CLASE. TIPO.

Clase no es igual a tipo. ¿De qué tipo es un objeto? Puede ser de muchos.
Un tipo es un conjunto de mensajes que entiende un objeto.

- Los tipos me sirven para restringir los objetos que pueden pasar por ciertos lugares. Esa restricción en Smalltalk se hace a mano.

Ejemplo: una Golondrina es

- del tipo Ave (el tipo puede ser su superclase),
- pero también puedo verlo como un objeto cualquiera (Object) –por ejemplo, si quiero mostrarlo por pantalla con un printOn:-.

O sea, hasta acá una clase puede ser un tipo (depende cómo quiera restringirlo). Pero no es la única posibilidad:

- también puedo ver a la golondrina como un animal que migra (aunque no exista la clase AnimalMigratorio). De hecho podría definir un método **migrarA:** otroLugar. Y pepita podría implementar el método migrarA:, como los osos, los elefantes marinos, los manatíes, las ballenas y los peces. ¿Qué tienen en común todos estos animales? Todos son “animales migratorios”.

¿Tiene sentido que ballena herede de Golondrina? No, claramente sus implementaciones son diferentes y no tiene mucho sentido crear la clase AnimalMigratorio para agrupar a Golondrina y a Ballena. Lo que sí comparten Ballena y Golondrina es la interfaz:




A todos les puedo enviar el mensaje **migrarA:** otroLugar

Entonces tenemos a elefantes marinos, ballenas, osos y golondrinas como objetos polimórficos, sin necesidad de que haya herencia entre ellos. Son polimórficos en el contexto de verlos como migratorios.

Y nuevamente preguntamos: ¿para quién tiene sentido que sean polimórficos? ¿para el que codifica a la golondrina o para el que usa los animales indistintamente?

SELF

¿Cómo me mando un mensaje a mí mismo?

```
vestite
   self poneteCamisa.
   self ponetePantalón.
```

objeto mensaje parámetro

self soy yo mismo: self apunta al objeto receptor del mensaje.

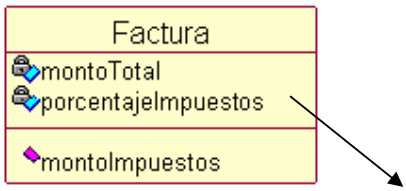
Me estoy enviando a mí mismo los mensajes poneteCamisa y ponetePantalón, para lo cual tienen que estar definido los dos métodos.

ACCESO DIRECTO VS. ACCESO INDIRECTO

Hay dos opciones cuando queremos usar un atributo (o variable) dentro de un método.

Acceso directo

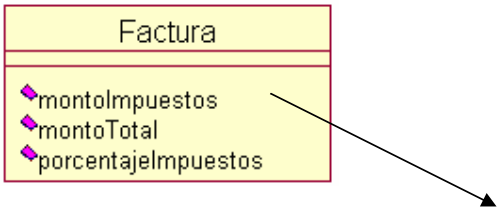
montoImpuestos (#Factura[1])
  ^montoTotal * porcentajeImpuestos



Acá estamos accediendo a las variables de instancia dentro de la clase Factura

Acceso indirecto

montoImpuestos (#Factura)
  ^self montoTotal * self porcentajeImpuestos



Acá lo que estamos haciendo es llamar a dos métodos: montoTotal y porcentajeImpuestos:

montoTotal
  ^montoTotal

porcentajeImpuestos
  ^porcentajeImpuestos

Métodos *getters*

¿Qué ventaja tiene si escribo más? Supongamos que no tengo el monto total en la factura, sino que lo tengo que obtener en base a los productos que me llevo. Entonces el cálculo es más complejo, pero yo lo tendría encapsulado en el método montoTotal (sólo cambio la implementación del método una vez).

Fíjense que en el Diseño cuando quise poner énfasis en los métodos desaparecieron las variables. *Tip del Diseñador:* no tengo que mostrar todo el diseño junto, sólo lo que me interesa comunicar en un diagrama.

Lo importante: no confundir, self es para poder enviarme un mensaje a mí mismo, sobre todo cuando un método delega parte de la responsabilidad a otro que puede estar codificado:

1) en la misma clase,
2) en una superclase,
3) en alguna subclase.

[1] **Convención:** #Factura o >>Factura implica que es el método es de la clase Factura

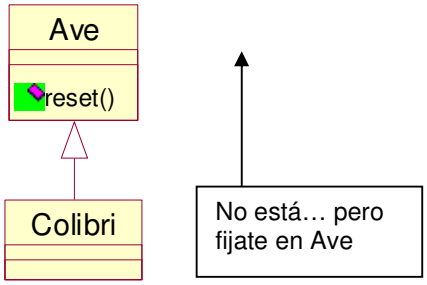
METHOD LOOKUP

Cuando envío el mensaje comióAlpiste: 20 a un colibrí, ¿cómo se resuelve el código que hay que ejecutar?

pepe **comióAlpiste:** 20 → como pepe es un colibrí, busco en la clase Colibrí la definición del método comióAlpiste: y lo invoco pasando el objeto 20 como argumento.

Ahora, ¿qué sucede si envío el mensaje

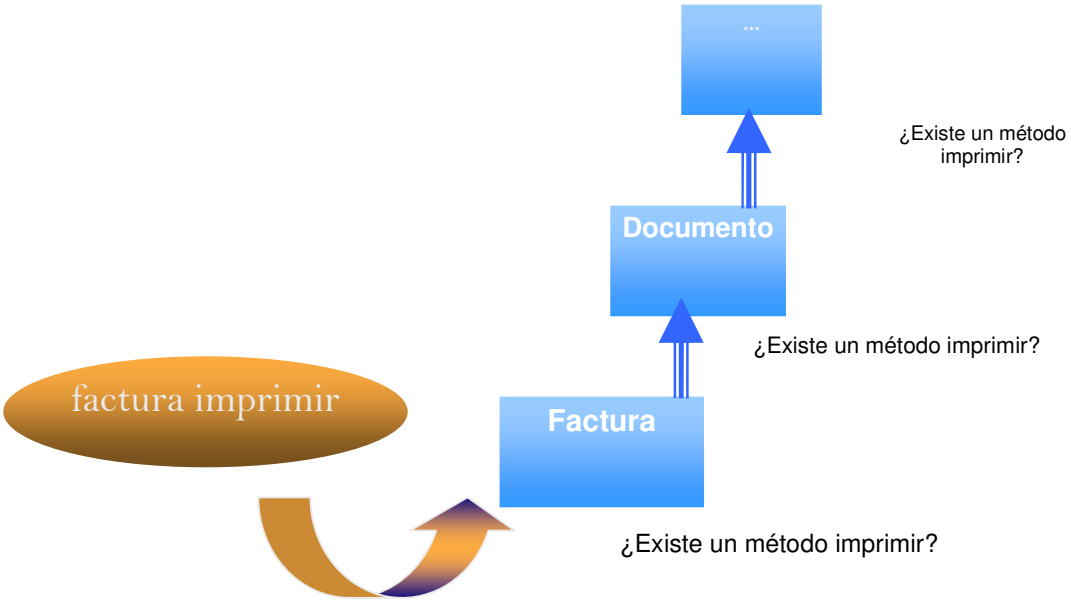
pepe reset? → como pepe es un colibrí, busco en la clase Colibrí la definición del método reset. Pero no está. ¿Entonces?



Bueno, aquí es donde la superclase entra a jugar su papel de partenaire. Cuando en Colibrí no encontré un método reset, voy a buscar la definición a la clase Ave. Pero que no nos confunda dónde voy a buscar la definición, pepe sigue siendo un colibrí, por lo tanto self sigue apuntando **siempre** al objeto receptor, que es pepe (o el colibrí en cuestión).

¿Y si no existiera en Ave? Lo iría a buscar a la superclase de Ave y así sucesivamente. Si al final de la cadena no puede encontrar una definición del método devuelve un error, indicando que el objeto no puede entender lo que yo le pedí que hiciera ("does Not Understand", que quizás les suene familiar).

La búsqueda de la definición del método se conoce como "*Method lookup*" y la estrategia que se elige en Smalltalk es comenzar por el objeto receptor y luego ir hacia arriba en la jerarquía de clases hasta que se encuentra el método o hasta que se llega a la última de las clases de la jerarquía.



SUPER

Super también apunta al objeto receptor, pero la búsqueda del método comienza a partir de la superclase del objeto receptor.

Ejemplo: hay un Colibrí africano del cual nos interesa registra la cantidad de veces que vuela.

¿Cómo queda el método **voló:**?

```
#ColibriAfricano
voló: minutos
        energíaCalculada := energíaCalculada – 30.
        cantidadDeVecesQueVolo := cantidadDeVecesQueVolo + 1.
```

El tema es que estoy diciendo dos veces la forma en la que un Colibrí vuela:

- en Colibri
- en ColibriAfricano

¿Qué código se repite?

```
voló: minutos
        energíaCalculada := energíaCalculada – 30.
        cantidadDeVecesQueVolo := cantidadDeVecesQueVolo + 1.
```

Ok, entonces podemos aprovechar el código que ya definimos en Colibri. Pero en lugar de enviar el mensaje self **voló:** minutos que nos hace entrar en loop infinito, usamos super para decirle a Smalltalk que empiece buscando el método voló: a partir de Colibri:

```
#ColibriAfricano
voló: minutos
        super voló: minutos.
        cantidadDeVecesQueVolo := cantidadDeVecesQueVolo + 1.
```

Recalcamos una vez más: super no apunta a la superclase, super sigue referenciando al objeto receptor (super == self), la diferencia es solamente en el method lookup.

Lo mismo hacemos en el método reset, donde aprovechamos el reset definido en Colibri y agregamos la inicialización propia de un colibrí africano:

```
#ColibriAfricano
reset
    super reset.  ← evitamos así duplicar la inicialización de Colibrí en la subclase
    cantidadDeVecesQueVolo := 0.
```