1: Repaso teórico de conceptos fundamentales

1.1. Conceptos de lenguajes

Vamos a comenzar esta sección para preguntarnos qué es un programa. Se nos puede inmediatamente venir una definición simple a la cabeza, que vemos reproducida una y otra vez como incendio forestal:

"Un programa es una secuencia de instrucciones para solucionar un problema computacional"

La parte de que debe solucionar un problema computacional, muchas veces es omitida en algunas definiciones incluso más simplistas. La definición no es del todo incorrecta, pero luego de comprender los diferentes paradigmas de programación existentes, cabe preguntarse si esa definición es apropiada para los lenguajes funcionales. En este paradigma el concepto de instrucción es inexistente, y por tanto la definición anterior no es válida. Una definición funcional sería:

"Un programa es un conjunto de funciones que pueden reducirse para expresar la solución a un problema computacional"

Nuevamente, es una buena definición, pero solo para los lenguajes funcionales. El paradigma estructurado no puede hacer uso de dicha definición. Tampoco lo puede hacer el paradigma de objetos. Nosotros entonces optamos por una definición más pragmática, que resulta independiente del paradigma y del lenguaje usado:

Definición 1.1
Un programa es un texto que describe la solución a un problema computacional

Esta definición es útil ya que resalta la idea de que la solución debe ser expresada como texto (en algún lenguaje de programación cualquiera), pero no habla de qué forma debe tener dicho texto, o cuál debe ser la forma de interpretarlo. Así, esta definición sirve para cualquier paradigma.

Pero cabe preguntarse ¿Por qué cuando cambia el paradigma cambiaría la definición de programa? El tema es que el paradigma precisamente define nuestra interpretación de un "programa". Para poder definir un paradigma hay que poder responder dos preguntas clave:

- ¿Cómo se representan los datos en el paradigma?
- ¿Cómo avanza el cómputo del programa?

Pensemos en el paradigma estructurado en primera instancia. Las expresiones son la forma de representar datos, más aún, lo que llamamos valores. Por otro lado, el cómputo avanza mediante una transición de estados, producto de la ejecución de los comandos definidos.

En el paradigma funcional, por otro lado, los datos no son más que funciones (incluso aquellos que muchas veces no pensamos o concebimos como funciones, como los números o los booleanos). El cómputo avanza mediante el proceso de beta-reducción de las funciones definidas.

Podemos entonces elaborar la siguiente tabla:

Tabla 1.1: Conceptualización de los paradigmas estructurado y funcional

	Paradigma estructurado	Paradigma funcional
Representación de los datos	Valores	Funciones
Avance del cómputo	Transición de estados mediante la ejecución de comandos	Beta-reducción de las funciones definidas

Pasemos ahora a la programación orientada a objetos. ¿Qué define a este paradigma? Para esto debemos contestar las mismas dos preguntas, pensando en un lenguaje orientado a objetos puro.

En el paradigma, son los objetos los que cumplen el rol del dato. Todo, absolutamente todo, debería ser un objeto (en el paradigma no hay concepción de tipos primitivos como tienen algunos lenguajes). Por otro lado, el programa avanza en su cómputo a través de la comunicación de esos objetos entre sí, lo cual ocurre mediante el envío de mensajes. De esta forma, el paradigma puede resumirse en dos palabras, “objeto” y “mensaje”.

Tabla 1.2: Conceptualización del paradigma orientado a objetos

	Paradigma orientado a objetos
Representación de los datos	Objeto
Avance del cómputo	Envío de mensajes

El paradigma viene conceptualizado únicamente en el concepto de “objeto-mensaje”, del cual, se desprenden muchos otros conceptos que son inherentes al paradigma, como veremos más adelante. Ahora bien, muchos otros conceptos, que a veces se explican como una parte fundamental del mismo, como pueden ser las clases o la herencia, son en realidad elementos completamente accesorios, que brindan comodidad al programador para conseguir ciertas tareas fundamentales, pero que no son en absoluto necesarias para plantear un lenguaje orientado a objetos puro.

Como resumen de esta sección, es importante comprender que cuando alguien pregunte ¿De qué trata la programación orientada a objetos? nuestra respuesta debe ser automática, “objeto-mensaje”.

1.2. Conceptos previos de la programación orientada a objetos

Ahora vamos a repasar algunos conceptos que ya conocemos de la programación orientada a objetos, y que hemos adquirido principalmente a nivel práctico, aunque tengamos ideas teóricas también sobre estos. Es importante recordar y comprender mejor de qué estamos hablando para poder comprender los conceptos avanzados que trabajaremos más adelante.

1.2.1. Estado y comportamiento

Estamos acostumbrados al concepto de estado del paradigma estructurado. En dicho paradigma, uno tiende a pensar en el estado como un elemento global. El cómputo del programa avanza mediante transformaciones de estado, que ocurren al ejecutar un comando. El estado está representado entonces por distintos elementos, como pueden ser las variables globales, locales, parámetros, etc. En una visión de bajo nivel, podemos pensar la equivalencia de estado con memoria.

En la programación orientada a objetos, el estado también es una parte importante del paradigma. **Cada objeto tiene su propio estado**, y es responsable de mantenerlo. **La suma de los estados de todos los objetos del sistema, compone el estado del sistema**.

En ese sentido, e**l avance del cómputo implica transformaciones de estado**, similar al paradigma estructurado. La salvedad es que **el estado ahora está distribuido en múltiples entidades independientes**, los objetos.

La forma de manipular el estado de un objeto, es mediante el comportamiento que dicho objeto expone. Lo que para el paradigma estructurado eran procedimientos y funciones, para el paradigma orientado a objetos es comportamiento que un objeto tiene asociado.

Además **el comportamiento puede incluir funcionalidad que no transforma estado, sino información**, donde la información puede estar proporcionada como parámetros, o ser tomada del estado del programa.

Entonces surge una segunda definición importante del paradigma de objetos:

Definición 1.2
Un objeto es una entidad independiente del sistema, que tiene un estado y un comportamiento asociados.

1.2.2. Referencias

Para poder hacer uso de un objeto, hay que conocerlo, y la forma de conocerlo, es mediante una **referencia**. Una **referencia consiste en una forma de identificar a un objeto**, y a bajo nivel, resulta en un "puntero" a dicho objeto (algo que veremos en más detalle en futuras secciones).

Una referencia puede consistir en un **nombre global**, es decir, **un nombre que puede usarse en cualquier parte del sistema para identificar a un objeto particular**. Un ejemplo de esto son los objetos "global" y "window" en JavaScript, los objetos "true" y "false" en Smalltalk, o el objeto None en Python.

También las **variables locales** y los **parámetros**, que "almacenan" datos, son referencias a objetos (porque todo dato es un objeto).

Como es bien sabido, un mismo objeto puede ser referenciado de muchas formas, es decir, **pueden existir múltiples referencias a un mismo objeto**. Sin embargo, no tiene sentido la existencia de un objeto sin que existan referencias al mismo, ya que sí no se tiene una forma de hablar de él. Podemos decir entonces que, **un objeto que no puede ser referenciado es inútil**.

Definición 1.3
Una referencia consiste en un identificador que apunta a un objeto, permitiendo mencionar al mismo. Pueden ser variables locales, parámetros, variables globales, atributos o auto-referencias, y pueden estar inicializadas o sin inicializar (referencia nula)

Algunos autores hablan incluso de "**referencias anónimas**", por ejemplo, el resultado de cómputo puede dar por resultado un objeto, al cual puedo manipular, pero sin necesidad de almacenarlo en una variable local.

Un caso interesante y particular es la **referencia nula**, que muchos lenguajes presentan como construcción (Java, C++, C#, entre otros). En estos lenguajes uno puede declarar una variable sin inicializar, y por tanto, sin "valor". Si una variable no hace referencia a un objeto, entonces ¿Qué referencia? La respuesta de estos lenguajes ha sido la de introducir el concepto de "apuntar a

nada", mediante una referencia nula. El "valor" de dicha referencia suele ser tratado como un caso especial de "no objeto", y se lo suele llamar **null**. Notar que "null" en Java es muy distinto a "nil" en Smalltalk o "None" en Python, que son, en realidad, nombres globales de objetos.

Una referencia muy importante que existe en todo lenguaje es la **auto-referencia** (llamada "**self**", "**this**" o "**me**" según el lenguaje). El poder auto-referenciarse es una característica necesaria de todo lenguaje orientado a objetos. La auto-referencia permite a un objeto enviarse mensajes a sí mismo, algo que resulta necesario en el paradigma para poder realizar división en subtareas, reutilización, recursión, entre otras varias características.

Ejemplo 1.1. Auto-referencia explícita en Smalltalk

```
Persona>>nombreYApellido
   "En Smalltalk siempre las auto-referencias deben ser explícitas"
   ^ self nombre, ' ', self apellido
```

En algunos lenguajes, el uso de una auto-referencia es explícito (hay que escribir "self" o "this" necesariamente cada vez que se desea enviar un mensaje al objeto), mientras que en otros es implícito (el uso de una invocación "suelta" en el código, refiere más bien a un mensaje que se envía al objeto, incluso sí el objeto no se menciona).

Ejemplo 1.2. Auto-referencia implícita y explícita en Java

```
class Persona {
    ...
    public String nombreYApellido() {
        // En Java pueden ser implícitas, como en el caso de nombre,
        // o pueden ser explícitas como en el caso de apellido.
        return nombre() + " " + this.apellido()}
    ...
}
```

1.2.3. Interfaces, encapsulamiento, visibilidad y abstracción

El paradigma orientado a objetos tiene por objetivo principal permitir una mejor escalabilidad y reutilización de código, logrado principalmente a través de la abstracción de los datos en objetos encapsulados que solo permiten a terceros ver e interactuar con su interfaz. Pero, ¿Qué quiere decir todo esto?

El **encapsulamiento refiere al hecho de que cada objeto tiene todos sus datos** (estado) **agrupados con todas las operaciones que trabajan con esos datos** (comportamiento) **en un mismo lugar** (por ejemplo en el lugar en donde se define el objeto). Es decir, el objeto se presenta como una "cápsula" con todos sus elementos relevantes dentro de esta.

El concepto de **visibilidad refiere a quién puede ver esos datos y operaciones, y de qué forma**. En la conceptualización original existen dos tipos de visibilidad, **privada** (solo dentro de la definición del objeto se puede hacer uso de ese elemento) y **pública** (se puede hacer uso tanto de dentro de la definición del objeto como desde otros lugares del programa, por ejemplo, desde otros objetos). Otros lenguajes agregan otros conceptos de separación adicionales de los elementos (por ejemplo, agrupando objetos en un paquete) y por tanto proveen otros niveles de visibilidad. Más aún, algunos lenguajes permiten determinar la visibilidad mediante alguna palabra clave (ej. Java o C#), mientras que en otros, la visibilidad tiene que ver con el tipo de elemento, y no puede modificarse (ej. Smalltalk o Python).

Por último, la **abstracción refiere a trabajar al objeto como una "caja negra"**. Es decir, el usuario de un objeto, no tiene por qué saber en qué forma el objeto está definido, sino que le basta conocer conceptualmente las operaciones que sabe responder. La abstracción vista desde esta perspectiva, permite **desacoplar** objetos, un concepto fundamental sobre el cual volveremos más adelante.

Sin embargo, cuando hablamos de encapsulamiento, la mayoría de las veces queremos hablar de los tres conceptos juntos, encapsulamiento, visibilidad y abstracción. En este escrito seguimos la idea de muchos autores de no separar estos tres conceptos, pues son todos conceptos fuertemente relacionados y que tienen todos el mismo objetivo final. Así, **cuando digamos encapsulamiento, vamos a referirnos al hecho de que todos los datos y comportamiento están asociados, y dispuestos de forma tal que, solo el comportamiento público está expuesto para terceros, pero los datos y el comportamiento privado no están expuestos, y son propios del objeto que los define, para así lograr abstracción.**

Definición 1.4
El encapsulamiento refiere a que un objeto tiene su estado y comportamiento asociados, y solo expone a terceros el comportamiento relevante para estos, para así proveer abstracción.

El encapsulamiento es lo que redunda en que lo relevante de un objeto sea su **interfaz**. La **interfaz** es un **contrato** sobre lo que el **objeto sabe responder**, es decir, sobre los mensajes que entiende, su **comportamiento**. Cuando un objeto es referenciado, los mensajes que sabe responder son aquellos que responden a alguna de sus interfaces. **El acceso o la modificación de los datos que posee un objeto, sólo puede ser realizado mediante la interfaz que el objeto provee.**

Y es que un objeto tiene múltiples interfaces. Por ejemplo, si un objeto sabe responder a "volar", "comer" y "dormir", decimos que **su interfaz es el conjunto de todos estos mensajes y todos los subconjuntos de dichos mensajes**. Es decir, hay una interfaz "{volar, comer, dormir}", otra "{volar, comer}", otra "{volar, dormir}", otra "{comer, dormir}" y luego "{volar}", "{comer}" y "{dormir}". La interfaz con conjunto vacío no es realmente útil a la definición, pero también puede ser considerada en algunos casos.

De esta forma, distintos objetos pueden compartir una misma interfaz, lo que no significa que necesariamente su comportamiento es idéntico o que sean el mismo objeto, sino que saben responder al mismo mensaje. Cuando esto ocurre, decimos que son **polimórficos** con respecto a dicha interfaz. El concepto de lo polimorfismo surge de forma natural luego de incorporar el concepto de interfaz, y es uno de los conceptos fundamentales de la programación orientada a objetos, tan así, que sí el lenguaje incorpora alguna estructura que represente "código ejecutable", y polimorfismo, entonces las estructuras de control de flujo tradicionales, como la repetición y la alternativa no son necesarias.

Definición 1.5
Las interfaces de un objeto consisten en todos los subconjuntos del conjunto de mensajes que el objeto sabe responder.

Notar que el concepto de interfaz está relacionado, pero no es idéntico a, el concepto que se define bajo la palabra clave "interface" en lenguajes como Java o C#. Más adelante veremos detalles de este concepto y su vinculación con el concepto teórico de interfaz.

1.2.3. Métodos y atributos

Una de las formas de **representar estado** en los lenguajes orientados a objetos, aunque no la única, es mediante el concepto de **atributo**. Un atributo es básicamente **una referencia que un objeto tiene, y que conoce mediante un nombre cuyo alcance está limitado al objeto en sí**.

Salvo objetos especiales que suelen venir incluidos en el sistema (por ejemplo, números, booleanos, etc.), todo objeto que el programador defina tiene su estado representado mediante la suma de todos los estados de los objetos referenciados por los atributos que el objeto en cuestión posee. Es decir, el estado de un objeto, es la sumatoria de los estados de los objetos que conoce en sus atributos.

Por otro lado, el concepto de **método** es una de las formas de **representar comportamiento**. **Un método es una porción de código que se activa ante un determinado mensaje que el objeto reciba**, de forma similar a una función o un procedimiento. De hecho, los métodos pueden verse básicamente como funciones o procedimientos (dependiendo de sí describen un valor o modifican el estado del objeto), y pueden definir parámetros, que consiste en datos que deben ser dados al momento de enviar el mensaje asociado.

Definición 1.6
Un atributo es una referencia que un objeto posee, y que determina parte del estado del objeto.

Definición 1.7
Un método es una porción de código que determina el comportamiento de un objeto al recibir un mensaje determinado.

Así, **conocer los métodos que define un objeto permiten determinar los mensajes que este entiende, y por tanto, las interfaces que el objeto implementa**.

En la mayoría de los lenguajes, el **mensaje que se asocia a un método es básicamente el mensaje que tiene un nombre idéntico al del método que se está definiendo**. Sin embargo, no siempre es así. Python por ejemplo, define un concepto de **métodos mágicos**, donde por ejemplo, el método ”__eq__” permite definir el comportamiento del objeto ante el mensaje “=”.

La **signatura** de un método es la forma en la que nos referimos a un método particular de forma inequívoca. En algunos lenguajes consiste en el nombre, pero en algunos lenguajes además consiste en la cantidad, orden y tipos de los parámetros. Distinguir signaturas según parámetros da lugar a la **sobrecarga de métodos** o ***method overloading***, es decir dos métodos con mismo nombre, pero distinta cantidad, tipo u orden de parámetros, tienen distinta signatura, y por tanto, son métodos distintos.

En Java por ejemplo, los métodos definidos como “sumar()” y “sumar(int x, int y)” **tienen distinta signatura, y por tanto son métodos distintos, que se activan ante mensajes distintos**, que se identifican no gracias al nombre del mensaje, sino mediante una combinación del nombre y los argumentos dados. El lenguaje determina el método a invocar gracias a los argumentos dados.

Para que exista el method overload el lenguaje tiene que distinguir, al menos, entre cantidad de parámetros, pero en general, también suele proveerse alguna forma de diferenciar los tipos de dichos parámetros.

1.2.4. Clases y herencia

Otro de los conceptos que aparece de forma casi constante en cualquier curso, taller o escrito sobre la programación orientada a objetos, es el concepto de clases. Sin embargo, las clases no son una parte fundamental del paradigma, y se puede tener programación orientada a objetos sin clases.

Las **clases** son una estructura que permite definir la “forma” que tendrá un objeto. Puede pensarse como una **plantilla para crear objetos**. Los objetos que se crean a partir de una determinada clase se conocen como **instancias de la clase**. Las clases permiten crear a partir de una única definición muchos objetos iguales, **reutilizando código y evitando la duplicación**.

En los lenguajes que tienen clases, **la clase define**, por ejemplo, **los atributos que tendrá un objeto y su comportamiento**, mediante métodos, y luego, **las instancias son las que finalmente implementan esa estructura**.

Las clases en realidad, definen muchas otras cosas además de la plantilla que define los objetos, pero indagaremos en estos más adelante.

Definición 1.8
Una clase es una estructura que permite definir múltiples objetos a partir de ella, los cuales comparten estructura y comportamiento. Es, a modo informal, una plantilla a partir de la cual se crean objetos.

Otro concepto asociado a las clases es el de herencia. La **herencia es un mecanismo que permite extender una estructura a partir de otra**, por lo que podemos definir una clase mediante la extensión de otra clase.

Al heredar, la clase que es heredada, llamada **superclase**, **clase-base** o **clase-padre**, le "hereda" la estructura y comportamiento a la clase que hereda, llamada **subclase**, **clase-derivada** o **clase-hijo**.

Esto **permite una aún mayor reutilización del código**, pero a la vez genera problemáticas asociadas a la ubicación del código de nuestros objetos, ya que ahora, parte del comportamiento y del estado no están definidos en la misma clase de nuestro objeto, sino en distintas clases (lo cual en algunos lenguajes implica incluso muchos archivos). Esto **hace que sea difícil visualizar la estructura completa de la clase**, y por tanto entender completamente la estructura de nuestros objetos.

Definición 1.9
La herencia es el mecanismo mediante el cual una clase, llamada subclase, puede definir comportamiento y estructura de los objetos que se crean a partir de esta, mediante la reutilización de los elementos definidos en otra clase, llamada superclase.

Además, **la herencia trae asociada una dependencia fuerte entre la superclase y la subclase**, de forma tal que cuando la superclase cambia, este cambio puede afectar seriamente el código de la subclase, debiendo reescribir en muchas ocasiones grandes partes de la misma. Esto va en contra de la idea de objetos, en donde lo que se busca es que los objetos estén lo más desacoplados entre sí, para lograr un código altamente escalable.

Según la cantidad de superclases que una clase pueda tener, la herencia puede catalogarse de distintas formas:

- **Herencia simple**: Una clase puede tener solo una única superclase.
- **Herencia múltiple**: Una clase puede tener tantas superclases como se desee.

Además también tenemos una categorización según la cantidad de subclases:

- **Herencia única**: Una superclase puede tener una única subclase.
- **Herencia jerárquica**: Una superclase puede tener muchas subclases.

Y por último según la cantidad de niveles que tenga esa herencia:

- **Herencia mononivel**: Una clase que es subclase de otra no puede actuar de superclase.
- **Herencia multinivel**: Una clase que es subclase de otra puede a su vez actuar de superclase de otra clase.

Y por supuesto, estas categorizaciones son ortogonales, pudiendo tener sistemas de herencia que son combinaciones de estas. Es raro encontrar lenguajes con herencias únicas o mononivel, y además, a excepción de C++ y algunos pocos lenguajes experimentales, la mayoría de los lenguajes optan por herencia simple. Por ejemplo, Java tiene una herencia simple, jerárquica y multinivel, y este tipo de herencia suele ser la más elegida por los lenguajes de programación modernos.

Así, en los sistemas de herencia simple, jerárquicos, multinivel (que son los más extendidos) toda clase tiene una superclase, y esta, a su vez, por ser una clase, puede tener una superclase, y así siguiendo. Por supuesto, en muchos lenguajes suele haber una clase que rompe la regla y que actúa de cabeza de la jerarquía, y de la que todas las clases heredan (generalmente llamada "object"). Así, podemos hablar de la **cadena de clases**, o **cadena de herencia** de una clase, como la cadena que lleva de esa clase a su superclase, y de esta a la siguiente, y así siguiendo hasta llegar a "object".

Tanto las clases como la herencia se enfocan entonces en un aspecto clave de la programación, la **reutilización**. Las clases permiten entonces definir estructuras y comportamiento reutilizable a nivel objetos, mediante la definición de plantillas, mientras que la herencia permite definir elementos reutilizables a nivel estructural, es decir, porciones de plantillas, que otras plantillas pueden reutilizar.

Un concepto que surge cuando se tiene herencia es el de **sobreescritura de métodos**, o ***method override***.

Definición 1.10
La sobreescritura de métodos ocurre cuando una clase define un método que tiene la misma signatura que un método de la superclase o alguna clase en la cadena de herencia.

Cuando ocurre una sobreescritura de métodos hay un cambio en el comportamiento de los objetos de la subclase. Al recibir el método sobreescrito, el comportamiento depende mucho del lenguaje, siendo generalmente el método de la subclase el cual es ejecutado, en lugar de el método de la superclase. Esto, en principio, puede llegar a ser poco deseable, ya que en este caso es necesario no solo ver el comportamiento declarado en una clase, sino el declarado en todas las subclases de la cadena para poder entender el comportamiento final que tendrá el objeto en cuestión.

Junto con el concepto de sobrescritura de métodos aparece la idea de poder reutilizar el código declarado en una superclase, incluso sí el comportamiento fue sobreescrito. La forma en la que se implementa este mecanismo depende ampliamente del lenguaje, siendo en ocasiones una palabra clave adicional, en otras una función, en otras una referencia, o incluso otros mecanismos. Este concepto en general es denominado **super** o **base**. La gracia del concepto tiene que ver con alterar el sistema de resolución de mensajes normal, algo que veremos en profundidad en siguientes secciones.

1.3. Lo inherente vs. lo accesorio del paradigma

En base a lo analizado arriba, podemos observar que no todos los elementos que en general tenemos en mente a la hora de hablar de programación orientada a objetos no son realmente propios del paradigma, mientras que otros no pueden ser eliminados del mismo.

Por ejemplo, la idea de **objeto** como la unidad mínima de transferencia de información es la base fundamental del paradigma, así como la idea de **mensaje**, como elemento fundamental para avanzar el cómputo del programa. Además de esto, es fundamental la idea de que el objeto es el elemento que posee **estado** y **comportamiento**, y por tanto es quien recibe el mensaje. Esto hace que sea imposible separar la idea de **polimorfismo** del paradigma, ya que distintos objetos

pueden reaccionar al mismo mensaje activando distintos comportamientos. Por supuesto, sí hablamos de comportamiento en un objeto, entonces la idea de **interfaz** también es inseparable del paradigma. Por último, podemos mencionar las ideas de conocer a un objeto, mediante una **referencia**, ya que de lo contrario no podríamos interactuar con los objetos. Esto incluye también que un objeto se conozca a sí mismo, mediante una **auto-referencia**, sin la cual no podría existir separación en subtareas dentro del objeto.

Decimos entonces que estas construcciones son **inherentes al paradigma**. Vienen con el mismo y no puede existir programación orientada a objetos sin ellos. Es decir, sí un lenguaje se dice orientado a objetos, entonces tiene que tener polimorfismo, tiene que tener auto-referencias, etc.

Por otro lado, muchas construcciones que en general se asocian al paradigma son simplemente **accesorias**. Es decir, no tienen porqué existir en un lenguaje para que este se diga orientado a objetos. Un ejemplo claro son las **clases**. Una clase solo es un mecanismo del lenguaje para construir objetos que tienen el mismo comportamiento y estructura, pero no es la única forma que existe para lograr esto. El **clonado** de objetos por ejemplo, es otro mecanismo que permite una dinámica similar. Y por supuesto, sí las clases son accesorias, todo lo que viene asociado a estas también lo son, como la **herencia**, el concepto de "**super**" o "**base**", la **sobreescritura de métodos**, entre otros. De hecho, la misma idea de **método** y de **atributo** son conceptos accesorios, ya que existen otras formas de lograr estado y comportamiento, sin tener esas herramientas.

Pero, ¿Por qué queremos separar lo inherente de lo accesorio? Principalmente porque no todos los lenguajes orientados a objetos funcionan igual, y es importante entender que cosas voy a poder aplicar siempre y cuáles solo voy a poder usar en lenguajes específicos. Entender en profundidad qué conceptos aplican y cuáles no, me va a permitir explotar mejor el lenguaje, cualquiera sea. Pero por otro lado, al entender mejor los conceptos voy a poder comparar lenguajes y mecanismos que estos proveen, pudiendo seleccionar mejor un lenguaje u otro dependiendo de la tarea a realizar (en los casos en los que el lenguaje sea una elección posible).

Por supuesto, que los conceptos son un elemento fundamental para volverse un experto en cualquier lenguaje, pero también deberemos saber detalles de implementación y otras cosas puntuales. El presente no trata de ahondar en algún lenguaje particular, pero te permitirá saber dónde mirar para volverte un experto en un lenguaje, sí así lo deseas.

Bibliografía y enlaces útiles

- Stéphane Ducasse, Oscar Nierstrasz, Andrew P. Black, Damien Pollet. **Pharo by example**. Square Brackets Associates, 2010. https://books.pharo.org/updated-pharo-by-example/
- Martín Abadi, Luca Cardelli**. A theory of objects**. Springer, 1996.