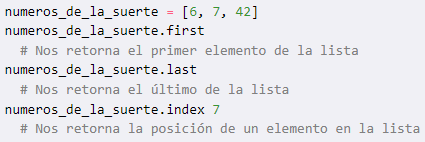
* no admiten elementos repetidos;
* sus elementos no tienen un orden determinado.

Vamos a ver un ejemplo transforma una lista en un set utilizando to\_set:



Algo importante a tener en cuenta es que tanto las listas como los sets tienen mensajes en común. Dicho de otro modo, son polimórficos para algunos mensajes. Por ejemplo: push, delete, include? y size.

Sin embargo, los siguientes mensajes...



... no podemos enviárselos a un set porque sus elementos no están ordenados.

Hay una diferencia notable entre los primeros dos mensajes (push y delete) y los otros dos (include? y size):

1. push y delete, al ser evaluados, *modifican* la colección. Dicho de otra forma, producen un **efecto** sobre la lista en sí: agregan o quitan un elemento del conjunto.
2. include? y size sólo nos retornan información sobre la colección. Son métodos **sin efecto**.

Los *bloques* son **objetos** que representan un mensaje o una secuencia de envíos de mensajes, **sin ejecutar**, lista para ser evaluada cuando corresponda. La palabra con la que se definen los bloques en Ruby es *proc*. Por ejemplo, en este caso le asignamos un *bloque* a incrementador:

un\_numero **=** 7

incrementador **=** proc { un\_numero **=** un\_numero **+** 1 }

Ahora avancemos un pasito: en este segundo ejemplo, al *bloque* { otro\_numero = otro\_numero \* 2 } le enviamos el mensaje *call*, que le indica que **evalúe la secuencia de envíos de mensajes** dentro de él.

otro\_numero **=** 5

duplicador **=** proc { otro\_numero **=** otro\_numero **\*** 2 }.call

¿Cuánto vale un\_numero luego de las primeras dos líneas? Tené en cuenta que la secuencia de envío de mensajes en el *bloque* del primer ejemplo está **sin ejecutar**. En cambio, en el ejmplo de otro\_numero estamos enviando el mensaje call. Por lo tanto:

* un\_numero vale 7, porque el bloque incrementador no está aplicado. Por tanto, no se le suma 1.
* otro\_numero vale 10, porque el bloque duplicador se aplica mediante el envío de mensaje call, que hace que se ejecute el código dentro del bloque. Por tanto, se duplica su valor.

Los bloques también pueden recibir argumentos para su aplicación. Por ejemplo, sumar\_a\_otros\_dos recibe dos argumentos, escritos entre barras verticales | y separados por comas:

un\_numero **=** 3

sumar\_a\_otros\_dos **=** proc { **|**un\_sumando, otro\_sumando**|** un\_numero **=** un\_numero **+** un\_sumando **+** otro\_sumando }

Para aplicar el bloque sumar\_a\_otros\_dos, se le pasan los argumentos deseados al mensaje call:

> sumar\_a\_otros\_dos.call(1,2)

**=>** 6

si de una lista de números queremos los mayores a 3.

Lo que usamos es el mensaje select de las colecciones. select recibe un bloque con un parámetro que representa un elemento de la colección y una condición booleana como código, y lo que devuelve es una nueva colección con los elementos que la cumplen. El select **no** produce efecto.





¿Y si en vez de **todos** los elementos que cumplan una condición, sólo queremos uno? ¡Usamos find! Mientras que select devuelve una colección, find devuelve **únicamente** un elemento. ¿Y si ningún elemento de la colección cumple la condición? Devuelve nil, que, como aprendiste antes, es un objeto que representa la nada - o en este caso, que ninguno cumple la condición.





Para saber si **todos** los elementos de una colección cumplen un cierto criterio podemos usar el mensaje all?, que también recibe un bloque. Por ejemplo, si tenemos una colección de estudiantes, podemos saber si todo el grupo aprueba



De manera muy similar podemos saber si **algún elemento** de la colección cumple cierta condición mediante el mensaje any?. Siguiendo el ejemplo anterior, ahora queremos saber si por lo menos alguien aprobó:



¿Qué tienen de distinto all? y any? respecto aselect y find?

Mientras que select devuelve una colección y find un elemento o nil, all? y any? siempre devuelven un valor booleano: true o false.

El mensaje map nos permite, a partir de una colección, obtener **otra colección** con cada uno de los resultados que retorna un envío de mensaje a cada elemento.

En otras palabras, la nueva colección tendrá lo que devuelve el mensaje que se le envíe a cada uno de los elementos. Por ejemplo, si usamos map para saber los niveles de energía de una colección de golondrinas:

> [Pepita, Norita].map { **|**una\_golondrina**|** una\_golondrina.energia }

**=>** [77, 52]

Al igual que el resto de los mensajes que vimos hasta ahora, map no modifica la colección original ni sus elementos, sino que devuelve una **nueva** colección.

count nos dice cuántos elementos de una colección cumplen la condición. Por otro lado, para calcular sumatorias tenemos el mensaje sum. Si queremos conocer la suma de todas las notas de la colección de estuidantes, por ejemplo, podemos hacer:





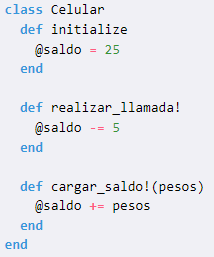
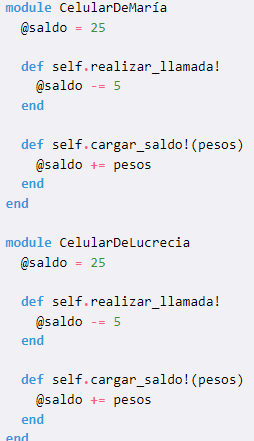
cuando queremos hacer algo con cada elemento, para mandarle un mensaje a cada uno con la intención de **producir un efecto**. Es en este caso que nos resulta de utilidad el mensaje each.

Resumen:

* Qué son las colecciones
* Qué mensajes básicos entienden: agregar y quitar elementos mediante push y delete, preguntar si existe un elemento usando include?, y saber el tamaño de la colección enviándole size
* Qué son los bloques, cómo se aplican y cómo se les pasan parámetros
* Mensajes más complejos que utilizan bloques como select, find, map, all?, any?, count, sum y each. ¡Son un montón!

**6-Clases e Instancias:**

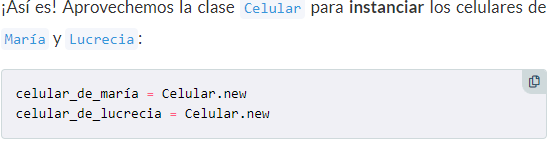
Si tenemos más de un objeto que se comporta **exactamente** de la misma forma, lo que podemos hacer es generalizar ese comportamiento definiendo una **clase.** ¡Atención! No todo es tan simple: notá que, a diferencia de en los objetos, los métodos en las clases **no se preceden** con la palabra self.

** **

Las clases sólo nos sirven para generalizar objetos que tengan el mismo comportamiento: **mismos métodos y mismos atributos**.

Si el código es parecido pero no puede ser generalizado para que sea el mismo, las clases no nos servirán.

La clase es un objeto que nos sirve como **molde** para crear nuevos objetos. Esto quiere decir que las clases pueden crear nuevos objetos

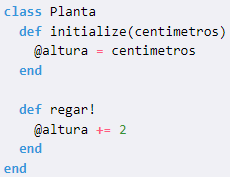
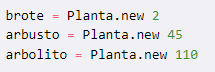


la salud no se comparte entre bouba y kiki a pesar de que ambos sean instancias de Zombi.

initialize. Al trabajar con clases tenemos que inicializar los atributos en algún lugar. ¡Para eso es que existe ese método!

El mensaje initialize nos permite especificar **cómo queremos que se inicialice** la instancia de una clase. ¡Es así de fácil!

initialize puede **recibir parámetros** que especifiquen **con qué valores** deseamos inicializar los atributos al construir nuestros objetos

Lo que hiciste recién fue **especificar un constructor**: decirle a la clase cómo querés que se construyan sus instancias.

Los constructores pueden recibir más de un parámetro. Por ejemplo, si de una Planta no sólo pudiéramos especificar su altura, sino también su especie y si da o no frutos...



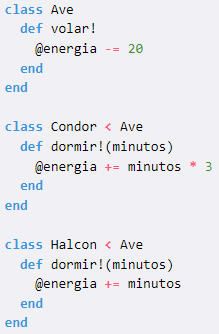
¡Ya conocemos todo lo necesario acerca de las clases!

Recordá que sirven para generalizar comportamiento idéntico entre objetos: si no es **exactamente igual**, es necesario crear una nueva clase, como hicimos con Zombi y SuperZombi, o con Sobreviviente y Aliado. En esos casos tuvimos que repetir parte de la lógica que coincidía entre las clases; pronto aprenderemos una alternativa.

**7-Herencia:**

Cuando tenemos clases que tienen una parte de comportamiento común pero otra que difiere, las herramientas que vimos hasta el momento nos quedan cortas, tendremos que crear otra clase.

Una forma de organizar las clases cuando programamos en objetos es establecer una **jerarquía**. En nuestro caso podemos pensar que Celular y Notebook se pueden englobar en algo más grande que las incluya, la idea de Dispositivo.



El símbolo < significa "hereda de": por ejemplo, Condor hereda de Ave, que está más arriba en la jerarquía. Otra manera de decirlo es que cada Condor es un Ave.

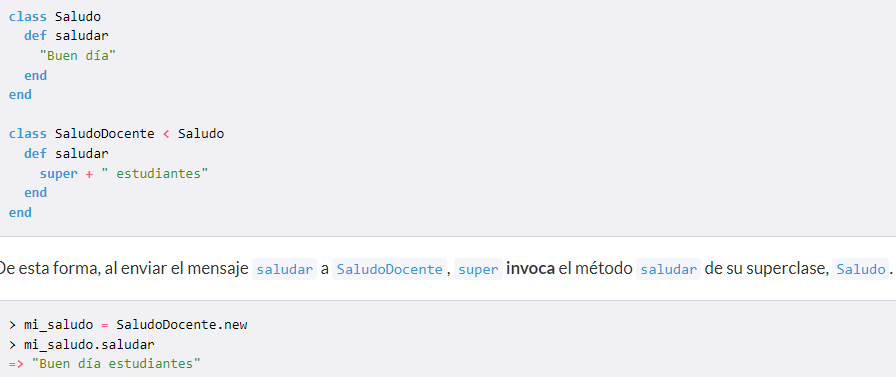
La herencia nos permite que las subclases (Condor y Halcon) posean los mismos métodos y atributos que la superclase Ave. Es decir, las instancias de Condor y de Halcon van a saber volar! de la misma forma, pero cuando les enviemos el mensaje dormir! cada una hará algo diferente.

Cuando dos objetos repiten lógica, creamos una clase con el comportamiento en común. En el caso que dos clases repitan lógica deberíamos crear una nueva clase a la cual llamamos superclase. A esta nueva clase llevaremos los métodos repetidos y haremos que las clases originales hereden de ella. Estas subclases que heredan de la superclase solo contendrán su comportamiento particular.

A este tipo de clases, como Dispositivo o Ave en el ejemplo del ejercicio anterior, se las llama **clases abstractas** porque, a diferencia de las **clases concretas** (como Celular o Notebook), nunca las instanciamos. En otras palabras, no creamos objetos con esa clase, solo nos sirven para proveer comportamiento a sus subclases.

Esto de la herencia está buenísimo. Porque nos permite heredar el comportamiento de una superclase pero redefinir aquellas cosas que nuestras subclases hacen distinto. Pero cuidado, si tenemos que redefinir todo probablemente no necesitemos heredar en primer lugar.

Al utilizar super en el método de una subclase, **se evalúa el método con el mismo nombre de su superclase**. Por ejemplo..



Recordá que la **herencia** es un concepto amplio que tiene muchas variantes: primero vimos clases **abstractas**, que no se instancian. También aprendimos a **redefinir** los métodos cuando heredamos un método pero queremos que se comporte de otra forma.

Luego vimos un caso de herencia de una clase concreta, y cómo las subclases **heredan** los métodos de su superclase. Por último aprendimos a usar super: cuando una subclase lo envía, se evalúa el método del mismo nombre de su superclase.