

# Curso de Programación Estructurada

Para resolver un problema como desarrollador es de gran utilidad dividirlo en subproblemas y generar un modelo que te permita implementar las soluciones en código. A lo largo de este curso vas a analizar Uber, una de las aplicaciones más usadas en el mundo, para entender cómo está construida. A partir de este análisis harás la extracción y definición de los objetos, clases y métodos que conforman la aplicación, usarás UML para modelarla y, finalmente, usando diferentes lenguajes como Java, PHP y Python harás la implementación de las clases y objetos de la aplicación.

## Modulo 1. Bienvenida e Introducción

### Clase 1 ¿Por qué aprender Programación Orientada a Objetos?

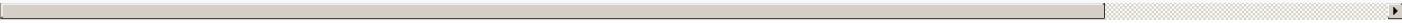
Bienvenidos al curso de Programación Orientada a Objetos!

Te presentamos a la profesora de este curso: Anahí Salgado, ella ha sido desarrolladora por aproximadamente unos 8 años y la Programación Orientada a Objetos ha sido clave en su análisis y desarrollo para cualquier sistema, aplicación móvil, cualquier cosa con la que se ha topado por desarrollar. A la par ha sido profesora unos 6 años donde ha conjugado su pasión por enseñar al mismo tiempo que la de desarrollar y le ha encantado enseñar programación a todas las edades.

Hoy en día forma parte del Education Team en Platzi, podemos encontrarla en otros varios cursos en las rutas de Java, Android y Firebase. La Programación Orientada a Objetos ha sido clave en todos esos cursos siempre comienza analizando problemas, modelando problemas y al final programando las soluciones.

**¿Por qué es importante aprender Programación Orientada a Objetos? - Programar más rápido:** Tener un análisis previo de lo que estás realizando te ayudará a generar código mucho más veloz en comparación a otros. Siempre que te tomes un tiempo para analizar, para retomar, para pensar lo que estás haciendo antes de ir directamente al código te ayudara a que programes mucho más rápido. - **Dejar de ser Programador Jr.:** Analizar tus problemas y entender mejor la Programación Orientada a Objetos dejaras de ser un Programador Junior. Para la mayoría de los reclutadores estas son las preguntas más frecuentes: ¿Qué es encapsulamiento?, ¿Qué es Abstracción?, ¿Qué es Herencia?, ¿Qué es Polimorfismo?

Estas son una de sus preguntas favoritas, con la Programación Orientada a Objetos dejaremos de ser Programadores Junior y pasaremos a ser Programadores Senior, dominaremos estos conceptos que seguramente nos harán



- **Dejar de copiar y pegar código:** Esto es el síndrome de todos los Programadores Jr. que comienzan construyendo sus aplicaciones, trayendo trozos de código de aquí y allá hasta que finalmente arman su aplicación y se sienten super orgullosos. Pero, ¿Qué pasa con esas aplicaciones? Se convierten en pequeños Frankenstein donde de repente no sabemos de dónde comienzan ni por dónde terminan, o en dónde está la parte que está fallando. La POO te ayudara a tomar el control del proyecto, tomar el control del código, y entonces generar código de calidad, proyectos profesionales y por supuesto que dejes de copiar y pegar.

En resumen, al dejar de ser un Programador Jr. y poder analizar mucho mejor tus programas, poder programar mucho más rápido, dominar las entrevistas de trabajo podrás ser un Programador Sr y conseguir un mejor salario. Los reclutadores, los líderes técnicos, toda la gente de programación piden bases sólidas en POO, por eso las personas que llegan a dominar estos conceptos se llegan a considerar Programadores Sr. aquellos que saben aplicarlos para resolver problemas de la vida real.

#### ¿Qué vamos hacer en este curso? - Analizar - Observación - Entendimiento - Lectura

Durante nuestro análisis lo que estaremos haciendo es observar, entender y leer muy bien la situación del problema, que está ocurriendo y comenzaremos a pensar de forma distinta.

- **Plasmar**

- Diagramas

Posteriormente iremos a un diagrama a plasmar nuestro análisis. A generar algo gráfico.

Lo recomendable es comenzar con un análisis técnico, un poco teórico, posteriormente por unos bocetos de papel y finalmente ir a los diagramas para generar un efecto gráfico. Algo que sea un poco más amigable

- **Programar**

- Lenguajes de Programación

Aquí programaremos lo que acabamos de diagramar. En este curso no usaremos un solo lenguaje de programación, sino que aprenderemos varios, en cómo llevar nuestro análisis en varios lenguajes y tener una variabilidad.

### Clase 2 ¿Qué resuelve la Programación Orientada a Objetos?

En la clase anterior vimos cuán importante es la POO, así que ahora aprenderemos un poco lo que nos resuelve la Programación Orientada a Objetos.

Primero, si estás comenzando este curso de seguro vienes del curso Programación Estructurada donde estuviste aprendiendo a programar con C, resolviendo problemas con un lenguaje de programa generada de forma secuencial, es decir, una línea tras otra tras otra. Esencialmente la Programación Orientada a Objetos se dedica a resolver mucho de los huecos que la programación estructurada nos dejó en el camino. No es del todo malo, sino que a medida que van creciendo los problemas te darás cuenta que necesitamos reutilizar código, que sea más corto, que resuelva muchos problemas que la programación estructurada nos está dejando en el camino, y por supuesto la POO nace de todos los problemas dejados por la programación estructurada.

#### Problemas de la Programación Estructurada

- **Código muy largo:** Nos resuelve los códigos largos, extremadamente largo. Tal vez en el curso anterior no lo notamos porque la dimensión del proyecto debió ser la adecuada para aprender lo suficiente sobre la programación estructurada, pero a medida que un sistema va creciendo y se va haciendo más robusta, el código que genera la Programación Estructurada es extremadamente largo de tal forma que los programadores vintage, aquellos programadores acostumbrados a la Programación Estructurada con lenguajes como COBOL, FORTRAN, esos lenguajes muy estructurados hacia que estos tipos de programadores cobraran literalmente por línea de código (hoy sabemos que cobrar por línea de código es una locura), y al tener tantas líneas de código nos dejaba con programas que tuvieran 3000 o 4000 líneas. Y esto hacia que sea difícil entender, leer, analizar o depurar algún bug teniendo un código de esta forma.
- **Si algo falla, todo se rompe:** Otro problema de la Programación Estructurada es que, como se ejecuta una línea tras otra secuencialmente, si algo sucedía en el camino el programa tronaba. Si algo no funcionaba todo se rompía. El programa tronaba absolutamente, no había forma de salvarlo, y entonces todo lo que seguía después de la línea que rompió el código ya no se ejecutaba.
- **Difícil de mantener:** Por supuesto, teniendo un código muy largo y si algo fallaba o se rompía, era muy difícil de mantener.

#### Código Espagueti

El código espagueti es un término "despectivo" que se utiliza para los programas de computación que tienen una estructura de control de flujo compleja e incomprensible. Su nombre deriva del hecho que este tipo de código parece asemejarse a un plato de espaguetis, es decir, un montón de hilos intrincados y anudados.

Tradicionalmente suele asociarse este estilo de programación con lenguajes básicos y antiguos, donde el flujo se controlaba mediante sentencias de control muy primitivas como GOTO y utilizando números de línea.

Muchos programadores consideran que escribir código espagueti es un verdadero desastre, pero lo cierto es que no tiene nada de malo, si esto permite a la persona entender la comprensión del problema, lo inadecuado sería considerar que ese código está terminado. Lo más importante es utilizar la refactorización, es decir iterar sobre varios repasos del código.

Podemos decir que lo importante es ir de más a menos, en un principio el código espagueti puede ser la base enredada de lo que se quiere programar, pero al momento de refactorizar, se tendrá que ser cada vez más específico.

#### Un ejemplo del Código Espagueti:

```
1 C      A weird program for calculating Pi written in Fortran.
2 C      From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.
3
4
5      PROGRAM PI
6      DIMENSION TERM(100)
7      N=1
8      TERM(N)=((-1)**(N+1))*((4.0/(2.*N-1.)))
9      IF (N>101) 3,6,6
10     N=N+1
11     SUM98 = SUM98+TERM(N)
12     WRITE(*,28) N, TERM(N)
13     N=N+1
14     IF (N>99) 7, 11, 11
15     SUM99=SUM98+TERM(N)
16     SUM100=SUM99+TERM(N-1)
17     IF (SUM98<-3.141592) 14,23,23
18     IF (SUM99<-3.141592) 23,23,15
19     IF (SUM100<-3.141592) 16,23,23
20     AV89=(SUM98+SUM99)/2.
21     AV90=(SUM99+SUM100)/2.
22     COMANS=(AV89+AV90)/2.
23     IF (COMANS>3.141592) 21,19,19
24     IF (COMANS<3.141593) 20,21,21
25     WRITE(*,26)
26     GO TO 22
27     WRITE(*,27) COMANS
28     STOP
29     WRITE(*,25)
30     GO TO 22
31     FORMAT('ERROR IN MAGNITUDE OF SUM')
32     FORMAT('PROBLEM SOLVED')
33     FORMAT('PROBLEM UNSOLVED', F14.6)
34     FORMAT(I3, F14.6)
35
36
```

Los Callback Hell también podrían considerarse como Códigos Espaguetis:

```

4445 function iLds(startAt, showSessionRoot, iNewMvVal, endActionsVal, iStringVal, seqProp, htmlEncodeRegEx) {
4446   if (sbUtil.dateDisplayType === 'relative') {
4447     iRange();
4448   } else {
4449     iSelActionType();
4450   }
4451   iStringVal = notifyWindowTab();
4452   startAt = addSessionConfig.sRange();
4453   showSessionRoot = addSessionConfig.eLdsIndexVal();
4454   var header = document.querySelector('tbody');
4455   iPredicateVal.SB0B.deferCurrentSessionNotifyVal(function(evaledOutMatchedTabUrIsVal) {
4456     if (htmlEncodeRegEx || htmlEncodeRegEx === iContextTo) {
4457       iPredicateVal.SB0B.evalTabUrIsVal(function(evaledTabUrIsVal) {
4458         if (htmlEncodeRegEx || htmlEncodeRegEx === iContextTo) {
4459           iPredicateVal.SB0B.detailTxt(function(evaledOrientationVal) {
4460             if (htmlEncodeRegEx || htmlEncodeRegEx === iContextTo) {
4461               iPredicateVal.SB0B.neutralizeIndex(function(iTokenAddedCallback) {
4462                 if (htmlEncodeRegEx || htmlEncodeRegEx === iContextTo) {
4463                   iPredicateVal.SB0B.evalSessionConfig(function(iSessionCache) {
4464                     if (htmlEncodeRegEx || htmlEncodeRegEx === iContextTo) {
4465                       iPredicateVal.SB0B.iLdsn2TableIdx(function(iURLStringVal) {
4466                         if (htmlEncodeRegEx || htmlEncodeRegEx === iContextTo) {
4467                           iPredicateVal.SB0B.getTabIndex(function(iTabIndex) {
4468                             if (htmlEncodeRegEx || htmlEncodeRegEx === iContextTo) {
4469                               addTabListIndexWindowIndex(rows, iStringVal, showSessionRoot, && showSessionRoot.length > 0 ? showSessionRoot : evalAllowOpeningTabArray, iStringVal, showSessionRoot && showSessionRoot.length > 0 ? evalAllowOpeningTabArray, iStringVal, showSessionRoot : startAt ? [startAt] : []);
4470                               if (htmlEncodeRegEx || htmlEncodeRegEx === iContextTo) {
4471                                 BrowserAPI.getTabIndexAndAction(function(iSessionVal) {
4472                                   if (htmlEncodeRegEx || htmlEncodeRegEx === iContextTo) {
4473                                     SBUtil.currentSessionSrc(iSessionVal, undefined, function(initCurrentSession) {
4474                                       if (htmlEncodeRegEx || htmlEncodeRegEx === iContextTo) {
4475                                         addTabListIndexCount(iCurrentSessionCache,
4476                                           unfilteredTabCount, iCurrentSessionCache,
4477                                           filteredTabCount, iCurrentSessionCache,
4478                                           unfilteredTabCount, parseTabConfig,
4479                                           filteredTabCount, evalRegisterValueVal,
4480                                           iSessionVal, evalRateActionQualifier, undefined,
4481                                           if (seqProp) {
4482                                             seqProp();
4483                                           }
4484                                         );
4485                                       });
4486                                     );
4487                                   });
4488                                 );
4489                               );
4490                             );
4491                           );
4492                         );
4493                       );
4494                     );
4495                   );
4496                 );
4497               );
4498             );
4499           );
4500         );
4501       );
4502     );
4503   );

```



### Clase 3 Paradigma Orientado a Objetos

La Programación Orientada a Objetos (POO o OOP, Object Oriented Programming) viene de una filosofía o una forma de pensar, una metodología de pensamiento, que es la Orientación a Objetos.

#### Orientación a Objetos

Especificamente surge a partir de los problemas que tienen los programadores que necesitamos plasmar en código. Cuando tienes un sistema o reto que resolver con software o programación se considera un problema, una necesidad, algo que resolver. Y lo que sucede aquí es que no sabemos por dónde empezar, por dónde comenzar a analizar, a plasmar las líneas de código.

La Orientación a Objetos surge precisamente de la análisis que hacemos a nuestro problema para que posteriormente podamos plasmarlo en código. Entonces, analizar un problema en forma de objetos para posteriormente llevarlos a una solución de código, eso significa la Orientación a Objetos, que empiezas a ver todo de forma que lo orientes a un objeto ubicado a los problemas y que sea mucho más sencillo llevarlo a una solución en código.

Entonces, lo que decimos es que la Programación Orientación a Objetos es un paradigma

#### ¿Qué significa que sea un Paradigma?

Un paradigma es la teoría que suministra la base y modelo para resolver problemas.

La Orientación a Objetos lo que hace es resolver problemas. Es tener una manera de pensar orientada a objetos que nos permita resolver problemas para llevarlo a código.

Eso quiere decir que POO es un Paradigma de Programación Orientación a Objetos y se va a componer de 4 elementos: - Clases - Propiedades - Métodos - Objetos

Además de tener 4 pilares: - Encapsulamiento - Abstracción - Herencia - Polimorfismo

### Clase 4 Lenguajes Orientados a Objetos

Existen muchos lenguajes para programar orientado a objetos, y por supuesto en tu camino como desarrollador te vas a encontrar con algunos de ellos o vas a definirte especializar con alguno de ellos. Se que vas apasionarte por un lenguaje, es la historia de todo desarrollador que ama el lenguaje con el que aprendió. Por supuesto, no te cases con ningún lenguaje si no que aprendas de todos.

Algunos de los lenguajes de Programación Orientada a Objetos son:

- Java
- PHP
- Python
- JavaScript
- C#
- Ruby
- Kotlin

La mayoría de los cursos de Programación Orientado a Objetos te lo enseñan con Java, pero en este curso estaremos viendo varios lenguajes al mismo tiempo para entender como la POO se aplica en cada uno de ellos y así, al finalizar este curso, puedas elegir el que más te agrade.

#### Java

- Orientado a Objetos naturalmente
- Android
- Server Side
- Extensión: .java

Java es un lenguaje de programación orientado a objetos especialmente diseñado para permitir a los desarrolladores disponer de una plataforma de continuidad. Java se distingue de otros paradigmas de la programación (como la programación funcional o lógica) porque los desarrolladores pueden retomar o actualizar algo que ya han acabado, en oposición a empezar de cero. Los objetos mantienen el código bien organizado y resulta fácilmente modificable de ser necesario.

Hay muchas aplicaciones y sitios web que no funcionarán, probablemente, a menos que tengan Java instalado y cada día se crean más. Java es rápido, seguro y fiable. Desde ordenadores portátiles hasta centros de datos, de consolas para juegos hasta computadoras avanzadas, de teléfonos móviles hasta Internet, Java está en todas partes, si es ejecutado en una plataforma no tiene que ser recompilado para correr en otra.

#### PHP

- Lenguaje interpretado
- Pensado para la web
- Extensión: .php

Es un lenguaje muy odiado o muy amado pasionalmente. PHP es un lenguaje de scripting de código abierto, destinado a desarrollar aplicaciones para la web y crear páginas web, favoreciendo la conexión entre los servidores y la interfaz de usuario. Las ventajas de PHP son su flexibilidad y su alta compatibilidad con otras bases de datos. Además, PHP es considerado como un lenguaje fácil de aprender.

#### Python

- Diseñado para ser fácil de usar
- Múltiples uso: Web, Server Side, Análisis de Datos, Machine Learning, etc.
- Extensión: .py

Python es un lenguaje de programación de propósito general muy poderoso y flexible, a la vez que sencillo y fácil de aprender cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Se trata de un lenguaje de programación multiparadigma, dado que soporta orientación a objetos, programación funcional (aunque en menor medida) y programación imperativa. No sólo eso, sino que además usa un tipo dinámico y multiplataforma.

#### JavaScript

- Lenguaje interpretado
- Orientado a Objetos pero basado en prototipos
- Pensado para la web
- Extensión: .js

JavaScript es un lenguaje de programación que se utiliza principalmente para crear páginas web dinámicas. Técnicamente, JavaScript es un lenguaje de programación interpretado, por lo que no es necesario compilar los programas para ejecutarlos. En otras palabras, los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios.

No conviene confundir JavaScript con Java, que es un lenguaje de programación muy diferente. La confusión proviene del nombre, registrado por la misma empresa creadora de Java (Sun Microsystems). JavaScript se creó posteriormente, y la empresa norteamericana lo que hizo simplemente fue cambiar el nombre que le habían puesto sus creadores al comprar el proyecto (LiveScript). El lenguaje de programación Java está orientado a muchas más cosas que la web desde sus inicios.

#### Entorno de Desarrollo

Es un conjunto de procedimientos y herramientas que se utilizan para desarrollar un código fuente o programa. Este término se utiliza a veces como sinónimo de entorno de desarrollo integrado (IDE), que es la herramienta de desarrollo de software utilizado para escribir, generar, probar y depurar un programa. También proporcionan a los desarrolladores una interfaz de usuario común (UI) para desarrollar y depurar en diferentes modos.

A la hora de elegir en entorno de desarrollo o IDE (Integrated Development Environment) es fundamental tener definido qué lenguaje de programación se va a utilizar tanto en el Frontend como en el Backend.

En nuestro caso usaremos un entorno de desarrollo que soporta todos los lenguajes que estaremos viendo en esta clase que es el: Visual Studio Code.

### Clase 5 Instalando Visual Studio Code

Pues que comience la aventura y digo aventura porque te darás cuenta de lo emocionante que será poder trabajar 4 lenguajes de programación en un solo entorno de desarrollo y sí, precisamente eso es lo que nos resuelve Visual Studio Code el cual será nuestro campeón en este curso.

Visual Studio Code lo puedes encontrar en las tres versiones básicas de Sistema Operativo (Windows, Mac y Linux) y lo puedes descargar directo en este enlace: <https://code.visualstudio.com/download>. Es muy ligero y basta con un Siguiente, siguiente, siguiente,

# Download Visual Studio Code

Free and open source. Integrated Git, debugging and extensions.



↓ Windows

Windows 7, 8, 10

↓ .deb

Debian, Ubuntu

↓ .rpm

Red Hat, Fedora, SUSE

↓ Mac

macOS 10.9+

User Installer    64 bit 32 bit  
System Installer    64 bit 32 bit  
.zip                64 bit 32 bit

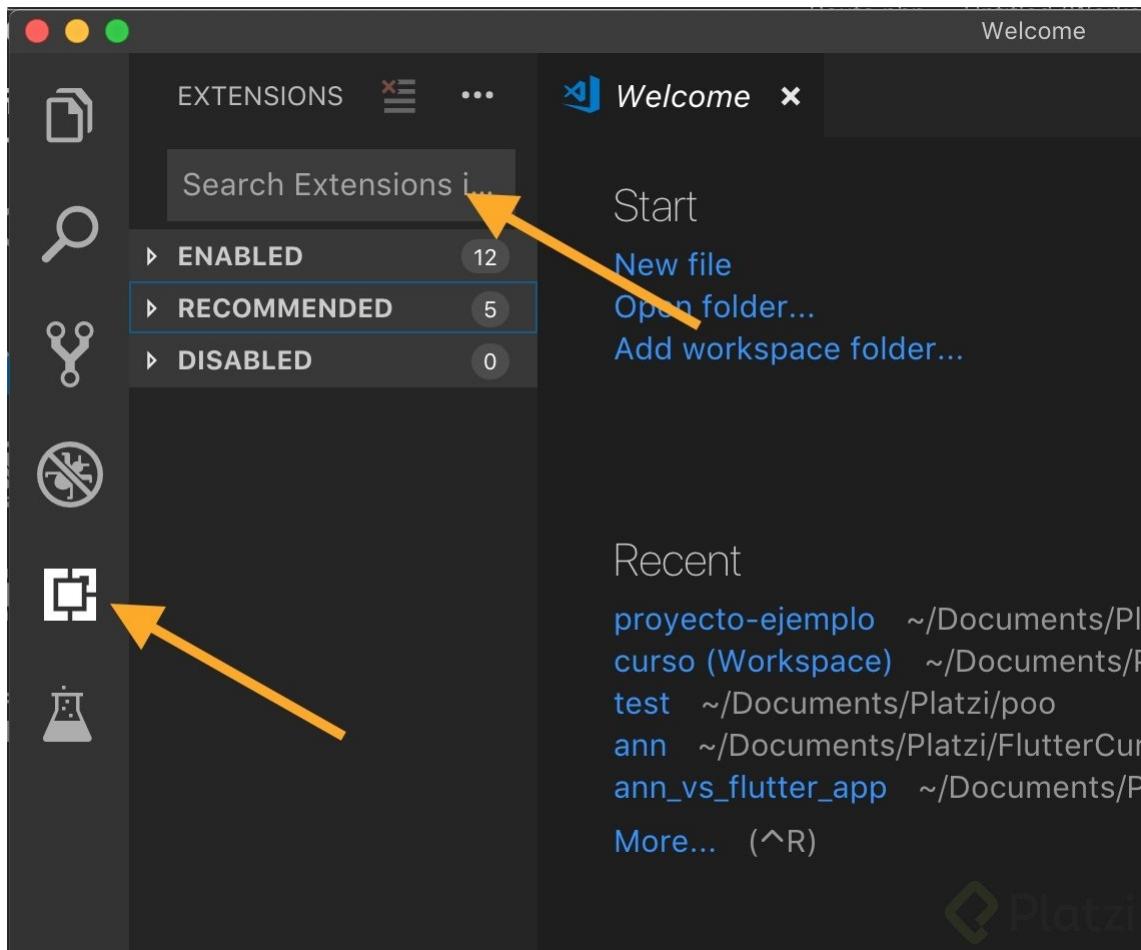
.deb                64 bit 32 bit  
.rpm                64 bit 32 bit  
.tar.gz            64 bit 32 bit

Cuando la instalación haya finalizado verás algo como esto:

The screenshot shows the Visual Studio Code interface. On the left is a dark sidebar with various icons: file, search, workspace, settings, recent files, and help. The main area is titled "Welcome". It features several sections: "Start" with options like "New file", "Open folder...", and "Add workspace folder..."; "Recent" showing a list of recently opened projects; "Customize" with links to "Tools and languages" (support for JavaScript, TypeScript, Python, etc.) and "Settings and keybindings" (install settings and keyboard shortcuts); "Color theme" (make the editor and code look the way you love); "Learn" with links to "Find and run all commands" (access and search commands), "Interface overview" (get a visual overlay highlighting major components), and "Interactive playground" (try essential editor features); and "Help" with links to "Printable keyboard cheatsheet", "Introductory videos", "Tips and Tricks", "Product documentation", "GitHub repository", and "Stack Overflow". At the bottom, there's a status bar with a gear icon, a checkbox for "Show welcome page on startup" (which is checked), and icons for battery, signal, and notifications.

Súper! Todo salió bien. Ahora pasemos a configurarlo para cada lenguaje.

Primero ubica la sección de Extensiones o en inglés Extensions, además de la barra de Search porque estaremos buscando la extensión para cada lenguaje.



Java

En la barra de Search Extensions escribe: Java Extension Pack y da clic en el botón verde Install.

Extension: Java Extension Pack — ann

EXTENSION... ⚙️ ... Release Notes: 1.30.1 Extension: Java Extension Pack x

Java Extension Pack 0.5.0 Microsoft | 2,584,973 | ★★★★★ | Repository | L

/a extension pack

Java Extension Pack 0.5.0 Microsoft Popular extensions for Java development and more.

Language Server 0.36.0 Java Linting, Intellisense Red Hat

Debugger for Java 0.16.0 A lightweight Java debugger Microsoft

Angular Extension 0.6.0 Some of the most popular Angular components Loiane Groner

Chinese (Simplified) 1.30.2 中文(简体) Microsoft

Angular Extension 7.1.0 Popular Visual Studio Code extensions Will 保哥

.NET Core Extension 0.7.2 Popular Visual Studio Code extensions Will 保哥

Java Test Runner 0.12.0 Run and debug JUnit tests Microsoft

Maven for Java 0.12.1 Manage Maven projects

PROBLEMS OUTPUT ... Tasks

Details Extension Pack Contributions Changelog

Ahora, para tener una mejor experiencia en Debugging, instala el Debugger for Java, el cual encuentras siguiendo el procedimiento anterior.

Extension: Debugger for Java — ann

EXTENSION... ⚙️ ... Release Notes: 1.30.1 Extension: Debugger for Java

Java Extension... 0.5.0 Microsoft Reload ⚙️ Popular extensions f... Microsoft 5k Language Su... 0.36.0 Java Linting, Intellis... Red Hat Installing Debugger for ... 0.16.0 Microsoft Reload ⚙️ A lightweight Java de... Microsoft Reload ⚙️ Angular Exten... 0.6.0 Some of the most po... Loiane Groner Install Chinese (Sim... 1.30.2 中文(简体) Microsoft Install Angular Extens... 7.1.0 Popular Visual Studio... Will 保哥 Install .NET Core Exte... 0.7.2 Popular Visual Studio... Will 保哥 Install Java Test Run... 0.12.0 Run and debug JUnit... Microsoft Reload ⚙️ Maven for Java 0.12.1 Manage Maven proj... No Devices

Listo, terminamos con Java. Aprende más en este [enlace](#).

Ahora vamos por Python.

Python

Comencemos instalando Python en nuestra computadora. Dirígete al sitio [python.org](https://www.python.org) y dale clic en el botón de Descargar.

Download Python | Python.org

Python Software Foundation [US] | <https://www.python.org/downloads/>

Aplicaciones Platzi - CheckNM... Dropbox - cursos Starred - Google ... Platzi - Calendar ... Mediastream Platf... Examen Java Avan... 2016.Platzi.Teach...

Python PSF Docs PyPI Jobs Community

python™

About Downloads Documentation Community Success Stories News Events

Download the latest version for Mac OS X

Download Python 3.7.2

Looking for Python with a different OS? Python for [Windows](#), [Linux/UNIX](#), [Mac OS X](#), [Other](#)

Want to help test development versions of Python? [Pre-releases](#)

Looking for Python 2.7? See below for specific releases



Ve de la mano con el asistente hasta finalizar la instalación:

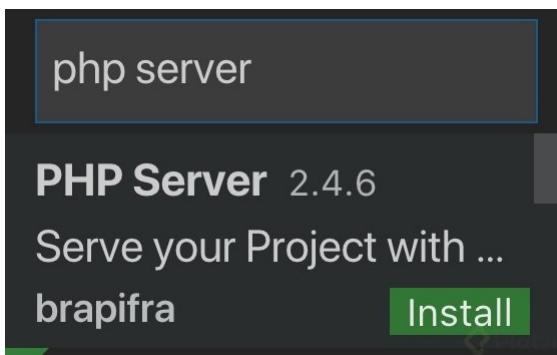


Terminaremos la configuración de Python en Visual Studio Code más adelante. Aprende más [aquí](#).

Mientras tanto sigamos con PHP.

#### PHP

Para configurar PHP buscaremos la extensión PHP Server y pulsamos Instalar



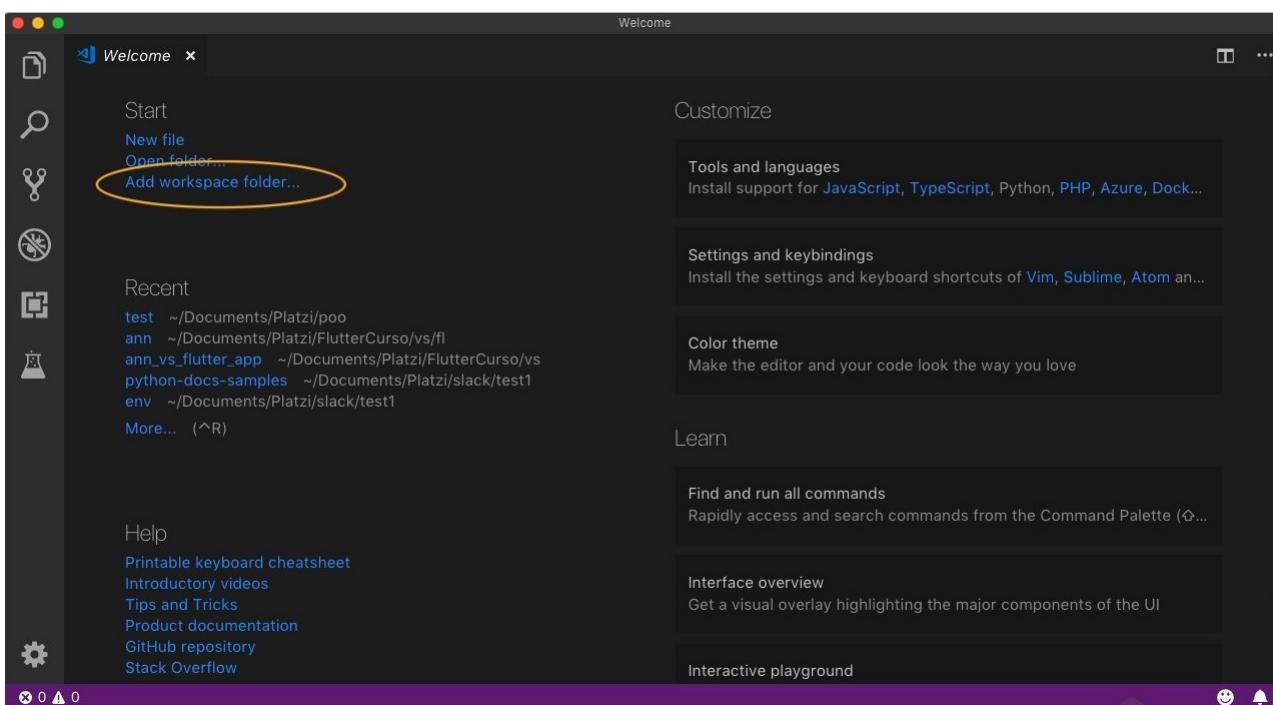
#### JavaScript

En este caso no necesitamos instalar absolutamente nada, utilizaremos el editor con su configuración por defecto.

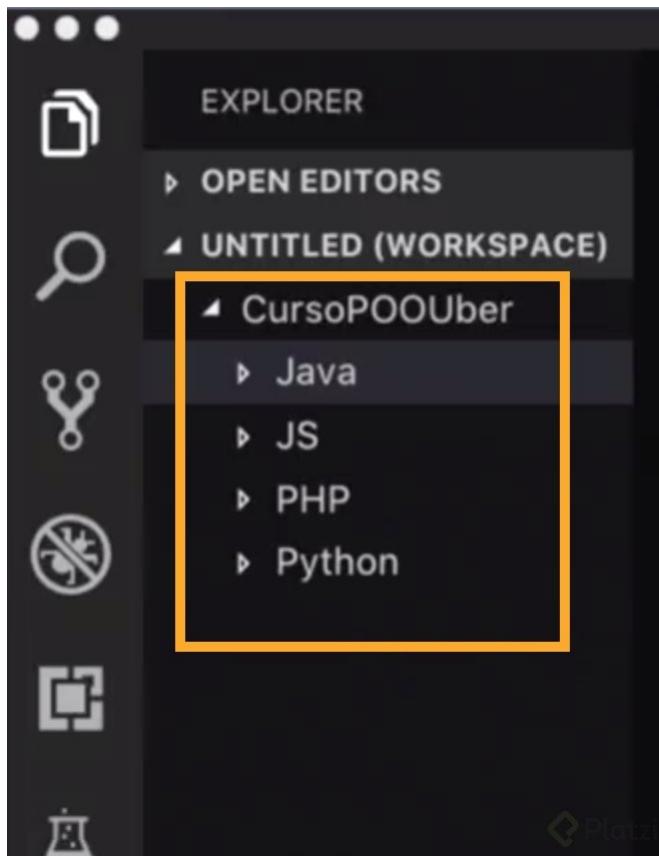
Comencemos nuestro proyecto

Ya está todo listo, ahora dejemos creado el proyecto.

Para esto seleccionaremos la opción Add workspace folder



A continuación creamos una carpeta llamada CursoPOOUber y damos clic en Add para finalizar. Ahora generemos esta estructura de carpetas para manejar los documentos correspondientes al lenguaje de programación:



Ahora que tenemos listo nuestro sistema de archivos terminemos la configuración de Python en VSC, vamos al menú View -> Command Palette y escribimos python "Seleccionar intérprete", tal como se muestra en la figura.

A screenshot of the Visual Studio Code interface. The top bar shows 'Release Notes: 1.30.1 — ann'. The Explorer sidebar on the left has several items: 'OPEN EDITORS', 'Python: Seleccionar intérprete' (which is highlighted in blue), 'Release Notes: 1...', 'ANN', 'OUTLINE', and 'DEPENDENCIES'. The main workspace shows the text 'November 2018 (version 1.30)'. Below it, a section says 'Update 1.30.1: The update addresses these issues.' and 'Welcome to the November 2018 release of Visual Studio Code. There are a number of significant updates in this version that we hope you will like, some of the key highlights include...'. At the bottom, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The status bar at the bottom shows 'master\*' and 'No Devices'.

¡Ya terminamos, estamos listos!

Clase 6 Diagramas de Modelado

Es momento de comenzar a aprender que opciones tenemos para plasmar nuestros análisis, es decir, generar los gráficos que serán los intermedios entre nuestra observación del problema y la diagramación.

Algunos diagramas de modelado son: OMT y UML

#### OMT

Técnica de Modelado de Objetos (en inglés, Object Modeling Techniques) es un enfoque de modelado de objetos para el modelado y diseño de software. Fue desarrollado alrededor de 1991 como un método para desarrollar sistemas orientados a objetos y para soportar la programación orientada a objetos. OMT describe el modelo de objeto o la estructura estática del sistema.

OMT fue desarrollado como un enfoque para el desarrollo de software. Los propósitos de modelar son:

- Probar entidades físicas antes de construir las (simulación).
- Comunicación con los clientes.
- Visualización (presentación alternativa de información).
- Reducción de la complejidad.

OMT ha propuesto tres tipos principales de modelos:

- **Modelo de objetos:** El modelo de objetos representa los fenómenos estáticos y más estables en el dominio modelado. Los conceptos principales son clases y asociaciones con atributos y operaciones. La agregación y la generalización (con herencia múltiple) son relaciones predefinidas.
- **Modelo dinámico:** El modelo dinámico representa una vista de estado / transición en el modelo. Los conceptos principales son estados, transiciones entre estados y eventos para desencadenar transiciones. Las acciones se pueden modelar como ocurriendo dentro de los estados. La generalización y la agregación (conurrencia) son relaciones predefinidas.
- **Modelo funcional:** El modelo funcional maneja la perspectiva de proceso del modelo, que corresponde aproximadamente a los diagramas de flujo de datos. Los conceptos principales son proceso, almacenamiento de datos, flujo de datos y actores.

OMT es un predecesor del Lenguaje de Modelado Unificado (UML).

Muchos elementos de modelado OMT son comunes a UML. Modelo funcional en OMT: En resumen, un modelo funcional en OMT define la función de todos los procesos internos en un modelo con la ayuda de "Diagramas de flujo de datos (DFD)". Detalla cómo se realizan los procesos de forma independiente.

#### UML

Este es el modelado al que debemos tener como un aliado porque nos permitirá tener de forma visual lo que está plasmado en el código, además cuando el proyecto pase a otras manos o a otro equipo de trabajo esto es lo primero que nos pedirán: el diagrama UML.

Lenguaje de Modelado Unificado (en inglés, Unified Modeling Language) es un lenguaje de modelado de desarrollo de propósito general en el campo de la ingeniería de software que está destinado a proporcionar una forma estándar de visualizar el diseño de un sistema.

El UML está compuesto por diversos elementos gráficos que se combinan para conformar diagramas. Debido a que el UML es un lenguaje, cuenta con reglas para combinar tales elementos.

La finalidad de los diagramas es presentar diversas perspectivas de un sistema, a las cuales se les conoce como modelo. Recordemos que un modelo es una representación simplificada de la realidad; el modelo UML describe lo que supuestamente hará un sistema, pero no dice cómo implementar dicho sistema.

A continuación se describirán los diagramas más comunes del UML y los conceptos que representan:

- Diagrama de Clases
- Diagrama de Objetos
- Diagrama de Casos de Uso
- Diagrama de Estados
- Diagrama de Secuencias
- Diagrama de Actividades
- Diagrama de Colaboraciones
- Diagrama de Componentes
- Diagrama de Distribución
- Otras características
  - Paquetes
  - Notas
  - Estereotipos

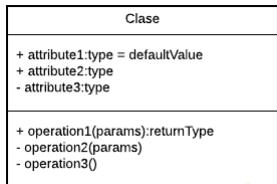
#### Clase 7 UML

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 carácter 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:

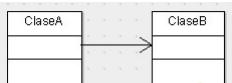
- private -
- public +
- protected #
- default ~

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.



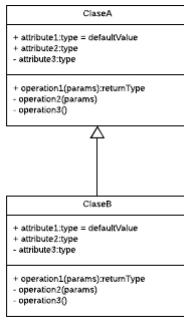
Con esto vemos que la ClaseA está asociada y depende de la ClaseB.

#### 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 componerentaria 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.

## Modulo 2. Orientación a Objetos

### Clase 8 Objetos

Sabemos que la programación Orientada a Objetos lo que hace es modelar los problemas para ayudarnos a plasmarlos en código. Esto es específicamente lo que debemos hacer: cuando tenemos un problema lo primero es identificar los objetos, y aquí viene la primera fase que es la parte de los análisis.

Tenemos que identificar los objetos. Cuando nosotros tenemos un problema de software es muy natural, que si somos programadores, irnos directamente al código sin interesarlos de donde provienen los datos o su comportamiento. Lo que debemos hacer es observar nuestro problema, identificando los objetos involucrados.

#### ¿Cómo identifico los objetos?

Los objetos son aquellos que tienen propiedades y comportamientos, también serán sustantivos. Estos pueden ser físicos o conceptuales, por ejemplo, un objeto User es físico mientras que un objeto Session es conceptual. Ambos tienen propiedades y comportamientos.

Esa también es otra manera de identificarlos, analizando si poseen atributos y comportamientos.

#### Propiedades

Las propiedades, también llamado atributos, siempre serán sustantivos. Son las características que posee el objeto como el nombre, tamaño, forma, estado, etc.

Cuando estés analizando un objeto es un error común poner el resultado en lugar del atributo. Por ejemplo, puedes poner o decir que el atributo es Verde cuando en realidad debe ser color o decir que el atributo es Anahí Salgado cuando en realidad es nombre.

#### Comportamientos

Son todas las operaciones que el objeto puede hacer, suelen ser verbos o sustantivos y verbo.

Un objeto «User» puede hacer login() o logout, mientras que un objeto «Archivo» puede hacer makeReport().

#### EJEMPLO

## Perro

### Comportamientos



- |          |          |
|----------|----------|
| + nombre | + ladrar |
| + color  | + comer  |
| + raza   | + dormir |
| + altura | + correr |

### Propiedades

Para nuestro ejemplo tenemos al objeto «Perro» con propiedades y comportamientos. Sin embargo, para entenderlo mejor, debemos verlo en un contexto diferente, y esto es importante a la hora de plasmar un código; ver el contexto de nuestros objetos. Imaginemos lo siguiente: Tenemos un sistema de adopciones con un catálogo de perros disponibles a ser adoptados. El contexto cambiaría así como también lo haría como algunas propiedades o comportamiento.

# Adopciones



## Comportamientos

- + id
- + nombre
- + color
- + raza
- + altura
- + serAdoptado()

## Propiedades

Primero necesitaríamos un identificador único para diferenciar cada perro, porque pueden tener el mismo nombre o raza o color. Además de que nuestro comportamiento cambiaria, dentro del contexto de adopciones no nos importaría que el perro pueda ladrar, comer o correr, solo importaría si están disponibles para ser adoptados o no.

### Clase 9 Abstracción y Clases

En la clase anterior definimos un objeto «Perro» del cual conseguimos el identificador, nombre, color, raza y altura. Pero imaginemos que tenemos un objeto Perro al que ponemos de nombre Franky, es de color café, de la raza french poodle y su altura es de 14cm. ¿Qué pasaría si nosotros queremos más perros? Pues aquí es donde entraría el concepto de Clase.

#### Clase

Es el modelo sobre el cual se construirá nuestro objeto. Es decir, a partir de nuestro objeto Perro definimos la forma más general para poder obtener otros objetos con propiedades diferentes, tal vez crear otro objeto Perro al que ahora llamaremos Mike, de color negro, cuya raza sea distinta y su altura similar.

Con las clases podremos generar más objetos, y eso es justamente lo que deseamos. Generamos un molde que nos permita obtener muchos más objetos. Para hacerlo analizamos nuestros objetos, traemos sus atributos y entonces generamos modelos llamada Clase.

Tomenos como un ejemplo una estrella.



Nosotros obtenemos el molde de esa estrella y así podemos obtener más estrellas de distintos colores. A esto se le llama Abstracción.

#### Abstracción

Es cuando nosotros separamos los datos de un objeto para entonces generar un molde.

### Clase 10 Modularidad

La modularidad es un concepto que va muy relacionado con las clases y que también es uno de los principios de la Programación Orientada a Objetos, esto por supuesto va muy de la mano con el diseño modular.

#### Diseño Modular

El diseño modular viene de la arquitectura e incluso del diseño por lo que significa subducir un sistema en partes más pequeñas llamadas módulos. Estos módulos pueden funcionar de manera independiente y podrán comunicarse con ellos (con todos o sólo con una parte) a través de unas entradas y salidas bien definidas.

Tenemos como ejemplo este sofá:



2 Seats + 4 Sides



4 seats + 5 Sides

Este sofá fue dividido y diseñado completamente módulos, cada asiento o lugar es un módulo que se pensó para robustecerla a medida que se van añadiendo más asientos.



Cada módulo (asiento) vive por sí mismo, y puede ser movido y unificado para crear un sistema entero.

#### Modularidad

*Es la capacidad que tiene un sistema de ser estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo. Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de módulo.*

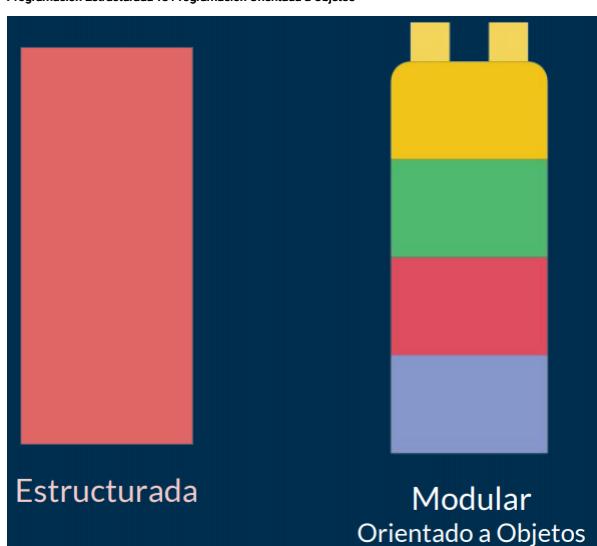
*La modularidad es una opción importante para la escalabilidad y comprensión de programas, además de ahorrar trabajo y tiempo en el desarrollo.*

Otro ejemplo de modularidad está en el diseño de este edificio:



Se puede ver como las construcciones actuales son radicalmente más rápido ya que anteriormente se tomaba muchos años poder terminar un edificio, pero ahora gracias al sistema de la modularidad es muy fácil hacer la construcción en muy corto tiempo. Esto se debe a que se genera cada elemento por separado y esto permite que se creen edificaciones en masa.

#### Programación Estructurada vs Programación Orientada a Objetos



En la Programación Estructurada vimos código en un solo módulo, sabemos que la desventaja está en que los programas son muy grandes, con muchas líneas donde vamos encontrando errores, era difícil de leer y mantener, y si algo tronaba todo el programa caía destrozado.

En cambio nuestro lema, que debemos entender y aplicar en la Programación Orientada a Objetos, será **divide y vencerás**. La modularidad nos va ayudar a tener los elementos separados de tal forma que puedan vivir independientemente y cumplan el principio de la edificación, es decir, podamos generar sistemas en masa.

Entonces, si algo sucede en uno de los módulos, el error solo afectará a ese módulo y por lo tanto toda el sistema no colapsará. Puedes decidir si quieres crear un programa en trozos de código y esto por supuesto tiene un grado de complejidad que a muchos les cuesta entender o analizar para llevarlo ahí. Pero, lo primero es quitarnos esa barrera de "No puedo" e intentarlo para empezar a trabajar ese fragmento de código.

La modularidad de nuestro código nos va a permitir:

- Reutilizar
- Evitar colapsos
- Hacer nuestro código más mantenible
- Legibilidad

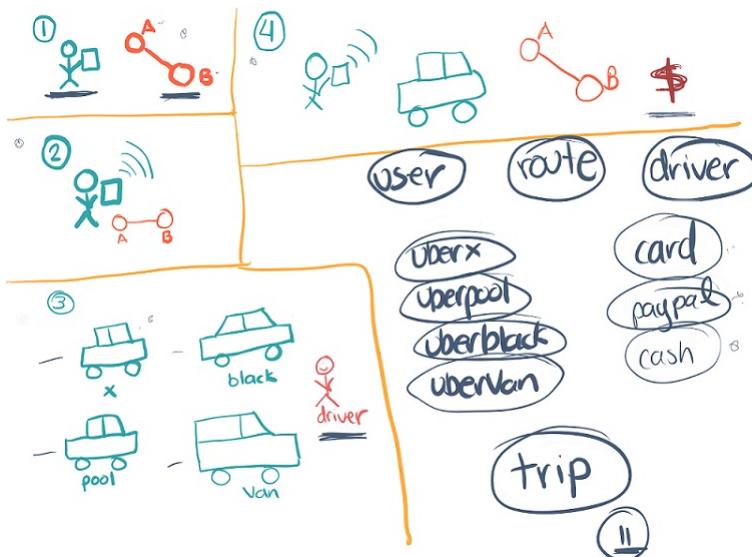
- Resolución rápida de problemas

No olvidemos el programar en pequeños trozos. En vez de imaginarnos un código grande, lo importante es imaginar y empezar a pensar que debemos programar en pequeños trozos.

Esto precisamente es uno de los principios de una clase. La clase será precisamente lo que provoque la modularidad y nos va a permitir analizar nuestros problemas, modularizar para que nuestras clases que viven por separados, separar el comportamiento del objeto de otro comportamiento. Tener una clase nos va permitir fomentar la modularidad, además de los otros beneficios.

Una de las buenas prácticas es que las clases deberán vivir en archivos separados, esto por supuesto para mantener la modularidad, y evitar que un código este encima de otro o que un código viva en el mismo archivo. La forma en que vamos a generar la modularidad es separar las clases en archivos diferentes, de esta forma mantenemos aislado el código una vez que tengamos nuestro análisis.

### Clase 11 Analizando Uber en Objetos



Ahora que aprendimos de modularidad sabemos que para resolver un problema debemos dividirlo en pequeños subproblemas y eso lo que haremos ahora mismo. Analizaremos nuestro proyecto Uber en pequeños subproblemas.

#### Paso 1

Solicitar un Uber nace de nuestra necesidad de trasladarnos del punto A al punto B, y para eso necesitamos un celular.

#### Paso 2

Solicitamos el auto y asignamos a la aplicación de donde a donde queremos desplazarnos.

#### Paso 3

Nos aparece un catálogo de autos a elegir: X, Pool, Black y Van. Además, también se encuentra el Conductor puesto que, sin importar el tipo de Uber a elegir, siempre estará involucrado.

#### Paso 4

Una vez elegido el auto, nosotros tenemos que únicamente esperar mientras el auto se dirige hacia nuestra ubicación y finalmente nos llevara del punto A al punto B. Ya, una vez nos esté llevando, será visible un saldo a cobrar por el viaje realizado.

Y ese es nuestro breve análisis de como funcionara nuestra aplicación Uber. Lo siguiente que aprendimos es que debemos analizar nuestros objetos y extraerlos.

#### Objetos

En el paso 1 tenemos a nuestro objeto **User (Usuario)** que estará solicitando el auto. Así como el objeto **Route (Ruta)** para trasladarnos.

En el paso 3 tenemos un catálogo con diferentes tipos de autos y eso significa diferentes tipos de objetos: **UberX**, **UberPool**, **UberBlack** y **UberVan**. Además, también tenemos nuestro objeto **Driver (Conductor)**.

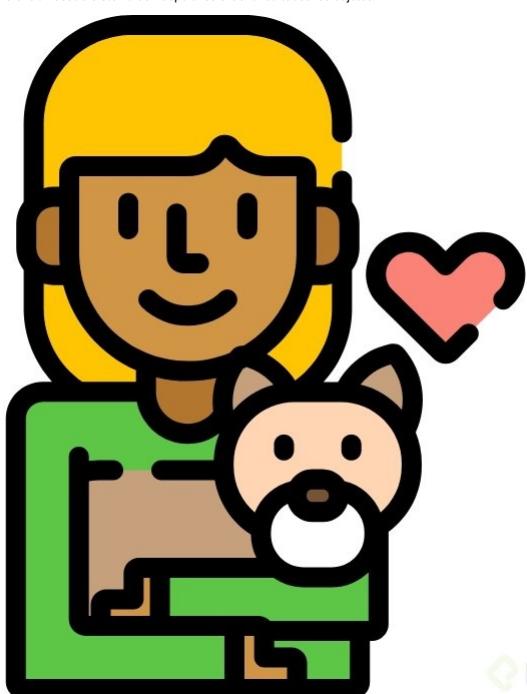
Finalmente en el paso 4 podemos ver que cuando nuestro viaje fue realizado se nos cobrara un monto y eso significa que tendremos distintas formas de pagar ese viaje. Ya sea a través de un objeto **Card (Tarjeta)**, **PayPal** o **Cash**.

También tenemos un último objeto que es del tipo conceptual y está presente durante todo nuestro análisis, ese es el objeto **Trip (Viaje)** que captura quien ejecuta el viaje, a donde quieras ir, que auto elegiste y que forma de pago realizaras.

### Clase 12 Reto 1: identificando objetos

Ya estás listo para resolver tu primer reto y poner en práctica todo lo que aprendiste para identificar objetos en un problema.

Toma como referencia nuestro Sistema de Adopciones e identifica todos los objetos.



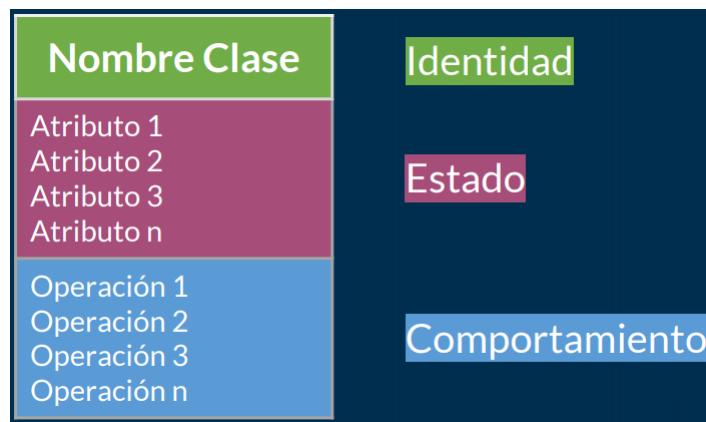
Comprétenos tu análisis en la sección de discusiones.

### Modulo 3. Programación Orientada a Objetos. Análisis

#### Clase 13 Clases en UML y su sintaxis en código

En esta clase veremos cómo podemos definir las clases para plasmarlas en un diagrama UML. Recordemos que nuestro proceso es: identificar el problema, identificar los objetos, definir las clases y finalmente plasmarlas en un diagrama.

#### Clases en UML

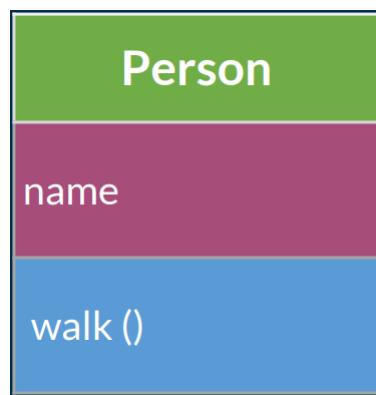


En UML, una clase será representada como un rectángulo con tres zonas:

- **Superior:** Es donde colocaremos el nombre de la clase
- **Intermedio:** Aquí definiremos los atributos
- **Inferior:** Es donde estarán los comportamientos, es decir, los que serán las funcionalidades.

Con esto le daremos a nuestras clases en UML una identidad (nombre de la clase), estados (atributos o propiedades) y operaciones (comportamientos).

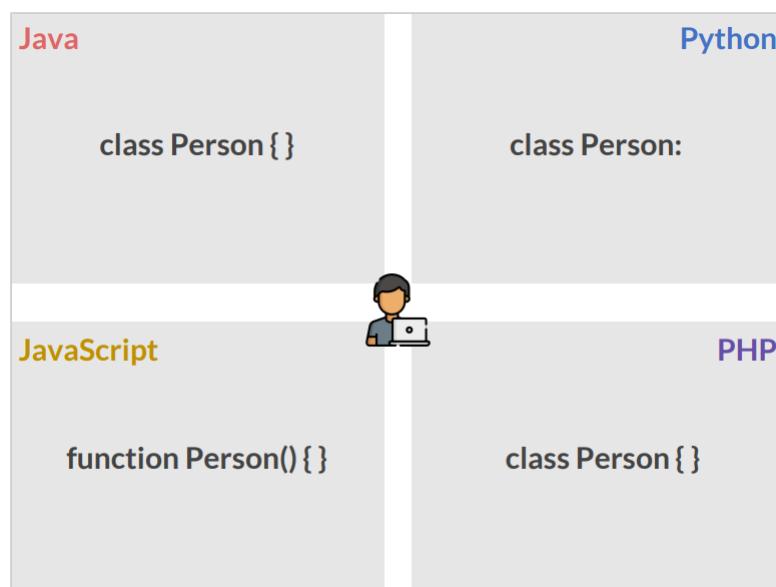
Para nuestro ejemplo, imaginemos que tenemos una clase **Person** cuyo atributo o propiedad es **name** y su comportamiento sea **walk()**:



Ya, una vez identificado el objeto Person y puesto en un diagrama UML, nuestro siguiente paso es definirlo en código.

#### SINTAXIS EN CÓDIGO

Nosotros ya habíamos dicho que trabajaremos en cuatro lenguajes: Java, Python, JavaScript y PHP.



- **Java:** Para declarar una clase utilizamos la palabra reservada **class**, seguido del nombre de la clase y finalizamos con llaves.

**class nombreClase {}**

- **Python:** Aquí usamos la palabra reservada **class**, seguido del nombre de la clase y finalizamos con dos puntos.

**class nombreClase:**

- **JavaScript:** Como sabemos este lenguaje se maneja de forma distinta y eso se debe a que todo es a través de prototipos. Sin embargo, utiliza la Programación Orientada a Objetos para analizar problemas y posteriormente poder plasmarlos en código de la mejor forma, por lo tanto al tenerlo en prototipos utilizaremos "funciones especiales" para definir las clases. Usaremos la palabra reservada **function** seguido del nombre de la clase con paréntesis y finalizaremos con llaves.

**function nombreClase() {}**

- **PHP:** Para declarar una clase nueva es totalmente idéntico a como declaramos en Java.

#### DEFINIR ATRIBUTO Y COMPORTAMIENTO

## Java

```
class Person {  
    String name = "";  
    void walk() {}  
}
```

## Python

```
class Person:  
    name = "";  
    def walk():
```

## JavaScript

```
Person.prototype.walk = function (){  
}
```



## PHP

```
class Person {  
    $name = "";  
    function walk() {}  
}
```

- **Java:** Para declarar un atributo es necesario poner el tipo de dato seguido del nombre, y para declarar un método ponemos el tipo seguido del nombre con dos paréntesis y finalizamos con llaves.

```
class nombreClase {  
    tipo variable1;  
    tipo variable2;  
    tipo variableN;  
  
    tipo método1(parámetro) {  
        Cuerpo del método  
    }  
  
    tipo método2() {  
        Cuerpo del método  
    }  
  
    tipo métodoN(parámetro, parámetro) {  
        Cuerpo del método  
    }  
}
```

- **Python:** Este lenguaje no es estricto en su tipado, por lo que para definir variables es simplemente necesario poner el nombre, en nuestro ejemplo ponemos comillas dobles para que Python infiera que es un string, y para declarar un método utilizamos la palabra reservada def. La forma que usa Python para agrupar declaraciones es mediante indentaciones, por lo que en el intérprete interactivo debes teclear un tabulador o espacio(s) para cada línea indentada.

```
class nombreClase:  
    variable = ""  
  
    def método:  
        Cuerpo del método  
  
    def método2:  
        Cuerpo del método
```

- **JavaScript:** Cuando se empieza a programar en un lenguaje como JavaScript, es decir, permisivo hasta no poder más, dar los primeros pasos puede resultar realmente complicado. Para declarar nuestras propiedades se utiliza la palabra reservada this y los métodos son declarados fuera usando la palabra reservada prototype seguido de la función.

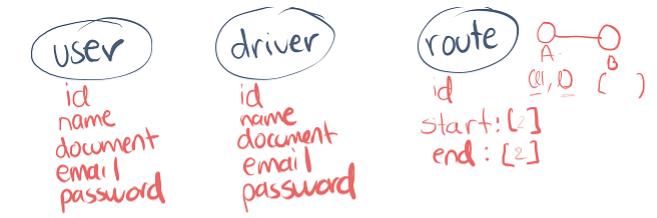
```
function Person(nombre) {  
    this.name = nombre  
}  
  
Person.prototype.walk = function () {  
    Cuerpo del método  
}
```

- **PHP:** Siendo también un lenguaje bastante flexible así que para declarar una variable usamos el símbolo \$, sin importar el tipo de dato, y para declarar un método es lo mismo que una función.

```
class nombreClase {  
    $variable = " ";  
  
    function método() {}  
}
```

## Clase 14 Modelando nuestros objetos Uber

Acabamos de aprender como plasmar objetos en diagramas de clase en UML, hagamos esto mismo en nuestro proyecto Uber. Analicemos los objetos y quitemos todos sus atributos.



- User

- Id: El identificador. Es bastante útil cuando manejamos una base de datos.
- Name
- Document: Es el documento que dependiendo del país puede ser numérico o alfanumérico. En este caso puede ser la CI, la CURP o el RFC.
- Email
- Password

- Driver

- Id
- Name
- Document
- Email
- Password

- Route: Como sabemos que se compone de un punto A y un punto B, sabemos que las ubicaciones tienen una latitud y una longitud.

- Id
- Start [2]: Representa el punto A y será un arreglo que contenga tanto la latitud como la longitud.
- End [2]: Representa el punto B y será un arreglo que contenga tanto la latitud como la longitud.

- UberX

- Id
- License: Será la placa del vehículo
- Driver: El conductor designado del vehículo
- Passenger: La cantidad de pasajeros
- Brand: Marca del vehículo
- Model: Modelo de vehículo

- UberPool

- Id
- License
- Driver
- Passenger
- Brand
- Model

- UberBlack

- Id
- License
- Driver
- Passenger
- typeCarAccepted []: Existe un catálogo de vehículos únicamente aceptados por Uber y se componen de la marca, modelo y año.
- seatsMaterial []: Los Uber Black necesitan tener interior con materiales de piel o vinilo.

- UberVan

- Id
- License
- Driver
- Passenger
- typeCarAccepted []
- seatsMaterial []

Si quieres saber más de los requerimientos de autos puedes ingresar en [Requisitos de autos](#).

Ahora analizaremos los últimos objetos que nos quedan:



- Card

- Id
- Number: Es el número de la tarjeta.
- CVV: El número escondido por detrás.
- Date: Es la fecha de vencimiento.

- PayPal

- Id
- Email: El correo asociado a la cuenta.

- Cash

- Id
- Cash no necesitará nada más que el identificador del tipo de pago ya que en este caso no tenemos registro de este tipo.

Con esto ya tenemos analizado todos nuestros objetos, pero hay algo de redundancia en el diagrama. En la siguiente clase veremos cómo podemos solucionarlo con la herencia.

## Clase 15 ¿Qué es la herencia?

En la clase anterior notamos que nuestro ejemplo tenía atributos repetidos y no solo fue uno, sino que fueron varios. Muchas clases entre sí tenían atributos que estaban siendo redundantes entre ellas, pues esto estaba violando una de las leyes del código.

«Don't repeat yourself» es una filosofía que promueve la reducción de la duplicación en la programación. Siempre nos inculcará que no tengamos líneas de código duplicadas y, en este caso, todavía no hemos hecho código estamos a un paso de hacerlo, pero si lo llevamos así como esta estaríamos violando esta filosofía. Por lo tanto, toda pieza de información no debería ser duplicada debido a que la duplicación incrementa la dificultad en los cambios y su evolución.

Si nosotros dejamos esto así como está, se nos va a dificultar que en el futuro podamos ejecutar cambios e incluso involucrar un objeto o un elemento más en el proyecto, hará que el código sea más difícil de leer y entender, y hace un mantenimiento se nos va a

complicar bastante. Por lo tanto no debemos tener líneas duplicadas en la medida posible.

#### ¿Qué debemos hacer?

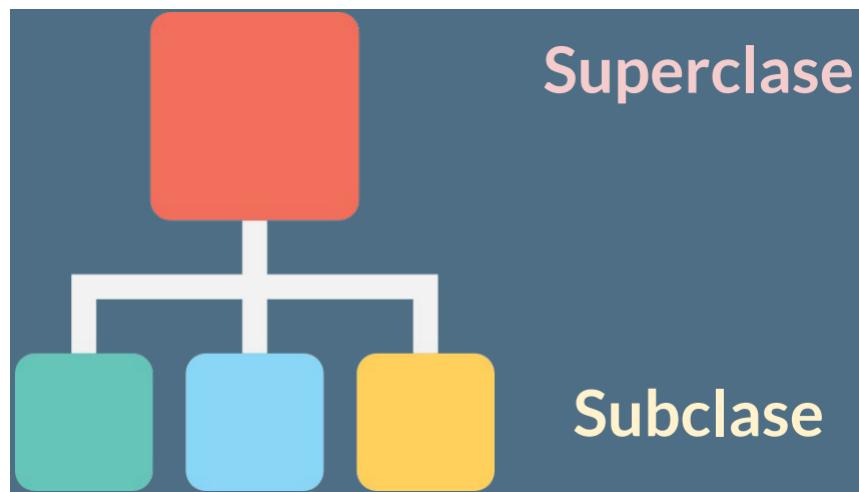
Haremos uso de uno de los principios de la programación orientada a objetos, la reutilización de código. La herencia es una de las piezas clave a la hora de reutilizar líneas de código a más no poder.

#### Herencia

La herencia es un pilar importante dentro de la programación orientada a objetos y nos permitirá crear nuevas clases a partir de otras.

Podemos definir la herencia como la capacidad de crear clases que adquieren de manera automática los miembros (atributos y métodos) de otras clases que ya existen, pudiendo al mismo tiempo añadir atributos y métodos propios.

Lo que haremos es que, una vez detectemos elementos duplicados, ejecutaremos una abstracción de tal manera que podremos generar una clase que sea la más general y, entonces, esa clase general nos permitirá crear nuevas clases. Tendremos una jerarquía, una estructura de padre e hijo, y es que un padre puede tener tantos hijos como sea necesario. Es común encontrar que un padre solo tenga un único hijo, pero, como en nuestro ejemplo, un padre puede tener bastantes hijos a través de la abstracción.



- Clase Padre: También llamada Super Clase, será la clase cuyas características se heredan.
- Clase Hijas: Llamada también Sub Clase. Son las clases que heredan de Clase Padre, puede agregar sus propios campos y métodos, además de los campos y métodos de la superclase.

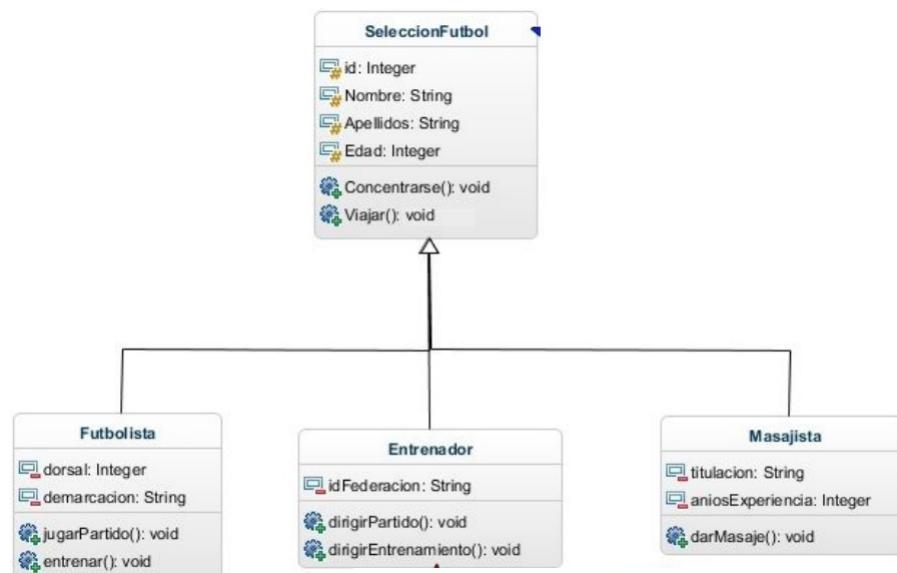
#### EJEMPLO

Para ejercitarnos y poder identificar herencias tenemos el siguiente ejemplo:

Futbolista	Entrenador	Masajista
<ul style="list-style-type: none"><li>▢ id: Integer</li><li>▢ Nombre: String</li><li>▢ Apellidos: String</li><li>▢ Edad: Integer</li><li>▢ dorsal: Integer</li><li>▢ demarcacion: String</li><li>▢ Concentrarse(): void</li><li>▢ Viajar(): void</li><li>▢ jugarPartido(): void</li><li>▢ entrenar(): void</li></ul>	<ul style="list-style-type: none"><li>▢ id: Integer</li><li>▢ Nombre: String</li><li>▢ Apellidos: String</li><li>▢ Edad: Integer</li><li>▢ idFederacion: String</li><li>▢ Concentrarse(): void</li><li>▢ Viajar(): void</li><li>▢ dirigirPartido(): void</li><li>▢ dirigirEntrenamiento(): void</li></ul>	<ul style="list-style-type: none"><li>▢ id: Integer</li><li>▢ Nombre: String</li><li>▢ Apellidos: String</li><li>▢ Edad: Integer</li><li>▢ Titulacion: String</li><li>▢ aniosExperiencia: Integer</li><li>▢ Concentrarse(): void</li><li>▢ Viajar(): void</li><li>▢ darMasaje(): void</li></ul>

En donde tenemos tres clases (Futbolista, Entrenador y Masajista), si analizamos estas clases podemos ver que comparten cuatro atributos y además tienen en común dos métodos.

En programación orientada a objetos, cuando detectamos que hay elementos repetidos, esto nos indica que debemos hacer algo. Algo no está bien y que seguramente, si lo dejamos así, nos traerá problemas a futuro. Una vez detectada una relación de estos elementos, podemos generar una abstracción de eso y entonces crear una clase que tengan todos estos elementos en común.



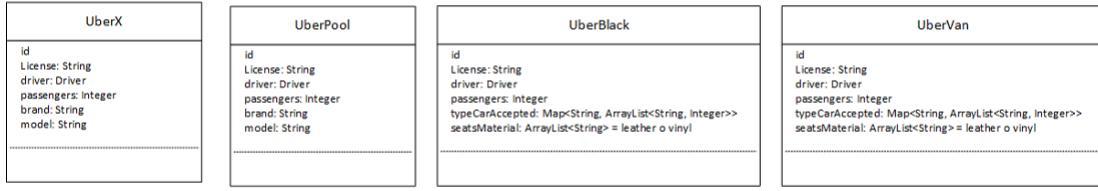
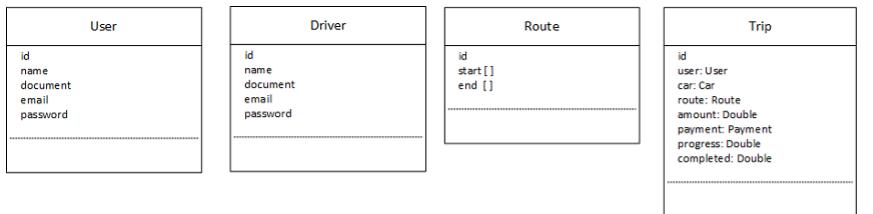
En este caso creamos **SeleccionFutbol** que será la superclase y de ella se estarán heredando: **Futbolista, Entrenador y Masajista**. Cuando ellos heredan, esto significa que automáticamente todos los atributos y métodos que tenemos ahí serán heredados a las subclases y no tenemos que escribirlo en código, simplemente aplicando la herencia automáticamente van a aparecer en cada clase.

Esta es una forma de analizar herencia, hay otra forma y es partiendo de los elementos en común. En general podemos tener elementos que no tengan ningún atributo en común, pero la lógica del negocio nos va a decir que esto debe considerarse como una clase más general, deben agruparse en una clase más general aunque y esa se puede llamar una clase padre.

#### Clase 16 Aplicando Herencia a nuestro proyecto Uber

Aplicaremos lo aprendido en la clase anterior y eso es detectar todos los atributos que son redundantes en nuestro proyecto Uber.

Tenemos plasmado nuestro proyecto en forma de diagrama de clase:

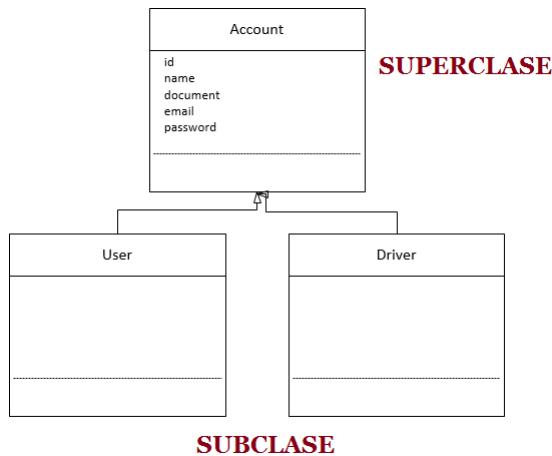


Y ahora vamos a detectar la redundancia entre estos elementos para aplicar la herencia.

#### USER – DRIVE

Comencemos por las clases User y Drive que tienen todos sus atributos en común. Si sacamos esos elementos comunes podemos crear la clase Account, que poseerá la jerarquía principal y se convertirá en la Super Clase o Clase Padre, mientras que User y Driver se heredaran de Account convirtiéndose en Sub Clases o Clases Hijas.

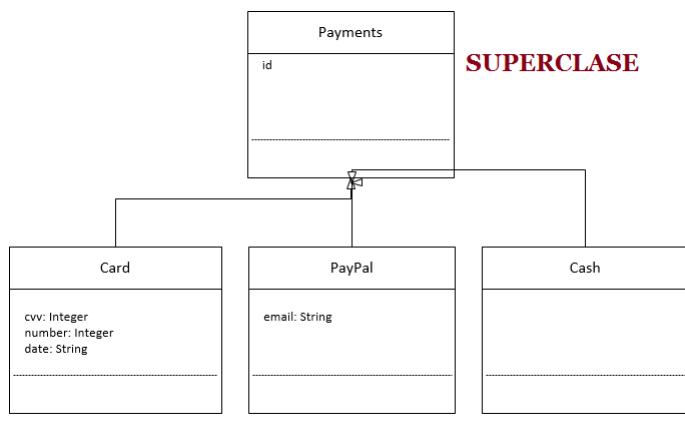
Para exemplificar la herencia usamos flechas vacías que apuntan hacia la Clase Padre.



#### SUBCLASE

#### CARD – PAYPAL – CASH

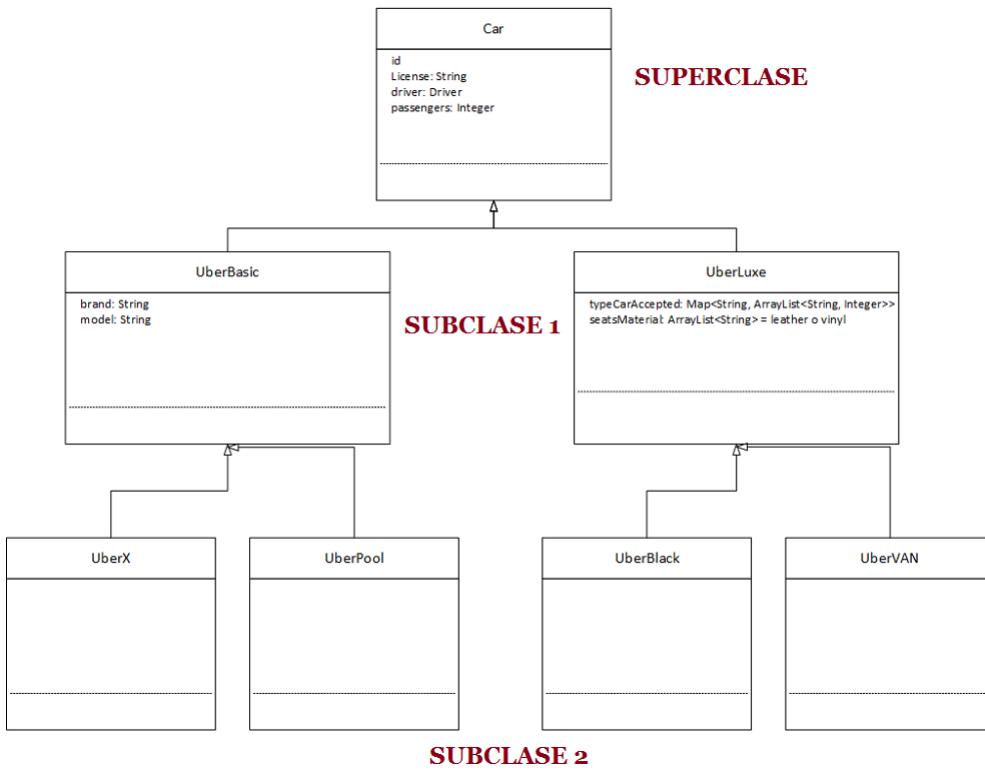
En el caso de las clases Card, PayPal y Cash no tenemos ningún elemento en común, sin embargo, todos son del mismo tipo así aplicaremos la otra forma de herencia: según la lógica de negocios.



#### SUBCLASES

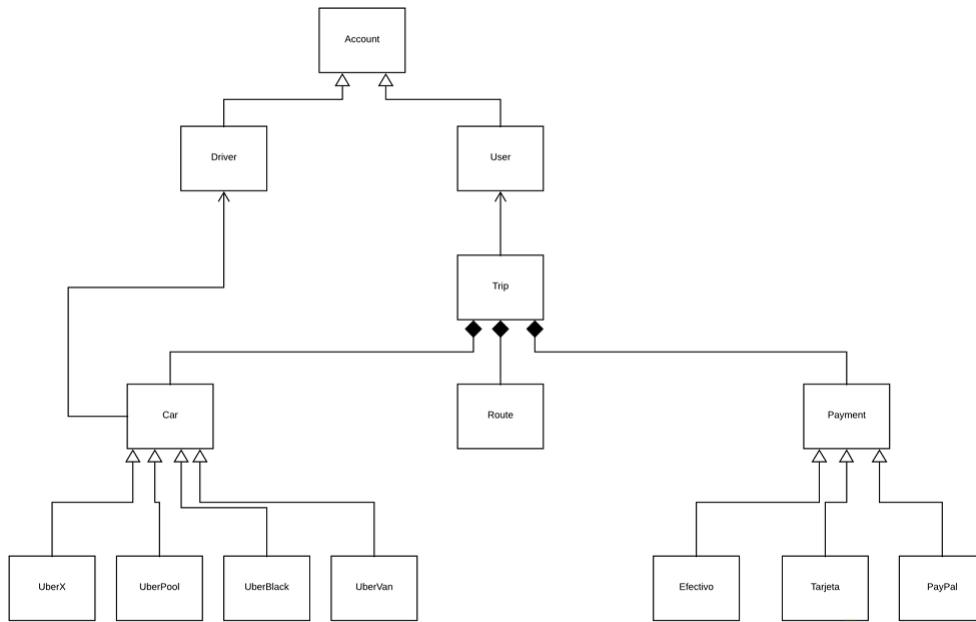
#### UBERX – UBERPOOL – UBERBLACK – UBERVAN

En el caso de los automóviles, las cuatro clases Uber tienen cuatro atributos en común que formaran parte de la Super Clase, sin embargo, todavía existen elementos comunes entre UberX y UberPool, y UberBlack y UberVan así que podemos hacer uso nuevamente de la herencia para otra Sub Clase.



#### Modelo Simplificado

Así es como estaría quedando nuestro sistema Uber.



#### Clase 17 Reto 2: analicemos un problema

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

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

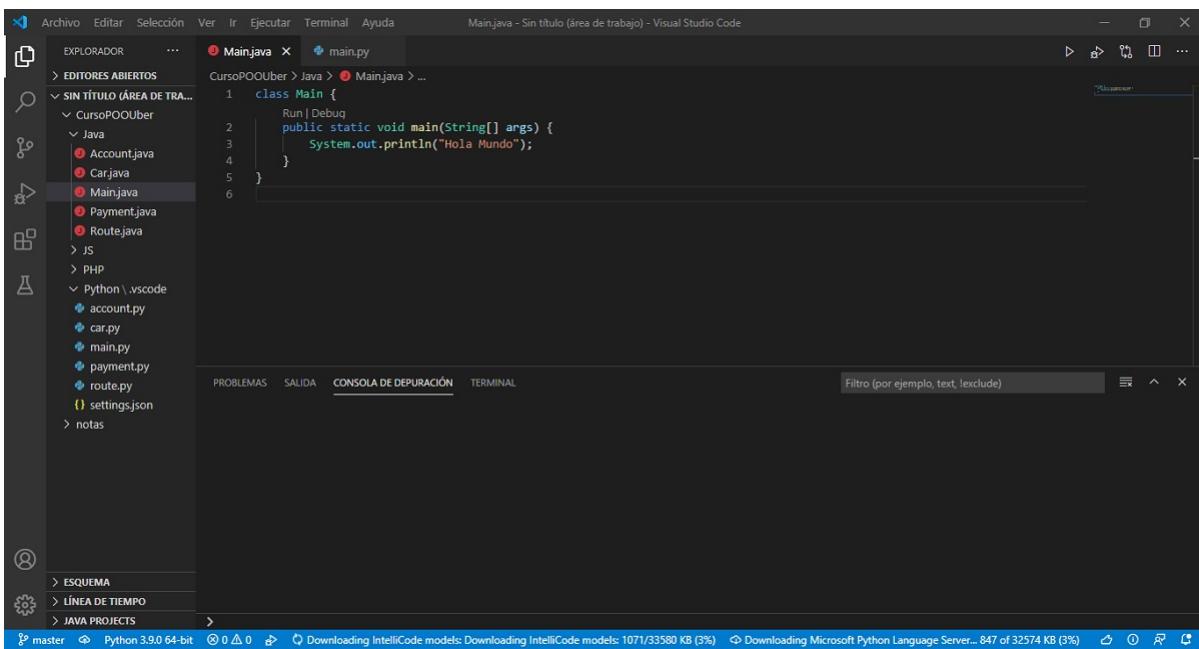
Comparte tus resultados en la sección de discusiones.

#### Modulo 4. Clases, Objetos y Método Constructor

#### Clase 18 Definiendo clases en Java y Python

Ahora que está listo nuestro modelo podemos comenzar con la etapa de programación de nuestro proyecto. Como vimos, todo el módulo anterior se basó en el análisis de los objetos; analizar y obtener los objetos, abstraerlos para convertirlos en clases y finalmente ejecutar un análisis de herencia para tener una mejor versión de nuestros objetos.

[Visual Studio Code](#)



Primero estaremos creando cuatro carpetas llamadas Java, Python, JavaScript y PHP en donde iremos depositando los documentos para sus correspondientes lenguajes.

Siguientemente nuestro diagrama de herencia que hicimos del proyecto estaremos creando cuatro archivos para las clases principales: Account, Car, Payment y Router. Además de un archivo Main que será donde generaremos el punto de entrada al programa, todo lo que queremos que funcione y sea visto debemos declararlo en este documento.

#### JAVA

- **Main:** Lo que debes saber en primer lugar es que el método main() es el punto de entrada de la aplicación, es decir, es el punto en el que comienza la ejecución de esta. Es por ello que ha de ser public (accesible desde fuera de la clase) y static (se puede ejecutar sin una instancia de la clase).

```
1 class Main {
2     Run| Debug
3     public static void main(String[] args) {
4         System.out.println("Hola Mundo");
5     }
}
```

- **Account:** Es la super clase de la que se derivaran las clases Drive y User.

```
1 class Account {
2     Integer id;
3     String name;
4     String document;
5     String email;
6     String password;
7 }
```

- Para declarar variables en Java ponemos primero el tipo de dato que estará manejando nuestro usuario y posteriormente el nombre.

- **Car:** Es la super clase de la que se derivaran las clases UberX, UberPool, UberBlack y UberVan.

```
1 public class Car {
2     Integer id;
3     String license;
4     String drive;
5     Integer passenger;
6 }
```

- Momentáneamente estaremos declarando 'drive' de tipo string, aunque sabemos que es del tipo Drive.

- **Payment:** Es la super clase de la que se derivaran las clases Card, PayPal y Cash.

```
1 class Payment {
2     Integer id;
3 }
```

- **Router:** Es la clase que contendrá las ubicaciones de nuestros puntos A y B.

```
1 import java.util.ArrayList;
2
3 class Route {
4     Integer id;
5     ArrayList<Double> start;
6     ArrayList<Double> end;
7 }
```

- Una opción muy conveniente cuando usamos una clase o interfaz de otro paquete, es el uso de la palabra clave import. Si usamos import, no es necesario escribir el nombre completo de la clase con el paquete incluido. Es suficiente con el nombre de la clase.
- En este caso, nosotros usaremos ArrayList que pertenece a la biblioteca java.util. Por tanto, para emplearla en nuestras clases o programas escribiremos como código en cabecera import java.util.ArrayList (o de forma más genérica import java.util.\*).
- ArrayList: Es una clase que permite almacenar datos en memoria de forma similar a los Arrays, con la ventaja de que el número de elementos que almacena, lo hace de forma dinámica, es decir, que no es necesario declarar su tamaño como pasa con los Arrays.

#### PYTHON

- **Main:** Este main, básicamente, comprueba si un módulo \*.py se está importando a nuestro código y establece, de ser así (y de ahí el condicional if) que sea el módulo actual el principal, el que asume el papel de main(), aquél que deberá ejecutarse primero, mientras que el módulo importado se ejecutará a continuación.

```
1 if __name__ == "__main__":
2     print("Hola Mundo")
```

- **Account:** Como podemos ver, somos capaces de declarar el tipo de dato que queremos que sean de esta forma.

```
1 class Account:
2     id = int
3     name = str
4     document = str
5     email = str
6     password = str
```

- Para declarar variables en Python, en cambio, va primero el nombre y después podemos asignarle el tipo de datos.

- **Card**

```
1 class Card:  
2     id = int  
3     license = str  
4     drive = str  
5     passenger = int
```

- Payment

```
1 class Payment:  
2     id = int
```

- Router

```
1 class Route:  
2     id = int  
3     start = []  
4     end = []
```

- Las variables de tipo arrays en Python se declaran poniendo corchetes vacíos.

## Clase 19 Definiendo Clases en JavaScript

Si estás interesado en aprender JavaScript desde ahora debes saber que el concepto de clases no existía como tal hasta el nuevo estándar ECMAScript 6. El reto de encontrar sistemas construidos con este estándar es alto por esa razón te explicaré cuál fue por mucho tiempo su equivalente.

Los Prototipos fue la forma de crear clases en JavaScript y las representaremos partiendo de la declaración de una función.

Creemos nuestras clases:

- Account
- Car
- Payment
- Route

Para esto crearemos el siguiente sistema de archivos dentro de la carpeta JS de nuestro proyecto:

- Account.js
- Car.js
- Payment.js
- Route.js
- index.js

El archivo index.js será el lugar equivalente al punto de entrada de la aplicación donde estaremos declarando nuestros objetos basado en las clases. Para esta clase lo dejaremos en blanco.

Ahora veamos el código archivo por archivo:

### Account.js

```
1 function Account() {  
2     this.id;  
3     this.name;  
4     this.document;  
5     this.email;  
6     this.password;  
7 }  
8 }
```



### Car.js

```
1 function Car() {  
2     this.id;  
3     this.license;  
4     this.driver;  
5     this.passenger;  
6 }  
7 }
```



### Payment.js

```
1 function Payment() {  
2     this.id;  
3 }  
4 }
```



### Route.js

```
1 function Route () {  
2     this.id;  
3     this.init;  
4     this.end;  
5 }  
6 }
```



Aquí podemos ver el código del proyecto.

En este código notarás el uso de la palabra reservada this. Normalmente cuando usamos la sintaxis punto siempre lo haremos a partir de un objeto instanciado, en este caso con this, se hace una simulación al objeto en cuestión, a pesar de que en ese momento visualmente sigue siendo una clase.

```
1 function Route () {  
2     this.id;  
3     this.init;  
4     this.end;  
5 }
```



Digamos que se adelanta un poco al momento de ejecución y visualiza al objeto con sus atributos, más adelante verás la forma en que podemos asignar datos a un atributo del objeto en otros lenguajes y verás que es exactamente la misma sintaxis.

Si intentáramos poner this en el momento de ejecución nos traería un listado de todos los componentes de la clase que en este caso son solo estos tres: id, init y end.

This hace referencia al objeto instanciado. Para comprender del todo esta última frase mira la siguiente clase donde hablamos de objetos.

#### Reto

- En la carpeta de nuestro proyecto PHP declara estas mismas clases: Puedes utilizar esta [clase](#) de apoyo.
- Inténtalo y compártenos tus resultados, compáralos con tus compañeros.

#### Clase 20 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.

Lo anteriormente hecho solo fueron clases, no pudimos nada en pantalla más que unos mensajes que decían "hola mundo", pero sin poder hacer verdaderamente nada. Para poder utilizar los elementos declarados dentro de esas clases empezaremos a trabajar con los objetos.

Recordemos que los objetos nos ayudaran a crear instancias de una clase, es decir, es el resultado de lo que moldeamos, de los parámetros declarados y usaremos los objetos para que nuestras clases cobren vida.

#### Declarar objetos

Java	Python
Person person = new Person();	persona = Person()
JavaScript	PHP
var person = new Person();	\$person = new Person();

- **Java:** Al momento de crear objetos en Java, debemos tener claras dos cosas indispensables, la primera es el nombre de la clase para la cual vamos a crear el objeto y segundo el constructor que dicha clase posee, es decir, si el constructor recibe o no parámetros. Para crear objetos en Java, el lenguaje nos proporciona el comando new, con este comando le decimos a Java que vamos a crear un nuevo objeto de una clase en específico y le enviamos los parámetros (en caso de ser necesario) según el constructor.

**NombreClase miObjeto = new NombreClase();**

- **Python:** Haciendo valer su fama como un lenguaje flexible, para declarar un objeto es bastante sencillo ya que solo necesita el nombre del objeto y la clase a la que hará instancia.

**miObjeto = nombreClase()**

- **JavaScript:** Al igual que con muchas cosas en JavaScript, la creación de un objeto a menudo comienza con la definición e iniciación de una variable. También hacemos uso del comando new para crear un nuevo objeto de una clase específica.

**var miObjeto = new NombreClase();**

- **PHP:** Para crear una instancia de una clase, se debe emplear la palabra reservada new. Un objeto se creará siempre a menos que el objeto tenga un constructor que arroje una excepción en caso de error. Las clases deberían ser definidas antes de la instanciación (y en algunos casos esto es un requerimiento).

**\$miObjeto = new NombreClase();**

#### Método constructor

Un constructor es un método especial de una clase que se llama automáticamente siempre que se declara un objeto de esa clase. Su función es inicializar el objeto y sirve para asegurarnos que los objetos siempre contengan valores válidos.

Para nosotros, las paréntesis representan los métodos.

## Java

```
public Person(String name){  
    this.name = name;  
}
```

## Python

```
def __init__(self, name):  
    self.name = name
```

## JavaScript

```
function Person(name) {  
    this.name = name  
}
```

## PHP

```
public function __construct($name){  
    $this->name = name;  
}
```

- **Java:** En el lenguaje Java, si para una clase no se define ningún método constructor se crea uno automáticamente por defecto. El constructor por defecto es un constructor sin parámetros que no hace nada. Los atributos del objeto son iniciados con los valores predeterminados por el sistema.

```
public nombreClase(String parámetro) {  
    this.variable = parámetro;  
}
```

- **Python:** En Python, el método constructor siempre se llama `__init__` (dos subrayados antes y después de `init`).

```
def __init__(self, parámetro):  
    self.variable = parámetro
```

- **JavaScript:** En JavaScript, la función sirve como el constructor del objeto, por lo tanto, no hay necesidad de definir explícitamente un método constructor. Cada acción declarada en la clase es ejecutada en el momento de la creación de la instancia.

```
function nombreClase(parámetro) {  
    this.variable = parámetro;  
}
```

- **PHP:** Debemos definir un método llamado `__construct` (es decir utilizamos dos caracteres de subrayado y la palabra `construct`). El constructor debe ser un método público (`public function`).

```
public function __construct($parámetro) {  
    this->variable = parámetro;  
}
```

*NOTA: No te preocupes por entender `this`, no te compliques, lo estaremos viendo más adelante.*

Pasar datos por parámetros

Dependiendo del tipo de dato que envíamos a la función, podemos diferenciar dos comportamientos: Paso por valor (se crea una copia local de la variable dentro de la función) y Paso por referencia (se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera).

## Java

```
Person person = new  
Person("Ann");
```

## Python

```
person = Person("Ann")
```

## JavaScript

```
var person = new  
Person("Ann");
```

## PHP

```
$person = new Person("Ann");
```

- **Java**

```
NombreClase miObjeto = new NombreClase("Ann");
```

- **Python**

```
MiObjeto = NombreClase("Ann")
```

- **JavaScript**

```
var MiObjeto = new NombreClase("Ann")
```

- **PHP**

```
$MiObjeto = new NombreClase("Ann");
```

Como es visible en todos los casos, para enviar un valor es únicamente necesario ponerlos dentro de los paréntesis. Como en este caso estamos enviando una cadena de caracteres o string, ponemos comillas dobles, algo que no es necesario cuando enviamos como valor un número.

Ya aprendimos la forma de crear objetos y sabemos la sintaxis básica en los cuatro lenguajes que vamos estudiando, ya es hora de ver como declarar esos objetos en Java y Python.

Recordemos que anteriormente dejamos nuestras clases listas. Por el momento, y por el bien de la práctica, usaremos la clase Car ya que es la que más sentido se nos hace.

#### JAVA

Primeramente, para crear objetos en el lenguaje Java, debemos ir a nuestra clase Main que recordemos tiene actualmente esto:

```
class Main {  
    public static void main(String[] args) {  
        System.out.println("Hola Mundo");  
    }  
}
```

Debemos recordar que para crear un objeto se sigue esta sintaxis: el tipo de la clase, nombre del objeto, igualamos (=), usamos la palabra reservada new y terminamos con el método constructor que trae por defecto las clases de Java.

En nuestro caso quedaría así:

```
class Main {  
    Run | Debug  
    public static void main(String[] args) {  
        System.out.println("Hola Mundo");  
  
        Car car = new Car();  
        car.license = "AMQ123";  
        car.driver = "Andres Herrera";  
        car.passenger = 4;  
        System.out.println("License: " + car.license);  
  
        Car car2 = new Car();  
        car2.license = "QWE567";  
        car2.driver = "Mary Blackstone";  
        car2.passenger = 3;  
        System.out.println("License: " + car2.license);  
    }  
}
```

- **Car:** Sería la clase que estaríamos usando.
- **car, car2:** Son el nombre de los objetos.
- **new:** Palabra reservada para la creación de objetos.
- **Car():** Es como llamamos al método constructor.
- **Operador punto ():** Nos permite acceder a los distintos atributos de la clase. Cuando tenemos un objeto de un tipo determinado y queremos acceder a uno de sus atributos solo tenemos que poner el identificador asociado al objeto seguido por un punto y por el identificador que hace referencia a un miembro concreto de la clase a la que pertenece el objeto.
  - **car.license()**: Accedemos al atributo license en el que guardamos la licencia que tiene el vehículo.
  - **car.driver ()**: Accedemos al atributo driver en el que guardamos el valor del nombre que tiene nuestro conductor.
  - **car.passenger()**: Accedemos al atributo passenger en el que guardamos la cantidad de pasajeros que podrá llevar ese vehículo.
- **System.out.println()**: En Java hay algunos objetos que existen por defecto (en cualquier entorno de desarrollo). Uno de ellos es el objeto denominado System.out. Este objeto dispone de un método llamado println que nos permite imprimir algo por pantalla en una ventana de consola.

Para no tener que estar declarando System.out.println() cada vez que deseamos imprimir el valor de nuestro objeto, podemos crear un método en la clase Car:

```
public class Car {  
    Integer id;  
    String license;  
    String driver;  
    Integer passenger;  
  
    void printDataCar() {  
        System.out.println("License: " + license + " Drive: " + driver);  
    }  
}
```

- Debajo de los atributos escribimos el método: primero el tipo y después el nombre, como no recibirá nada no necesita parámetros.
- La palabra void indica que el método printDataCar no retorna ningún valor, solamente imprimirá en pantalla la licencia y el conductor.

Finalmente, en nuestra clase Main cambiamos los System.out.println() por **car.printDataCar()** y **car2.printDataCar()**.

De esta forma, con los objetos, estamos accediendo a los atributos y a los objetos. Reutilizamos código para imprimir datos de los objetos.

#### PYTHON

Para crear objetos dentro de Python es necesario importar la clase de la que estaremos usando los elementos eso se hace escribiendo al principio de toda la clase Main:

```
from car import Car
```

- Donde car es el nombre del archivo y Car es el nombre de la clase.

Como en Python no es necesario escribir el tipo de clase y tampoco la palabra reservada new, simplemente creamos los objetos directo:

```
if __name__ == "__main__":  
    print("Hola Mundo")  
  
    car = Car()  
    car.license = "AMS234"  
    car.driver = "Andres Herrera"  
    print(vars(car))  
  
    car2 = Car()  
    car2.license = "QWE567"  
    car2.driver = "Mary Blackstone"  
    print(vars(car2))
```

En general permanece bastante similar al Java, excepto por algunos cambios obvios:

- **car, car2:** Son el nombre de los objetos.
- **Car():** Es como llamamos al método constructor.
- **Operador punto:** Nos permite acceder a los distintos atributos de la clase.

El cambio está en la forma de imprimir un objeto, no necesitamos crear un método especial para poder hacerlo, sino que simplemente usamos vars dentro de un print y le pasamos el objeto como parámetro:

- **print:** Es lo que nos permite mostrar texto en pantalla.
- **vars:** Toma como máximo un parámetro y devuelve los atributos del objeto. En este caso lo imprime en formato JSON, pero si queremos otro formato ya podemos ponerle un método. Sin embargo, en este caso, para fines demostrativos, nos funciona super bien.

## Clase 22 Declarando un Método Constructor en Java y JavaScript

En la clase anterior vimos como declarar el método constructor en el caso de Java y Python, hoy vamos a ponerlo en acción para el caso particular de JavaScript y Java.

#### JAVA

Primeramente lo veremos en el caso de Java para entenderlo mejor, porque en JavaScript es bastante peculiar.

Lo que haremos es ir a la clase Car en donde crearemos nuestro método constructor:

```

public class Car {
    Integer id;
    String license;
    Account driver;
    Integer passenger;

    public Car(String license, Account driver) {
        this.license = license;
        this.driver = driver;
    }

    void printDataCar() {
        System.out.println("License: " + license + " Name Drive: " + driver.name);
    }
}

```

- **public:** Indica que es un método accesible a través de una instancia del objeto.
- **Car():** Es el nombre que tendrá nuestro método.
  - Dentro de las paréntesis van los parámetros obligatorios para crear un objeto de tipo Car. En este caso, lo mínimo necesario es license y drive.
- Dentro del constructor ponemos this., y esto no es más que una buena práctica porque se acostumbra mucho que los parámetros tengan el mismo nombre que las propiedades.
  - **this.license:** Hace referencia al license, la variable global de la clase.
  - **license:** Hace referencia al parámetro license, la variable local que existen únicamente dentro del método.
- Ahora en nuestro método printDataCar si lo dejamos tal cual no imprimirá lo que estamos queriendo, por eso debemos cambiarlo y poner **driver.name**.

Hasta ahora también hemos estado manejando driver como un string, pero sabemos que en realidad es de tipo Account, así que debemos ir a la clase Account para crear su método constructor:

```

class Account {
    Integer id;
    String name;
    String document;
    String email;
    String password;

    public Account(String name, String document) {
        this.name = name;
        this.document = document;
    }
}

```

Ya hecho una vez todo esto y ejecutamos nuestro programa nos indicará un error, porque no hemos realizado los demás cambios. Cuando sobrescribimos el método automáticamente el método vacío que teníamos se pierde, así que para pasar los datos debemos hacerlo dentro de las paréntesis:

```

class Main {
    Run | Debug
    public static void main(String[] args) {
        System.out.println("Hola Mundo");
        Car car = new Car("AMQ123", new Account("Andres Herrera", "AND123"));
        car.passenger = 4;
        car.printDataCar();

        Car car2 = new Car("QWE567", new Account("Mary Blackstone", "NTS027"));
        car2.passenger = 3;
        car2.printDataCar();
    }
}

```

- Como podemos ver, siendo nuestro driver del tipo Account debemos declararlo para que se cree un objeto.

Con esto tenemos los datos mínimos necesarios para que un vehículo exista dentro de nuestra aplicación Uber.

#### JAVASCRIPT

Primero debemos crear un archivo HTML llamado index que funcionara como nuestro Main, esto es porque JavaScript necesita un navegador que nos permita visualizar todo.

En nuestro archivo index.html tendremos lo siguiente:

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Programación Orientada a Objetos en JavaScript</title>
</head>
<body>

    <script src="Account.js"></script>
    <script src="Car.js"></script>
    <script src="index.js"></script>

</body>
</html>

```

Es un esqueleto básico del HTML, lo importante está dentro del body donde tenemos tres scripts que nos permitirá incluir o llamar nuestro código de tipo JavaScript. Es importante ponerlos en ese orden porque de lo contrario no nos funcionara, esto es porque JavaScript empieza a compilar y renderizar desde el principio a medida que va leyendo línea por línea.

Ahora vamos a nuestra clase Car:

```

function Car(license, driver) {
    this.id
    this.license = license
    this.driver = driver
    this.passenger
}

Car.prototype.printDataCar = function () {
    console.log(`License: ${this.license} Name Driver: ${this.driver.name}`)
}

```

- Como podemos ver, no es necesitamos colocar el tipo de dato que tendrán los parámetros. Además los atributos que creamos ya son tomados como parte del método constructor y simplemente debemos igualar.
- Es en la creación del método **printDataCar** donde está lo interesante. Sabemos que JavaScript toma los métodos como funciones especiales y son declarados fuera.
- Para imprimir nuestro resultado usamos el comando **console.log** que imprime en la consola.
- Una forma de concatenar string es la interpolación de cadenas y utilizan las comillas invertidas o backticks para delimitar sus contenidos. Las partes variables se denominan placeholders y utilizan la sintaxis \${ } para diferenciarse del resto de la cadena.

Ahora vamos a nuestra clase Account para pasar el nombre y el documento:

```

function Account(name, document) {
    this.id
    this.name = name
    this.document = document
    this.email
    this.password
}

```

¿Recuerdas a index.js? Hasta el momento ha permanecido vacío, pero lo usaremos para poner ahí todas las llamadas de nuestras clases y posteriormente mostrarlas. Entonces, vamos a index.js:

```
1 var car = new Car("AW456", new Account("Andres Herrera", "AND123"))
2 car.passenger = 4
3 car.printDataCar()
```

- Declaramos un objeto `car` al que pasaremos los datos de la matrícula y del conductor.
- Incluso podemos usar el operador punto para acceder a la cantidad de pasajeros y usar nuestro método `printDataCar`.

Finalmente, para ver nuestros resultados, debemos abrir nuestro archivo HTML en un navegador. En el navegador damos click derecho, inspeccionar y vamos a consola, y ahí está imprimido en pantalla nuestra clase de hoy.

### Clase 23 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:

```
1 class Car {
2
3     constructor(license, driver) {
4         this.id;
5         this.license = license;
6         this.driver = driver;
7         this.passenger;
8     }
9
10    printDataCar() {
11        console.log(this.driver)
12        console.log(this.driver.name)
13        console.log(this.driver.document)
14    }
15
16 }
17 }
```



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

Ahora veamos a la clase `Account`:

```
1 class Account {
2
3     constructor(name, document) {
4         this.id;
5         this.name = name;
6         this.document = document;
7         this.email;
8         this.password;
9     }
10
11 }
12 }
```



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

```
1 class Route {
2     constructor(){
3         this.id;
4         this.init;
5         this.end;
6     }
7 }
```



```
1 class Payment {
2     constructor() {
3         this.id;
4     }
5
6 }
```



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

```
1 var car = new Car("AW456", new Account("Andres Herrera", "QWE234"))
2 car.passenger = 4;
3 car.printDataCar();
4
5
```



Y los resultados serán los mismos:

The screenshot shows the browser's developer tools with the 'Console' tab selected. The output in the console is as follows:

```
▶ Account {name: "Andres Herrera", document: "QWE234"}
Andres Herrera
QWE234
> |
```



#### Clase 24 Declarando un Método Constructor en Python

En Python encontrarás un concepto denominado Métodos Mágicos, estos métodos son llamados automáticamente y estrictamente bajo ciertas reglas. El método constructor en Python forma parte de esta familia de métodos y como aprendimos en la clase anterior lo declaramos usando `__init__`, aunque si nos ponemos estrictos este método no construye el objeto en sí. El encargado de hacer esto es `new` y el método `__init__` se encargará de personalizar la instanciación de la clase, esto significa que lo que esté dentro de `__init__` será lo primero que se ejecute cuando se cree un objeto de esta clase.

Para nuestro proyecto tenemos la necesidad de que algunas variables se inicialicen obligatoriamente cuando ocurra la instanciación. Así que declaremos el método `__init__` en las clases de nuestro proyecto con las propiedades obligatorias.

Para la clase Account quedaría algo así, notarás que usamos la palabra clave `self`, esta es muy parecida a lo que venimos trabajando a otros lenguajes con `this`. Y como su nombre lo dice hace referencia a los datos que componen la clase, en este caso `self.name` está llamando al atributo `name` que se encuentra en la línea 3 de la clase y, le está asignando el dato que se pasa en el método `__init__` de la línea 8.

```
1 class Account:
2     id          = int
3     name        = str
4     document    = str
5     email       = str
6     password    = str
7
8     def __init__(self, name, document):
9         self.name      = name
10        self.document = document
```



Ahora veamos la clase Car:

```

1  from account import Account
2
3  class Car:
4      id          = int
5      license     = str
6      driver      = Account("", "")
7      passenger   = int
8
9      def __init__(self, license, driver):
10         self.license    = license
11         self.driver     = driver

```



Lo que notarás de diferente es que cambiamos el tipo de dato de driver, ahora es de tipo Account y como ves está solicitando los dos datos obligatorios para instanciar un objeto de este tipo. Esto lo verás más en acción en el próximo fragmento de código del archivo main.py. Además, mucho ojo, en la primera línea observamos que es importante importar la clase para poderla usar.

Nuestro archivo main.py ahora se verá así:

```

1  from car import Car
2  from account import Account
3
4  if __name__ == "__main__":
5      print("Hola Mundo")
6
7      car = Car("AMS234", Account("Andres Herrera", "ANDA876"))
8      print(vars(car))
9      print(vars(car.driver))
10
11

```



Observa que estamos importando las dos clases que usaremos y las estamos instanciando en los métodos constructores.

Los resultados serán los siguientes:



El código de este ejemplo lo encuentras en este [enlace](#).

#### Reto 3

- Ahora que ya viste cómo creamos un método constructor en Python, mira esta [clase](#) y hazlo también para PHP. Compártenos tus resultados en la sección de discusiones.

#### Modulo 5. Herencia

##### Clase 25 Aplicando herencia en lenguaje Java y PHP

Ya sabemos cómo funciona la herencia de manera conceptual, pero aún lo hemos visto expresada en código dependiendo del lenguaje de programación que elijas. Y es que esto varía dependiendo de lo que estés eligiendo:

## Java

```
class Student extends Person
```

## Python

```
class Student(Person):
```

## JavaScript



```
student.prototype = new Person();
```

## PHP

```
class Student extends Person
```

- **Java:** Para crear una subclase se usa la palabra reservada `extends`, esto le indica a la clase hija cual va a ser su clase padre.

```
class ClaseHija extends ClasePadre
```

- **Python:** Usamos la palabra `class` seguido del nombre de la clase hija, se la pone entre paréntesis pasamos la clase padre como parámetro.

```
class ClaseHija (ClasePadre):
```

- **JavaScript:** Ha sido nuestro amigo rebelde durante todo el curso, por lo que no es de extrañar que JavaScript herede de una manera peculiar y es que simplemente toma a la clase hija seguido de la palabra `prototype` e inmediatamente instancia la clase padre.

```
claseHija.prototype = new ClasePadre()
```

- **PHP:** Esté lenguaje maneja la herencia de manera similar a Java, usando la palabra `extends`.

## JAVA

Siguiendo con la clase Car es tiempo de crear los objetos que descienden de ésté, para eso creamos cuatro archivos que serán nuestras clases UberX, UberPool, UberBlack y UberVan.

UberX y UberPool

```
public class UberX extends Car {  
    String brand;  
    String model;  
  
    public UberX(String license, Account driver, String brand, String model) {  
        super(license, driver);  
        this.brand = brand;  
        this.model = model;  
    }  
}
```

```
public class UberPool extends Car {  
    String brand;  
    String model;  
  
    public UberPool(String license, Account driver, String brand, String model) {  
        super(license, driver);  
        this.brand = brand;  
        this.model = model;  
    }  
}
```

- Creamos las clases y usamos `extends` para señalar que estamos heredando de la clase Car, creamos los atributos propios y después creamos un constructor que reciba cuatro parámetros.
- `super()`: sirve para llamar al constructor de la clase padre.

UberBack y UberVan

```
import java.util.ArrayList;  
import java.util.Map;  
  
public class UberBlack extends Car {  
    Map<String, Map<String, Integer>> typeCarAccepted;  
    ArrayList<String> seatsMaterial;  
  
    public UberBlack(String license, Account driver,  
                    Map<String, Map<String, Integer>> typeCarAccepted, ArrayList<String> seatsMaterial) {  
        super(license, driver);  
        this.typeCarAccepted = typeCarAccepted;  
        this.seatsMaterial = seatsMaterial;  
    }  
}
```

```
import java.util.ArrayList;  
import java.util.Map;  
  
public class UberVan extends Car {  
    Map<String, Map<String, Integer>> typeCarAccepted;  
    ArrayList<String> seatsMaterial;  
  
    public UberVan(String license, Account driver,  
                  Map<String, Map<String, Integer>> typeCarAccepted, ArrayList<String> seatsMaterial) {  
        super(license, driver);  
        this.typeCarAccepted = typeCarAccepted;  
        this.seatsMaterial = seatsMaterial;  
    }  
}
```

- Importamos el `ArrayList` y `Map` de la biblioteca `java.util`.
- `Map`: Es una interfaz que define el conducto general de una estructura que se hace relación de clave/valor.
- Creamos las clases y usamos `extends` para señalar que estamos heredando de la clase Car, creamos los atributos propios y después creamos un constructor que reciba cuatro parámetros.
- `super()`: sirve para llamar al constructor de la clase padre.

## PHP

Al igual que en Java, también creamos cuatro archivos nuevos que serán nuestras clases.

Tenemos nuestra clase Car:

```
<?php
require_once('account.php');

class Car {
    public $id;
    public $license;
    public $driver;
    public $passenger;

    public function __construct($license, $driver){
        $this->license = $license;
        $this->driver = $driver;
    }

    public function printDataCar() {
        echo "Licencia: $this->license Driver: ".$this->driver->name;
    }
}
```

Y ahora crearemos una clase UberX:

```
<?php
require_once('car.php');

class UberX extends Car {
    public $brand;
    public $model;

    public function __construct($license, $driver, $brand, $model){
        parent::__construct($license,$driver);
        $this->brand = $brand;
        $this->model = $model;
    }
}
```

- **require\_once**: Inserta en nuestro programa un código procedente de otro archivo, si el archivo no existe o contiene errores, nuestro programa no funcionará y obtendremos un fatal error.
- **extends**: Indicamos a la clase hija cual va a ser su clase padre.
- Los constructores padres no son llamados implícitamente si la clase hija define un constructor. Para ejecutar un constructor padre, se requiere invocar a **parent::\_\_construct()** desde el constructor hijo. Si el hijo no define un constructor, entonces se puede heredar de la clase madre como un método de clase normal (si no fue declarada como privada).

#### Reto 4

- Termina las clases de PHP en Uber.

#### Clase 26 Solución del reto de herencia en PHP

En la clase anterior nos quedamos con el reto de terminar la composición de las demás clases que heredan de Car, ahora vamos a ver lo que debimos haber hecho.

UberPool:

```
<?php
require_once('car.php');

class UberPool extends Car {
    public $brand;
    public $model;

    public function __construct($license, $driver, $brand, $model){
        parent::__construct($license,$driver);
        $this->brand = $brand;
        $this->model = $model;
    }
?>
```

- **require\_once**: Inserta en nuestro programa un código procedente de otro archivo, si el archivo no existe o contiene errores, nuestro programa no funcionará y obtendremos un fatal error.
- **extends**: Indicamos a la clase hija cual va a ser su clase padre.
- Los constructores padres no son llamados implícitamente si la clase hija define un constructor. Para ejecutar un constructor padre, se requiere invocar a **parent::\_\_construct()** desde el constructor hijo. Si el hijo no define un constructor, entonces se puede heredar de la clase madre como un método de clase normal (si no fue declarada como privada).

Y, las clases UberBlack y UberVan:

```
<?php
require_once('car.php');

class UberBlack extends Car {
    public $typeCarAccepted;
    public $seatsMaterial;

    public function __construct($license, $driver, $typeCarAccepted, $seatsMaterial){
        parent::__construct($license,$driver);
        $this->typeCarAccepted = $typeCarAccepted;
        $this->seatsMaterial = $seatsMaterial;
    }
?>
```

```
<?php
require_once('car.php');

class UberVan extends Car {
    public $typeCarAccepted;
    public $seatsMaterial;

    public function __construct($license, $driver, $typeCarAccepted, $seatsMaterial){
        parent::__construct($license,$driver);
        $this->typeCarAccepted = $typeCarAccepted;
        $this->seatsMaterial = $seatsMaterial;
    }
?>
```

- En el caso de UberBlack y UberVan existe una ventaja de que PHP no es estrictamente tipado, por lo que, a diferencia de Java en donde tuvimos que escribir toda la composición de la variable, en esta ocasión no es necesario.

Ahora vamos a probar nuestro código:

```

<?php
require_once("car.php");
require_once("uberX.php");
require_once("uberPool.php");
require_once("account.php");

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

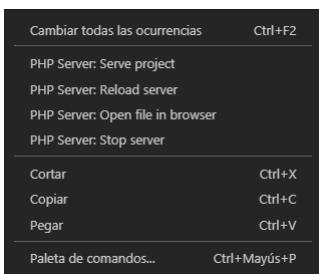
$uberPool = new UberPool("TYU567", new Account("Mary Blackstone", "QWE753"), "Dodge", "Attitude");
$uberPool->printDataCar();

```

- Usando `require_once()` traemos las clases Car, UberX, UberPool y Account.
- El símbolo \$ seguido de un nombre representan a nuestras variables.
  - `$uberX, $uberPool`: Son los objetos de tipo UberX y UberPool respectivamente.
- Entre las paréntesis van las variables que aceptan cada clase
  - Primero va license (matrícula), seguido del objeto Account que aceptan name (nombre) y document (documento), después va el brand (marca) del vehículo y finalmente el model (modelo).
- La sintaxis para llamar a un método en PHP es: `Variable -> método()`.

*Nota: Todavía no hemos ejemplificado en código la herencia de la clase Driver, pero con lo aprendido a estas alturas serías capaz de hacerlo por ti mismo.*

Finalmente para ejecutar nuestro código damos click derecho en algún lugar del editor y vamos en PHP Server: Serve project para arrancar un servidor.



Una vez hecho eso nos aparecerá un navegador con los datos imprimidos.

Licencia: CVB123 Driver: Andres Herrera Licencia: TYU567 Driver: Mary Blackstone

Como vez, todavía no imprimimos los datos de la marca y modelo, pero eso lo estaremos viendo más adelante.

### Clase 27 Aplicando herencia en lenguaje Python y JavaScript

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
- UberPool.py
- UberBlack.py
- UberVan.py

```

1  from car import Car
2
3  class UberX(Car):
4      brand = str
5      model = str
6
7      def __init__(self, license, driver, brand, model):
8          super().__init__(license, driver)
9          self.brand = brand
10         self.model = model
11

```



```

1  from car import Car
2
3  class UberPool(Car):
4      brand = str
5      model = str
6
7
8      def __init__(self, license, driver, brand, model):
9          super().__init__(license, driver)
10         self.brand = brand
11         self.model = model
12

```



```
1 from car import Car
2
3 class UberBlack(Car):
4     typeCarAccepted = []
5     seatsMaterial  = []
6
7     def __init__(self, license, driver, typeCarAccepted, seatsMaterial):
8         super().__init__(license, driver)
9         self.typeCarAccepted = typeCarAccepted
10        self.seatsMaterial = seatsMaterial
11
```



```
1 from car import Car
2
3 class UberVan(Car):
4     typeCarAccepted = []
5     seatsMaterial  = []
6
7     def __init__(self, license, driver, typeCarAccepted, seatsMaterial):
8         super().__init__(license, driver)
9         self.typeCarAccepted = typeCarAccepted
10        self.seatsMaterial = seatsMaterial
11
```



El código completo puedes verlo [aquí](#)

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.

```
1 class UberX extends Car {
2     constructor(license, driver, brand, model) {
3         super(license, driver)
4         this.brand = brand;
5         this.model = model;
6     }
7 }
```



```
1 class UberPool extends Car {
2     constructor(license, driver, brand, model) {
3         super(license, driver)
4         this.brand = brand;
5         this.model = model;
6     }
7 }
```



```

1 class UberBlack extends Car {
2     constructor(license, driver, typeCarAccepted, seatsMaterial) {
3         super(license, driver)
4         this.typeCarAccepted = typeCarAccepted;
5         this.seatsMaterial = seatsMaterial;
6     }
7 }
```



```

1 class UberVan extends Car {
2     constructor(license, driver, typeCarAccepted, seatsMaterial) {
3         super(license, driver)
4         this.typeCarAccepted = typeCarAccepted;
5         this.seatsMaterial = seatsMaterial;
6     }
7 }
```



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.

```

12 </body>
13 <script src="Account.js"></script>
14 <script src="Car.js"></script>
15 <script src="UberX.js"></script>
16 <script src="index.js"></script>
17 </html>
```



Nuestro ejemplo se verá así:

```

1 var car = new Car("AW456", new Account("Andres Herrera", "QWE234"))
2 car.passenger = 4;
3 car.printDataCar();
4
5 var uberX = new UberX("AW456", new Account("Andrea Ferran", "ANDA765"), "Ch")
6 uberX.passenger = 4;
7 uberX.printDataCar();
8
```

El código completo puedes verlo [aquí](#)

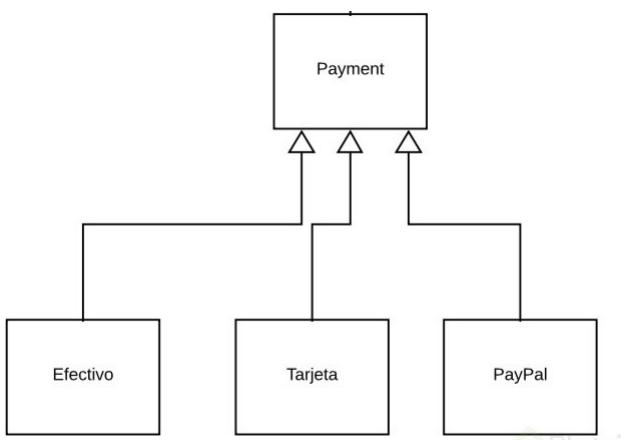
#### Clase 28 Otros tipos de Herencia

A partir de ahora las clases que estén siendo heredadas 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 necesitaremos 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.

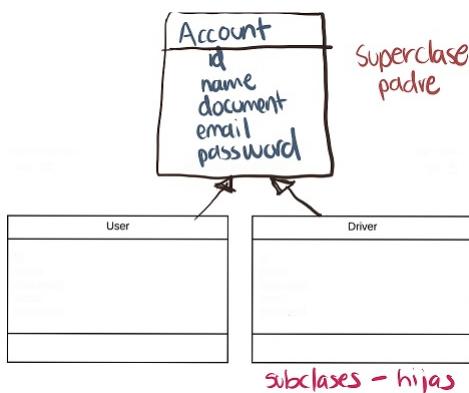
Ahora nos faltarán crear las clases y aplicar su herencia.

## Clase 29 Reto 4

Nos queda la Jerarquía Account pendiente.

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

Compartenos tus resultados.



## Modulo 6. Encapsulamiento

### Clase 30 Encapsulamiento

Ya casi estamos terminando nuestro proyecto y estamos dando los últimos detalles que claro no son menos importantes porque aquí estaremos viendo las restricciones que tendrá cada clase.

Recordemos nuestra clase Car ahora añadiremos para imprimir la cantidad de pasajeros:

```
class Car {  
    Integer id;  
    String license;  
    Account driver;  
    Integer passenger;  
  
    public Car(String license, Account driver) {  
        this.license = license;  
        this.driver = driver;  
    }  
  
    void printDataCar() {  
        System.out.println("License: " + license + " Name Drive: " + driver.name + " Passengers: " + passenger);  
    }  
}
```

Mientras que el archivo Main definimos una variable de tipo UberX:

```
class Main {  
    Run | Debug  
    public static void main(String[] args) {  
        System.out.println("Información");  
        UberX uberX = new UberX("AMQ123", new Driver("Andres Herrera", "AND123"), "Chevrolet", "Sonic");  
        uberX.passenger = 4;  
        uberX.printDataCar();  
    }  
}
```

*RECUERDA: UberX hereda de Car, por lo que tiene todos sus métodos y atributos.*

¿Qué pasa si ejecutamos el programa?

Pues no imprimirá la licencia, el nombre del conductor y la cantidad de pasajeros que puede llevar. ¿Y si cambiamos passenger de 4 a 3? Pues nada extraños, nos seguirá imprimiendo todo perfectamente.

Pero si lo meditamos un poco, las reglas de Uber dicen que los vehículos de categoría UberX deben tener al menos cuatro pasajeros sin contar al conductor, es decir, aceptan cinco personas dentro pero cuatro lugares deben quedar disponibles para ese vehículo. Entonces, si colocamos que passenger sea igual a tres generará una inconsistencia en nuestros datos. Una inconsistencia que podemos prevenir al hacer que nadie pueda alterar ese parámetro.

Y precisamente de eso trata nuestra clase de hoy: no alterar un parámetro, que nadie más tenga acceso, y la única forma que tenemos en Java (y en la programación orientada a objetos) será escondiendo ese parámetro, dejarlo invisible o al menos invalidado para los demás implementando la encapsulación.

#### Encapsulamiento

Este concepto consiste en la ocultación del estado o de los datos miembro de un objeto, de forma que sólo es posible modificarlos mediante los métodos definidos para dicho objeto. Es decir, limitamos el acceso a las variables de nuestras clases.

Y es justamente eso lo que nosotros deseamos hacer, esconder el atributo passenger para que no pueda ser alterado o que al menos no le pongan datos que ni siquiera tengan que ver con la lógica de nuestro negocio.

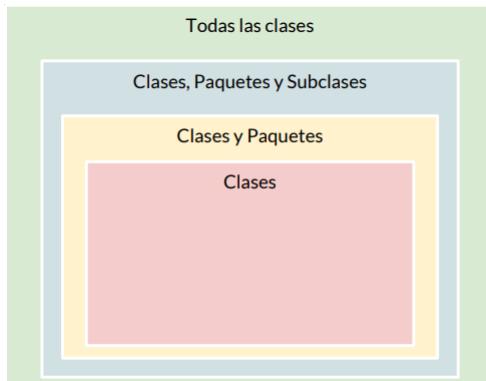
#### ¿Cómo se encapsulan los datos?

Cuando realizamos una abstracción en una clase para luego instanciarla y crear un objeto no se necesita conocer a fondo la implementación solo se necesita poder instanciar esa clase tampoco necesita conocer todas las propiedades de un objeto o acceder a ellas de forma directa, por ello podemos crear diferentes métodos y forzar a utilizar métodos definidos para modificar estas propiedades.

Para realizar el proceso anterior se necesita conocer los modificadores de acceso. Los modificadores de acceso permiten dar un nivel de seguridad mayor a nuestras aplicaciones restringiendo el acceso a diferentes atributos, métodos, constructores asegurándonos que el usuario deba seguir una "ruta" especificada por nosotros para acceder a la información.

Es muy posible que nuestras aplicaciones vayan a ser usadas por otros programadores o usuarios con cierto nivel de experiencia; haciendo uso de los modificadores de acceso podremos asegurarnos de que un valor no será modificado incorrectamente por parte de otro programador o usuario. Generalmente el acceso a los atributos se consigue por medio de los métodos get y set, pues es estrictamente necesario que los atributos de una clase sean privados.

Teniendo en cuenta la siguiente imagen:



- El recuadro verde hace referencia al modificador de acceso público (**public**). Es el más permisivo de todos, esto quiere decir que si el componente de una clase es public, tendremos acceso a él desde cualquier clase o instancia sin importar el paquete o procedencia de ésta.
- El recuadro azul hace referencia al modificador de acceso protegido (**protected**). Nos permite acceso a los componentes con dicho modificador desde la misma clase, clases del mismo paquete y clases que hereden de ella (incluso en diferentes paquetes).
- El recuadro amarillo hace referencia al modificador de acceso por defecto (**default**). Java nos da la opción de no usar un modificador de acceso y al no hacerlo, el elemento tendrá un acceso conocido como default, acceso por defecto, que permite que tanto la propia clase como las clases del mismo paquete accedan a dichos componentes (de aquí la importancia de declararle siempre un paquete a nuestras clases).
- El recuadro rojo hace referencia al modificador de acceso privado (**private**). Es el más restrictivo de todos, básicamente cualquier elemento de una clase que sea privado puede ser accedido únicamente por la misma clase por nada más. Es decir, si por ejemplo, un atributo es privado solo puede ser accedido por los métodos o constructores de la misma clase. Ninguna otra clase sin importar la relación que tengan podrá tener acceso a ellos.

### Clase 31 Encapsulando atributos en Java

Ahora que ya entendimos sobre encapsulamiento y cuáles son los datos a encapsular vamos a hacerlo en nuestras clases.

Estamos en nuestro proyecto y quedamos que passenger sea validado, el hecho de que tuviera 3 lugares disponibles no es algo que va con la regla de nuestro negocio. Para eso vamos a arreglar esto poniendo un modificador de acceso en la clase Car. ¿Por qué la clase Car? Porque ahí precisamente es donde se encuentra nuestro atributo passenger y actualmente se ve así:

```
class Car {
    Integer id;
    String license;
    Account driver;
    Integer passenger;

    public Car(String license, Account driver) {
        this.license = license;
        this.driver = driver;
    }

    void printDataCar() {
        System.out.println("License: " + license + " Name Drive: " + driver.name + " Passengers: " + passenger);
    }
}
```

El atributo passenger no tiene ningún modificador, es decir, su acceso es default. Podemos escribir **public Integer passenger;** para que sea de acceso público, pero nosotros queremos esconderlo y que sea accesible únicamente para la clase, por lo que su modificador será private.

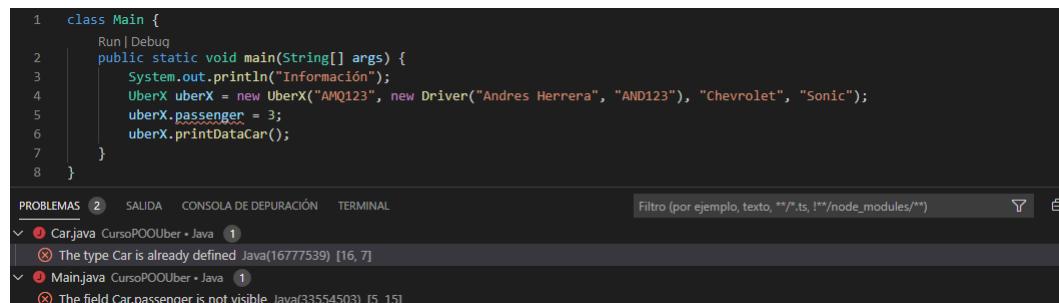
Entonces quedará de este modo:

```
class Car {
    Integer id;
    String license;
    Account driver;
    private Integer passenger;

    public Car(String license, Account driver) {
        this.license = license;
        this.driver = driver;
    }

    void printDataCar() {
        System.out.println("License: " + license + " Name Drive: " + driver.name + " Passengers: " + passenger);
    }
}
```

Pero si lo ejecutamos nos saldrá error:



Y eso se a que el atributo passenger ya no se encuentra visible para la clase Main, pero si continua siendo visible para la clase Car. Por eso, si nosotros queremos, podemos imprimir passenger desde el constructor de la siguiente forma:

```
class Car {
    Integer id;
    String license;
    Account driver;
    private Integer passenger;

    public Car(String license, Account driver) {
        this.license = license;
        this.driver = driver;
        passenger = 3;
        System.out.println("Passenger: " + passenger);
    }

    void printDataCar() {
        System.out.println("License: " + license + " Name Drive: " + driver.name + " Passengers: " + passenger);
    }
}

class Main {
    Run | Debug
    public static void main(String[] args) {
        System.out.println("Información");
        UberX uberX = new UberX("AMQ123", new Driver("Andres Herrera", "AND123"), "Chevrolet", "Sonic");
        //uberX.passenger = 3;
        uberX.printDataCar();
    }
}
```

Los que hicimos fue en la clase Car, dentro del constructor, darle al atributo passenger un valor de 3 (asientos disponibles) y debajo imprimimos en pantalla. Mientras que en la clase Main simplemente eliminamos o comentamos nuestro acceso al atributo passenger. Si esto lo ejecutamos podemos ver cómo nos corre perfectamente, sin embargo, si lo intentamos correr directamente a través de Main nos dará error porque el atributo passenger es privado y solo visible dentro de su clase.

Ahora vamos a darle un poco de forma y sentido. ¿Por qué lo dejamos accesible dentro de la clase? Porque necesitamos validarla, lo que nosotros deseamos es que cuando alguien quiera ingresar los datos sobre la cantidad de asientos disponibles definitivamente debe ser de cuatro para los de tipo UberX.

Por eso, y para acceder a los datos privados, usaremos métodos especiales.

Los métodos get y set, son simples métodos que usamos en las clases para mostrar (get) o modificar (set) el valor de un atributo. El nombre del método siempre será get o set y a continuación el nombre del atributo, su modificador siempre es public ya que queremos mostrar o modificar desde fuera la clase.

Agregando los métodos get y set a nuestro programa quedaría así:

```

class Car {
    Integer id;
    String license;
    Account driver;
    private Integer passenger;

    public Car(String license, Account driver) {
        this.license = license;
        this.driver = driver;
    }

    void printDataCar() {
        System.out.println("License: " + license + " Name Drive: " + driver.name + " Passengers: " + passenger);
    }

    public Integer getPassenger() {
        return passenger;
    }

    //No devolvemos nada, solo estamos asignando los datos
    public void setPassenger(Integer passenger) {
        this.passenger = passenger;
    }
}

```

```

class Main {
    Run | Debug
    public static void main(String[] args) {
        System.out.println("Información");
        UberX uberX = new UberX("AMQ123", new Driver("Andres Herrera", "AND123"), "Chevrolet", "Sonic");
        uberX.setPassenger(4);
        uberX.printDataCar();
    }
}

```

Creamos los métodos getPassenger y setPassenger.

- **getPassenger:** Retorna nuestro atributo passenger.
- **setPassenger:** Asigna el valor similar a como lo hicimos dentro del constructor.

Con esos dos métodos creados, en la clase Main podemos acceder a setPassenger y por parámetro enviar el valor para passenger. Y si este código lo ejecutamos nos seguirá corriendo perfectamente. Podríamos pensar que es exactamente lo mismo solo que cambiamos la variable por un método, pero no es así. Ahora seremos capaces de validar los valores enviados a passenger para que no permitir que ningún otro valor diferente a cuatro sea agregado o incluso enviar un parámetro vacío.

```

class Car {
    Integer id;
    String license;
    Account driver;
    private Integer passenger;

    public Car(String license, Account driver) {
        this.license = license;
        this.driver = driver;
    }

    void printDataCar() {
        System.out.println("License: " + license + " Name Drive: " + driver.name + " Passengers: " + passenger);
    }

    public Integer getPassenger() {
        return passenger;
    }

    public void setPassenger(Integer passenger) {
        if(passenger == 4) {
            this.passenger = passenger;
        } else {
            System.out.println("Error, necesitas asignar 4 pasajeros");
        }
    }
}

```

Dentro del método setPassenger validamos para que el valor del atributo passenger siempre sea cuatro, en caso de que se ingrese otro, entonces nos marcará error. Con esto ya todos los futuros conductores estarán obligados a poner cuatro asientos disponibles para que les funcione la aplicación.

Incluso podríamos validar los datos dentro de printDataCar para que todos atributo sean diferente a null, es decir, que siempre tengan un valor.

## Modulo 7. Polimorfismo

### Clase 32. 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

Estamos llegando casi al punto final de nuestro proyecto y es momento de ver una de las partes más importantes, una pieza fundamental para nuestro proyecto, que es el polimorfismo.

#### Polimorfismo

Viene de «Poli» que significa mucho y «Morfismo» que significa formas, es decir, muchas formas.

*Observación: No es Poliformismo como algunos lo llaman.*

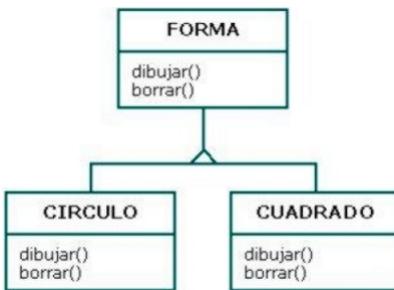
El polimorfismo es la capacidad que tienen los objetos de una clase en ofrecer respuesta distinta e independiente en función de los parámetros (diferentes implementaciones) utilizados durante su invocación. Dicho de otro modo el objeto como entidad puede contener valores de diferentes tipos durante la ejecución del programa.

En JAVA el término polimorfismo también suele definirse como 'Sobrecarga de parámetros', que así de pronto no suena tan divertido pero como veremos más adelante induce a cierta confusión. En realidad suele confundirse con el tipo de poliformismo más común, pero no es del todo exacto usar esta denominación.

Por lo general diremos que existen 3 tipos de polimorfismo:

- **Sobrecarga:** El más conocido y se aplica cuando existen funciones con el mismo nombre en clases que son completamente independientes una de la otra.
- **Paramétrico:** Existen funciones con el mismo nombre pero se usan diferentes parámetros (nombre o tipo). Se selecciona el método dependiendo del tipo de datos que se envíe.
- **Inclusión:** Es cuando se puede llamar a un método sin tener que conocer su tipo, así no se toma en cuenta los detalles de las clases especializadas, utilizando una interfaz común.

Un ejemplo clásico que podemos ver es esta:



En la que tenemos una jerarquía de clases donde nuestra clase padre se llama Forma y sus clases hijas son Circulo y Cuadrado.

Las clases pueden compartir atributos y métodos, como ya hemos aprendido en clases anteriores, pero en este caso particular el Circulo va a dibujar de una forma distinta al Cuadrado. Esto es un comportamiento diferente que tiene el Circulo con respecto al Cuadrado, pero ese método dibujar proviene de la clase padre por lo que puede que se le haya dado un comportamiento por defecto o incluso puede que solo se haya dejado en blanco para que cada quien implemente el comportamiento que así lo desea, esto específicamente es lo que llamamos polimorfismo es donde tenemos un método que se comparte entre clases y cada una de esas clases le da el comportamiento que necesita o que desea.

Ahora vamos a nuestro proyecto y analicemos un momento. Ya hemos encapsulado y validado la variable passenger para que acepte únicamente el valor de 4, y entendemos que el método setPassenger puede variar dependiendo de cada clase. En este caso las clases UberX, UberPool y UberBlack aceptaran cuatro como la cantidad de asientos disponibles, pero con UberVan nos surge la necesidad de validar el dato a seis lugares disponibles, esa es la condición que nos pone Uber para tener un auto de tipo UberVan en la plataforma. Entonces es ahí, en UberVan, donde tenemos un comportamiento diferente.

El comportamiento involucra la asignación del dato, la validación del dato para seis en lugar de cuatro, y aquí ya estamos viendo impregnado el polimorfismo.

Vamos en nuestra clase Car:

```

class Car {
    private Integer id;
    private String license;
    private Account driver;
    protected Integer passenger;

    public Car(String license, Account driver){
        this.license = license;
        this.driver = driver;
    }

    void printDataCar() {
        if(passenger != null){
            System.out.println("License: " + license + " Name Driver: " + driver.name + " Passengers: " + passenger);
        }
    }

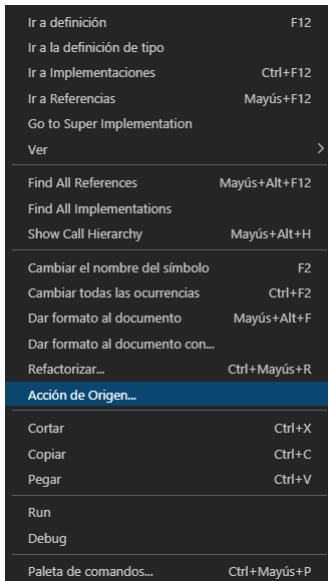
    public Integer getPassenger(){
        return passenger;
    }

    public void setPassenger(Integer passenger) {
        if(passenger == 4){
            this.passenger = passenger;
        }else{
            System.out.println("Necesitas asignar 4 pasajeros");
        }
    }
}

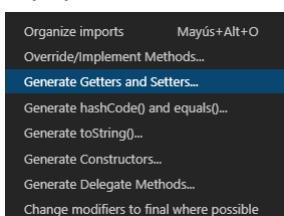
```

Encapsulamos todos nuestros demás datos, además de cambiar el modificador de acceso de Passenger y lo volvemos protected para que pueda ser usada por las subclases.

Después, con un click derecho, seleccionamos Source Action o Acción de Origen:



Y luego elegimos «Generate Getters and Setters»:



Esto nos trae automáticamente todos los getters y setters de todos los atributos que encapsulamos.

Entonces, la validación para que los asientos disponibles sea 4 ya queda heredado para las clases Car, UberX, UberPool y UberBlack, pero en el caso de UberVan es diferentes. Creemos un objeto de tipo UberVan y veamos que sucede.

The screenshot shows the Visual Studio Code interface. On the left is the code editor with Java code. At the top, there are tabs for PROBLEMAS, SALIDA, CONSOLA DE DEPURACIÓN, and TERMINAL. The TERMINAL tab is active, showing the command line output. The output shows the execution of a Java program named Main, which creates instances of UberX and UberVan and prints their data. The terminal window has a title bar "3: Java Process Console".

```
1 class Main {
2     Run | Debug
3     public static void main(String[] args) {
4         System.out.println("Información");
5         UberX uberX = new UberX("AMQ123", new Driver("Andres Herrera", "AND123"), "Chevrolet", "Sonic");
6         uberX.setPassenger(4);
7         uberX.printDataCar();
8
9         UberVan uberVan = new UberVan("TYU987", new Driver("Cleo Ramirez", "AND123"));
10        uberVan.setPassenger(6);
11        uberVan.printDataCar();
12    }
13 }
```

```
Microsoft Windows [Versión 6.3.9600]
(c) 2013 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Silvana\Desktop\CursoPOOUber> c:\Users\Silvana\.vscode\extensions\vscode-
va\vscode-java-debug-0.29.0\scripts\launcher.bat "C:\Program Files\Java\jdk-15.0
.1\bin\java.exe" --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -Dfile.encoding=UTF-8 -cp C:\Users\Silvana\AppData\Roaming\Code\User\workspaceStorage\b4c80fb4c4006fe7f8ba50067f56fdc\redhat.java\jdt_ws\CursoPOOuber_197938db\bin Main
Información

License: AMQ123 Name Driver: Andres Herrera Passengers: 4
Necesitas asignar 4 pasajeros

C:\Users\Silvana\Desktop\CursoPOOUber>[
```

Por el bien del ejemplo hemos cambiado el constructor de UberVan por uno más sencillo para poder manipular mejor los datos, pero igualmente podemos ver cuando enviamos 6 como la cantidad que debe tener passenger no nos imprime ningún resultado, incluso cuando imprimimos nuestro objeto UberX, y nos dice que debemos poner 4 pasajero lo cual no es el dato que nosotros queremos que tenga.

Para arreglar esto vamos a la clase UberVan:

```
import java.util.ArrayList;
import java.util.Map;

class UberVan extends Car {
    Map<String, Map<String, Integer>> typeCarAccepted;
    ArrayList<String> seatsMaterial;

    public UberVan(String license, Account driver){
        super(license, driver);
    }

    @Override
    public void setPassenger(Integer passenger) {
        if(passenger == 6){
            this.passenger = passenger;
        } else{
            System.out.println("Necesitas asignar 6 pasajeros");
        }
    }
}
```

¿Para qué sirve @Override?

Pues no sirve para nada. El uso de la anotación @Override es opcional.

Para lo único que realmente sirve es a modo de documentación:

- Un método marcado con @Override debería estar sobreescritiendo un método de alguna clase padre.
- Si esto no es así, el compilador de java generará un error

En este caso nosotros estamos trayendo el método setPassenger y usamos el polimorfismo de sobrecarga para cambiar la validación, es decir, sobre escribimos sobre el método para que tenga un nuevo comportamiento. En este caso, nuestra validación pasa de aceptar 4 a aceptar 6.

También podemos usarlo en UberX:

```
public class UberX extends Car {
    String brand;
    String model;

    public UberX(String license, Account driver, String brand, String model) {
        super(license, driver);
        this.brand = brand;
        this.model = model;
    }

    @Override
    void printDataCar() {
        super.printDataCar();
        System.out.println("Modelo: " + model + " Marca: " + brand);
    }
}
```

En el caso de UberX nosotros traemos al método printDataCar y, aparte de decirle super.printDataCar() que nos imprima lo normal (licencia, nombre y documento), también hacemos que nos imprima la marca y el modelo del vehículo.

Ahora, si ejecutamos el código, podemos ver cómo nos imprime perfectamente tanto los datos del UberX como del UberVan:

```

1  class Main {
2      Run | Debug
3      public static void main(String[] args) {
4          System.out.println("Información");
5          UberX uberX = new UberX("AMQ123", new Driver("Andres Herrera", "AND123"), "Chevrolet", "Sonic");
6          uberX.setPassenger(4);
7          uberX.printDataCar();
8
9          UberVan uberVan = new UberVan("TYU987", new Driver("Cleo Ramirez", "AND123"));
10         uberVan.setPassenger(6);
11         uberVan.printDataCar();
12     }
13 }

```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL

3: Java Process Console + ☰ 🗑️ ⌂ ⌄ X

```

.C:\bin\java.exe" --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -Dfile.encoding=UTF-8 -cp C:\Users\Silvana\AppData\Roaming\Code\User\workspaceStorage\b4c80fb4c4006fe7f8ba50067f56fdc\redhat.java\jdt_ws\CursoPOOUber\197938db\bin Main
Información

License: AMQ123 Name Driver: Andres Herrera Passengers: 4
Necesitas asignar 4 pasajeros

C:\Users\Silvana\Desktop\CursoPOOUber> cd c:\Users\Silvana\Desktop\CursoPOOUber && c:\Users\Silvana\.vscode\extensions\vscjava.vscode-javascript-debug-0.29.0\scripts\launcher.bat "C:\Program Files\Java\jdk-15.0.1\bin\java.exe" --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -Dfile.encoding=UTF-8 -cp C:\Users\Silvana\AppData\Roaming\Code\User\workspaceStorage\b4c80fb4c4006fe7f8ba50067f56fdc\redhat.java\jdt_ws\CursoPOOUber_197938db\bin Main
Información

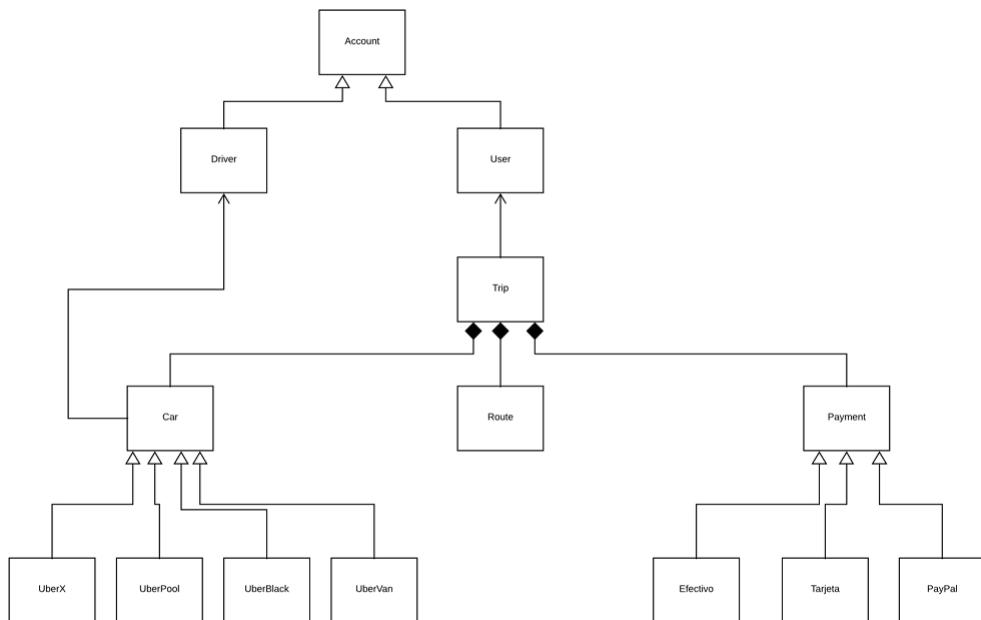
License: AMQ123 Name Driver: Andres Herrera Passengers: 4
Modelo: Sonic Marca: Chevrolet
License: TYU987 Name Driver: Cleo Ramirez Passengers: 6

```

## Modulo 8. Cierre del curso

### Clase 33 El Diagrama UML de Uber

Este es el diagrama que finalmente obtuvimos, aquí solo faltaría añadirle los atributos que posee cada clase.



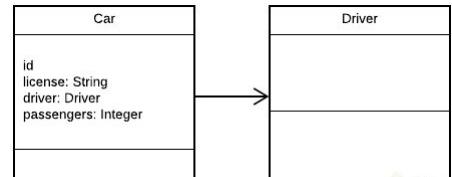
Recopilemos todo lo que hemos aprendido para explicar los últimos detalles.

En primer lugar notarás que tenemos 3 tipos de flechas:

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. Si recuerdas la clase Car, este contenía una instancia de Driver. La flecha apuntará hacia la dependencia.



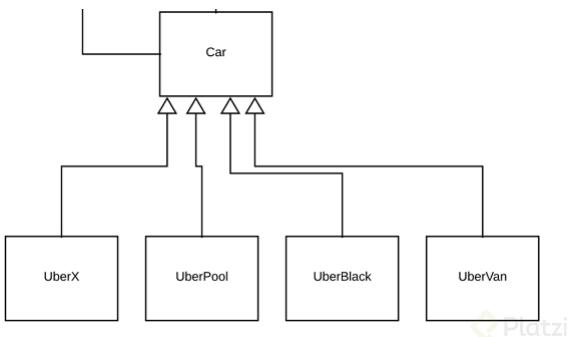
Herencia



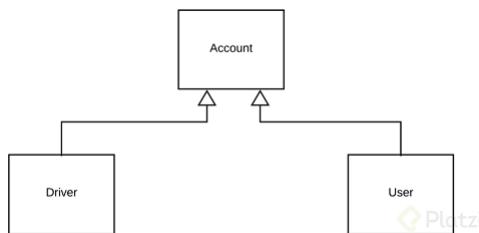
Siempre que veamos este tipo de flecha se estará expresando la herencia.

En nuestro diagrama tuvimos al menos tres familias conviviendo. La dirección de la flecha irá desde el hijo hasta el padre.

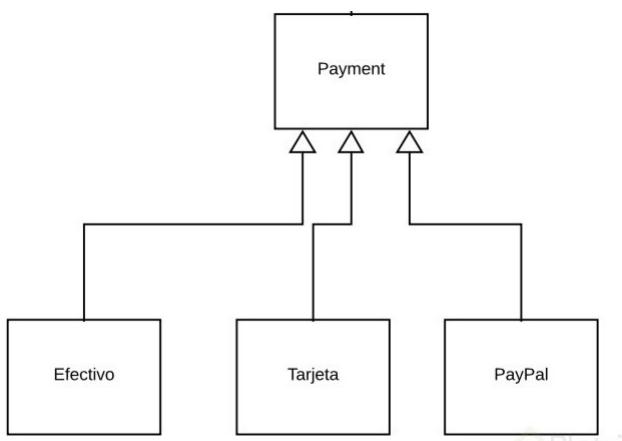
Familia Car



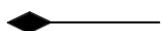
Familia Account



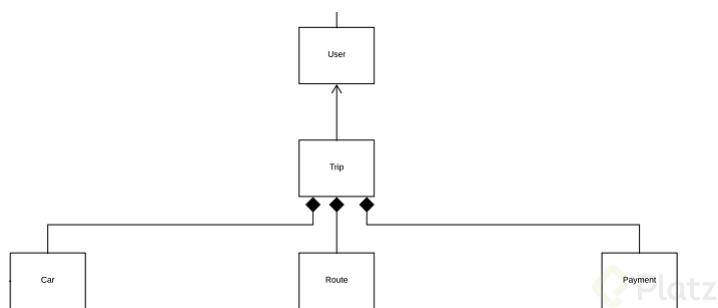
Familia Payment



Composición



Pasemos a una de nuestras piezas clave, pues notarás en el centro del diagrama la clase Trip que está vinculada a User, Car, Route y Payment. La composición va a significar una asociación entre estas clases con la diferencia de que para que esta clase pueda vivir forzosamente necesita a las demás. Es decir que estas clases son obligatorias para que la clase Trip pueda existir, esta dependencia obligatoria podríamos expresarla en el método constructor de la clase Trip, pues para que un objeto pueda ser creado dependerá de que los demás existan.



Esta clase Trip poseerá la lógica más fuerte del negocio aquí será donde se concentrarán la mayor cantidad de clases.

Esto es todo nuestro diagrama de clases, que quedó totalmente expresado en nuestro proyecto.

### Clase 34 Conclusiones

Has llegado al final del curso de Programación Orientado a Objetos y fue un placer para todos ir en esta travesía de aprendizaje donde aprendimos primeramente a analizar un problema y no cualquier problema, sino que fue un problema de la vida real. Con nuestro proyecto Uber pudimos aprender el análisis que luego convertimos en gráficas y finalmente en un diagrama UML que nos permitió llevarlo directamente a la fase de programación. Y no solo fue un único lenguaje de programación, estuvimos aprendiendo cuatro lenguajes: Java, Python, PHP y JavaScript.

No olvides los pilares de la Programación Orientado a Objetos :

- Encapsulamiento
- Abstracción
- Herencia
- Polimorfismo

Y por supuesto no olvides las clases, los objetos y todo lo que hemos aprendido aquí.

### Clase 35 Bonus: Qué es la Programación Orientada Objetos

Imaginemos que tenemos un videojuego de fútbol en el cual debemos representar a los jugadores de cada equipo, con la Programación Orientada a Objetos podemos hacer una abstracción de todo esto.

La **abstracción** es cuando nosotros separamos los datos de un objeto para entonces generar un molde y ese molde se llama clase. La **clase** se compone de dos cosas: atributos y métodos. Los **atributos** son todas las características que corresponden al jugador, mientras que los **métodos** son todas las acciones que podrá hacer el jugador. Por ejemplo, los atributos de nuestro jugador pueden ser el nombre y apellido, y uno de los métodos podría ser la de correr.

A partir de clases podemos crear objetos. El **objeto** es la base de la Programación Orientada a Objetos y son las instancias de las clases, es decir, las clases son el molde de los objetos. Con las clases podemos crear tantos jugadores como queramos. Cada jugador tendrá características e incluso acciones diferentes, por ejemplo, podemos tener jugadores con diferentes colores de camisetas, shorts distintos o también puede que corran a velocidades diferentes.

Además, con la Programación Orientada a Objetos podemos usar otro concepto importante que es la herencia. La **herencia** nos ayudara a crear nuevas clases a partir de otras. Es probable que en nuestro videojuego debamos crear la clase arbitro, la clase jugador y la clase portero, estas clases pueden tener características que sean muy similares entre sí.

También tenemos otro concepto importante que es el **encapsulamiento** que puede significar esconder algo. En nuestro videojuego podría ser que a nuestros jugadores quisieramos esconderle la velocidad a la cual corren, es decir, hacerlo invisible a los demás.

jugadores.

Por ultimo tenemos el **polimorfismo** que significa muchas formas y también es una base importante. Nuestro entrenador podría asignarle un mensaje a cada uno de nuestros jugadores y cada uno irá al campo a ejecutar lo que interprete de ese mensaje.

## Final:

---

Si es que este resumen te gusto considera ver [este repositorio](#) donde tengo esta información mas ordenada y tambien subo la resolución de los ejercicios.

Esta contribución saca casi toda la información de [este repositorio](#).