RESUMEN DE OBJECT BROWSER

- Workspace
    - Las variables permanecen (entonces me doy cuenta de que hay **efecto colateral**).

Para qué uso el

- Evaluate It: cuando necesito enviarle un mensaje a un objeto; no me importa si devuelve algo. *Ejemplo:* pepita energia: 30 (el resultado es el efecto colateral sobre pepita, no es relevante lo que devuelve).
- Show It: cuando le quiero enviar un mensaje a un objeto y sí me interesa lo que devuelve. *Ejemplo:* pepita energia (acá sí me interesa saber qué energía tiene pepita).
- Inspect It: cuando envío un mensaje a un objeto y quiero ver cómo quedó el estado interno del objeto que me devuelve ese método. Ejemplo: pepita energia: 30 y luego marco pepita y hago Inspect. ¿Qué concepto se está rompiendo? El encapsulamiento, porque estoy viendo la implementación de un objeto por adentro. Ok, a veces me puede resultar útil…

¿Qué pasa cuando un objeto recibe un mensaje?
Pasa que se evalúa un método, un pedazo de código con el mismo selector que el mensaje. A la búsqueda de ese método se la llama "**Method Lookup**".

¿Y dónde se busca ese método? ¿Dentro del objeto?
Depende… todo depende:

- En el caso de los objetos que nosotros mismos creamos, sí: el método está en el objeto. Cuando un objeto recibe un mensaje, se fija dentro de sí mismo a ver si tiene el método correspondiente. En el caso de los objetos copiados, se copia la estructura interna (atributos, sin sus valores) y los métodos, por lo tanto trabajan de igual forma.
- Peeero… siempre hay un pero… en el caso de los objetos clonados, los clones trabajan un poco distinto: los clones no tienen los métodos, sino que usan los métodos del objeto original.

¿Pero cómo…? Si el clon invoca los métodos del objeto original, ¿no va a modificar las variables del original también?
El truco para que esto no pase es que el clon no "invoca" al método, sino que "se trae" el método y lo evalúa en su propio contexto, no en el del objeto original. El método lo podemos imaginar como a una receta de cocina: dice qué se tiene que hacer, pero corresponde al cocinero (el objeto receptor) hacer eso que se indica.

INTRODUCCIÓN AL CONCEPTO DE CLASE

Bien, tenemos nuestra consultora de sistemas (¡qué bajo hemos caído![1]) y decidimos contratar a nuestros mejores alumnos como programadores, dándoles un sueldo acorde.
Entonces tenemos tres empleados:

- Matías Karlsson
- Andrés Fermepin
- Sebastián Alvarez

Voy a tener tres objetos que representan el concepto empleado y que tienen características similares. Es lógico que cuando les pregunte su sueldo, cada uno me diga lo que gana (preguntarle a cada uno):

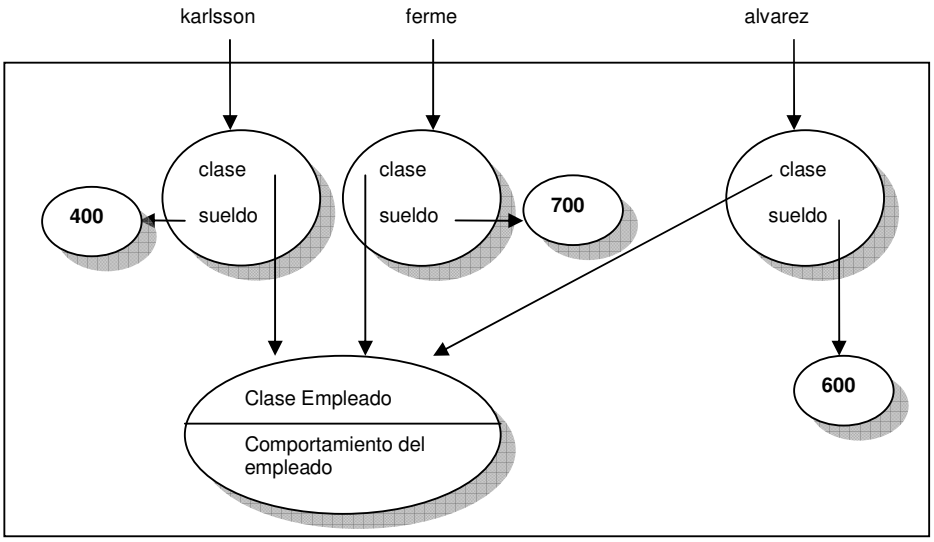
karlsson cuantoGanas 400
ferme cuantoGanas 700
etc.etc.

Si lo tengo que hacer en el Object Browser

- Una opción es definir un objeto *prototipo*: hay un empleado que sirve como concepto guía para los demás y es responsable de definir el comportamiento en ese único lugar, los otros empleados se clonan a partir del objeto inicial. El tema es qué empleado elijo como prototipo: karlsson, ferme o alvarez.
- Si no tengo que programar n veces los mismos métodos.

Método de karlsson	Método de alvarez	Método de ferme
cuantoGanas	cuantoGanas	cuantoGanas
^sueldo	^sueldo	^sueldo

Cada uno gana distinto, pero la forma en que obtienen el sueldo es el mismo. Cuando pregunto cuánto ganás, eso depende del sueldo que le asignó la consultora (además, podría decidir calcular el sueldo con retenciones, familiares a cargo, etc. y tendría que repetir el mismo cálculo en cada uno de los empleados, lo cual tiene escasa onda).
Cuando el comportamiento se repite, podemos agrupar los objetos en **clases**.



[1] El lector podrá ubicar esta frase entre el chiste, la ironía sutil o la cruda realidad, según su propia experiencia.

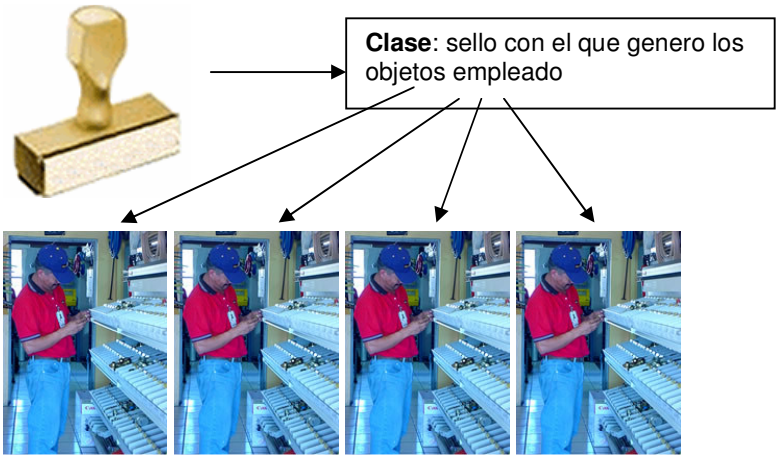
Primero aparecen los objetos, que es la abstracción que pide el paradigma. Cuando aparecen muchos objetos similares, abstraigo la definición del estado interno y su comportamiento y los modelo en una unidad que llamo clase.

Ojo, no confundirse. En la clase yo defino el estado interno: todos los empleados van a tener un atributo sueldo. Pero cada empleado tiene un valor diferente para ese atributo (cada empleado gana distinto pero todos tienen la cualidad de ganar un sueldo).

Que haya clases no es primordial para el paradigma, tenemos varios intentos por formalizar lenguajes que sólo tengan objetos (puristas hubo siempre, claro) como **Self**, que es un Smalltalk sin clases.

Javascript, por otra parte, es un lenguaje que en lugar de utilizar clases trabaja con *prototipos* de objetos[2].

No obstante, la mayoría de lenguajes orientados a objetos actuales (C++, Python, Ruby, Java, Smalltalk, .NET) utilizan la noción de clase como "template" para generar objetos y es el enfoque con el que nos vamos a quedar en la materia.

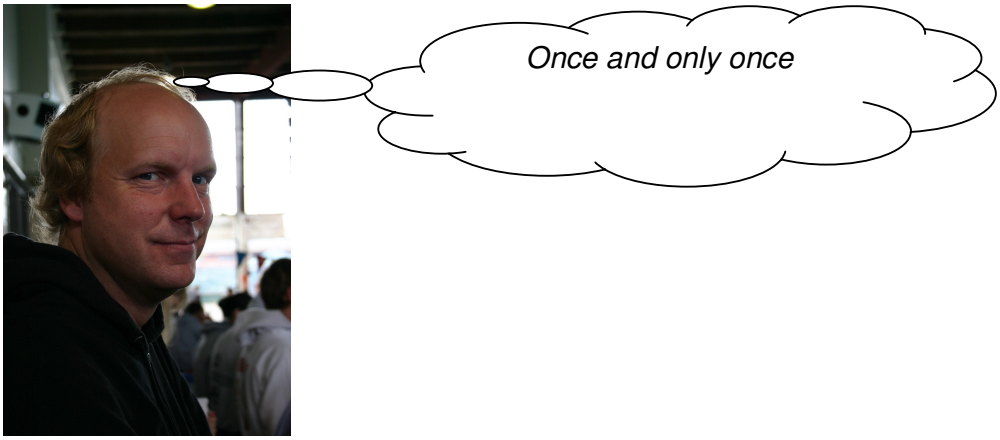


Entonces la clase Empleado tiene como definición:

- **Estado interno:** un atributo llamado sueldo que contiene el sueldo mensual de un empleado
- **Comportamiento:** "cuando le pregunto a un empleado cuánto ganás me responde cuál es su paga mensual"

Bien, nótese que en el comportamiento yo no dije "cuando le pregunto a un empleado cuánto ganás me responde con el valor que almacena la variable sueldo mensual". Eso es decir cómo lo hace y en la explicación yo no quiero detalles de implementación. Acá no hay motor de inferencia y sí tengo que decir cómo hacer las cosas, pero mejor que las haga en un solo lugar:

[2] El Object Browser implementa la idea de un lenguaje basado en instancias en lugar de clases, también conocida como programación orientada a prototipos



Es un consejo que les dejo de un amigo mío: Kent Beck. Si el código se repite (aun en la documentación) es algo que tenemos que arreglar. Decir las cosas una sola vez.

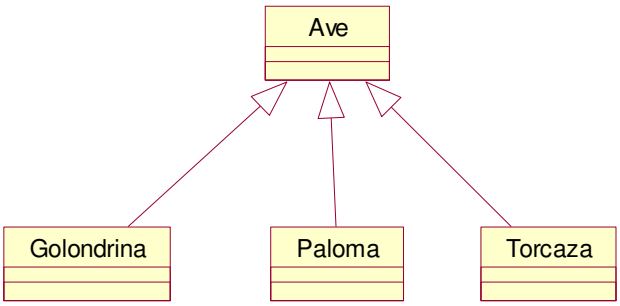
¿Y para qué me sirve tener una clase? Para crear nuevos objetos:
mike := Empleado new.
Golondrina new (y lo inspeccionamos)

Un objeto es instancia de **una y sólo una** clase y es buena práctica que durante todo el ciclo de vida pertenezca siempre a esa misma clase (decimos que conserva su identidad).

HERENCIA

Tenemos conceptos conocidos: golondrina, colibrí, torcaza, paloma, gorrión. Muchos de estos pájaros tendrán cosas en común pero se diferencian en algo…
Una vez que reconocí varios objetos, puedo abstraer una clase. Pierdo información, porque lo que obtengo es más general (una golondrina genérica, en lugar de esta o aquella golondrina, que era de color azul y yo llamaba pepita).

Ahora vamos a trabajar con otro tipo de abstracción: tengo varias clases (conceptos) y llego a una jerarquía de clases de la más general a las más particulares:



Otra forma de ver la herencia: subclasificamos un concepto conocido, lo refinamos. Si mi hija no conoce lo que es una tonina, yo le puedo explicar: “Y… es como un delfín pero negro” (una especie de … pero que …; mostrando tanto en lo que se parece como en lo que se diferencia).

Delfín

Tonina

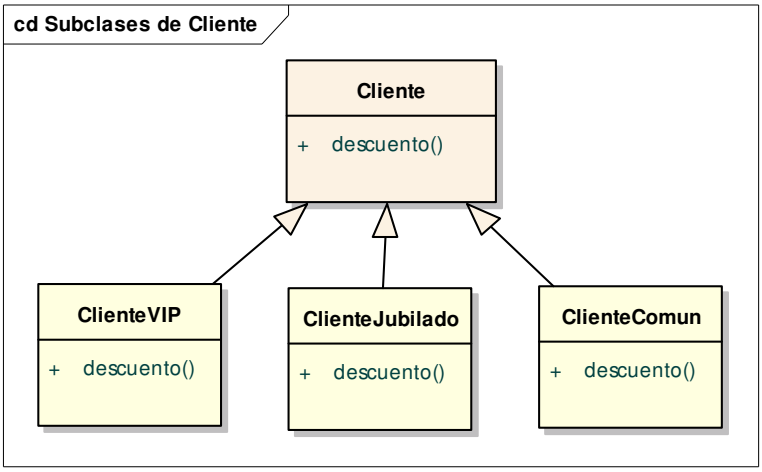
Primer tipo de abstracción: Flipper, Clementina → Delfín (instancia → clase).
Segundo tipo de abstracción: Delfín, Tonina → Delfínido, etc. (clase → superclase).
Golondrina, Colibrí, Torcaza y Paloma heredan de Ave.

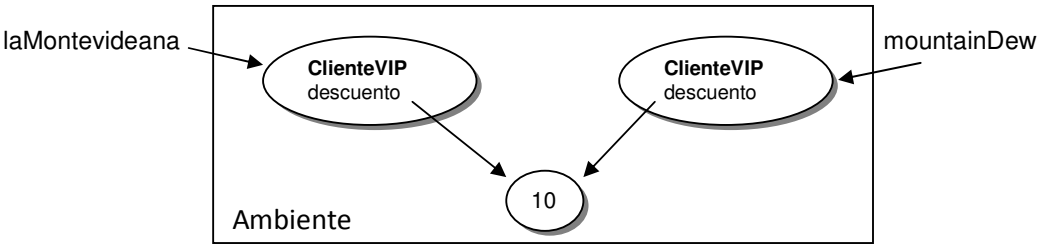
CRITERIOS PARA SUBCLASIFICAR

Ejemplos para que piensen:

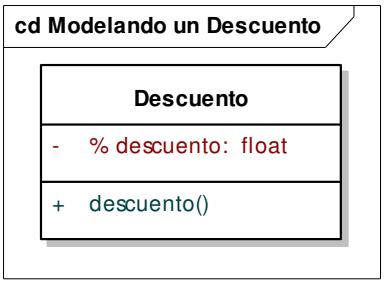
"Tengo 3 tipos de clientes. Los VIP tienen un 10% de descuento, los jubilados un 20% de descuento y los comunes tienen un 5% de descuento".

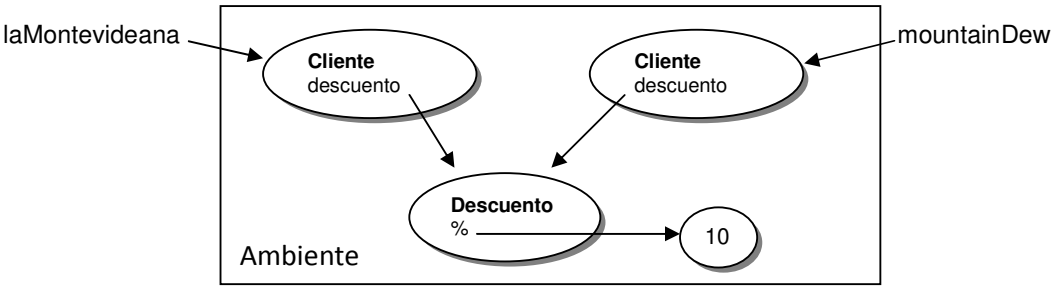
Suena tentador subclasificar Cliente en: ClienteVIP, ClienteJubilado y ClienteComún.





Si la única diferencia está en el % de descuento bien podríamos modelar el descuento con 3 instancias diferentes: un descuento para los VIP, otro para los jubilados y otro para los comunes:



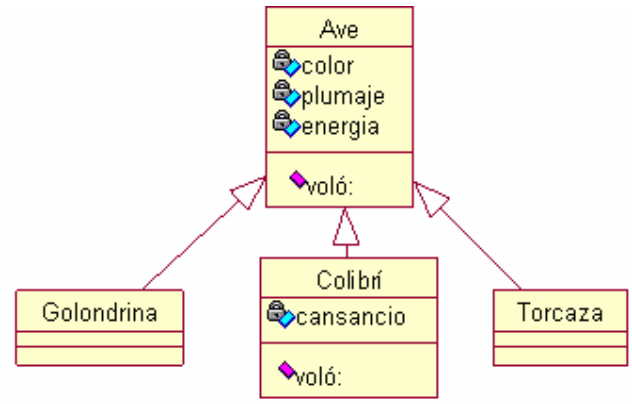


Otro ejemplo: "Modelar las piezas de ajedrez."
¿tengo una clase Caballo, otra Alfil, otra Rey o modelo todo con 32 instancias de Pieza?
Ojo, no siempre se justifica subclasificar y es una de las preguntas importantes cuando trabajo con objetos: ¿cuándo usar instancias y cuándo usar clases? Uso clases cuando

- hay comportamiento diferente
- cuando estoy encontrando un nombre útil para el dominio. Ej: si estoy modelando un juego de naipes quizás quiera darle una entidad de clase a los palos de la baraja: Oro, Copa, Espada y Basto, porque es importante para el que los va a usar (que puede ser tanto un usuario final como un desarrollador).

HERENCIA DE ATRIBUTOS Y DE MÉTODOS

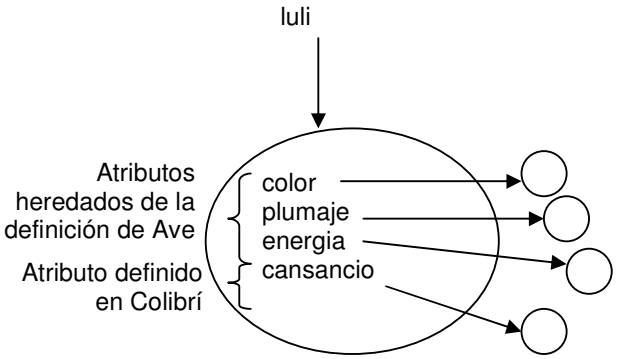
¿Qué heredo de la superclase hacia las subclases?



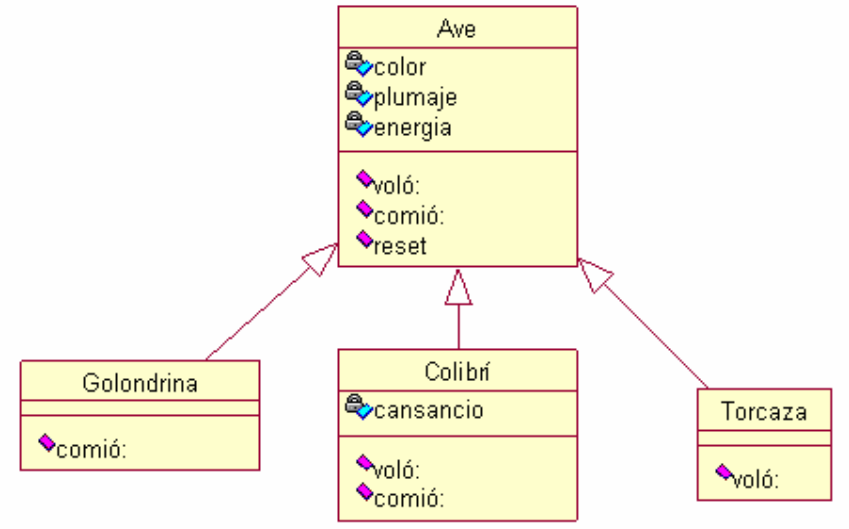
1) Ave define que tiene los siguientes atributos: color, plumaje y energía. La golondrina, el colibrí y la torcaza ya tienen incorporada esa definición: una golondrina tiene por lo

tanto su propio color, su propio plumaje y su energía (el valor siempre sigue estando en el objeto).

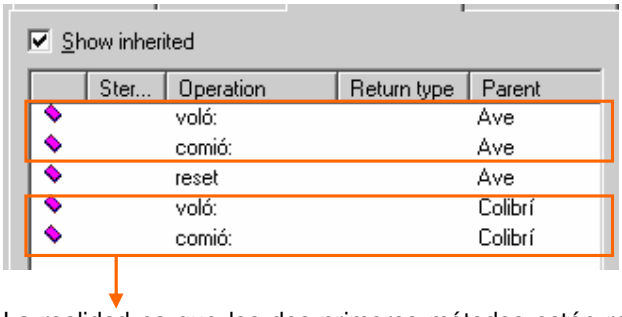
El colibrí, además de esos 3 atributos, le agrega el cansancio. Por lo tanto una instancia de colibrí tendrá los siguientes atributos:



2) Respecto a los métodos heredados, el comportamiento puede:
   - *Especializarse en la subclase*. El comportamiento default es redefinido en la subclase: un colibrí come de manera diferente a un ave, entonces no le sirve la definición de ese método: codifico otro método que tiene el mismo nombre. Mmm… ¿eso no es polimorfismo? Y… sí, y está bueno que superclase y subclase sean polimórficas.
   - *Heredarse*, en cuyo caso no es necesario hacer nada. El colibrí se resetea de la misma forma que todas las aves, entonces quien codifica Colibrí no tiene que preocuparse por escribir un método reset, sino que se codifica en Ave y el colibrí heredará ese comportamiento de la superclase Ave.



De hecho si vemos los mensajes que entiende Colibrí:



La realidad es que los dos primeros métodos están redefinidos en la subclase, por lo que la definición del método se *pisa* (vamos a ver más sobre esto pronto).

CLASE ABSTRACTA

Al generar superclases sigo perdiendo información, gano en generalidad. Un Ave quizás no tenga sentido instanciarlo. Si en mi aplicación no voy a instanciar aves (porque representan un concepto demasiado general) entonces la clase es **abstracta**.

Diferencia entre Smalltalk y Java:

1) En Smalltalk no tiene sentido crear un ave, aunque podría. La clase es abstracta cuando no tengo intención de crear instancias de esa clase (porque no tiene sentido).
2) En Java no puedo crear un ave aunque quisiera. La clase es abstracta y el compilador me impide generar instancias de esa clase. Son dos filosofías distintas.

```
public abstract class Ave {  ← forma parte de la definición misma de clase
   …
```

ENVIANDO MENSAJES A UNA CLASE

(En Smalltalk) la clase es un objeto. ¿Qué mensajes le puedo mandar a una clase? Abrimos el Class Browser, Class → Find… Class (la clase Class)

- **new** crea una nueva instancia de esa clase
- **allInstances** nos devuelve el conjunto de instancias de esa clase
- **fileOut:** bajo la definición de la clase a un archivo

Y para los curiosos, pueden ver estos métodos:

- **addInstVarName:** me agrega una nueva variable con ese nombre
- **addSelector:withMethod:** me agrega un nuevo método a la clase
- **allSelectors** me devuelve todos los mensajes que entiende esa clase
- **allSuperclasses** me devuelve el conjunto de las superclases de esa clase.

Con esto pueden agregar *comportamiento, variables y clases dinámicamente*.