**¿Por qué aprender Programación Orientada a Objetos?**

* **Vas a programar más rápido**. Tener un análisis previo de lo que estás realizando te ayudará a generar código mucho más veloz
* **Dejas de ser Programador Jr**. Podrás responder preguntas como ¿Qué es encapsulamiento?, ¿Qué es Abstracción?, ¿Qué es Herencia?, ¿Qué es Polimorfismo? en futuras entrevistas de trabajo
* **Dejar de Copiar y Pegar Código**.

La programación orientada a objetos tiene cuatro características principales:

**Encapsulamiento**. Quiere decir que oculta datos mediante código.

**Abstracción**. Es como se pueden representar los objetos en modo de código.

**Herencia**. Es donde una clase nueva se crea a partir de una clase existente.

**Polimorfismo**. Se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos.

En este curso, los pasos a seguir será.

Analisis

Plasmar

Programar

La mayoría solo aprende a hacer esto en un lenguajes de programación,a quí se tiene una variabilidad.

# ¿Qué resuelve la Programación Orientada a Objetos?

La programación Orientada a Objetos nace de los problemas creados por la programación estructurada y nos ayuda a resolver cierto problemas como:

* Código muy largo: A medida que un sistema va creciendo y se hace más robusta el código generado se vuelve muy extenso haciéndose difícil de leer, depurar, mantener.
* Si algo falla, todo se rompe: Ya que con la programación estructurada el código se ejecuta secuencialmente al momento de que una de esas líneas fallara todo lo demás deja de funcionar.
* Difícil de mantener.

# Paradigma Orientado a Objetos

La **Programación Orientada a Objetos** viene de una filosofía o forma de pensar que es la **Orientación a Objetos** y esto surge a partir de los problemas que necesitamos plasmar en código.

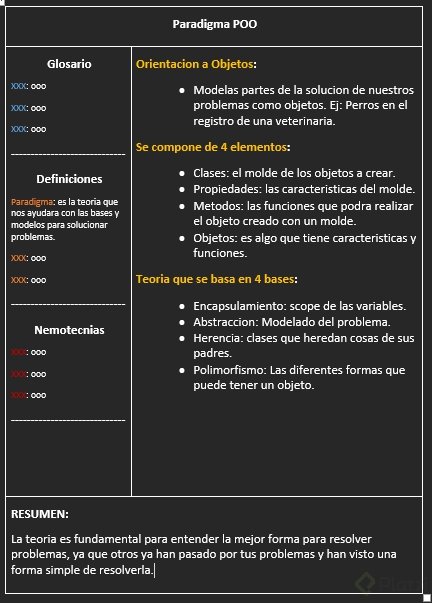
Es analizar un problema en forma de objetos para después llevarlo a código, eso es la **Orientación a Objetos**.

Un **paradigma** es una teoría que suministra la base y modelo para resolver problemas. La paradigma de Programación Orientada a Objetos se compone de 4 elementos:

* Clases
* Propiedades
* Métodos
* Objetos

Y 4 Pilares:

* Encapsulamiento
* Abstracción
* Herencia
* Polimorfismo



# Lenguajes Orientados a Objetos

**Correcciones:  
En el minuto 5:06. El navegador NO interpreta el código PHP. Lo hace el servidor**  
**Visual studio code y visual studio no es lo mismo.**  
**Visual Studio Code, el editor de código multiplataforma de Microsoft, es uno de los preferidos por muchos desarrolladores mientras que Visual Studio permite a los desarrolladores crear sitios y aplicaciones web, así como servicios web en cualquier entorno compatible con la plataforma .NET**

Algunos de los lenguajes de programación Orientados a Objetos son:

* **Java**:  
  – Orientado a Objetos naturalmente  
  – Es muy útilizado en Android  
  – Y es usado del lado del servidor o Server Side
* **PHP**  
  – Lenguaje interpretado  
  – Pensado para la Web
* **Python**  
  – Diseñado para ser fácil de usar  
  – Múltiples usos: Web, Server Side, Análisis de Datos, Machine Learning, etc
* **Javascript**  
  – Lenguaje interpretado  
  – Orientado a Objetos pero basado en prototipos  
  – Pensado para la Web
* C#
* Ruby
* Kotlin

# Diagramas de Modelado

**OMT**: Object Modeling Techniques. Es una metodología para el análisis orientado a objetos.

**UML**: Unified Modeling Language o Lenguaje de Modelado Unificado. Tomó las bases y técnicas de OMT unificándolas. Tenemos más opciones de diagramas como lo son Clases, Casos de Uso, Objetos, Actividades, Iteración, Estados, Implementación.

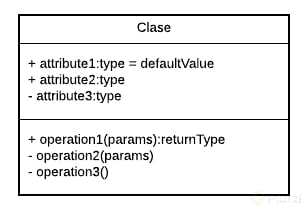
# Qué es UML

Como ya viste UML significa Unified Modeling Language el cual es un lenguaje estándar de modelado de sistemas orientados a objetos.



Esto significa que tendremos una manera gráfica de representar una situación, justo como hemos venido viendo. A continuación te voy a presentar los elementos que puedes utilizar para hacer estas representaciones.

Las **clases** se representan así:



En la parte superior se colocan los atributos o propiedades, y debajo las operaciones de la clase. Notarás que el primer caracter con el que empiezan es un símbolo. Este denotará la visibilidad del atributo o método, esto es un término que tiene que ver con Encapsulamiento y veremos más adelante a detalle.

Estos son los niveles de **visibilidad** que puedes tener:

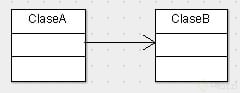
**-** private  
**+** public  
**#** protected  
**~** default

Una forma de representar las relaciones que tendrá un elemento con otro es a través de las flechas en UML, y aquí tenemos varios tipos, estos son los más comunes:

## Asociación



Como su nombre lo dice, notarás que cada vez que esté referenciada este tipo de flecha significará que ese elemento contiene al otro en su definición. La flecha apuntará hacia la dependencia.

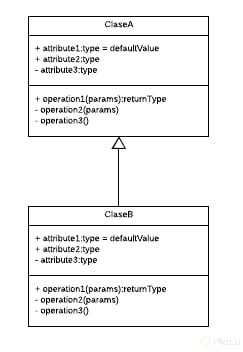


Con esto vemos que la ClaseA está asociada y depende de la ClaseB.

## Herencia



Siempre que veamos este tipo de flecha se estará expresando la herencia.  
La dirección de la flecha irá desde el hijo hasta el padre.

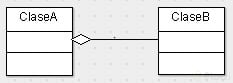


Con esto vemos que la ClaseB hereda de la ClaseA

## Agregación



Este se parece a la asociación en que un elemento dependerá del otro, pero en este caso será: Un elemento dependerá de muchos otros. Aquí tomamos como referencia la multiplicidad del elemento. Lo que comúnmente conocerías en Bases de Datos como Relaciones uno a muchos.

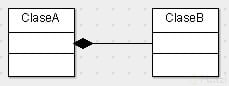


Con esto decimos que la ClaseA contiene varios elementos de la ClaseB. Estos últimos son comúnmente representados con listas o colecciones de datos.

## Composición



Este es similar al anterior solo que su relación es totalmente compenetrada de tal modo que conceptualmente una de estas clases no podría vivir si no existiera la otra.



Con esto terminamos nuestro primer módulo. Vamos al siguiente para entender cómo podemos hacer un análisis y utilizar estos elementos para construir nuestro diagrama de clases de Uber.

<https://www.youtube.com/watch?v=Z0yLerU0g-Q>

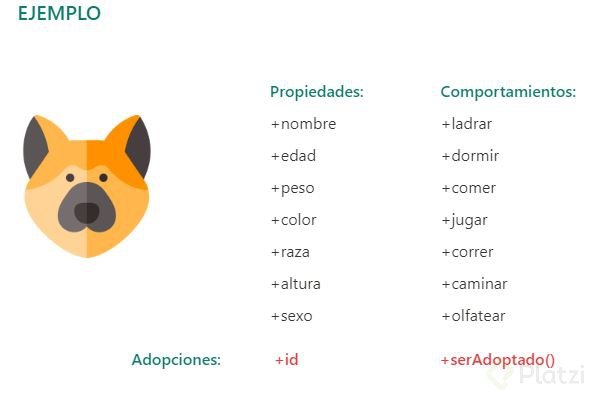
# Objetos

Los Objetos son aquellos que tienen propiedades y comportamientos, también serán sustantivos.

* Pueden ser Físicos o Conceptuales

Las **Propiedades** también pueden llamarse atributos y estos también serán sustantivos. Algunos atributos o propiedades son nombre, tamaño, forma, estado, etc. Son todas las características del objeto.

Los **Comportamientos** serán todas las operaciones que el objeto puede hacer, suelen ser verbos o sustantivos y verbo. Algunos ejemplos pueden ser que el usuario pueda hacer login y logout.



# Abstracción y Clases

Una **Clase** es el modelo por el cual nuestros objetos se van a construir y nos van a permitir generar más objetos.

Analizamos Objetos para crear **Clases**. Las **Clases** son los modelos sobres los cuales construiremos nuestros objetos.

**Abstracción** es cuando separamos los datos de un objeto para generar un molde.

Nota de clase:

En el vídeo anterior Anahi hablo muy bien de lo que es un Objeto y como esta compuesto por atributos o propiedades y a su vez estos tienen acciones o métodos.

Ahora el concepto se aumenta cuando todo esto se ve como una sola cosa llamada Clase.

Una Clase Es el modelo sobre el cual nuestros objetos se construyen.

Es decir si tenemos un objeto llamado perro y este tiene sus atributos que lo describen generalmente y a su vez tiene métodos donde se define las acciones que pueda hacer ese perrito. Una clase me permite generar mas objetos (mas perros) con mismos atributos y métodos pero con resultados diferentes. ej:

Objeto #1 llamado “Rocky”:

tributo\_1: color = marrón

atributo\_2: taman’o = pequen’o

atributo\_3: raza = chiguagua

metodo\_1: ladrar

metodo\_2: comer

metodo\_3: dormir

Objeto #2 llamado "Max"

atributo\_1: color = blanco

atributo\_2: taman’o = grande

atributo\_3: raza = hunky siberiano

metodo\_1: ladrar

metodo\_2: comer

metodo\_3: dormir

Para no repetir esto muchas veces de acuerdo a la cantidad de perros que es mi ejemplo de objeto, la idea es analizar todos estos objetos extraemos todos esos atributos y entonces generamos modelos. Esos modelos se le llaman Clases.

Una Clase son los modelos sobre los cuales construiremos Objetos

A este análisis se le conoce como Abstracción, simplemente consiste en generar un molde en base a esas propiedades y métodos de los objetos, abstraemos todos esos datos para generar dicho molde.

Resumen: Una clase es un molde para generar un objeto y este análisis se llama Abstracción

abstracción: recoger los datos y caracteristicas de un objeto para luego definir con ello una interfaz

# Modularidad

La **modularidad** va muy relacionada con las clases y es un principio de la Programación Orientado a Objetos y va de la mano con el Diseño Modular que significa dividir un sistema en partes pequeñas y estas serán nuestros módulos pudiendo funcionar de manera independiente.

La **modularidad** de nuestro código nos va a permitir

* Reutilizar
* Evitar colapsos
* Hacer nuestro código más mantenible
* Legibilidad
* Resolución rápida de problemas

Una buena práctica es separando las clases en archivos diferentes.

Modular: Dividir un sitema y así crear módulos independientes, lo que permite evitar un colapso masivo en nuestro código y mejorar la legibilidad.

La modularidad va de la mano con el Diseño Modular, este consiste en dividir nuestro sistema en partes pequeñas que funcionen de manera independiente.

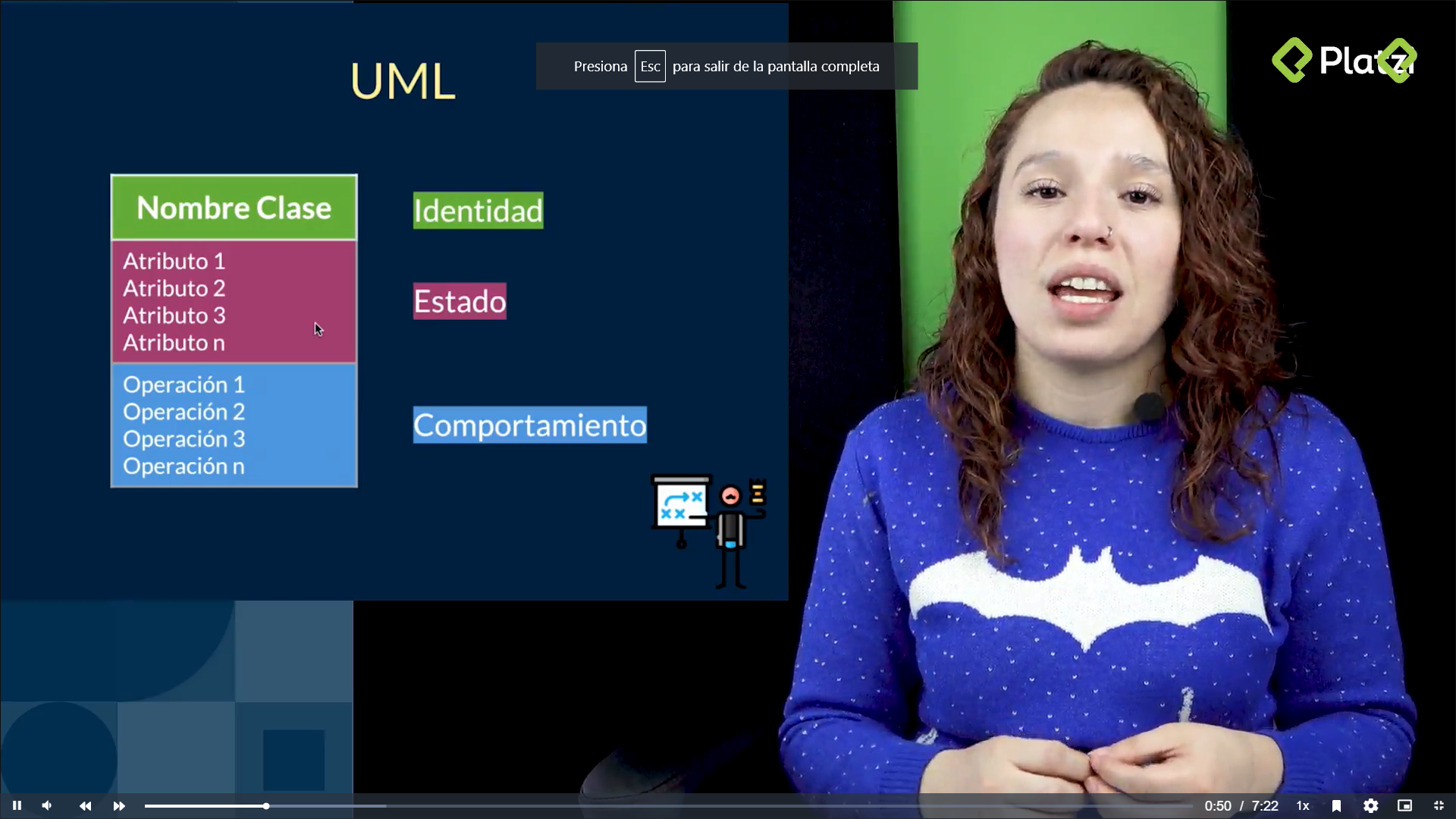


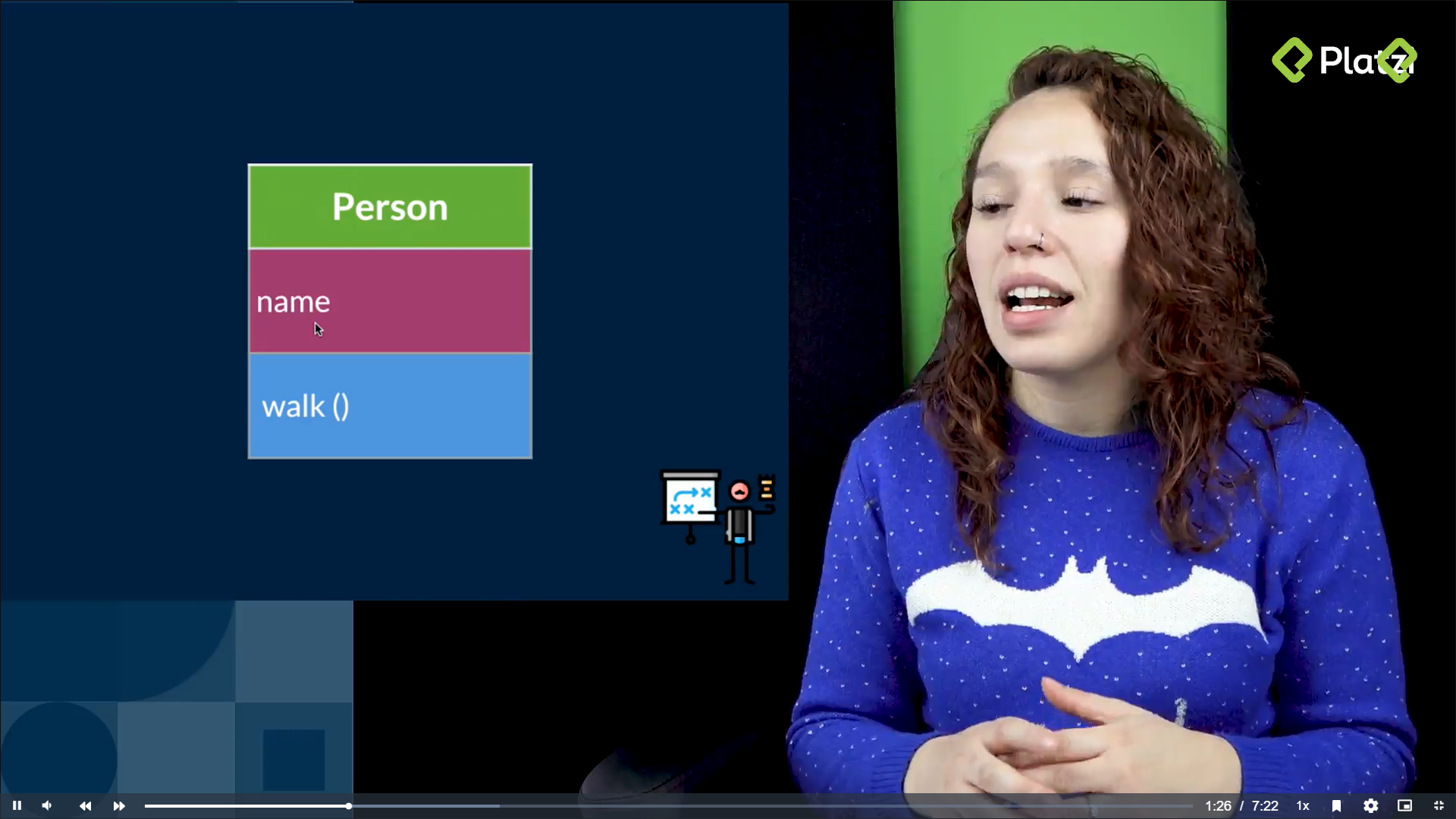
# Analizando Uber en Objetos

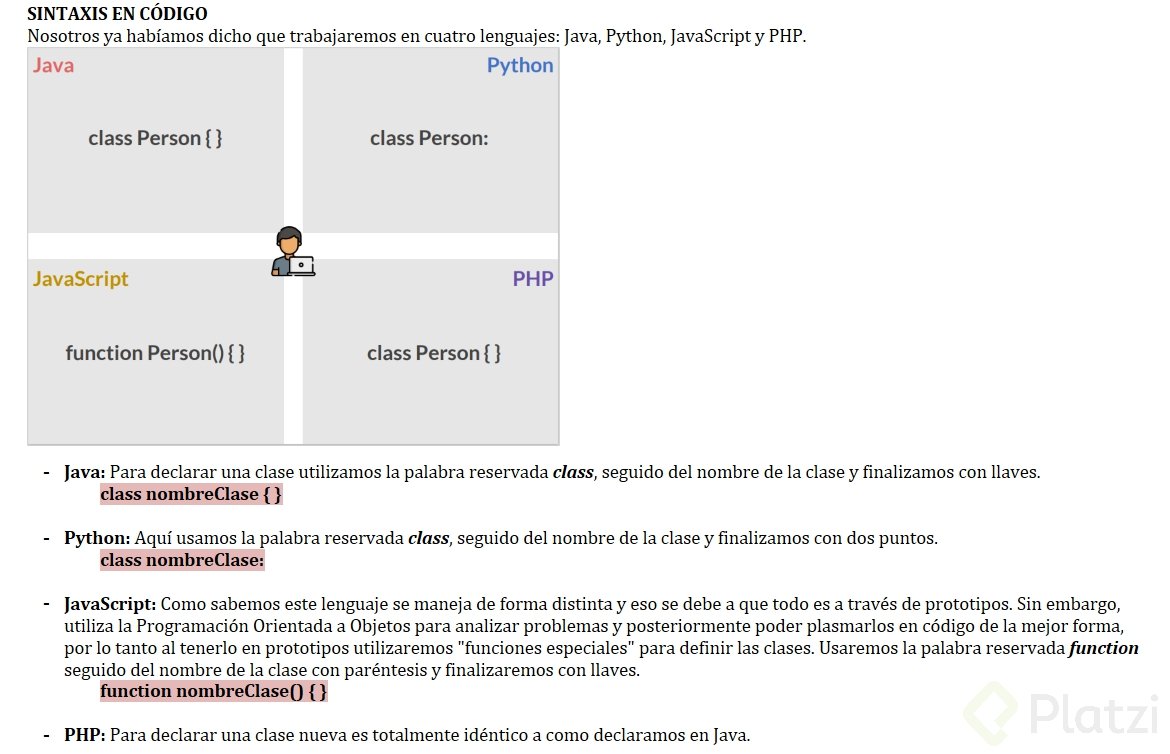
# 

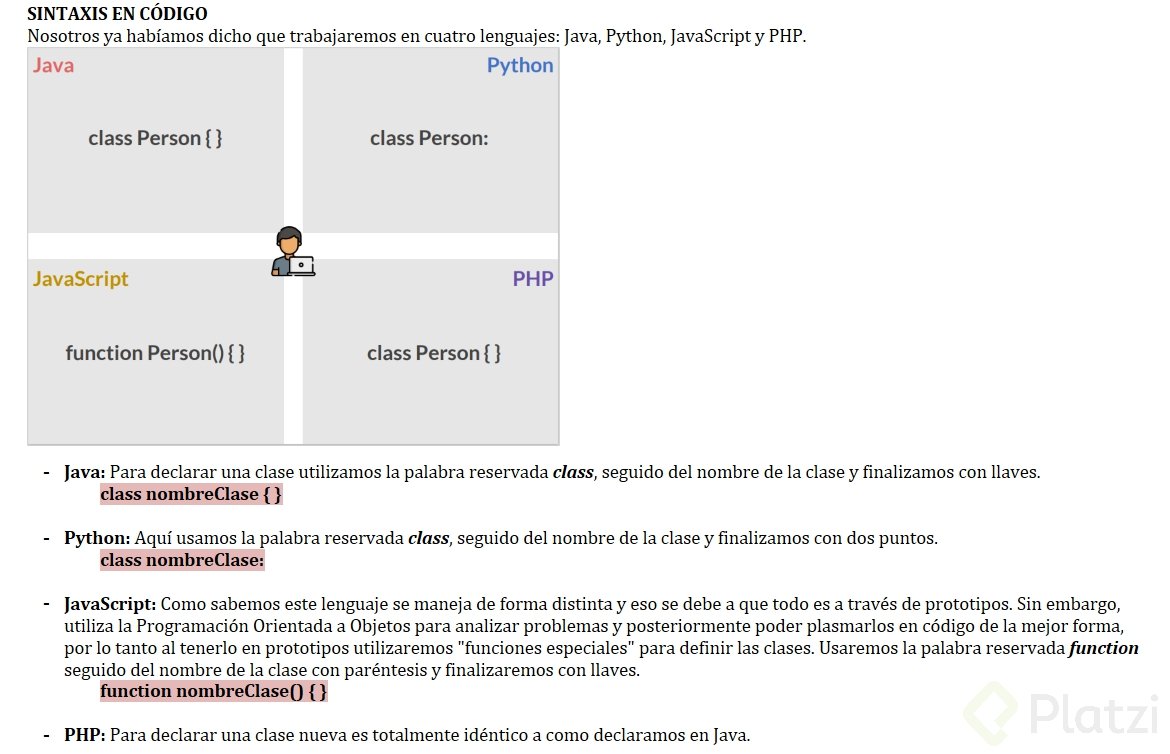
# Preview

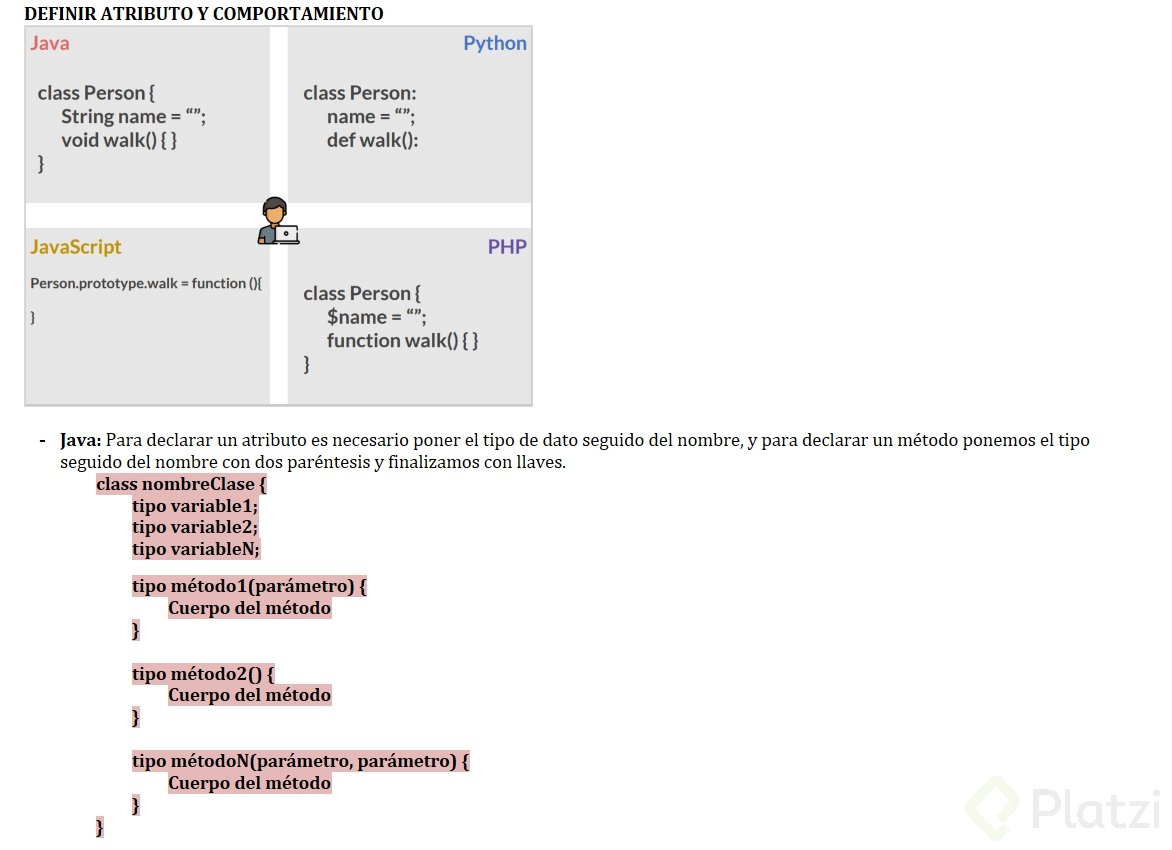
# Clases en UML y su sintaxis en código

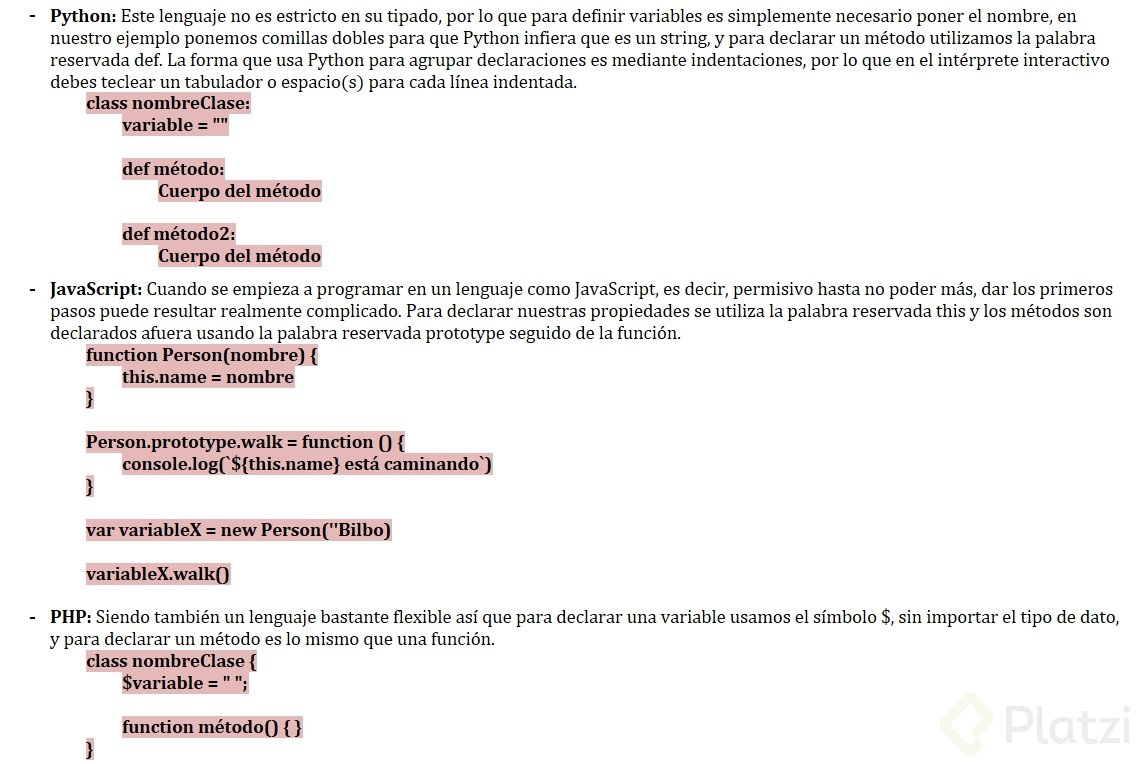




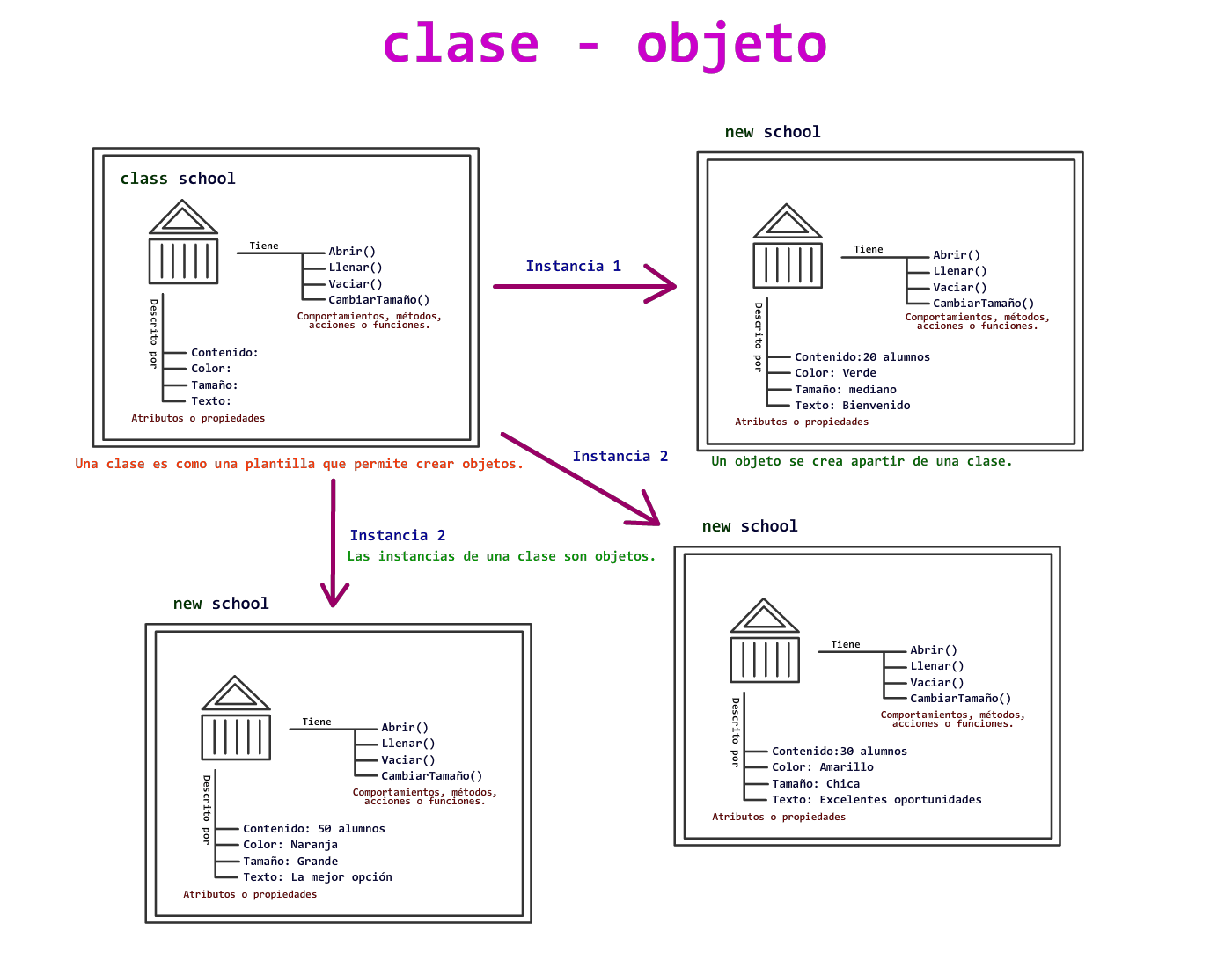


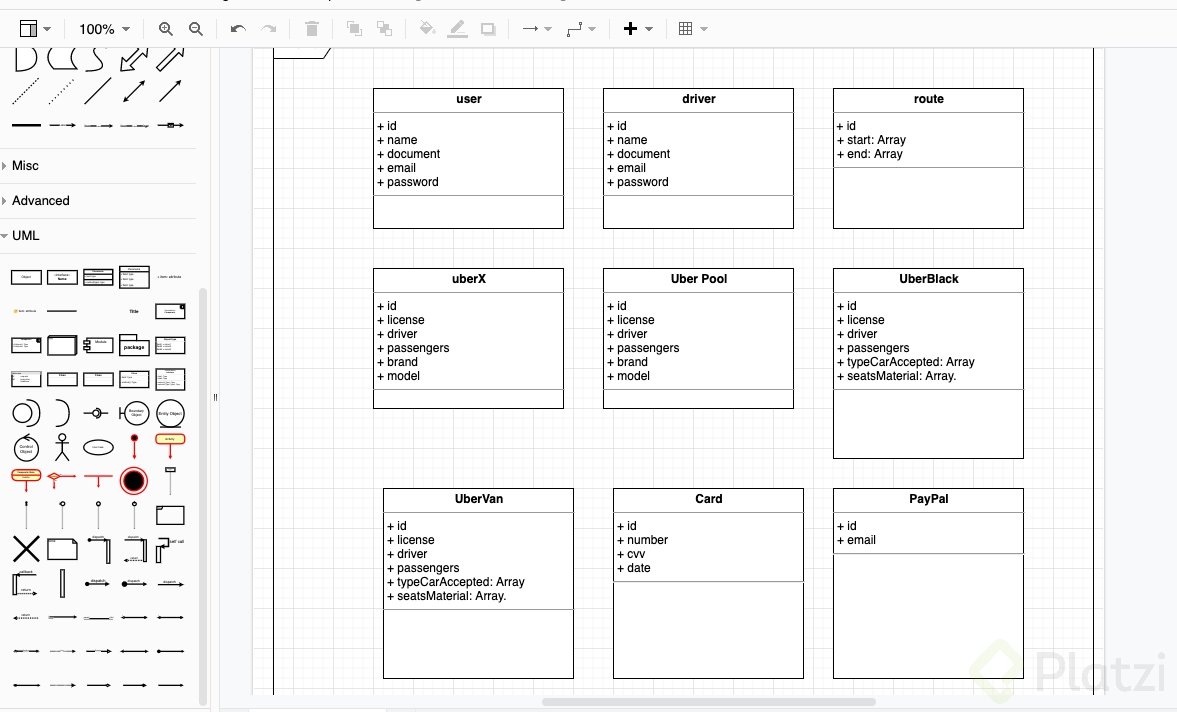


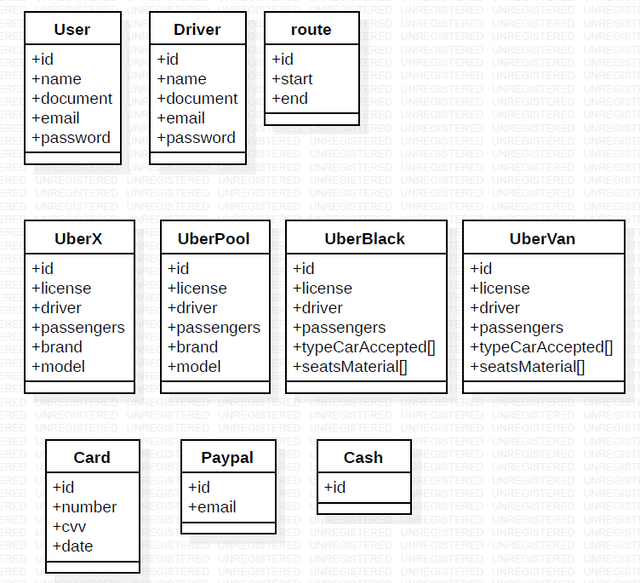




# Modelando nuestros objetos Uber







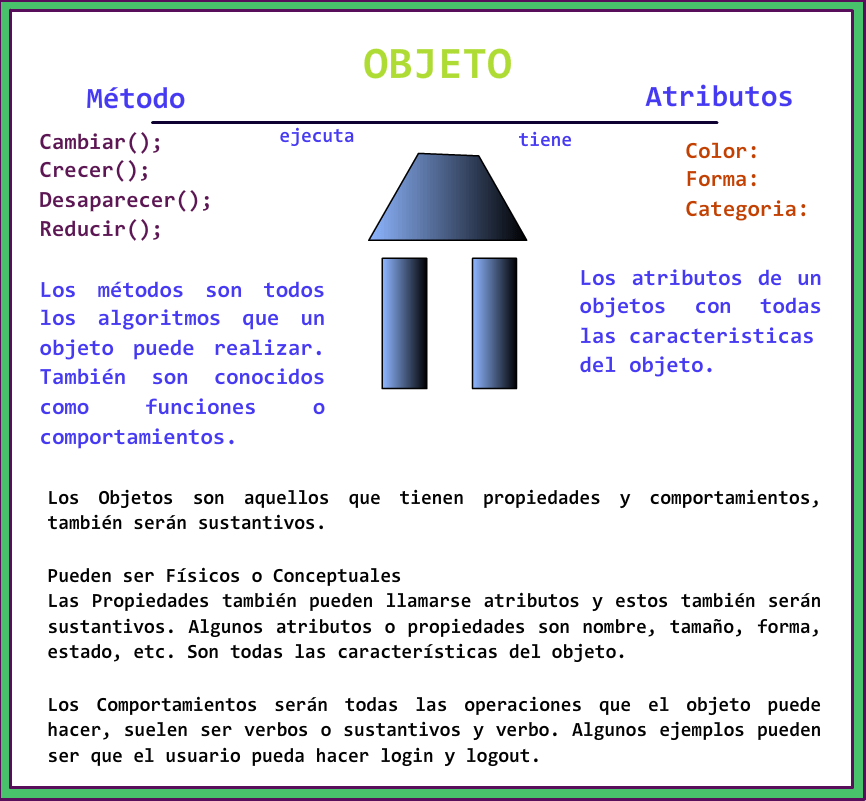
# ¿Qué es la herencia?

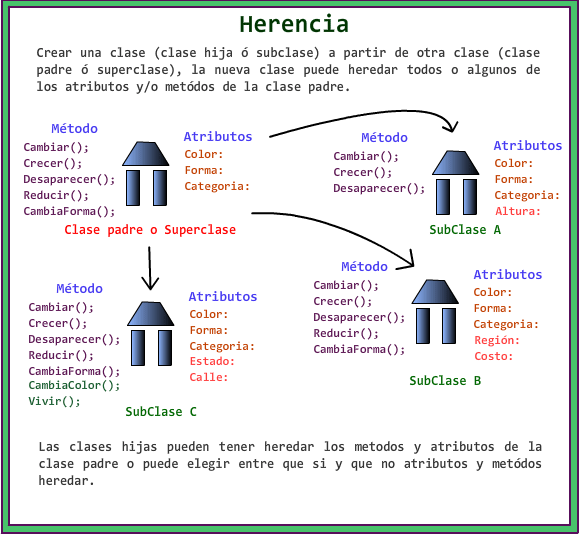
**Don’t repeat yourself** es una filosofía que promueve la reducción de duplicación en programación, esto nos va a inculcar que no tengamos líneas de código duplicadas.

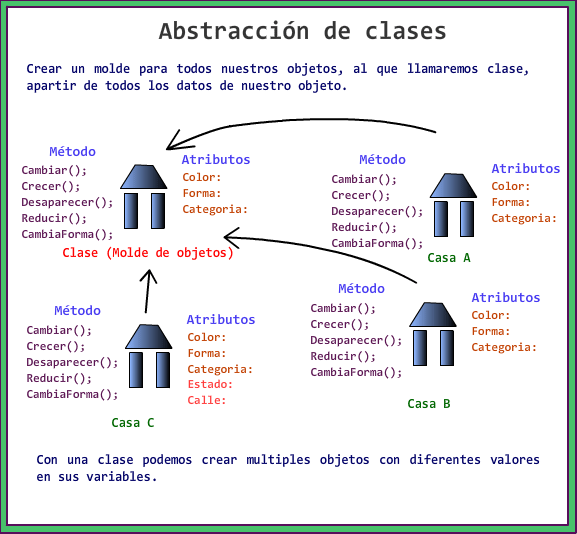
Toda pieza de información nunca debería ser duplicada debido a que incrementa la dificultad en los cambios y evolución

La **herencia** nos permite crear nuevas clases a partir de otras, se basa en modelos y conceptos de la vida real. También tenemos una jerarquía de **padre e hijo**.

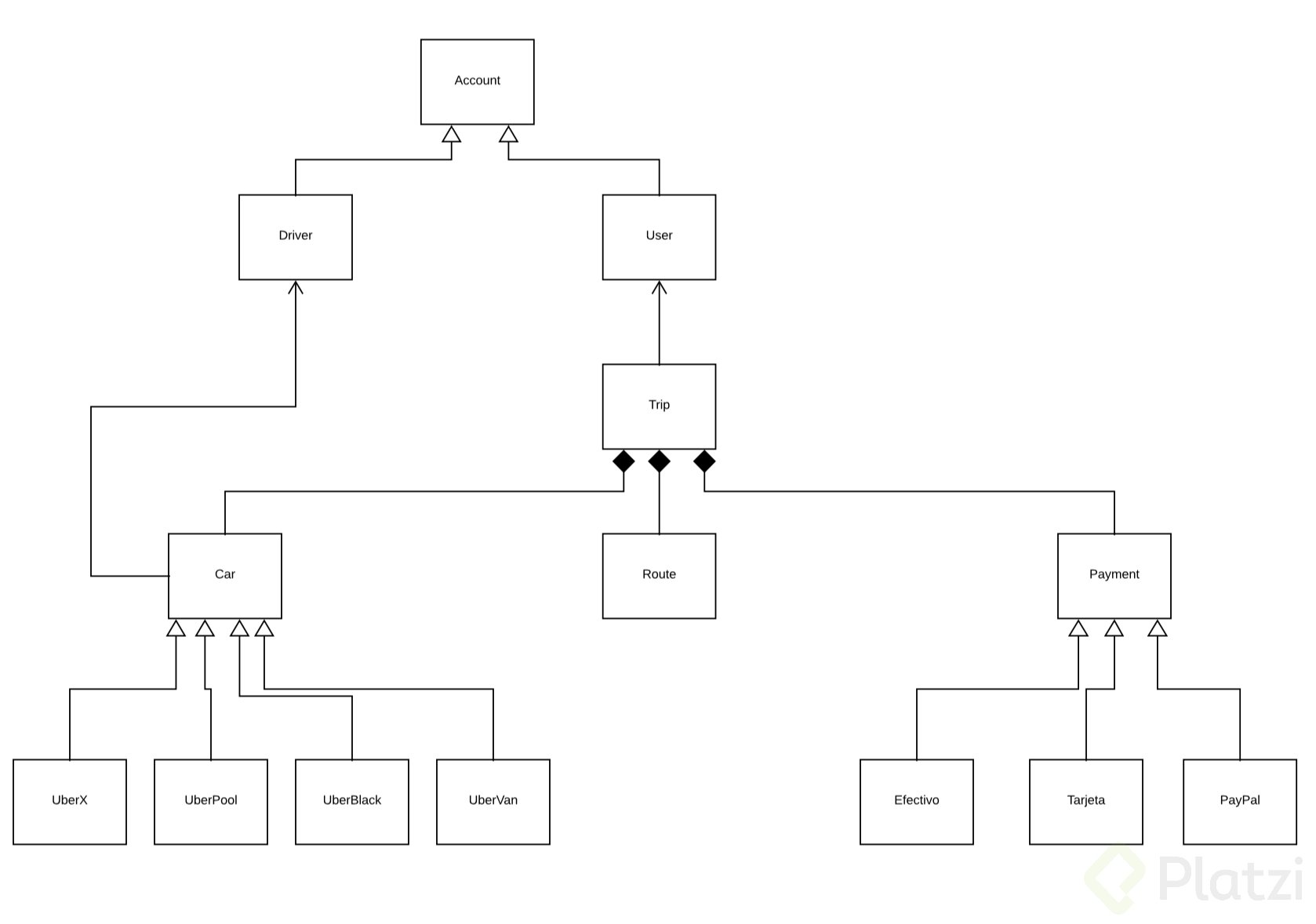


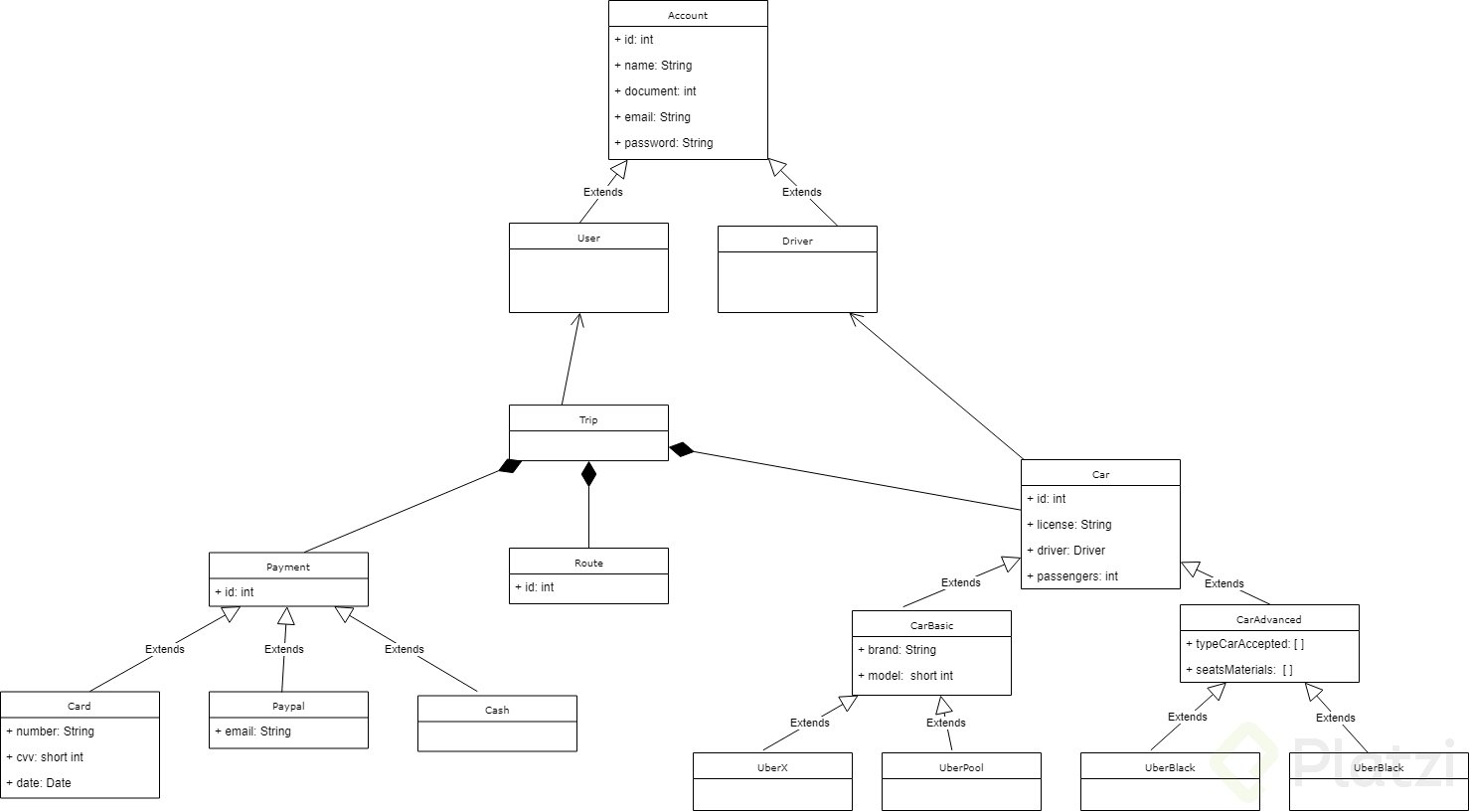




3

# Aplicando Herencia a nuestro proyecto Uber



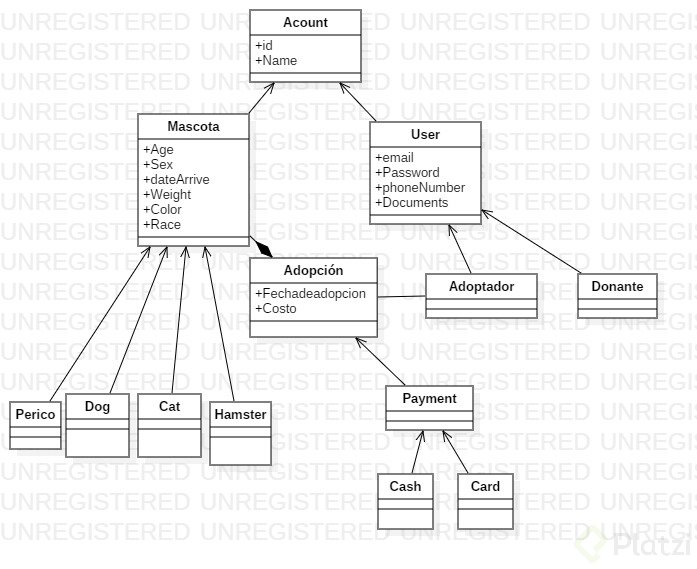


# Reto 2: analicemos un problema

Imagina que nuestro sistema de adopciones creció y ahora ofrece adoptar pericos, loros, gatos y hamsters.

Genera un nuevo análisis, aplica herencia para abstraer mejor el problema y lograr modularidad en el software.

Comparte tus resultados en la sección de discusiones.



# Creando nuestras carpetas iniciales para el curso

¡Hola! Ahora sí vamos a empezar a definir nuestras clases en 4 diferentes lenguajes de programación, pero antes de ello vamos primero a crear la estructura de nuestras carpetas.

La estructura de carpetas que estaremos manejando durante el curso será la siguiente:

CursoPOOUber

├── **Java**

├── **JS**

├── PHP

└── Python

Básicamente nuestra carpeta raíz será la carpeta CursoPOOUber y dentro crearemos las carpetas Java, JS, PHP y Python en las cuales iremos guardando nuestros archivos a medida que avanzamos en el curso.

## Configurando el workspace para Python

En una clase anterior instalamos Python, sin embargo, como estaremos guardando todo el código de este lenguaje dentro de la carpeta Python debemos informarle a Visual Studio Code que el código va a estar ahí, de esta forma podemos evitar problemas a futuro cuando usemos módulos.

Para hacer esto es muy sencillo, dentro de nuestra carpeta CursoPOOUber vamos a crear una nueva carpeta llamada .vscode (ojo, el punto al inicio del nombre es importante). Dentro de esta carpeta vamos a crear un nuevo archivo llamado settings.json y dentro de este archivo vamos a poner la siguiente información:

{

"python.autoComplete.extraPaths": ["./Python"],

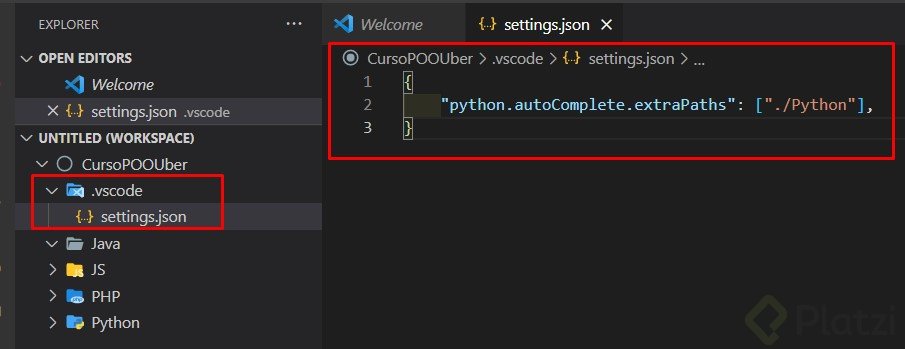
}

De esta forma le estamos diciendo a nuestro editor de código que todos nuestros archivos de Python van a estar dentro de la carpeta Python, por tanto, él sabrá que cualquier módulo deberá leerlo desde esa carpeta, y no desde la carpeta raíz, lo cual evitará que te marque errores en el futuro 😄.

**Es importante** que cuando abras tu proyecto en Visual Studio Code, lo abras desde la carpeta CursoPOOUber, de esta forma, el editor de código te mostrará en el árbol de carpetas a tus 5 carpetas que has creado:



¡Estamos listos para empezar a codear! Nos vemos en las siguiente clase donde empezaremos a definir clases con Java y Python.

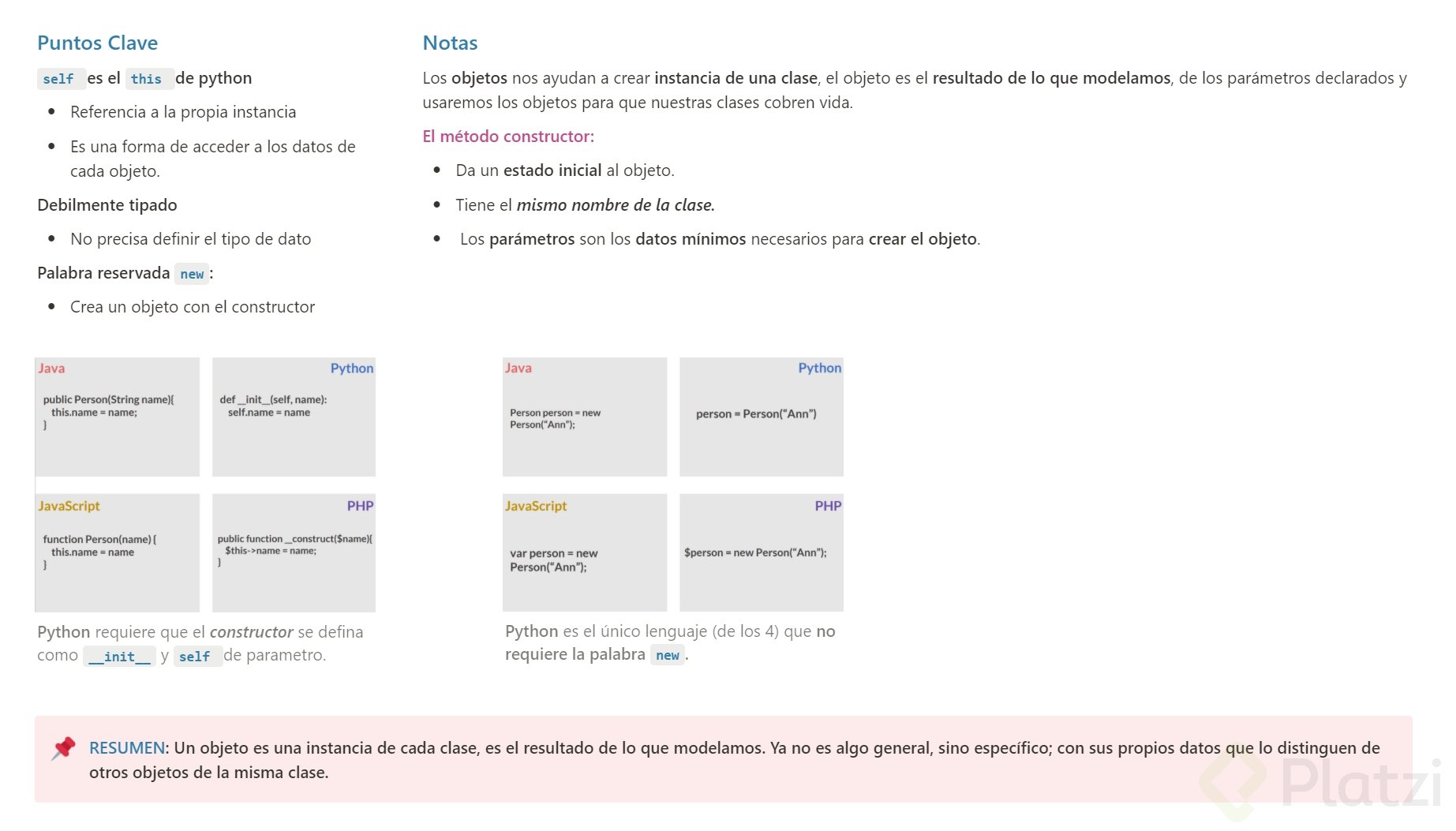


Material Icon Teme

# Objetos, método constructor y su sintaxis en código

Los **objetos** nos ayudan a crear instancia de una clase, el objeto es el resultado de lo que modelamos, de los parámetros declarados y usaremos los objetos para que nuestras clases cobren vida.

Los **métodos constructores** dan un estado inicial al objeto y podemos añadirle algunos datos al objeto mediante estos métodos. Los atributos o elementos que pasemos a través del constructor serán los datos mínimos que necesita el objeto para que pueda vivir.



# Objetos. Dando vida a nuestras clases en Java y Python

# Declarando un Método Constructor en Java y JavaScript

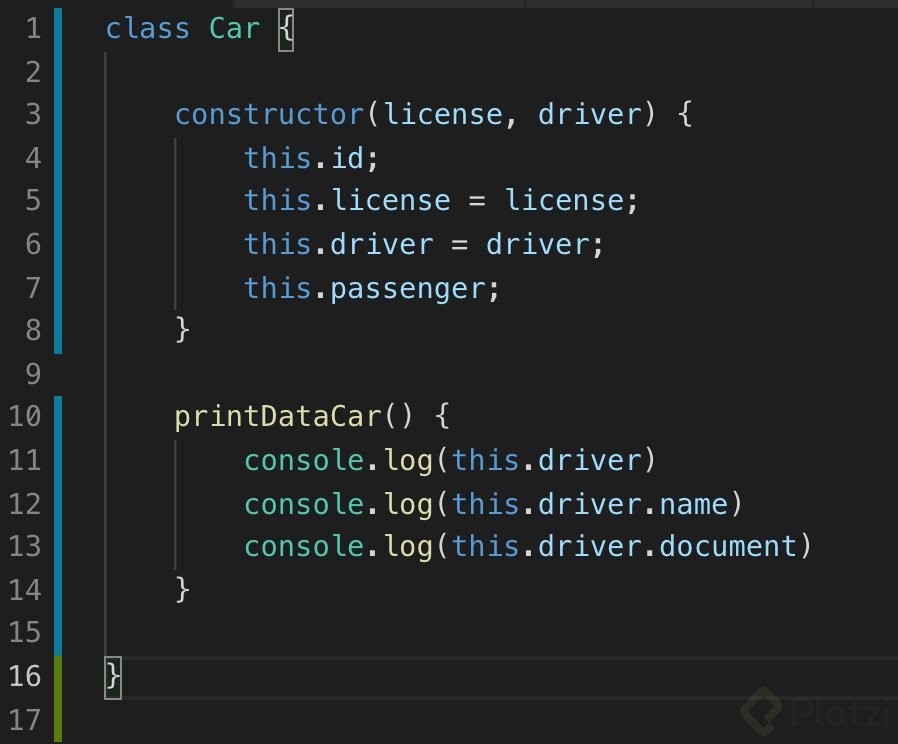
# JavaScript orientado a objetos, lo más nuevo

A partir de las nuevas especificaciones del EcmaScript 6 ya podemos declarar una clase con la palabra reservada class, aunque es importante aclarar que estos no dejan de ser prototipos, sino todo lo contrario.

Además tendremos una palabra clave para definir un constructor, y dentro de este estarán las propiedades de nuestra clase definidas listas para inicializarse.

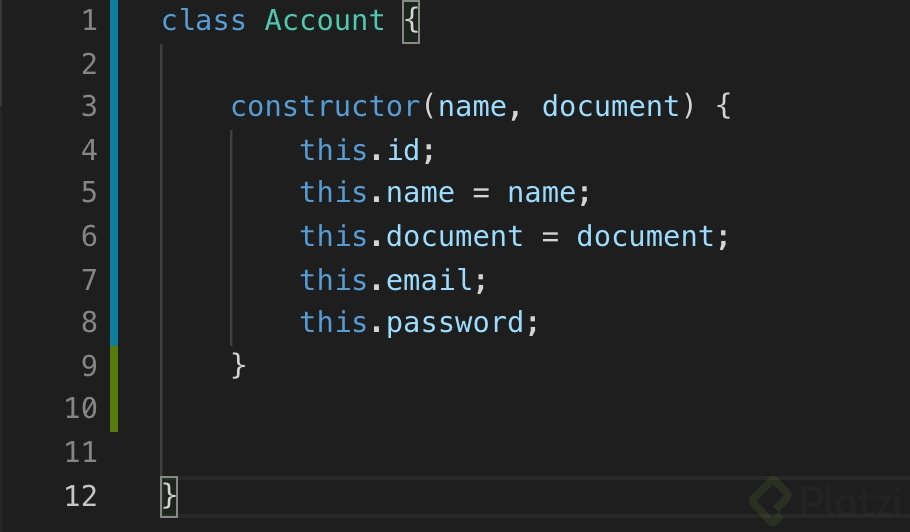
Transcribamos el código JavaScript que generamos en la clase anterior a este nuevo estándar.

La clase **Car** quedaría así:

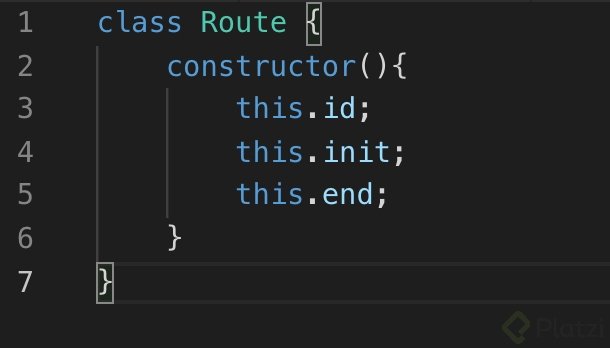


Si quisiéramos declarar un método, en esta nueva sintaxis dejaremos de utilizar la palabra clave function.

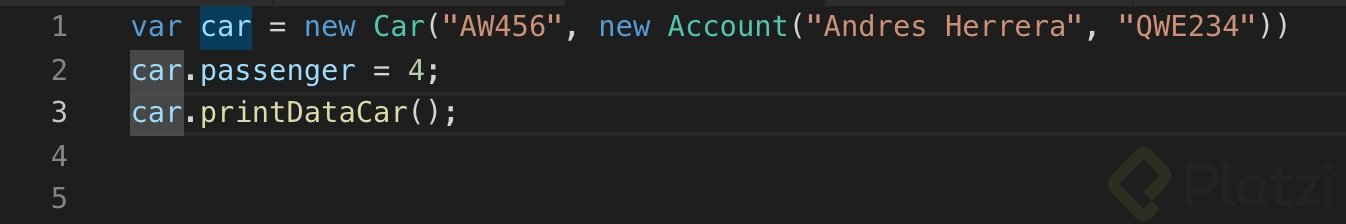
Ahora veamos a la clase **Account**:



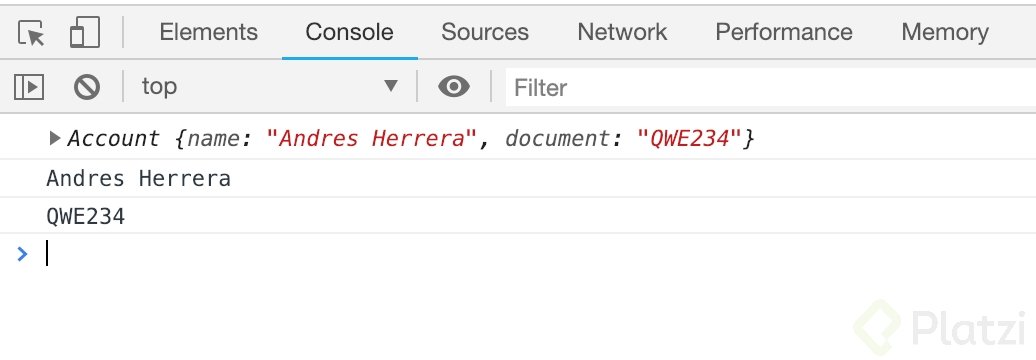
Y para finalizar aquí puedes ver las clases **Route** y **Payment**:



Notarás que para instanciar un objeto seguiremos usando la palabra clave new.



Y los resultados serán los mismos:



Aquí encuentras el código de este ejercicio: <https://github.com/anncode1/Curso-POO-Platzi/tree/3.1.POOJS>

# Aplicando herencia en lenguaje Java y PHP

## **Disclaimer:**

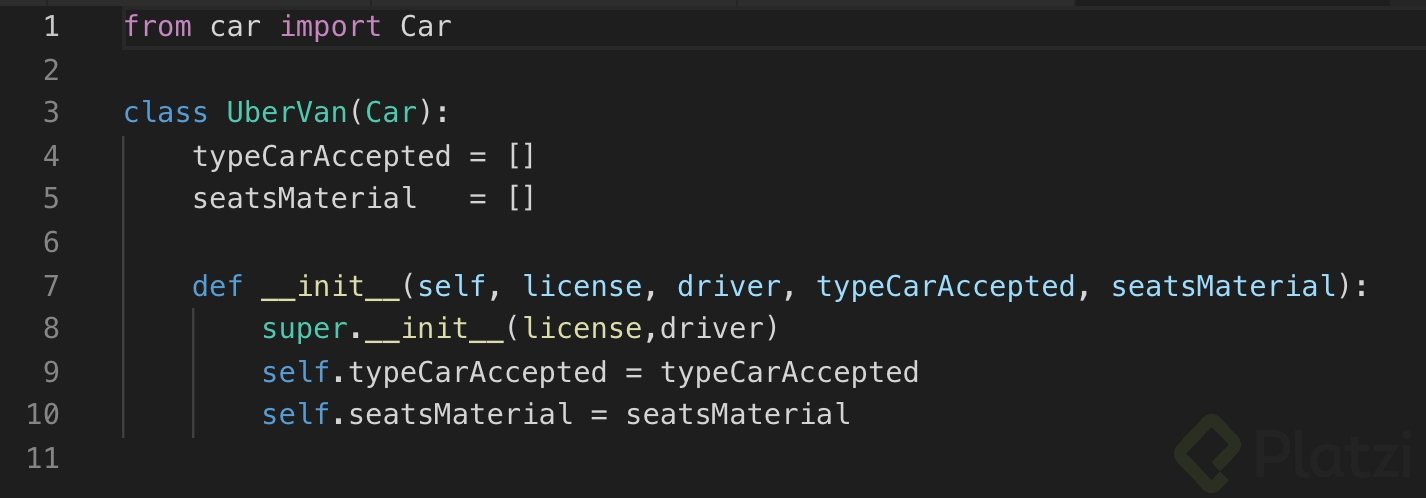
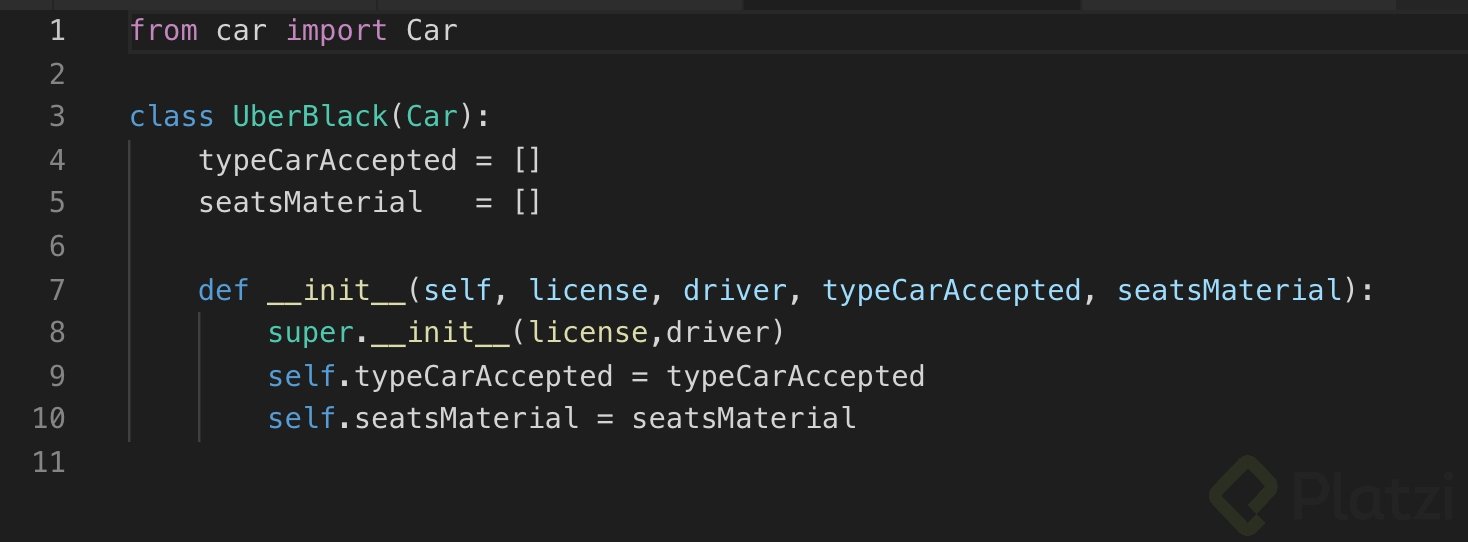
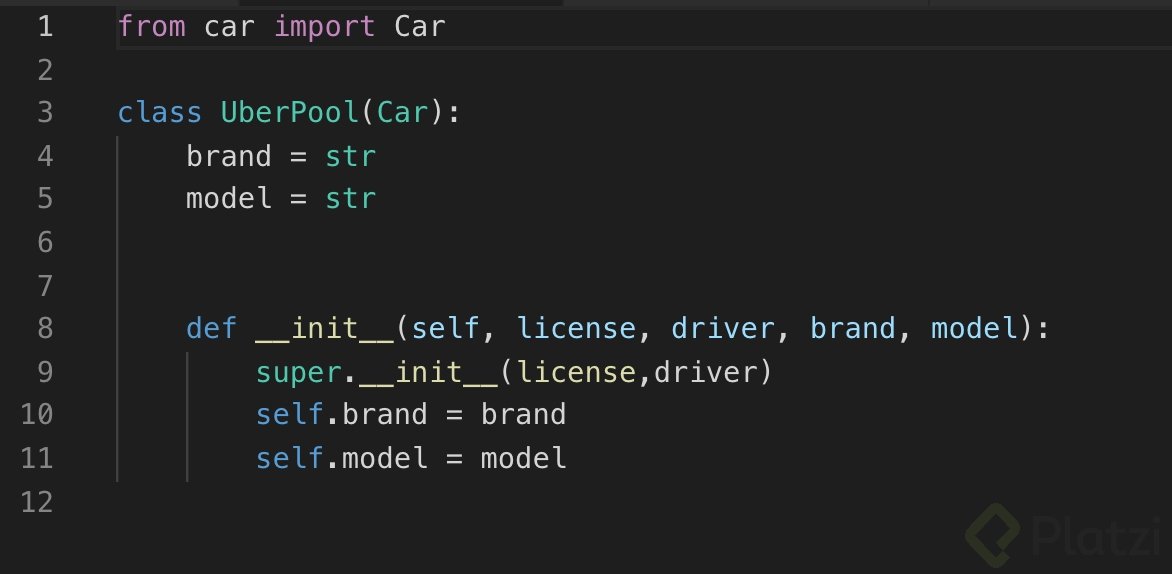
ArrayList solo permite un argumento, por ejemplo: ArrayList<String> myList;  
En este caso se necesitan dos argumentos quedaría algo así  
Map<String, Map<String, Integer>> typeCarAccepted;

# Aplicando herencia en lenguaje Python y JavaScript

## Python

Recuerdas que en Python la herencia se expresa de manera muy similar a un método constructor de otros lenguajes. Apliquemos herencia para nuestra familia Car, para esto crearemos las siguientes clases:

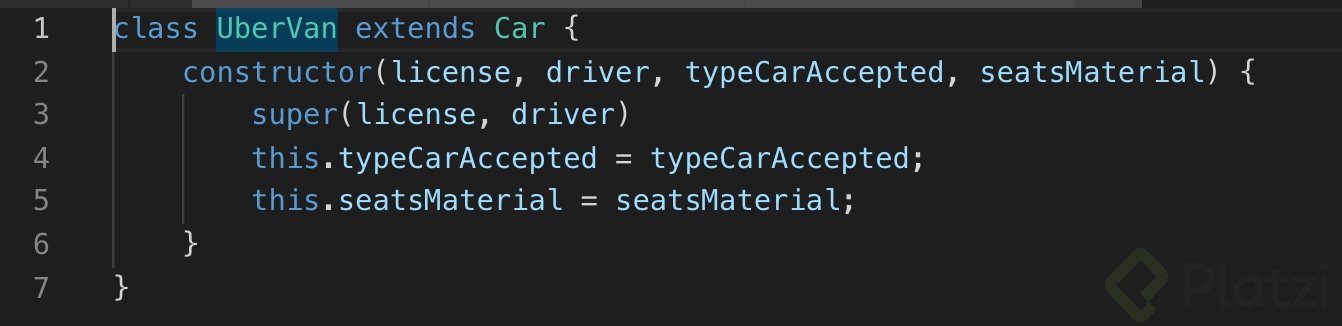
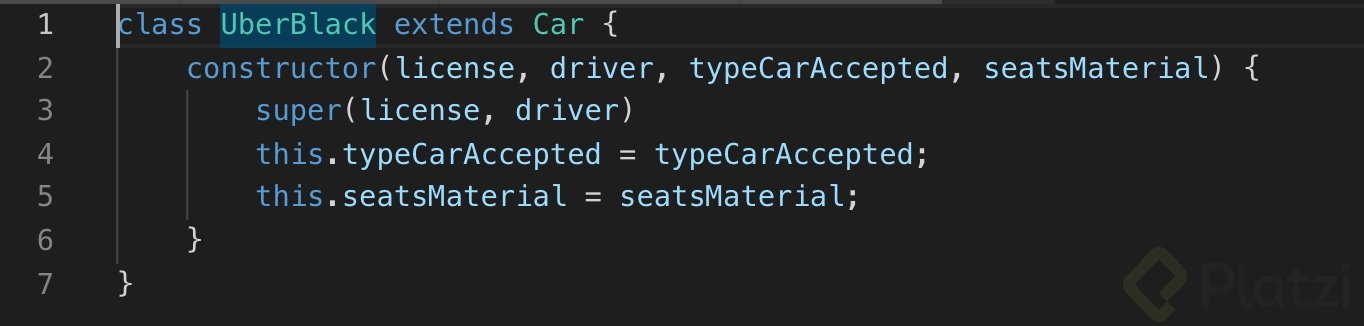
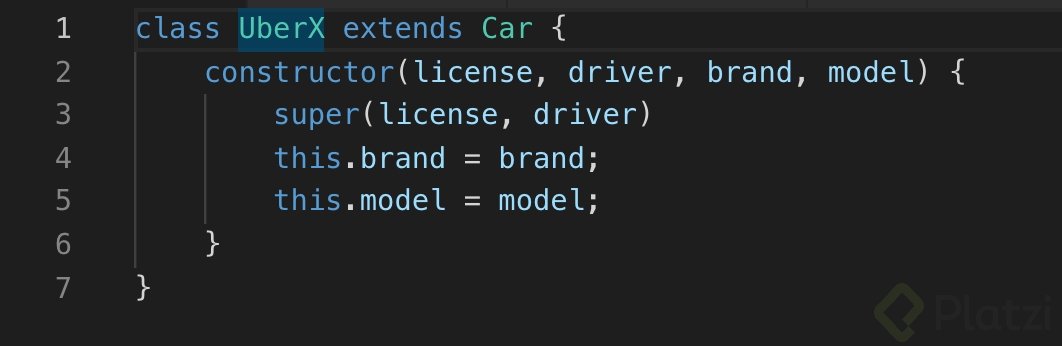
* [UberX.py](http://uberx.py/)
* [UberPool.py](http://uberpool.py/)
* [UberBlack.py](http://uberblack.py/)
* [UberVan.py](http://ubervan.py/)



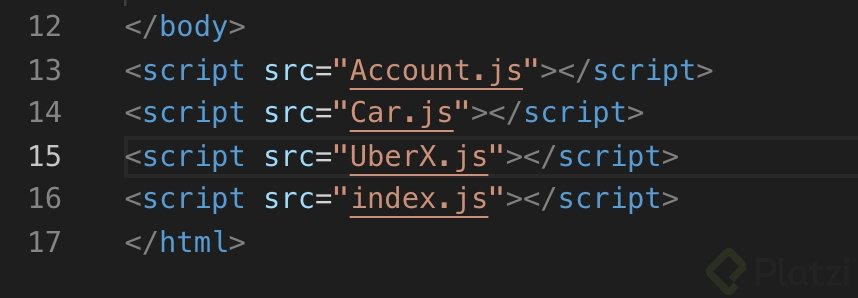
El código completo puedes verlo aquí: <https://github.com/anncode1/Curso-POO-Platzi/tree/2cbdf9db470a98323328f8a21bf6a9de941d008e/Python>

## JavaScript

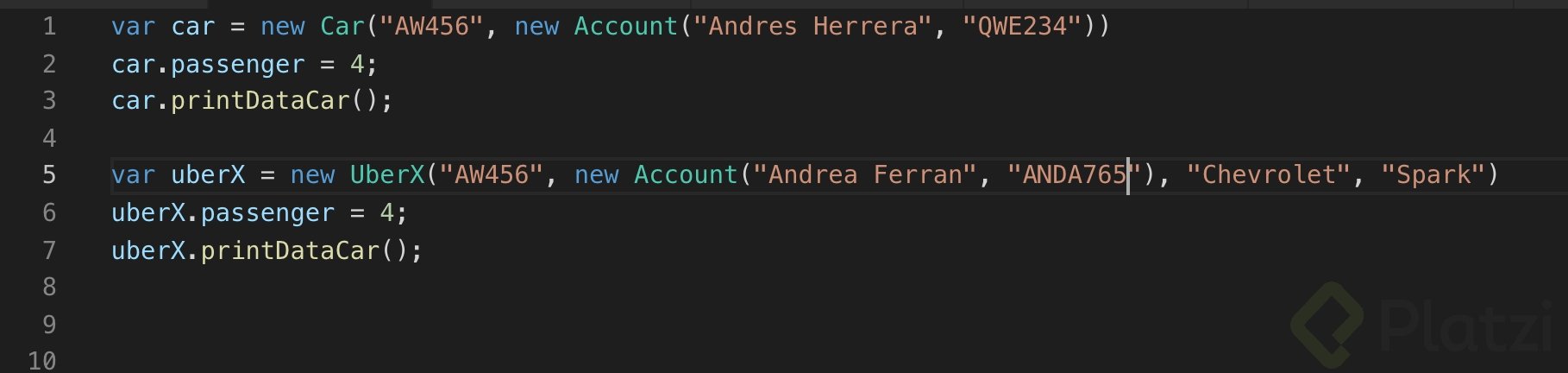
En clases anteriores te expliqué cómo ejecutar herencia en estándares anteriores al EcmaScript 6. Uno de los beneficios de utilizar este nuevo estándar que ejecutar herencia es tan simple como utilizar la palabra reservada **extends**.



Ahora para utilizar una de las clases y crear un objeto, por ejemplo de UberX, no olvides declarar la clase en el archivo **index.html**.



Nuestro ejemplo se verá así:



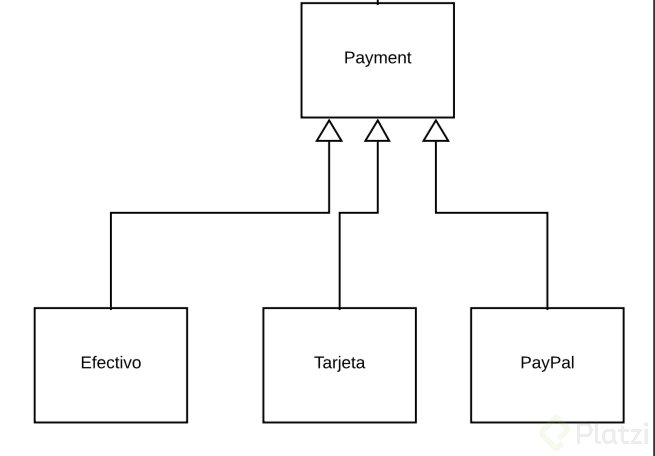
El código completo puedes verlo aquí: <https://github.com/anncode1/Curso-POO-Platzi/tree/9251101bdc2722ed13f9d93cb432ba8e9aba17b4/JS>

A partir de ahora las clases que estén siendo heredades las llamaremos familias.

Acabamos de aplicar herencia a la familia **Car**. Ahora apliquémosla a la familia **Payment**.

En clases anteriores te mencioné que otro punto de partida que puedes tomar para aplicar herencia es del hecho de que hay clases que lógicamente deberían estar en una familia, como es el caso de Payment.

Repasemos el diagrama de Payment



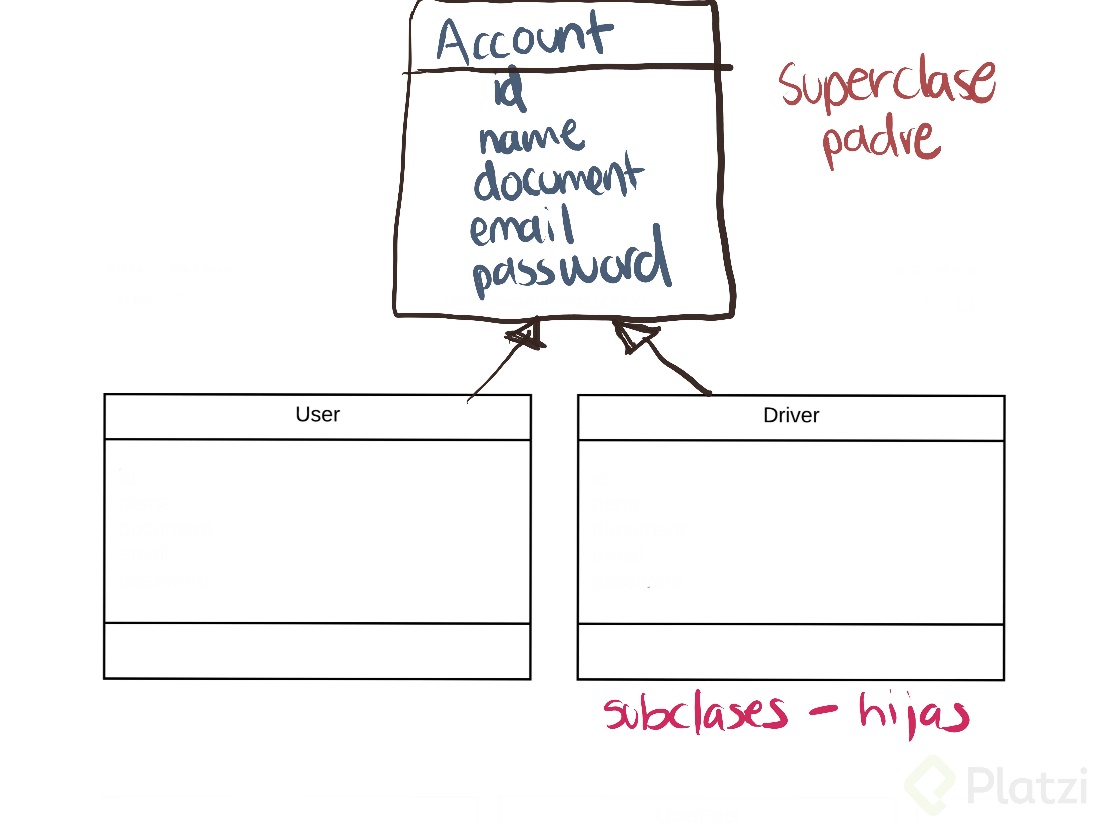
Notarás que a nivel de código parece inservible pero cuando estemos en el caso de uso Pagar un Viaje, probablemente en ese momento no sabremos cuál es el método de pago, y necesitemos ingresar un dato lo suficientemente genérico que conceptualmente nos dé la información que necesitamos, en este caso que es un Payment. Este es un tipo de Polimorfismo y uno de los principios SOLID del software que obedece a la Inyección de Dependencias. Lo veremos más adelante a detalle.

Ahora nos faltará crear las clases y aplicar su herencia.

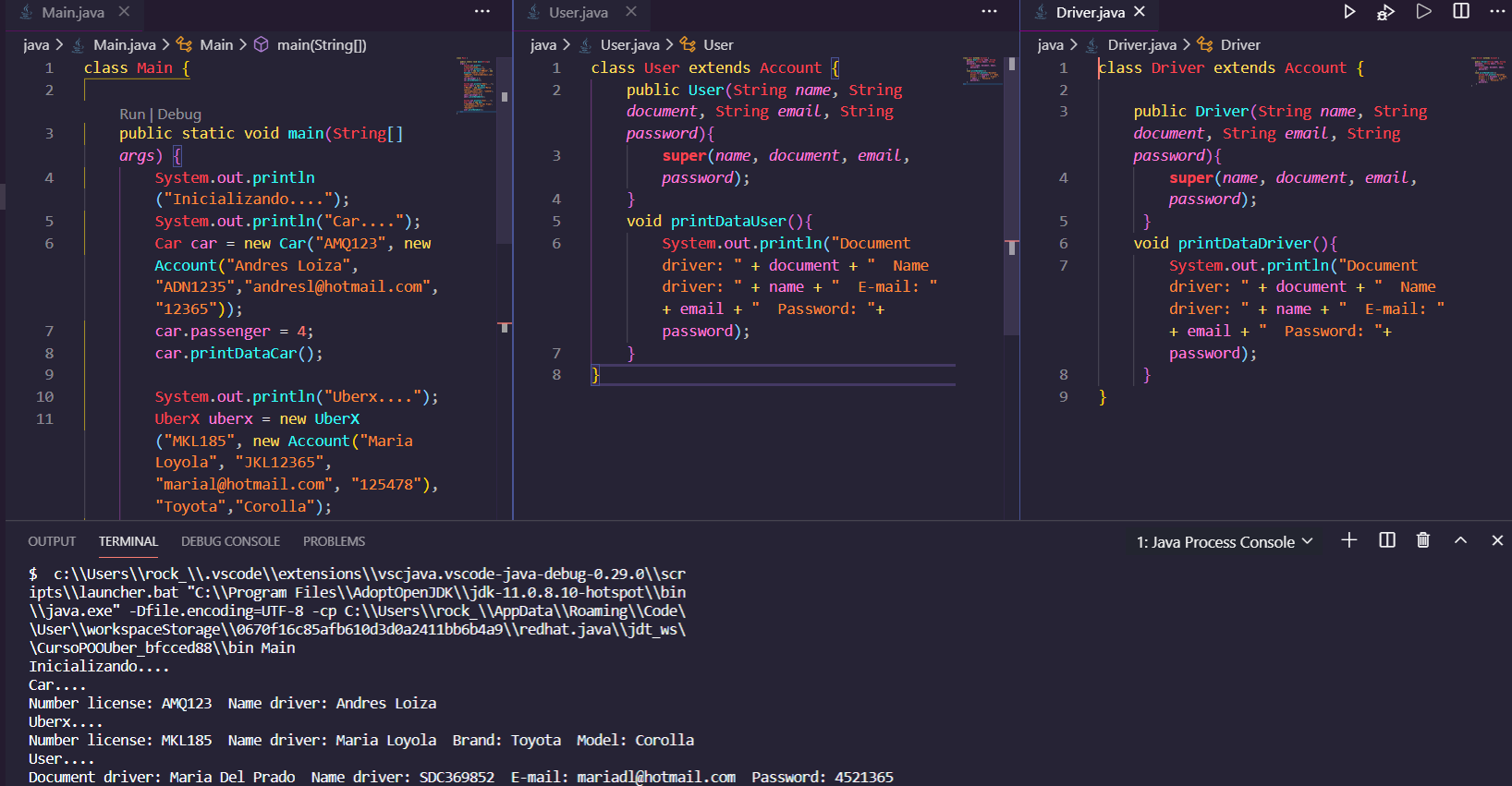
# Reto 4

Nos queda la Jerarquía Account pendiente.

Tomando como referencia nuestros diagramas. Plásmala en tu lenguaje de programación favorito.  
Compártenos tus resultados.



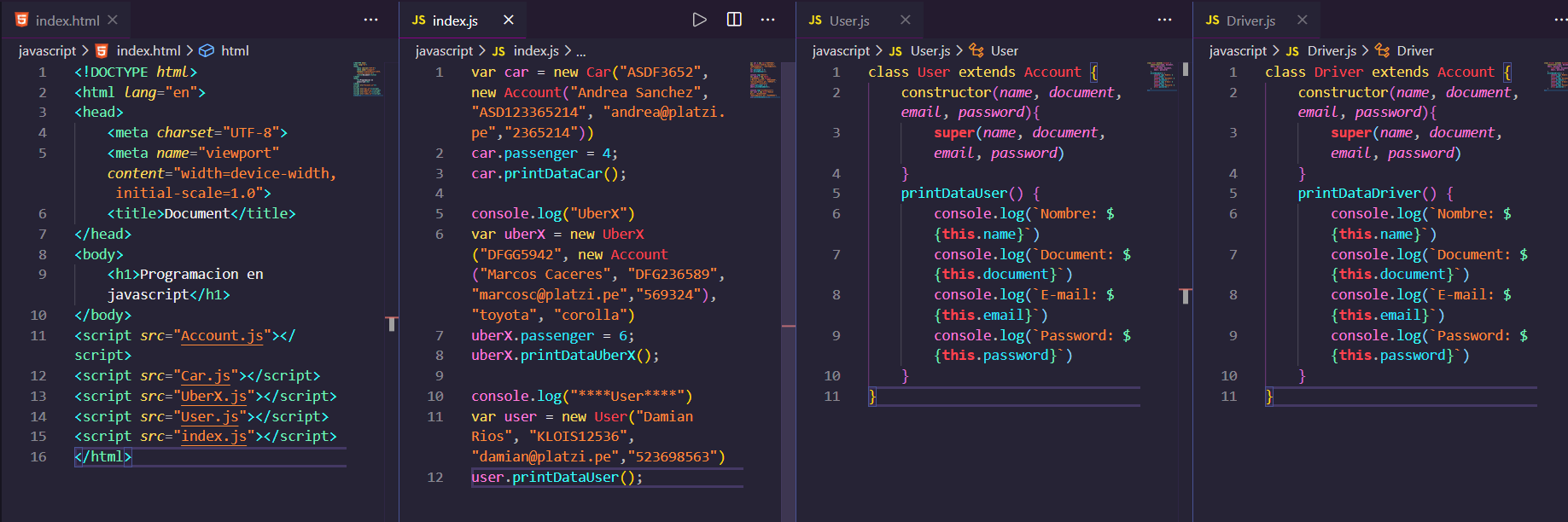
Java:

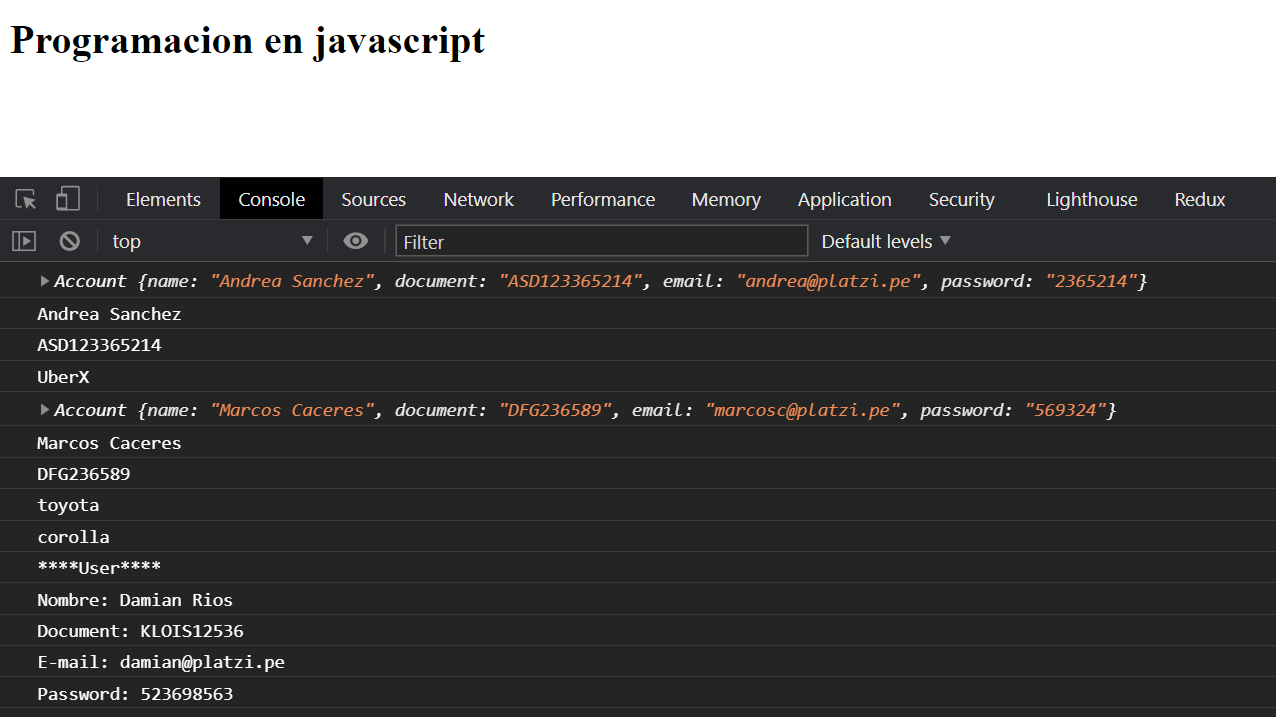


Python:

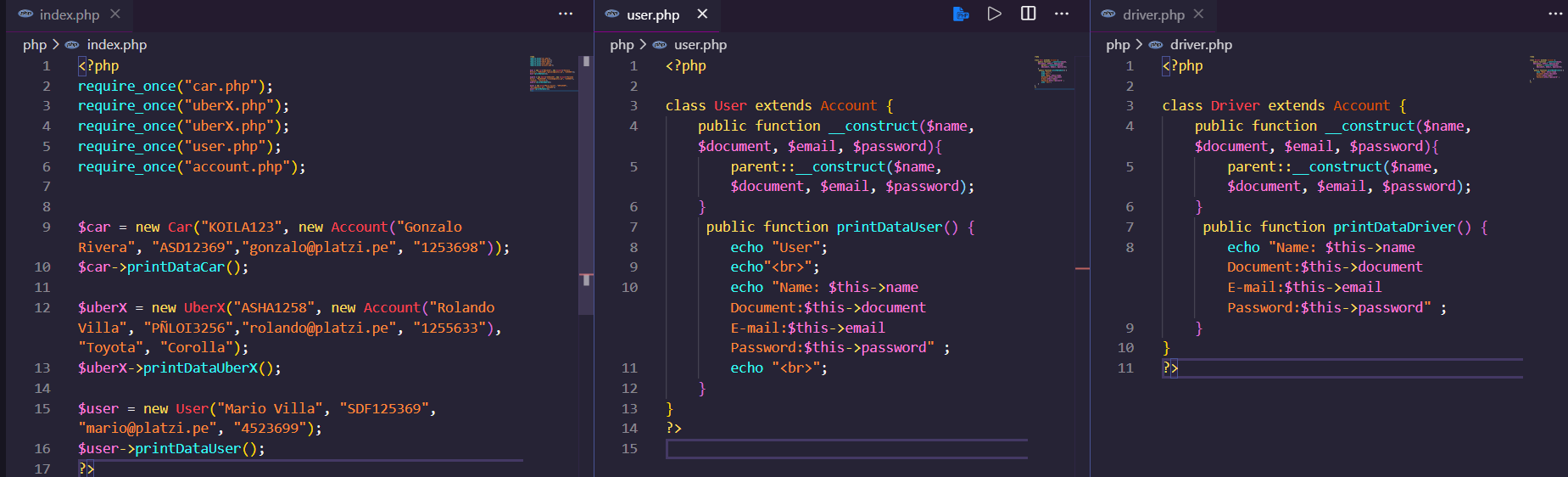


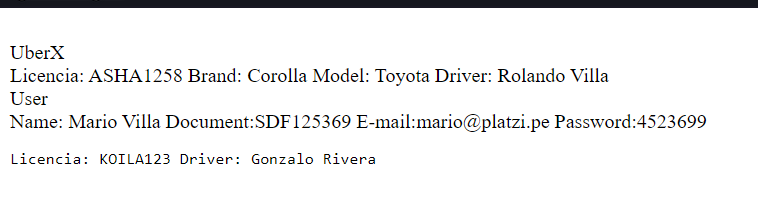
Javascript:





Php:





# Encapsulamiento

El **Encapsulamiento** es hacer que un dato sea inviolable, inalterable cuando se le asigne un modificador de acceso.

Los getters y setters son muy importantes porque nos permiten trabajar con propiedades que son privadas, pero no de manera directa, sino que a través de un getter podemos validar ese dato que estamos recibiendo:D!

Algo genial es que en algunos IDE’s para Java como NetBeans te permiten establecer los getters y setters dando un solo click en una opción xD

# Generando polimorfismo en Java

**Polimorfismo**: Muchas formas. Poli = muchas, morfismo = formas. **NO** es Poliformismo

Es construir métodos con el mismo nombre pero con comportamiento diferente

# Generando polimorfismo en PHP

Ahora que ya entiendes cómo funciona el polimorfismo y lo aplicaste a Java veamos cómo hacer el mismo ejemplo en PHP.

Aplicar polimorfismo en PHP es relativamente más sencillo, pues este es un lenguaje con un tipado más flexible (que no requiere que especifiques qué tipo de dato usar) y su sintaxis es muy sencilla.

Hasta ahora esto es lo que tenemos en nuestro archivo car.php:

**require\_once**('account.php');

**class** **Car** {

**public** $id;

**public** $license;

**public** $driver;

**protected** $passenger; // Es importante que esta propiedad esté marcada como protected

**public** **function** **\_\_construct**($license, $driver){

$this->license = $license;

$this->driver = $driver;

}

**public** **function** **printDataCar**() {

**echo** "Licencia: $this->license Driver: ".$this->driver->name;

}

}

?>

Aún nos falta añadir nuestros getters y setters, en PHP es sencillo, simplemente tenemos que escribir dos funciones, una que nos devuelva nuestro número de pasajeros, y otra que nos permita establecer cierto número de pasajeros.

La función getter es la más sencilla de todas, pues simplemente hay que retornar el valor de la propiedad passenger:

**public** **function** **getPassenger**() {

**return** $this->passenger;

}

Recuerda que para hacer nuestra función setter necesitamos hacer una validación. Uber únicamente nos permite tener 4 pasajeros (comportamiento que modificaremos más adelante con polimorfismo), por lo que nuestra función setter quedaría así:

**public** **function** **setPassenger**($passenger) {

**if** ($passenger == 4) {

$this->passenger = $passenger;

}

**else** {

**echo** "Necesitas asignar 4 pasajeros";

}

}

¡Genial!, ahora ya estamos listos para empezar a trabajar, recuerda que ya deberías tener tu clase UberVan creada hasta este punto 😉.

Antes de comenzar a aplicar polimorfismo veamos qué sucede si intento generar una instancia de 6 pasajeros con mi clase UberVan, recuerda que para hacer estas pruebas estamos trabajando sobre nuestro archivo index.php (no olvides hacer el require hacia el archivo uberVan.php 👀):

$uberVan = **new** UberVan("OJL395", **new** Account("Raúl Ramírez", "AND456"), "Nissan", "Versa");

$uberVan->setPassenger(6);

$uberVan->printDataCar();

Cuando trato de ejecutar este código, esto es lo que me imprime en mi navegador:



Me dice que solo puedo asignar 4 pasajeros, ¡pero para Uber Van deben ser 6 pasajeros! Es hora de usar el polimorfismo.

## Aplicando polimorfismo

Para permitir que la clase UberVan acepte 6 pasajeros sin afectar a las otras clases de Uber, necesitamos sobreescribir el método setPassenger, pero permitiendo que esta vez acepte 6 pasajeros.

Recuerda que cuando aplicamos herencia, todos los métodos de la clase Car fueron pasados a la clase UberVan, por tanto, esta clase también tiene acceso al método setPassenger, pero este método solo permite 4 pasajeros. Para aplicar polimorfismo, podemos sobreescribir dicho método dentro de nuestra clase UberVan alojada en el archivo uberVan.php, ¡pero ahora permitiendo 6 pasajeros!

**public** **function** **setPassenger**($passenger) {

**if** ($passenger == 6) {

$this->passenger = $passenger;

}

**else** {

**echo** "Necesitas asignar 6 pasajeros   
";

}

}

Si lo deseas, también puedes añadir la propiedad $passenger a la clase UberVan (aunque en PHP no es necesario):

**protected** $passenger;

Para hacerlo más interesante, vamos a cambiar el código de nuestro método printDataCar de la clase Car ubicada en car.php por el siguiente:

**public** **function** **printDataCar**() {

**echo** "

Licencia: $this->license

Driver: {$this->driver->name}

Número de pasajeros: $this->passenger

";

}

El código es el mismo, pero lo hicimos más legible, agregamos la información del número de pasajeros y agregamos un salto de línea al final.

¡Ahora pongamos a prueba nuestra clase polimórfica! Vayamos a nuestro index.php, aquí ya teníamos escrito el código para instanciar nuestra clase UberVan con 6 pasajeros, pero agreguemos el código para instanciar una clase de 4 pasajeros:

$uberX = **new** UberX("CVB123", **new** Account("Andres Herrera", "AND456"), "Chevrolet", "Spark");

$uberX->setPassenger(4);

$uberX->printDataCar();

¡Vualá!, ahora vemos que nuestra clase UberVan nos permite añadir 6 pasajeros, mientras que el resto de clases nos permite seguir añadiendo 4 pasajeros, ¡hemos creado una clase polimórfica!



## Reto

Ahora que ya sabes cómo trabajar con polimorfismo en PHP, te reto a que apliques el polimorfismo al método printDataCar de la clase Car en la clase UberX de tal forma que al llamarlo desde una instancia de UberX este también imprima la marca y el modelo.

Una vez que hayas completado el reto puedes dejar tu solución en los aportes de la clase y compararlo con la solución de otros estudiantes. ¡Nos vemos en la siguiente clase para dar un repaso al diagrama UML de Uber que obtuvimos al final!