# Objetos y Mensajes

Ambiente: universo de objetos.

Los objetos tienen identidad. Cada objeto es único y diferente a los demás.

La forma más sencilla de comparar los objetos es mediante la igualdad y la diferencia, de la siguiente forma:

pepita == pepita

pepita != anastasia

Para comunicarte con los objetos, le podés enviar *mensajes*. Si el objeto no entiende el mensaje, se produce un error. Si lo entiende, puede reaccionar de dos formas diferentes:

* devolver algo, o sea, *otro objeto*.
* *producir un efecto*, es decir, provocar un cambio.

Si el objeto sabe hacer algo como reacción a ese mensaje, lo hará. De lo contrario se lanzará un error.

un objeto puede entender múltiples mensajes; a este conjunto de mensajes que podés enviarle lo denominamos **interfaz**.

 la forma correcta de escribir el envío de mensajes, es decir su sintaxis:

1. pepita.volar() es un envío de mensaje, también llamado colaboración
2. volar() es el mensaje
3. pepita es el objeto receptor del mensaje

Puede ocurrir que varios objetos compartan mensajes y que puedas hacer las mismas preguntas a diferentes objetos y cada uno responde a su manera. Pero tampoco le podés pedir cualquier cosa a todo objeto. Depende de lo que cada objeto sepa hacer.

Un mensaje puede tener uno o más argumentos separados por comas. Se usan para indicar un valor a la hora de mandar un mensaje.

Si a un mensaje le paso una cantidad incorrecta de argumentos, saltará un error ya que no es un mensaje válido.

# Métodos y Estado

todos los números, cadenas de caracteres y booleanos son objetos y con solo expresarlos con la sintaxis adecuada (comillas dobles o simples para los strings, punto para los numeros con parte decimal, true y false para los valores de verdad) ya están disponibles para ser usados. Cada uno de ellos ya entiende una serie de mensajes predefinidos. Se les llama **literales**.

Por ejemplo, para contar con anastasia en el ambiente de objetos la tenemos que declarar de la siguiente forma:

object anastasia {

}

¿Cómo hacemos que un objeto entienda un mensaje? ¿Cómo sabe pepita qué cuando le piden cantar debe responder con su canto característico? Tenemos que declararle un **método**:

object pepita {

method cantar() {

return "pri pri pri"

}

}

El método se declara en el objeto, lleva por nombre el mismo que el mensaje y contiene una descripción de **qué hacer cuando se recibe un mensaje del mismo nombre**.

si a un objeto no le decimos **cómo** reaccionar ante un mensaje, y se lo envíamos igual, no lo entenderá y nuestro programa se romperá. Y la forma de evitar esto es declarando un **método**.

A continuación se declara dentro del objeto (pepita) un atributo variable (energía) con un valor inicial de 100

object pepita {

var energia = 100

method estaFeliz() {

return energia >= 90

}

}

Al conjunto de atributos se lo denomina **estado**.

Lo que sucede es que el estado es siempre **privado**, es decir, sólo el objeto puede utilizar sus atributos. A esto se lo llama **encapsulamiento.**

Y para reforzar la idea que es propio del objeto, se suele hablar de estado *interno*.

**los atributos NO son métodos**. No existe ningún metodo llamado energía() ni energia(valor) que reciba un parámetro.

Se pueden declarar métodos que accedan a un atributo del objeto.

Comúnmente se los denomina **accessors** y son métodos simples de declarar:

object pepita {

var energia = 100

var ciudad = rosario

method energia() {

return energia

}

method energia() {

energia = 5

}

}

El primer método es un **getter** ya que solamente retorna una variable.

El segundo método es un **setter** ya que modifica una variable.

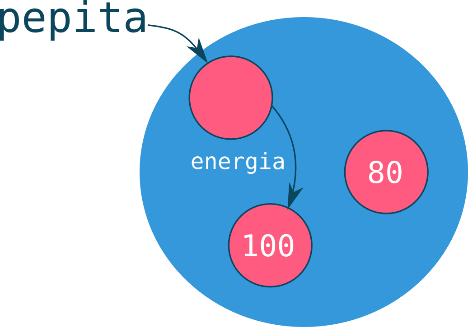
Si quiero que el valor asignado por el setter sea determinado al momento de enviar el mensaje, se puede usar un **parámetro**. Un parámetro es una referencia al valor que se pasa cuando se envía el mensaje, o desde el punto de vista del objeto receptor, al valor que el método recibe:

method energia(valor) {

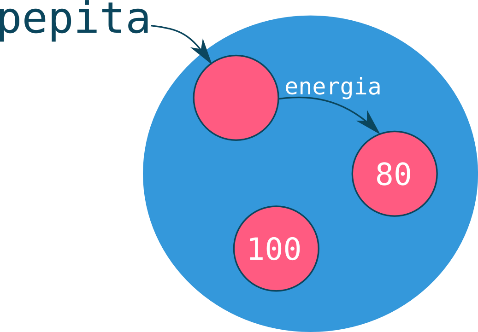
energia = valor

}

En objetos trabajamos con *referencias*: energia (un atributo) es una referencia a un objeto, en particular una referencia variable, que inicialmente *apunta* al objeto 100.



Luego, la operación de asignación cambia ese apuntador: La variable energia pasa a referenciar al resultado de la resta, que es el objeto 80.



Como regla práctica, si un método produce un efecto no debe retornar un valor.

*Delegar una responsabilidad*, o simplemente, **delegar**. Se trata de detectar tareas más pequeñas en las cuales descomponer una solución más compleja, identificar los objetos responsables de realizarlas y enviarles los mensajes correspondientes.

la forma en que un objeto delega un mensaje a **él mismo** , es utilizar la referencia self como objeto receptor del mensaje.

method volarHacia(destino) {

energia = energia - self.distancia(destino) / 10

ciudad = destino

}

También se puede crear un objeto al cual se le delegue una tarea, y luego que otro objeto llame a ese objeto.

# Referencias

la variable es una etiqueta que hace *referencia* a un objeto. De esta manera, cuando enviamos un mensaje a una variable, le estamos enviando el mensaje al objeto al que la variable hace referencia.

hay muchos tipos de referencias, y una de ellas son las variables del programa.

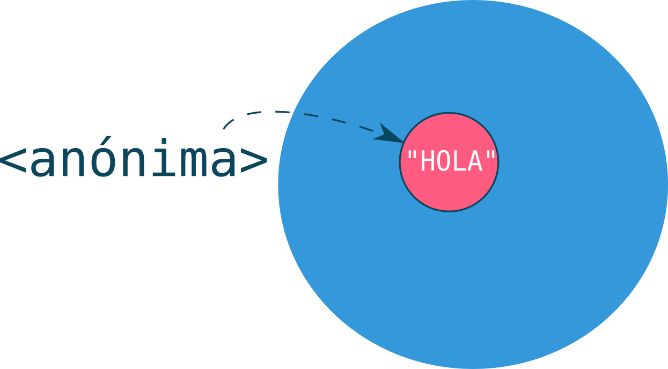
Cuando enviamos mensajes a objetos literales como el 2, el true o "hola", o expresiones, estamos conociendo a esos objetos a través de *referencias implícitas*, que pueden ser temporales (sólo existen durante ese envío de mensajes) y anónimas (no tienen un nombre asociado).

Var saludo = “hola”

saludo.capitalize().size()

^

+-- referencia implicita a "HOLA"



Pueden haber dos variables que sean referencias que apunten a un mismo objeto. En fin, asignar a una variable significa agregar una nueva referencia a un objeto existente.

Si queremos que una variable no varíe y haga referencia siempre al mismo objeto, entonces estamos hablando de una constante.

const otroSaludo = “buen día”

Los objetos bien conocidos tienen un comportamiento, un estado y un nombre. Este nombre es una referencia obvia.

Las referencias obvias son constantes y globales, es decir, cualquier objeto o programa las puede utilizar. Es decir, si el objeto1 y el objeto2 comparten al objeto compartido y el objeto1 modifica el estado del objeto compartido, entonces el objeto2 podrá ver esos cambios.

Se puede decir:

* *enviar un mensaje a la variable* *nieto*
* *enviar un mensaje al objeto* *nieto*
* *enviar un mensaje al* *nieto*

Lo importante es que entiendas que *siempre* estamos enviando el mensaje al objeto a través de una referencia.

# Programar en objetos es contar historias

Un sistema se entiende como un conjunto de objetos coordinados.

El conjunto de mensajes que un objeto entiende se lo llama comportamiento.

Hay que diseñar el modelado de los objetos, particularmente elegir qué variables son *atributos* y cuáles *métodos de consulta*. Por ejemplo, si una variable es *independiente* (**no depende de ningún otro valor**), es necesario modelarla como un *atributo*. En cambio, si es una *variable dependiente* (**se calcula a partir de otro valor**) es conveniente tener ese cálculo en un *método de consulta*.

# Práctica Objetos y Mensajes

Hay veces que es necesario que varios objetos comprendan el mismo mensaje, y esto implica tanto el nombre del mensaje como sus parámetros.

Es mala práctica "guardarse" un valor que depende de otros para ser calculado.

# Colecciones

Colección: una lista ordenada de objetos, la cual admite repetidos.

Ej. creamos una colección de objetos (*biblioteca de juegos*):

object biblioteca{

var juegos = [timbaElLeon, carlosDuty]

method juegos(){

return juegos

}

}

.add() 🡪 agrega elementos al conjunto

.remove() 🡪 quita elementos del conjunto.

Ej. agregamos devilMayLaugh y sacamos timbaElLeon:

Biblioteca.

object biblioteca{

var juegos = [timbaElLeon, carlosDuty]

method agregarJuego(){

juegos.add(devilMayLaugh)

}

method borrarJuego(){

juegos.remove(timbaElLeon)

}

}

.contains(elemento) 🡪 nos dice si el conjunto contiene cierto elemento.

.size() 🡪 nos devuelve la cantidad de elementos de la lista.

**Bloques**:

Son **objetos** que representan una secuencia de envíos de mensajes, sin ejecutar, lista para ser evaluado cuando corresponda.

var unNumero = 4

var incrementador = { unNumero = unNumero + 1 }

incrementador es un bloque. Si se lo quiere aplicar, es decir, que se ejecute su código, hay que hacer lo siguiente:

incrementador.apply()

Los bloques también pueden tener parámetros que se deben pasar a la hora de aplicarlo.

var sumarAOtrosDos = {numeroA, numeroB => unNumero + numeroA + numeroB }

sumarAOtrosDos.apply(1,2)

Un bloque siempre retorna el valor de la última sentencia ejecutada dentro del bloque, por lo que puedo usarla como si fuese una función anónima (¡como las expresiones lambda de funcional!)

**filter**: 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.

**find**: devuelve **sólo** un elemento, no una colección. Si ningún elemento cumple la condición, el programa rompe.

**all**: indica si todos los elementos de una colección cumplen un cierto criterio.

**any**: indica si alguno de los elementos de una colección cumple un cierto criterio.

**map**: a partir de una colección se obtiene **otra colección** con cada uno de los resultados que retorna el envío de un mensaje a cada elemento. El resultado de mapear una lista siempre va a ser una lista nueva con tantos elementos como la que recibió el mensaje map, pero modificados (mapeados).

c**ount:** sirve para saber cuántos elementos de mi colección cumplen la condición.

**sum**: calcula la sumatoria del resultado de evaluar el bloque contra cada elemento de la colección.

**forEach**: para cada elemento de la colección produce un efecto.

# Excepciones

Excepción: error explícito que **interrumpe el flujo del programa**.

Mediante la sentencia self.error(mensaje) se lanza una excepción.

las excepciones no abortan simplemente la evaluación del método, sino que también abortan la evaluación de toda la cadena de envío de mensajes.

Cuando trabajamos con excepciones, el orden es importante: lanzar una excepción interrumpe el flujo de ejecución a partir de ese momento, pero no descarta cambios realizados anteriormente.

# 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 declarando una **clase**. Dentro de esta clase iría todo ese código repetido. Si varios objetos tienen el código parecido, pero no puede ser generalizado para que sea el mismo, no nos sirve definir una clase.

La clase es nos sirve como **molde** para crear nuevos objetos.

A la acción de crear objetos a partir de una clase la llamamos **instanciar** y decimos que el objeto creado es una **instancia** de esa clase:

const objetoInstanciado = new Clase()

Al trabajar con clases tenemos que *inicializar* los atributos en algún lugar:

const objetoInstanciado = new Clase(atributo = 500)

Si en la clase ya se inicializa un atributo, al instanciarla es opcional enviar como parámetro un valor inicial para que sustituya al que ya está dado. Si no se envía, queda el que estaba.

En contrapartida, es obligatorio en el new() pasar como parámetro un valor inicial para cada atributo que no esté inicializado en la definición de la clase.

No necesitamos una referencia para cada uno. El objeto se crea con el new(), según el contexto vemos si lo queremos asignar en una referencia o si lo usamos para otra cosa.

const sobrevivientes = [new Sobreviviente(), new Sobreviviente(), new Sobreviviente()]

ciudad.agregar( new Sobreviviente() )

method crearZombi() { return new Zombi() }

También podemos instanciar un objeto, utilizarlo y nunca guardarlo ni referenciarlo, con lo cual luego que realizó su tarea lo perdemos:

const bouba = new Zombi()

bouba.salud() // Devuelve 100

new Sobreviviente(energia = 200).atacar( bouba )

bouba.salud() // Devuelve 60

Es importante tener en cuenta que nuestros objetos **también pueden crear otros objetos**, usando *new* con la clase que corresponda.

Por lo tanto, los casos en los que un objeto puede conocer a otro son:

* Cuando es un **objeto autodefinido**, como con los que veníamos trabajando hasta ahora.
* Cuando el objeto se pasa por parámetro en un mensaje, por ejemplo *juliana.atacar(bouba)* hace que *juliana* conozca a bouba durante la ejecución de *atacar*.
* Cuando tiene un atributo que haga referencia a ese objeto.
* Cuando un objeto crea otro usando *new*.

# Herencia

La herencia es un mecanismo que tiene por objetivo principal compartir código entre clases que tienen una naturaleza en común (o sea que pueden ser pensadas como que son del mismo tipo), y así evitar repeticiones.

Para decir que una clase hereda de otra usamos la palabra clave *inherits*:

class Bisonte {

var property peso

method comer(){

peso += 2

}

}

class BisonteVolador inherits Bisonte {

method yipYip(){

peso -= 1

}

}

Si declaro un atributo variable como una *property*, el *getter* viene incorporado (tanto en un objeto como en una clase)

Cuando un objeto recibe un mensaje, mediante Method lookup buscará el comportamiento requerido en la clase de la cual es instancia y, en caso de no tener un método para el mismo, en su superclase. Y si no está en su superclase, en la superclase de su superclase...

Wollok maneja la idea de **herencia simple**, eso significa que una clase tiene siempre una **única** superclase. Si al declarar una clase no especificamos cuál es la superclase, la misma por defecto es Object que es una clase que define el comportamiento básico de todos los objetos.

Si al seguir el method lookup para encontrar el método correspondiente al mensaje que recibió un objeto se llega hasta la clase Object y no está definido ahí, entonces va a fallar con el error que ya conocemos: el objeto no entiende el mensaje que le mandamos.

Cuando no tenemos una implementación que podamos generalizar, podemos declarar **métodos abstractos** en las superclases.

Los métodos abstractos se definen como un método común, pero sin cuerpo. Luego, sus subclases pueden aportar su implementación como lo venían haciendo, pero agregando la palabra clave **override** para explicitar que no es coincidencia que estén declarando un método cuyo nombre coincide con uno definido por su superclase.

class Animal {

// este método no tiene definición, es abstracto

method hacerRuido()

}

class Gato inherits Animal {

override method hacerRuido() {

return "Miau!!"

}

}

Es importante entender que esto funciona como esperamos gracias al **method lookup**. Cuando se le mande *esGroso* a un maestro agua, buscará una implementación en su propia clase *MaestroAgua* y al no encontrarla seguirá buscando en *Maestro*. Si la implementación que encuentra allí tiene un llamado a *self.poder()*, no buscará la implementación desde *Maestro*, sino que empieza nuevamente desde la clase a partir de la cual ese maestro fue instanciado, o sea *MaestroAgua*.

para usar la lógica heredada para un método que estamos redefiniendo tenemos que usar *super(),* para indicar que queremos usar la implementación definida en una superclase:

class Padre {

method m1() {

return 42

}

method m2(valor){

return self.m1() + valor

}

}

class Hijo1 inherits Padre {

override method m1() {

return super() - 1

}

}

class Nieto inherits Hijo1 {

method m3(){

return self.m2(8)

}

}

class Hijo2 inherits Padre {

override method m2(valor) {

return super(valor) \* 2

}

}

En la consola:

new Padre().m1()

=> 42

new Padre().m2(10)

=> 52

new Hijo1().m1()

=> 41

new Nieto().m3()

=> 49

new Hijo2().m2(10)

=> 104

*super()* no es un envío de mensaje como los que acostumbramos a ver:

* **¿Quién es el receptor?** *El receptor siempre es el objeto que recibió el mensaje inicialmente (o sea, igual que self)*
* **¿Cuál es el nombre del mensaje?** *El mensaje es aquel que se llama igual al que se está definiendo, porque es* ***la única situación*** *en la cual es conceptualmente correcto manipular de esta forma la búsqueda del método a ejecutar.*