

# Paradigmas de Programación

## Objetos

### CONCEPTOS IMPORTANTES

#### **Objeto:**

Definición de la cátedra: un objeto es una cosa que sabe hacer cosas por nosotros.

Definición técnica: un objeto es una representación computacional de un ente (una abstracción con razón de ser) que exhibe comportamiento. Tiene identidad, puede tener un estado interno y expone una interfaz (comportamiento). Instancia de una clase.

**Mensaje:** **son las cosas que le puedo pedir a un objeto que haga**; el nombre que le enviamos a nuestro objeto para decirle que haga algo, es la forma de hacer que un objeto realice una acción (forma que tenemos de interactuar con un objeto) que se denota:

```
unObjeto.unMensaje()
```

Un mensaje está dado por su nombre, su aridad (cantidad de elementos que recibe) y su tipo de retorno.

Para interactuar con un objeto mediante un mensaje, debe estar definido un método que respalde el mensaje.

**Método:** es lo que tiene que hacer el objeto cuando RECIBE/RECEPCIONA/SE LE ENVÍA un mensaje. Se lo define dentro de un objeto y es lo que hace el objeto cuando RECIBE el mensaje. Se denota dentro del objeto:

```
object perro{ ----- > Objeto
  method ladrado() {
    ...
  }
}
```

Clasificación de métodos:

→ Métodos que devuelven: **preguntas**

→ Métodos que tienen efecto: **órdenes**

Si queremos crear un método que nos devuelva el valor de un atributo del objeto, por convención de nombres, le colocamos el mismo nombre que el atributo.

**En objetos tratamos de que cada una sea por separado, es decir no podemos tener métodos que pregunten y realicen órdenes.**

**Para los métodos que tienen efecto**, le colocamos nombre de acciones en infinitivo.

**Para los métodos que devuelven**, le colocamos nombre de sustantivos.

**Atributos:** son las características del objeto. Se definen:

```
object nombreObjeto {
  var nombreAtributo = 100
  ...
}
```

**var** → VARIABLES

**const** → CONSTANTE

**La única forma de interactuar con un objeto es mediante MENSAJES, no podemos interactuar mediante nombres de atributos u otras cosas, SOLO MENSAJES**

**Para que un objeto entienda un mensaje, debe estar definido un método que respalde ese mensaje.**

**En objetos tenemos EFECTO y ESTADO, las cosas pueden cambiar.**

Cuando tenemos un atributo dentro de un objeto, se establece una “comunicación” entre ese objeto y su atributo, el cual es otro objeto.

**Los objetos son inmutables, no se pueden cambiar, y tienen identidad.**

**Cuando realizamos enviamos un mensaje a un objeto que implica un cambio en sus atributos, estos atributos no se modifican, sino que apuntan a otros atributos (otros objetos), ya que, como dijimos, los objetos son inmutables.**

Si un objeto no es apuntado por otro objeto entonces no puede recibir mensajes, por ende, no podemos interactuar con él.

## Garbage collector

Se lleva aquellos objetos que no son apuntados por ningún otro objeto. Si no son apuntados por ningún otro objeto entonces no puede recibir mensajes.

**MÁXIMA:** Si un objeto no tiene mensajes, no podemos interactuar con él.

**MÁXIMA:** Nunca arrancamos a plantear un objeto por sus atributos primero pensamos qué mensajes tiene que responder/entender.

## Getters y Setters

Aquellos métodos que consultan sobre los atributos de un objeto se denominan **getter**.

Aquellos métodos que setean un valor específico de un atributo en un objeto, se denomina **setter**. Se denota:

```
method nombreMetodo(unValorEspecifico) {  
  valorAtributo = unValorEspecifico  
}
```

Para otorgarle a un objeto un atributo que sea otro objeto ya definido la sintaxis es:

```
var atributo = objetoAtributo
```

De esta manera, referenciamos a un objeto como atributo de otro objeto.

Y en caso de que queramos que se modifique o consulte ese atributo objeto, a través de un método, se denota:

```
method random{ -----> método definido para el objeto  
  atributo.haceAlgo() -----> atributo también es un objeto referenciado  
  ...  
}
```

## Comparaciones

Identidad es distinto que Igualdad, en el caso de comparar por igualdad (==) alcanza con que ambos objetos sean parecidos para que dé TRUE, en caso de comparar por identidad (===) deben ser la misma “bolita” o el mismo objeto para que dé TRUE, incluso si los objetos son iguales, no dará TRUE. **IGUALDAD ≠ IDENTIDAD**

== -> comparación por igualdad

=== -> comparación por identidad

## Interfaz

Conjunto de operaciones o cosas que puedo hacer con un objeto, esto quiere decir que son los mensajes; por ende, la **interfaz** es el conjunto métodos (solo las firmas del método, no su comportamiento) que posee un objeto, dado que a través de ellos podremos interactuar con el objeto mediante mensajes.

**La INTERFAZ de un objeto define el TIPO del objeto.**

**Definición de cátedra:** una **interfaz** es el conjunto de mensajes que entiende un objeto, ya que los mensajes son solamente “la firma” del método, no abarca su comportamiento.


Todos los objetos definidos con “object” se los conoce como **Objetos Bien Conocidos** o **WKO (Well Known Object)**.

Para definir un método dentro de un objeto que implique la realización de otros métodos en secuencia, se declara:

```
method hacerRutina() {  
  self.comer(10)  
  self.salirAPasear()  
}
```

**self** es una referencia que tienen los objetos para referirse a sí mismos

Realizamos el override sobre la operación igualdad en un objeto, es decir redefinimos la operación mediante un método, mediante esto ese objeto será igual a todos los otros

IDAÑEZ, LUCIA MARÍA 

COSAS DE WOLLOK:-----

Para importar todos los objetos de un archivo:

**import nombreArchivo.\***

Para importar objetos uno por uno de un archivo:

**import nombreArchivo.nombreObjeto**

-----

IDAÑEZ, LUCIA MARÍA 🐱

En el mundo de objetos hay una diferencia entre DEVOLVER y HACER:

→ Los mensajes de **devolver** son como preguntas, estos suelen consultar por atributos del objeto.

→ Los mensajes de **hacer** son aquellos que producen un efecto, esto quiere decir que provocan un cambio, puede suceder que haya mensajes que no devuelvan nada, sin embargo, esto no quiere decir que no haya hecho nada, sino que hizo algo que no hemos visto en el momento.

Tomá nota 📝 de los elementos sintácticos que utilizamos:

- la palabra `object` delante del nombre del objeto.
- `{ }` para encerrar lo que estamos definiendo de ese objeto; en este ejemplo, todo el código.
- la palabra `method` delante del nombre de cada método del objeto.
- `()` al final del nombre, tal cual como lo usamos al enviar el mensaje.
- nuevamente `{ }` para encerrar todo lo que abarca el método.
- la palabra `return` para indicar cuál es la respuesta que se debe retornar en cada caso.
- Como detalle, a las cadenas de caracteres las encerramos entre `""`, los valores de verdad se son `true` y `false` y los numeros tal cual son.

## ESTADO

El **estado** de un objeto se conforma por su interfaz y los atributos que lo conforman, todo estado es siempre privado, esto quiere decir que sólo el objeto puede utilizar sus atributos.

Vamos a denominar **encapsulamiento** a esta característica de un objeto de ser el único que puede manipular sus propios atributos.

## Comportamiento

El **comportamiento de un objeto** está ligado a qué hace un objeto cuando recibe un mensaje, esto quiere decir que dos objetos pueden tener el mismo mensaje y, sin embargo, hacer cosas distintas.

### EJERCICIO CON EL QUE SE TRABAJA

#### Ejercicio 3 - Sueldo de Pepe

Implementar en WolloK los objetos necesarios para calcular el sueldo de Pepe.

El sueldo de pepe se calcula así:  $\text{suelo} = \text{neto} + \text{bono} \times \text{presentismo} + \text{bono} \times \text{resultados}$ .

El neto es el de la categoría, hay dos categorías: **gerentes** que ganan \$1000 de neto, y **cadetes** que ganan \$1500.

Hay dos bonos por presentismo:

- Es \$100 si la persona a quien se aplica no faltó nunca, \$50 si faltó un día, \$0 en cualquier otro caso;
- En el otro, nada.

Hay tres posibilidades para el bono por resultados:

- 10% sobre el neto
- \$80 fijos
- nada

No olvidar que existe la palabra reservada **return** para retornar un elemento

Jugar cambiándole a pepe la categoría, la cantidad de días que falta y sus bonos por presentismo y por resultados; y preguntarle en cada caso su sueldo.

Nota: acá aparecen varios objetos, piensen bien a qué objeto le piden cada cosa que hay que saber para poder llegar al sueldo de pepe.

->Primero determinamos qué es lo que necesitamos, esto consiste en conocer el sueldo de Pepe, esto quiere decir que declarando `pepe.suelo()` en la consola, nos diga su sueldo

->Comenzamos planteando un objeto Pepe, del cual queremos saber su sueldo, para ello creamos un método sueldo que lo calculará

```
object pepe {
  method sueldo() {
  }
}
```

Cuando no sabemos qué colocar, delegamos esa acción a uno mismo (al objeto sobre el que estamos trabajando) para luego resolverlo.

->Ahora tenemos en cuenta cómo se calcula el sueldo:

**neto + bono x presentismo + bono x resultados**

De esta manera nos encontramos con los problemas, el neto varía de acuerdo con la categoría, el presentismo de acuerdo con las faltas y los resultados son distintos tipos.

->Tratamos de resolver PRIMERO el problema de cálculo del sueldo, en este caso pondremos la palabra **return** dentro del método, seguido del cálculo, sin embargo, aún no sabemos qué colocar por lo tanto nos auto-delegamos esas acciones de cálculo que se necesitan.

```
method sueldo() {
  return self.neto() + self.bonoPresentismo() + self.bonoResultado()
}
```

->Comenzamos con el neto, para resolver esto se crea una variable **categoría** como atributo que se le asignará el objeto GERENTE o CADETE eventualmente; estos objetos deberemos crearlos para que retornen el sueldo neto correspondiente

```
object pepe {
  var categoria = gerente
}
```

Se le asigna el OBJETO GERENTE a la variable **categoría**

```
object gerente {
  method neto() {
    return 1000
  }
}
```

```
object cadete {
  method neto() {
    return 1000
  }
}
```

Se crea el objeto GERENTE/CADETE con un método que devuelve el sueldo neto que le corresponde

->Como nos auto-delegamos la responsabilidad de calcular el neto en base a nuestra categoría, deberemos crear un método que nos permita conocerlo:

```
object pepe {
  var categoria = gerente

  method sueldo() {
    return self.neto() + self.bonoPresentismo() + self.bonoResultado()
  }

  method neto() {
    return categoria.neto()
  }
}
```

Usamos **self.mensaje()** para hacer referencia a un método que tenemos nosotros (nuestro objeto)

Aunque categoría puede variar, se puede enviar un mensaje de esta manera ya que ambos objetos (gerente y cadete) reconocen el mensaje **neto**

->Para calcular el bonoPresentismo() definimos una variable **faltas** que representa los días que faltó Pepe y una variable **bonoPresentismo** a la cual se le asignará un objeto BONO que le corresponde a Pepe

```
object pepe {
  var categoria = cadete

  var faltas = 0

  var bonoPresentismo = bonoPorFaltas
}
```

->Para crear los objetos BONO, necesitaremos la cantidad de faltas que tiene Pepe (o el empleado), por eso debemos enviarlas por parámetro

->En este caso, como nos auto-delegamos la responsabilidad de calcular el total del bonoPresentismo, debemos crear un método que nos permita conocerlo:

```
method bonoPresentismo() {
  return bonoPresentismo.valor(faltas)
}
```

->El bonoPorFaltas da \$100 si la persona no faltó nunca y \$50 si la persona faltó una sola vez, en otro caso da \$0

```
object bonoPorFaltas {
  method valor(unaCantidadFaltas) {
    return (100 - 50 * unaCantidadFaltas).max(0)
  }
}
```

Esto también se puede hacer con if...

```
object bonoPorFaltas {
  method valor(unaCantidadFaltas) {
    if (unaCantidadFaltas == 0) {
      return 100
    } else if (unaCantidadFaltas == 1) {
      return 50
    } else {
      return 0
    }
  }
}
```

->También está el bono que no otorga nada, en este caso no es necesario pasarle un parámetro, SIN EMBARGO, debemos respetar el hecho de que le pasemos un parámetro debido a que, de no pasarle el parámetro, no será el mismo mensaje, ya que estos se definen por su nombre y aridad.

```
object bonoNada {
  method valor(unaCantidadFaltas) {
    return 0
  }
}
```

Le pasamos **unaCantidadFaltas** aunque no se use ya que, de no recibir las faltas, el objeto Pepe no se podría comunicar con bonoNada debido a que no presenta la interfaz adecuada para que se puedan comunicar

IDAÑEZ, LUCIA MARÍA 🐱

En el caso de los BONOS vemos que presentan un comportamiento distinto a pesar de entender el mismo mensaje.

->Para el resolver bonoPorResultado, creamos una variable **bonoResultado** donde le asignaremos un objeto que variará de acuerdo con el tipo de bono que le corresponda a Pepe (o empleado)

```
var bonoResultado = bonoPorcentual
```

->Nuevamente, como nos auto-delegamos la responsabilidad de calcular el total del bonoResultado, debemos crear un método que nos permita conocerlo:

```
method bonoResultado() {  
    return bonoResultado.valor(self)  
}
```

Nos pasamos a nosotros mismos (nuestro objeto) **self** por parámetro

->Para crear los objetos de los BONOS RESULTADOS necesitamos información adicional que consiste en conocer el tipo de sueldo neto de Pepe, para ello hay distintas maneras de hacerlo, podemos pasar como parámetro el sueldo neto, la categoría e, incluso, mandarse a uno mismo (a nuestro objeto) para realizar el cálculo (usamos la última porque es mejor). La diferencia es que en las dos últimas, pasamos un objeto, luego debemos usar los mensajes correspondientes para realizar el cálculo

```
object bonoResultadoNada {  
    method valor(unEmpleado) {  
        return 0  
    }  
}  
  
object bonoResultadoFijo {  
    method valor(unEmpleado) {  
        return 80  
    }  
}  
  
object bonoResultadoPorcentual {  
    method valor(unEmpleado) {  
        return unEmpleado.neto() * 0.10  
    }  
}
```

Para que el objeto Pepe se pueda comunicar con los distintos objetos bonosResultado indistintamente se debe colocar que su método **valor()** recibe por parámetro a unEmpleado aunque no lo usen, de esta manera tendrán todos la misma interfaz y entenderán el mensaje al momento que el objeto Pepe lo envíe

## Pilares de la Programación Orientada a Objetos

→ **Asignación de responsabilidades**: darle a un objeto la responsabilidad de hacer algo

→ **Polimorfismo**: capacidad que tiene un objeto de poder hablar indistintamente a otros objetos que sean distintos; un objeto trata polimórficamente a otros objetos cuando les envía los mismos mensajes y estos pueden entender y responder ya que tienen una interfaz común.

El polimorfismo se arraiga con la identidad de un objeto, porque el comportamiento de cada objeto es específico de ese objeto y, aun así, entienden el mismo mensaje, pero realizan distintas cosas, solo nos interesa que entienda el mismo mensaje.

Para que dos objetos sean polimórficos deben entender el mismo mensaje y que haya un tercero que los use indistintamente.

→ **Encapsulamiento**: no permitir que otros objetos modifiquen los atributos de un objeto.

El encapsulamiento se rompe cuando un objeto externo modifica los atributos de otro objeto directamente (ej.: por seteo de un valor), también se rompe cuando se "sabe" muchas cosas de otro objeto, esto quiere decir cuando se comienza a "preguntar" específicamente, en vez de directamente. Interactuar con un objeto mediante una anidación de mensajes también es romper el encapsulamiento.

Ejemplo de romper el encapsulamiento:

Todos los métodos en Wollok son públicos



```
object bonoResultadoPorcentual {
  method valor(unEmpleado) {
    return unEmpleado.categoria().neto() * 0.10
  }
}
```

Se rompe debido a que se puede preguntar directamente mediante `unEmpleado.neto()`, además de que se conoce mucho del objeto, es decir sabemos que tiene una categoría y, a su vez, esa categoría tiene un sueldo neto.

Clase 28/9/22

## Colecciones

Las **COLECCIONES** nos permiten agrupar objetos para luego poder operar sobre: un elemento en particular, un conjunto de elementos seleccionados mediante un filtro o sobre toda la colección.

Si bien una colección es un conjunto de objetos, es mejor pensar que es un conjunto de referencias, ya que la colección no tiene “adentro suyo” a los elementos, sino que la colección los conoce mediante una referencia.

### LISTAS

Las listas son objetos que referencian a sus elementos, los cuales también son objetos, mediante un índice numérico que indica la posición que tienen.

Las listas al ser objetos se pueden interactuar con ellas mediante una serie de mensajes.

#### MENSAJES MEDIANTE LOS QUE PODEMOS INTERACTUAR CON LAS LISTAS

Mediante el mensaje **get(...posición...)** a la lista y esta nos trae el elemento que esté en una posición específica que que mandemos como parámetro.

Mediante el mensaje **add(...elemento...)** la lista agrega un elemento.

Mediante el mensaje **size(...nombreLista...)** sabemos el largo de la lista, su cantidad de elementos.

Mediante el mensaje **count()** sabemos cuántos elementos de nuestra colección cumplen una condición.

Si queremos calcular la sumatoria de una lista usaremos **sum()**.

Se puede saber el primer elemento de la lista, es decir la cabeza, mediante **first(...nombreLista...)** y el último elemento mediante **last(...nombreLista...)**.

Se puede utilizar **drop(n)** como en funcional, esto quiere decir que elimina los “n” primeros elementos que determinemos, siendo “n” el número que tome como parámetro el mensaje.

Se puede utilizar **take(n)** que toma los primeros “n” elementos, siendo “n” el número que tome como parámetro el mensaje.

También se puede utilizar **contains(objeto)** el cual nos indica si un elemento pertenece a la lista.

Se puede utilizar **remove(objeto)** que elimina un elemento (objeto) de la lista, este debe ser colocado como parámetro.

Mediante **addAll([...])** agregamos una lista de objetos a nuestra lista.

Para filtrar mi lista/colección, es decir para quedarme con todos aquellos elementos que cumplan una condición, es necesario pasarle al mensaje **filter()** un bloque.

Los **bloques** son objetos que representan una porción de código que no se evalúa, este objeto entiende el mensaje **apply()** que, cuando se envía al bloque, evalúa el bloque de código.

También hay bloques que reciben parámetros (como las funciones lambda o funciones anónimas) y se denotan de la siguiente manera:

**{ numero => 2 + numero }**

Sin embargo, este bloque por sí mismo no realiza ninguna acción, debemos colocar delante de él el mensaje **apply()** junto con el parámetro que recibirá el bloque:

**{ numero => 2 + numero } . apply( 3 )**  
**>5**

**3** es el parámetro que recibirá el bloque y se evaluará

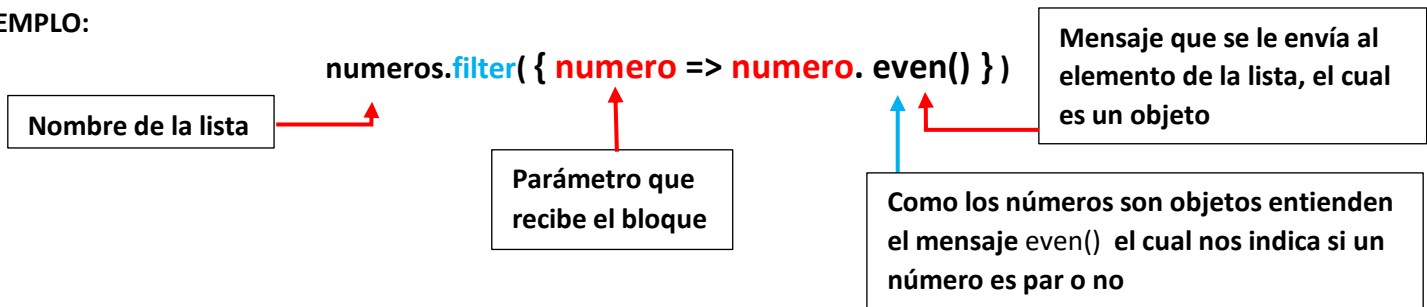
El bloque es parecido a una función lambda en funcional.



IDAÑEZ, LUCIA MARÍA 🐱

Como se mencionó, los **filter()** esperan bloques como parámetros, estos bloques deberán recibir un parámetro, el cual será un elemento de la lista, y evaluarán cada elemento de la lista mediante una condición booleana. Los bloques deben ser condiciones, es decir retornan un booleano.

**EJEMPLO:**



En este caso, al **filter()** le estamos pasando un objeto como parámetro y este objeto tiene la particularidad de que puede ser evaluado en cualquier momento. Nosotros creamos el bloque y el filter se encargará de mandarle al objeto que recibe por parámetro el mensaje **apply()**.

Si lo que queremos no son todos los elementos que cumplan una condición sino alguno que la cumpla, se utiliza **find()**, este mensaje recibe un bloque como parámetro y toma el primer elemento de la lista que cumpla la condición. Devuelve un solo elemento, no una lista.

**EJEMPLO:**

```
var algunosNumeros = [1, 2, 3, 4, 5]
algunosNumeros.find({unNumero => unNumero > 3})
// va a devolvernos el 4,
// porque es el primero que cumple la condición.
```

Así como existe el filter, también está el **map()** el cual aplica ese bloque, que recibe un parámetro, a todos los objetos de la lista.

`numeros.map( { numero => numero.even() } )`

En este caso, se devolverá una lista de booleanos.

También está el **all()**, que pregunta por si todos cumplen. Recibe un bloque como parámetro.

```
>>> numeros.all({ numero => numero > 0 })
```

El **any()**, pregunta por si alguno cumple una condición. Recibe un bloque como parámetro.

```
>>> numeros.any({ numero => numero.even() })
```

Para negar se utiliza **!**, como en C.

Con las LISTAS podemos tener elementos repetidos y admite la idea de un orden mediante índices numéricos que se le asignan a elementos de la lista.

Otro mensaje que entienden las listas es el **forEach()** el cual recibe un bloque, sin embargo la acción que se ejecuta en este bloque no debe retornar nada, esto quiere decir que solo va a tener un efecto en el sistema (solo va a tener efecto en el diagrama).

**EJEMPLO:**

```
>>> numeros.forEach({ numero => numero * 2 })
```

## Set

Hay otro tipo de colecciones que se denominan SET que no posee ninguna de las particularidades de las listas, es decir que no tiene un orden definido (no podemos acceder a los elementos de la lista por medio de un índice) y no admiten los elementos repetidos. Los SET se denotan de la siguiente manera:

`#{ 1, 2, 3 }`

Los SET no entienden mensajes de posiciones, como last, first, get(2), etc. ya que son mensajes que tienen una idea de orden.

Los SET entienden mensajes de add(), sin embargo, en caso de que el elemento ya esté en el SET, no lo vuelve a agregar.

IDAÑEZ, LUCIA MARÍA 🐱

El SET entiende todos aquellos mensajes que no tengan que ver con una idea de orden y de elementos repetidos. Sin embargo, al utilizar todos estos mensajes, en vez de devolver otro set, devuelve una lista.

## Clases

Las **clases** son **MOLDES** los abarcan a objetos que tienen características similares como comportamiento (interfaz) y atributos, por lo tanto, para no repetir lógica/código puedo agruparlos en clases, haciendo que los objetos pregunten por sus métodos y atributos a la clase que pertenecen.

- Las CLASES no son objetos, son MOLDES para instanciar los objetos.
- Las CLASES definen los atributos que tienen/tendrán los objetos.
- Las CLASES proveen los métodos que tienen/tendrán las instancias de esos objetos.

```
class NombreDeClase {  
    ...Atributos...  
  
    ...Interfaz...  
  
}
```

En casi todos los lenguajes de objetos, las clases arrancan con mayúsculas.

Al momento de hacer el test podemos definir a los objetos específicos:

```
const nombreObjeto = new NombreClase(...atributos específicos de ese objeto...)
```

```
const pepita = new Golondrina()
```

El OBJETO es “pepita” y está creado en base a la CLASE “Golondrina”.

```
> const pepito = new Golondrina(energía = 180)
```

El OBJETO es “pepito”, está creado en base a la CLASE “Golondrina” y tiene como atributo “energía = 180”.

### TENER EN CUENTA:

```
method vivirUnDia() {  
    enfermedades.forEach({ enfermedad => enfermedad.afectarA(self) })  
}
```

Este método toma una lista de enfermedades, a esa lista se le envía el mensaje **forEach()** con un bloque, este bloque recibe un parámetro que es cada elemento de la lista “enfermedades” y se le envía un mensaje el cual es **afectara(self)**, esto quiere decir que afectan al objeto que tiene el método (no devuelven nada, solo producen efecto en el sistema, en este caso el objeto).

### TENER EN CUENTA:

1. ¿De quién es la responsabilidad de...?
2. ¿Quién tiene la información mínima e indispensable para...?

Se pueden anidar los mensajes para un objeto (al igual que la composición en lógico).

EJERCICIOS: Hacer el DrCasa, guías de Mumuki = 6, 7, 8 y 10, guías de PdP = 1, 2, 3 y 4

**CLASE 5/10**

En Objetos tenemos un objeto que recibe mensajes, por lo tanto, va de atrás para adelante, es decir de izquierda a derecha, por lo tanto, el objeto que está a la izquierda (al comienzo de la línea) se le envía un mensaje (este mensaje se declara a su derecha) y así sucesivamente.

## Diagrama de Clases

El **diagrama de clases** es una herramienta que permite modelar las relaciones entre las entidades, estas entidades se representan mediante un rectángulo que posee tres divisiones: nombre, atributos y operaciones, también se pueden representar mediante el nombre y la interfaz.

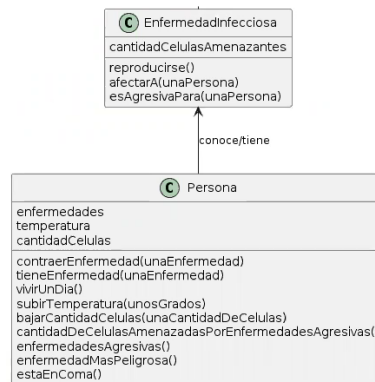
En el paradigma OO se utiliza el diagrama de clases, este diagrama es estático. A partir de este diagrama se conoce la estructura de las instancias de los distintos objetos.

El diagrama de clases es la representación más alta en abstracción de nuestro sistema, por lo tanto, el diagrama no va a contener el detalle algorítmico, sino que va a ser una representación de las clases con los atributos y los métodos más importantes y, a su vez, cómo se relacionan las clases entre sí.

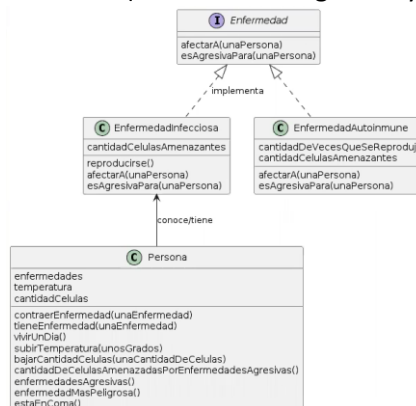
Los diagramas de clases deben ser siempre conexos, esto quiere decir que entre los objetos deben tener “flechas” y estas flechas son relaciones entre los objetos.

En los diagramas de clases hay distintos tipos de relación:

->Flechita llena: significa que una clase conoce a otra, la clase de la que sale la flecha conoce/tiene a la clase a la que llega la flecha.



->Flechita punteada con punta blanca: simboliza que una clase implementa una interfaz. Las interfaces contienen aquellos métodos que tienen en común dos clases (no cuentan los getters y setters).



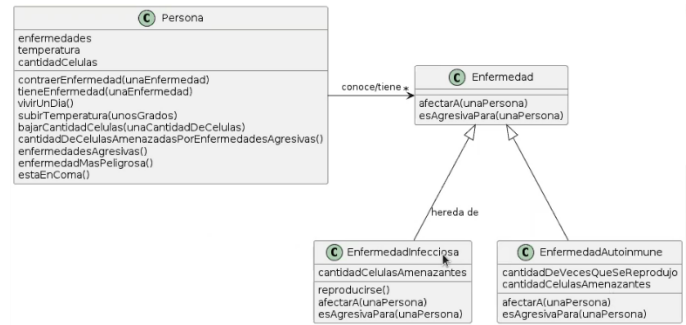
->Flechita de relación uno a muchos: se indica con un \* (asterisco) en la parte de muchos, es decir la flechita sale una clase y donde llega la flechita se pone el \*.

IDAÑEZ, LUCIA MARÍA 🐱

->Flechita llena con punta blanca: esta misma representa que dos clases HEREDAN de otra clase, y se suele colocar como extends.

→ Las clases nos permiten evitar la repetición de lógica entre objetos.

Cuando hay comportamiento distinto entre las SUPERCLASES y las clases herederas, es conveniente representar ese método/comportamiento de manera individual para cada uno, es decir que representamos el comportamiento en ambas clases



## Herencia

La lógica repetida entre distintas clases se maneja mediante la **herencia**. En Wollok el mecanismo de herencia que se utiliza es la HERENCIA SIMPLE.

La **HERENCIA SIMPLE** significa que una clase puede heredar únicamente de una clase, la cual llamamos SUPER CLASE. La herencia en Wollok se representa mediante la palabra reservada **inherits**, esta palabra se coloca seguido de la clase que queremos herede de otra clase.

En el caso de DR. CASA queremos que la clase Médico herede de la clase Persona, ya que de esta manera evitamos la repetición de código dado que un Médico también puede contraer una enfermedad, así como también una enfermedad lo puede afectar, etc.

```
class Medico inherits Persona{  
    ...  
}
```

En el caso de que tengamos dos clases parecidas, es decir que hay una tercera clase/objeto que trata a esas dos polimórficamente, pero tienen la lógica de sus métodos es distinta (las resoluciones de sus métodos son distintas), también podemos crear una clase de la que hereden.

Esto mismo sucede para EnfermedadInfecciosa y EnfermedadAutoinmune, tienen resoluciones de métodos distintas, pero ambas son enfermedades, esto mismo podemos simplificar haciendo que ambas hereden de una clase Enfermedad.

```
class EnfermedadAutoinmune inherits Enfermedad{  
    ...  
}
```

Estas clases (las SUPER CLASES) en particular no las queremos instanciar, ya que no nos interesan, aquello que nos interesa de estas clases son las clases que heredan de ellas dado que, a partir de estas, instanciaremos objetos. Estas clases son denominadas **CLASES ABSTRACTAS**, no nos interesa instanciarlas, sino que nos sirven para extraer comportamiento a partir de ellas.

Se las representa en el Diagrama de Clases mediante **abstract class NOMBRE\_SUPER\_CLASE{}**

Dentro de las clases abstractas habrá métodos obligatorios para todas aquellas clases que hereden de ellas, estos métodos se denominan **MÉTODOS ABSTRACTOS**. Los métodos abstractos son propios de cada clase hija que hereda de la clase abstracta, es decir que la lógica de los métodos abstractos depende de cada clase hija (clase heredera). En el código los métodos abstractos se representarán solo con su nombre y su aridad.

```
method afectarA[unaPersona]
```

Entonces, ¿qué hacemos con esos métodos para los cuales dos clases distintas tienen una lógica diferente?

Estos son los casos de `afectarA(unaPersona)` y `esAgresivaPara(unaPersona)`, cada uno de estos métodos está definido para una clase, por lo tanto, podemos decir que son métodos abstractos, es decir que son propios de cada clase.

La forma de solucionar esto es que, al momento de representar el método, junto con su propia lógica, en la clase heredera se debe colocar un **override** frente al método. De la siguiente manera:

```
override method afecta|rA(unaPersona) {  
    unaPersona.subirTemperatura(cantidadCelulasAmenazantes / 1000)  
}
```

Esto mismo se hace con todos los métodos abstractos de las clases herederas de una super clase.

Se puede hacer **override** de métodos que no sean abstractos

IDAÑEZ, LUCIA MARÍA 🐱

Volviendo a los médicos, el enunciado decía que “si un médico contrae una enfermedad, este trata de curarla atendiéndose a sí mismo”. Esto nos quiere decir que los Médicos contraen una enfermedad de manera “diferente” a como lo modelamos en Persona, por lo tanto, debemos modificarlo.

```
override method contraerEnfermedad(unaEnfermedad) {  
  enfermedades.add(unaEnfermedad)  
  self.atenderA(self)  
}
```

Esto tiene sentido, ya que hacemos un **override** de otro método que ya existe en nuestra SUPER CLASE. Sin embargo, para estos casos en particular, observamos que se sigue repitiendo un poco de lógica dado que la línea **enfermedades.add(unaEnfermedad)** era parte de la lógica del método **contraerEnfermedad(unaEnfermedad)** que está en la clase Persona, para evitar esta repetición de lógica en métodos donde los estamos “pisando” pero de manera parcial, es decir que solo lo hacemos para agregar una parte extra, utilizamos **super(ParámetroDelMétodoDondeEstemos)**. “super” lo que hace es ir a buscar comportamiento de la superclase para el método en el cual está, realiza ese comportamiento (esto quiere decir que lo ejecuta) y cuando termina, vuelve a la clase heredera de la superclase, es decir donde está parado, y ejecuta esa lógica extra que agregamos al método.

```
override method contraerEnfermedad(unaEnfermedad) {  
  super(unaEnfermedad)  
  self.atenderA(self)  
}
```

Como vemos acá, modificamos la parte de **enfermedades.add(unaEnfermedad)**, el cual era código repetido, y lo reemplazamos por **super(unaEnfermedad)**, de esta manera buscará comportamiento en la clase Persona para el método **contraerEnfermedad(unaEnfermedad)**, lo ejecutará y, cuando termina, vuelve a la clase Medico y ejecuta el resto de lógica extra que le corresponde al método en la clase Medico.

## Method Look-Up

El **method lookup** es el mecanismo que tiene un objeto de buscar comportamiento dentro de sí mismo.

Si un objeto recibe un mensaje, mediante Method lookup buscará el comportamiento requerido en la clase de la cual es instancia, y si no lo encuentra, irá a sus superclases, así hasta llegar a la raíz; en caso de no encontrarlo, el método no existe.

//En el caso de WKO, estos reciben un mensaje, mediante Method lookup, buscan el comportamiento requerido dentro de sí mismo y, si no lo encuentran irán a buscarlo a sus **superclases**; sucede lo mismo en caso de no encontrar el método//

Esta cadena de herencia puede tener tanto niveles como queramos.

El method lookup cada vez que encuentra un método va desde abajo hacia arriba, por lo tanto, si está en la superclase buscando comportamiento de un método y dentro de ese método hay otro, vuelve hacia abajo (arranca desde 0).

## SEGUIMOS CON EJERCICIO DR. CASA PARA EXPLICACIÓN

### TEMP. 2

Clases que entienden los mismos mensajes tienen la misma interfaz, esto hace que esas clases sean del mismo tipo. Esto quiere decir que podemos tener clases que sean de un tipo sin que hereden directamente de ese tipo, esto sucede con JefeDeDepartamento, nosotros podemos hacer que sea un Médico sin que herede directamente de la clase Médico; esto implica que repita código en algún momento.

→ Un método está dado por su nombre, aridad (cantidad de parámetros) y su tipo de retorno.

En el caso de LaMuerte (la enfermedad) es posible que sea un objeto, ya que no tiene atributos, el hecho de que no tenga atributos nos indica que es la misma Muerte para todos los objetos que instanciados de nuestro sistema. Esto mismo no sucede en el caso de, por ejemplo, enfermedadAutoinmune; esta misma tiene que ser una clase, ya que tiene el atributo cantidadDeVecesQueSeReprodujo el cual es propio para cada persona que contraiga la enfermedad.

## Estado

Hay objetos que tienen ESTADO (o StateFull), es decir que tienen atributos los cuales van a tener valores y esto implica que le proveen al objeto un estado. Por otro lado, hay objetos que no tienen estado (StateLess), esto es lo mismo que decir que no tienen atributos.

Ejemplo:

```
object laMuerte { // No tiene Atributos, No tiene ESTADO o es Stateless

  method afectarA(unaPersona) {
    unaPersona.morite()
  }

  method esAgresivaPara(unaPersona) {
    return true
  }

  method atenuar(unaCantidad) {
    // No hace nada
  }

}
```

En este caso, la Muerte es una enfermedad ya que entiende los mismos mensajes que las otras enfermedades, por lo tanto, tiene la misma interfaz, entonces es del mismo tipo.

**EL TIPO DE UN OBJETO VIENE DADO POR LOS MENSAJES QUE ENTIENDE (INTERFAZ) Y DESPUÉS POR EL TIPO PROPIO Y EL DE SU JERARQUÍA**



**property** nos genera el getter y setter automáticamente. En el caso que sea **const** solo nos genera el getter.

#### PUNTO POR REALIZAR

- **Cameron, que es A le intente donar a House, que es O, 1100 células.**

Comenzamos planteando un test de esto, esto quiere decir que debemos tener el objeto Cameron el cual entiende un mensaje que lo llamaremos donarA(..., ...) donde su primer parámetro será a quien dona, en este caso house, y el segundo la cantidad de células a donar, serán 1100.

```
cameron.donarA(house, 1100)
```

Este test dará en rojo, ya que cameron no entiende el mensaje donarA(..., ...), por lo tanto procedemos a realizarlo. Nos planteamos lo que sucede cuando una persona dona una cantidad de células, por enunciado, nos dice que disminuye su cantidad de células mientras que la persona que recibe las células (a quien le donaron las células) aumenta tantas células como la cantidad que le hayan donado.

```
method donarA(unaPersona, unasCelulas) {
  self.bajarCantidadCelulas(unasCelulas)
  unaPersona.aumentarCantidadCelulas(unasCelulas)
}
```

Sin embargo, debemos tener en cuenta las condiciones para donar a una persona una determinada cantidad de células, estas se dan por el enunciado de Dr. Casa:

- Las personas tienen que ser compatibles, según su factor sanguíneo:
  - Los A pueden donarle a los A, y los R, y reciben de A y de O
  - Los R sólo pueden donarle a los R, pero reciben de todos
  - Los O les donan a todos, pero sólo reciben de A
- La cantidad por donar tiene que ser mayor a 500 células, pero menor o igual a un cuarto de las células totales.

Para cumplir esto, realizaremos un método que se llame puedeDonarA(unaPersona, unasCelulas), de esta manera validaremos que la persona, y nosotros, cumplamos con las condiciones para donar.

VALIDACIÓN

```
method donarA(unaPersona, unasCelulas) {
  if (self.puedeDonarA(unaPersona, unasCelulas)) {
    self.bajarCantidadCelulas(unasCelulas)
    unaPersona.aumentarCantidadCelulas(unasCelulas)
  }
}
```

Ahora realizamos puedeDonarA(unaPersona, unasCelulas), primero analizamos cuándo una persona puede donar células a otra, este caso se da cuando el tipo sanguíneo de ambas personas son compatibles y cuando esa cantidad de células está entre un intervalo propiamente definido, entonces realizamos un método que valide cada condición.

```
method puedeDonarA(unaPersona, unasCelulas) {
  return self.esCompatibleCon(unaPersona) && self.puedeDonar(unasCelulas)
}
```

Procedemos a desarrollar ambos métodos:

```
method puedeDonar(unasCelulas) {
  return unasCelulas <= cantidadCelulas / 4 && unasCelulas >= 500
}
```

En el caso de esCompatibleCon(unaPersona) debemos analizar la compatibilidad de los factores sanguíneos de cada persona, para ello debemos agregar un atributo a la clase Persona que nos indique el factor sanguíneo de esa persona. Una vez hecho esto, la RESPONSABILIDAD de saber si un factor sanguíneo es compatible con otro (le puede donar a otro), es del factor sanguíneo, por lo tanto, este atributo debe entender el mensaje puedeDonarA(otroFactorSanguineo).

```
method esCompatibleCon(unaPersona) {
  return factorSanguineo.puedeDonarA(unaPersona.factorSanguineo())
}
```

Para saber si el factor sanguíneo es una clase o un objeto, primero, vamos a tener en cuenta qué es lo que tiene que hacer factor sanguíneo, es decir los mensajes que debe entender, en este caso puedeDonarA(otroFactorSanguineo). Para este caso, vamos a aprovechar la identidad de los objetos, ya que va a ser necesario una comparación entre objetos, a su vez los factores van a ser WKO, es decir van a ser globales para todo nuestro sistema. Además de que los factores sanguíneos son los mismos para todas las instancias de la clase Persona, es decir que no tienen estado interno de acuerdo con la persona.

```
object factorA {
  method puedeDonarA(otroFactorSanguineo) {
    return [self, factorR].contains(otroFactorSanguineo)
  }
}

object factorR {
  method puedeDonarA(otroFactorSanguineo) {
    return otroFactorSanguineo == self
  }
}

object factorO {
  method puedeDonarA(otroFactorSanguineo) {
    return true
  }
}
```

Lo importante de modelar los factores de esta manera, es que fue posible debido a que los factores solo exhiben un comportamiento.

Observemos porqué no podemos plantear esto mediante clases:

Como las clases sirven para representar instancias, cada instancia de un factor tendría una identidad distinta, cuando en realidad un factor sanguíneo es igual a otro (siempre y cuando sean del mismo tipo), por lo tanto, al momento de compararlos por igualdad, no será posible ya que no tienen la misma identidad.

- La identidad se ve en el diagrama de objetos, si es la misma bolita, es que es la identidad es la misma
- La identidad es fuerte cuando necesitamos comparar por igualdad para saber que nuestro objeto es único en nuestro sistema, es decir que la “bolita” de nuestro diagrama es única

Volvemos al test...

### MANEJO DE ERRORES

```
test "Cameron intenta donarle a House 1100 células" {
  const cameron = new Persona(factor = factorA, temperatura = 36, cantidadCelulas = 4000)
  const house = new Persona(factor = factorO, temperatura = 36, cantidadCelulas = 4000)
  cameron.donarA(house, 1100)
  assert.equals(4000, cameron.cantidadCelulas())
  assert.equals(4000, house.cantidadCelulas())
}
```

En este caso el test es correcto, ya que cameron no cumple las condiciones para donar sangre a house, por lo tanto, ambos conservan la misma cantidad de células. Sin embargo, el enunciado nos pide que en caso de que no se pueda realizar una transfusión, se debe informar correctamente.

La mejor forma de realizar esto es haciendo que nuestro programa falle, ¿cómo se hace esto? Lanzando un error.

Este error corta el flujo de la ejecución y es “agarrado” por quien pueda solucionarlo.

- Los métodos que hacen algo, no retornan nada. Un método hace algo o retorna un valor, no existe un método que haga algo y retorne un valor. No puede haber un método que haga un efecto y retorne.

Lo que vamos a hacer para que nuestro programa lance un error es agregar el comportamiento por la parte **else** del **if**.

### LANZAMIENTO DE EXCEPCIONES

IDAÑEZ, LUCIA MARÍA 🐱

El término que vamos a utilizar es EXCEPCIÓN, esto es la indicación de un problema (puede o no ser un error) que ocurre durante la ejecución del programa. La particularidad de las EXCEPCIONES es que cortan el flujo de ejecución del programa hasta que alguien lo resuelve.

Para realizar este **lanzamiento de excepciones** se lo representa con:

**throw new Exception**(message = “\* acá va el mensaje\*”)

IDAÑEZ, LUCIA MARÍA 🐱

En nuestro ejercicio quedó representado de la siguiente manera:

```
method donarA(unPersona, unasCelulas) {
  if (self.puedeDonarA(unPersona, unasCelulas)) {
    self.bajarCantidadCelulas(unasCelulas)
    unPersona.aumentarCantidadCelulas(unasCelulas)
  } else {
    throw new Exception(message = "No se pudo realizar la donación")
  }
}
```

En nuestro test debemos tener en cuenta que ESTÁ BIEN QUE FALLE, ya que eso es lo que buscamos por lo tanto vamos a utilizar: `throwsExceptionWithMessage(Mensaje, bloque de código que va a lanzar la excepción)`

```
test "Cameron intenta donarle a House 1100 células" {
  const cameron = new Persona(factor = factorA, temperatura = 36, cantidadCelulas = 40)
  const house = new Persona(factor = factor0, temperatura = 36, cantidadCelulas = 40)
  assert.throwsExceptionWithMessage("No se pudo realizar la donación", {
    cameron.donarA(house, 1100)
  })
}
```

### FAIL FAST

Concepto: FAIL FAST → Que algo falle rápido

→ Esto es una buena práctica

Para hacer que nuestro bloque falle rápido, debemos realizar PRIMERO nuestra validación, esto quiere decir que la validación debe ser lo primero en ejecutarse/evaluarse (debe estar más arriba en el código).

```
method donarA(unPersona, unasCelulas) {
  if (! self.puedeDonarA(unPersona, unasCelulas)) {
    throw new Exception(message = "No se pudo realizar la donación")
  }
  self.bajarCantidadCelulas(unasCelulas)
  unPersona.aumentarCantidadCelulas(unasCelulas)
}
```

Se realiza primero la validación, en este caso SI NO PUEDE DONAR

En caso de que no se pueda donar, se CORTA el flujo de ejecución, es decir que tira el mensaje de excepción y la ejecución del programa se detiene. Por otro lado, si puede donar, pasa no pasa el if y continua con la ejecución debajo del mismo.

Para que el código quede más limpio, es una buena práctica abstraer la validación en otro método. En este caso, abstraemos el if en otro método llamado `validarDonacion(unPersona, unasCelulas)`.

```
method donarA(unPersona, unasCelulas) {
  self.validarDonacion(unPersona, unasCelulas)
  self.bajarCantidadCelulas(unasCelulas)
  unPersona.aumentarCantidadCelulas(unasCelulas)
}

method validarDonacion(unPersona, unasCelulas) {
  if (! self.puedeDonarA(unPersona, unasCelulas)) {
    throw new Exception(message = "No se pudo realizar la donación")
  }
}
```