**Programación orientada a objetos en Java**

**¡Te damos la bienvenida al Curso de Java SE Orientado a Objetos!**

Los **paradigmas de programación** son una teoría que nos suministra una base y modelo estandarizado para resolver problemas con nuestro código.

La **Programación Orientada a Objetos** (POO) nos ayuda a analizar y entender todos estos problemas para resolverlos de la forma más sostenible en el futuro. Java surgió con este paradigma y es uno de los lenguajes que define en gran manera el rumbo que sigue la POO.

Este paradigma se compone de 4 elementos (que analizaremos a fondo más adelante):

* **Clases**
* **Propiedades**
* **Métodos**
* **Objetos**

Además, se basa en los siguientes 4 pilares:

* **Encapsulamiento**
* **Abstracción**
* **Herencia**
* **Polimorfismo**

Recuerda que antes de tomar este curso es muy recomendado que tomes el [Curso de Introducción a Java SE](https://platzi.com/clases/java-basico) y el [Curso de Programación Orientada a Objetos: POO](https://platzi.com/poo). Al terminar este curso puedes continuar con el *Curso de Java SE: Programación Funcional*.

# ¿Qué es un Objeto?

Los **Objetos** son todas las cosas físicas o conceptuales que tienen propiedades y comportamientos. Por ejemplo: usuario, sesión, auto, etc.

Las **Propiedades** o atributos son las características de nuestros objetos. Estos atributos siempre serán sustantivos y pueden tener diferentes valores que harán referencia a nombres, tamaños, formas y estados.

Por ejemplo: el color del auto es verde o rojo (color es el atributo, verde y rojo son posibles valores para este atributo).

Los **Comportamientos** o métodos serán todas las operaciones de nuestros objetos que solemos llamar usando verbos o sustantivos y verbos. Por ejemplo: los métodos del objeto sesión pueden ser login(), logout(), makeReport(), etc.

Los **Objetos** pueden ser  
Físicos: User  
Conceptuales: Session

Los objetos tendrán:  
**Propiedades** (estado)  
**Métodos** (comportamientos)

Las **propiedades** son características de los objetos y siempre serán de tipo sustantivo(nombre, tamaño, forma, estado).

-Los valores de las propiedes pueden ser sustantivos o adjetivos.  
Por ejemplo: Juan, Alta, Azul, apagado.

Los **comportamientos** serán todas las operaciones del objeto, suele ser verbos o sustantivos y verbo.  
ejemplos: login(), logout(), makeReport().

# Abstracción: ¿Qué es una Clase?

La **Abstracción** se trata de analizar objetos de forma independiente, sus propiedades, características y comportamientos, para *abstraer* su composición y generar un modelo, lo que traducimos a código como clases.

Las **Clases** son los modelos sobre los cuales construimos nuestros objetos, es decir, las clases son los “moldes” que nos permiten generar objetos. Cada clase debe tener identidad (con un nombre de clase único usando [Upper Camel Case](https://platzi.com/clases/1631-java-basico/21181-tecnica-de-naming-camel-ca-9/" \t "_blank)), estado (con sus atributos) y comportamiento (con sus métodos y operaciones).

Por ejemplo:

El ejemplo de clase más típico en Internet:

Nombre de la clase: Person

Atributos: Name, Age

Operaciones: Walk()



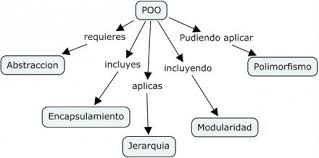
# Modularidad

La **Modularidad** consiste en dividir nuestro programa en diferentes módulos de forma que puedan unirse o separarse sin romperse entre ellos o perder alguna funcionalidad.

La Modularidad en Programación Orientada a Objetos nos ayuda a:

* Reutilizar código.
* Evitar colapsos.
* Que nuestro código sea mantenible.
* Mejorar la legibilidad.
* Resolución rápida de problemas.

**Modularidad;** Lo mejor para tener un codigo muy mantenible y extensible, comprender la modularidad. No solo en java.



# Creando nuestra primera Clase

Nuestro proyecto en este curso es construir un sistema que nos permita listar y agendar nuestras citas médicas, por lo que debemos crear algunas clases para cada integrante del sistema: doctores, pacientes, entre otras.

Así vamos a crear nuestra primer clase con sus métodos y atributos:

// Clases:

**public** **class** **Doctor** {

// Atributos:

**int** id;

String name;

String speciality;

// Comportamientos (métodos):

**public** **void** **showName**() {

// Instrucciones...

}

}

# Método constructor

El **Método Constructor** es el primer método que se ejecuta por defecto cuando creamos una clase, nos permite crear nuevas instancias de una clase. Lo invocamos con la palabra reservada new seguida del nombre con el que inicializamos la clase y paréntesis.

// nombreDeLaInstancia = new MétodoConstructor();

myDoctor = **new** Doctor();

El compilador de Java crea un método constructor en caso de que no definamos uno, pero de todas formas es muy buena idea programarlo nosotros, ya que nos permite definir y/o configurar el comportamiento de nuestros objetos usando argumentos.

**public** **class** **Doctor** {

// Atributos...

// Método Constructor:

Doctor(/\* parámetros \*/) {

// Instrucciones que se ejecutan al crear/instanciar

// un nuevo objeto con la clase Doctor...

}

}

El método constructor no debe regresar ningún valor (no necesitamos un return). Más adelante estudiaremos un poco más a fondo cómo funcionan la sobrecarga de métodos y sobrecarga de constructores.

# Static: Variables y Métodos Estáticos

Los métodos y variables estáticos nos ayudan a ejecutar o conseguir algún código desde clases no han sido instanciadas, ya que sus valores se guardan en la memoria de nuestro programa, no en diferentes objetos instanciados a través de una clase.

Las variables estáticas mantienen su valor durante todo el ciclo de vida de nuestro programa, por lo tanto, podemos alterar los valores de una variable estática desde una clase y consumir su valor alterado desde otra sin necesidad de conectar ambas clases.

También podemos importar los métodos estáticos de una clase para usarlos sin necesidad de escribir el nombre de la clase:

**import** **static** com.anncode.operaciones.Calculadora.\*

**import** **static** java.lang.Math.\*

**public** **class** **Principal** {

**public** **static** **void** (String[] args) {

**int** number = suma(3, 5);

System.out.println(number + PI);

}

}

¿Que es **Static**? se utiliza para definir métodos o variables de ese tipo estático. Utilizamos **Static** para llamar una variable sin utilizar un objeto. Se invoca en una clase que no tiene instancias de la clase.

¿Que nivel de **Scope** tiene una variable de tipo **Static**? Una variable de ese tipo, mantiene su ciclo de vida en todo el programa mientras este corriendo el programa.

<https://www.youtube.com/watch?v=mvBX4-5-A4o>

# Creando elementos estáticos

En muchos casos nuestro código necesita ejecutar métodos que no necesariamente deben pertenecer a un objeto o instancia en concreto, ya que pueden ser muy generales (así como Math.Random) o los valores que almacenamos deben ser los mismos, sin importar si los consumimos desde una o más clases.

En todos estos casos vale la pena usar variables y métodos estáticos.

# Final: Variables Constantes

He visto que muchos consideran final == const y esto no es verdad. Creo que es porque Ann debería de usar const según el ejemplo que mostró en la clase en lugar de final. Dejenme explicar.  
Aunque se puede usar indistintamente en algunos casos, hay que tener en cuenta las diferencias.  
¿Qué es final?  
final quiere decir que solamente podemos hacer una sola asignación. Esto es: Lo podemos definir sin asignar un valor, pero después en algún momento se tiene que inicializar y ese valor no podrá cambiar.  
¿Qué es const?  
const es un poco más complejo y tiene algunas propiedades interesantes. Cuando se usa const quiere decir que el valor el cual tiene asignado se puede determinar completamente en la compilación. Por ejemplo: un valor de const puede ser: “1+2” pues se calcula al momento de compilar, y un valor invalido seria “new DateTime.now()” porque este último se obtiene en runtime(Cuando ejecutas el programa).  
Diferencias:  
-Si usas const , lo tienes que declarar como “static const” en lugar de solamente “const”.  
-Cuando queremos una colección completamente inmutable, utilizamos const, pues todo lo que esté dentro de esa colección también será inmutable. En cambio si utilizamos final, todo lo que esté dentro no será final.  
-final debería de usarse en lugar de const cuando no conocemos el valor al tiempo de compilación y éste será calculado u obtenido en runtime.

Como los datos de los nombres de los meses no se obtiene en run-time sino en compile-time es mejor utilizar const en este caso.  
Incluso conceptualmente es mejor, pues los meses nunca cambian de nombre, siempre son constantes(const)

# Sobrecarga de métodos y constructores

A veces necesitamos que dos o más métodos de una misma clase tengan el mismo nombre, pero con diferentes argumentos o distintos tipos de argumentos/valores de retorno.

Afortunadamente, Java nos permite ejecutar código y métodos diferentes dependiendo de los argumentos que reciba nuestra clase.

**public** **class** **Calculadora** {

// Los dos parámetros y el valor de retorno son de tipo int

**public** **int** **suma**(**int** a, **int** b) {

**return** a + b;

}

// Los dos parámetros y el valor de retorno son de tipo float

**public** **float** **suma**(**float** a, **float** b) {

**return** a + b;

}

// Un parámetro es de tipo int, mientras que el otro parámetro

// y el valor de retorno son de tipo float

**public** **float** **suma**(**int** a, **float** b) {

**return** a + b;

}

}

El uso más común de la sobrecarga de métodos es la sobrecarga de constructores para instanciar objetos de formas distintas dependiendo de la cantidad de argumentos que enviamos.

**public** **class** **Doctor** {

**static** **int** id = 0;

String name;

String speciality;

**public** **Doctor**() {

**this**.name = "Nombre por defecto";

**this**.speciality = "Especialidad por defecto";

}

**public** **Doctor**(String name, String speciality) {

**this**.name = name;

**this**.speciality = speciality;

}

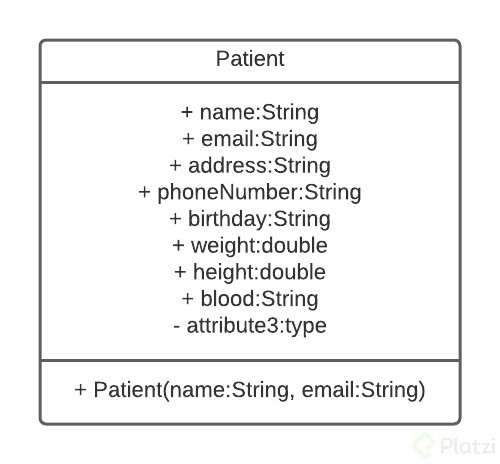
}

# ¡Reto!

Ahora estás listo para resolver tu primer reto que en realidad es muy sencillo de hacer.

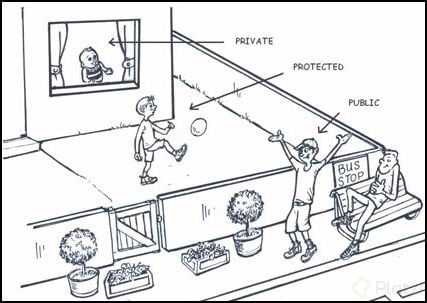
Mira el siguiente diagrama y construye la clase Patient.

Patient  
name: String  
email:String  
address: String  
phoneNumber: String  
birthday: String  
weight: double  
height: double  
blood: String  
Patient(name: String, email: String



# Encapsulamiento: Modificadores de acceso

Los **Modificadores de Acceso** nos ayudan a limitar desde dónde podemos leer o modificar atributos especiales de nuestras clases. Podemos definir qué variables se pueden leer/editar por fuera de las clases donde fueron creadas. Esto lo conocemos como **Encapsulamiento**.



# Getters y Setters

Los **Getters y Setters** nos permiten leer y escribir (respectivamente) los valores de nuestras variables privadas desde fuera de la clase donde fueron creadas. Con los Getters obtenemos los datos de las variables y con los Setters asignamos o cambiamos su valor.

También puedes usar los atajos de tu IDE favorito para generar los métodos getters y setters de todas o algunas de tus variables.

**public** **class** **Patient** {

**private** String name;

**public** String **getName**() {

**return** "Patient name is " + **this**.name;

}

**public** **void** **setName**(String newName) {

**this**.name = newName;

}

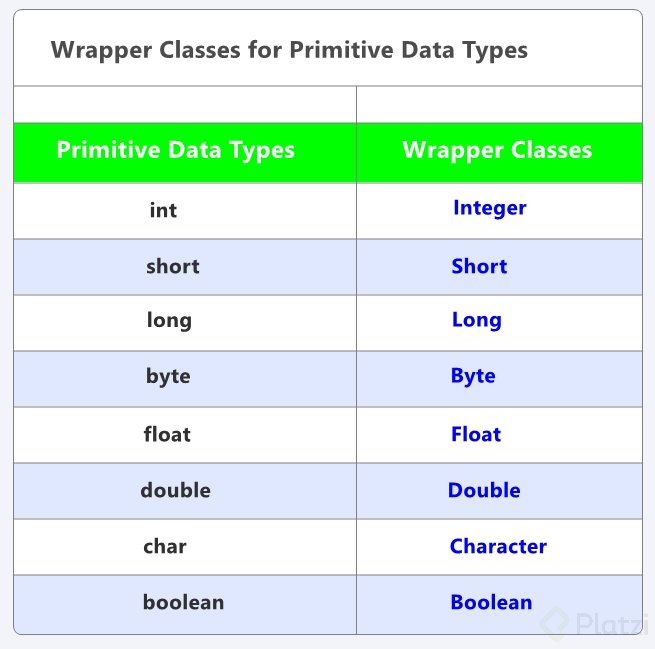
}

# Variable vs. Objeto

Las **Variables** son entidades elementales muy sencillas, pueden ser números, caracteres, booleanos, entre otras. Los **Objetos** son entidades complejas que pueden estar formadas por la agrupación de diferentes variables y métodos.

Los **Objetos Primitivos** o **Clases Wrapper** son variables primitivas que trabajan con algún tipo de dato y también tienen las características de los objetos.

Por ejemplo: Byte, Short, Integer, Long, Float, Double, Character, Boolean o String.



# Variable vs. Objeto: Un vistazo a la memoria

**Un objeto es una referencia a un espacio en memoria.** Cuando creamos objetos, Java los guarda en la memoria y nos devuelve coordenadas con las que podremos acceder a la información que almacenamos.

Existen dos tipos de memoria: **Stack** y **Heap**.

La memoria **Stack** es mucho más rápida y nos permite almacenar nuestra información de forma “ordenada”. Aquí se guardan las variables y sus valores de tipos de datos primitivos (booleanos, números, strings, entre otros).

Los objetos también usan la memoria Stack, pero no para guardar su información, sino para guardar las coordenadas a la verdadera ubicación del objeto en la memoria **Heap**, una memoria que nos permite guardar grandes cantidades de información, pero con un poco menos de velocidad.

Notas de la clase:

🎳 Los objetos son referencias a un espacio en memoria.  
🎩 La mayoría de lenguajes de programación utilizan dos tipos de memorias: **Stack** y **Heap**.  
📁 La memoria **Stack** es muy rápida, pero sin tanto espacio. Aquí guardamos los valores primitivos de nuestras variables (true, false, ‘JuanDC’, 16, etc).  
🌪 La memoria **Heap** es un poco más lenta y nos permite guardar grandes cantidades de información. Son como los tornados: grandes, lentos y desordenados (😬). En esta memoria guardamos los valores de los objetos, las instancias de nuestras clases.

//variable vs objeto: un vistazo a la memoria  
int i = **0;**int b = **2;**b = i**;**//b=0  
String name = "ann"**;**Patient patient = new Patient("Alejandra"**,** "alejandra@mail.com")**;**Patient patient2 = new Patient("Anni"**,** "Annia@mail.com")**;**System.*out*.println()**;**System.*out*.println(patient)**;** // imprime la localidades de memoria  
System.*out*.println(patient2)**;**System.*out*.println(patient.getName())**;** //  
System.*out*.println(patient2.getName())**;**patient2 = patient**;**System.*out*.println(patient.getName())**;** //  
System.*out*.println(patient2.getName())**;**//altere el valor ambos seran alterados  
System.*out*.println()**;**patient2.setName("manuel")**;**System.*out*.println(patient.getName())**;** //  
System.*out*.println(patient2.getName())**;**

Patient@75b84c92

Patient@6bc7c054

Alejandra

Anni

Alejandra

Alejandra

manuel

manuel

Patient@75b84c92

# Clases Anidadas

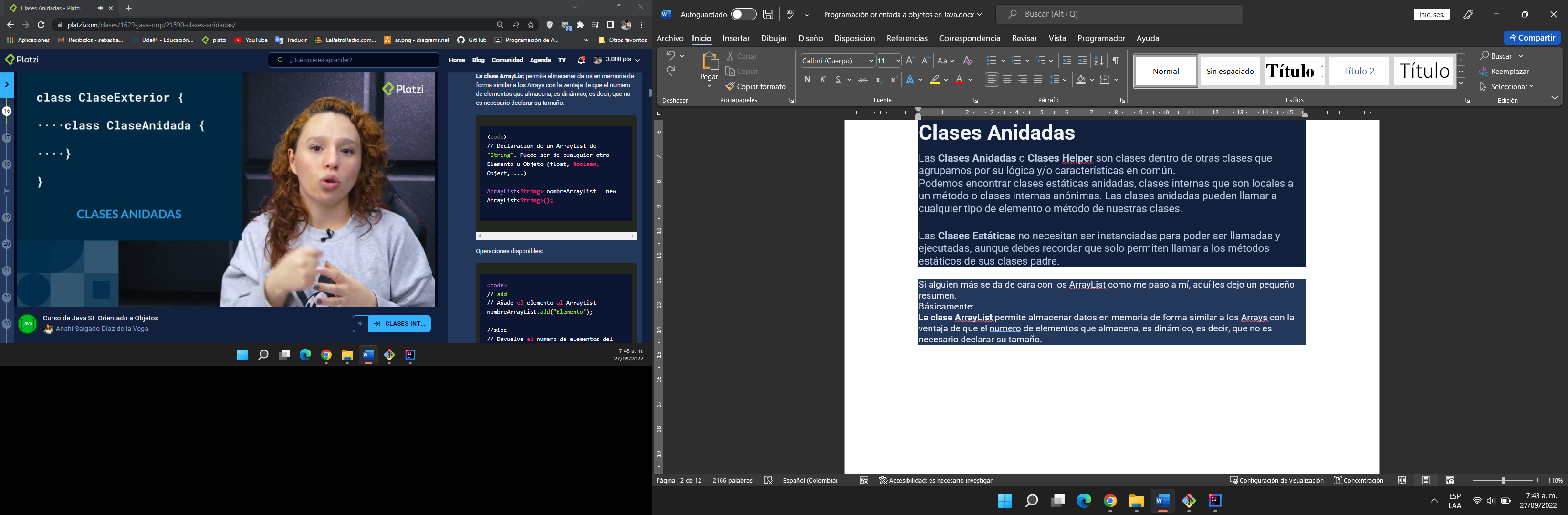
Las **Clases Anidadas** o **Clases Helper** son clases dentro de otras clases que agrupamos por su lógica y/o características en común.

Podemos encontrar clases estáticas anidadas, clases internas que son locales a un método o clases internas anónimas. Las clases anidadas pueden llamar a cualquier tipo de elemento o método de nuestras clases.

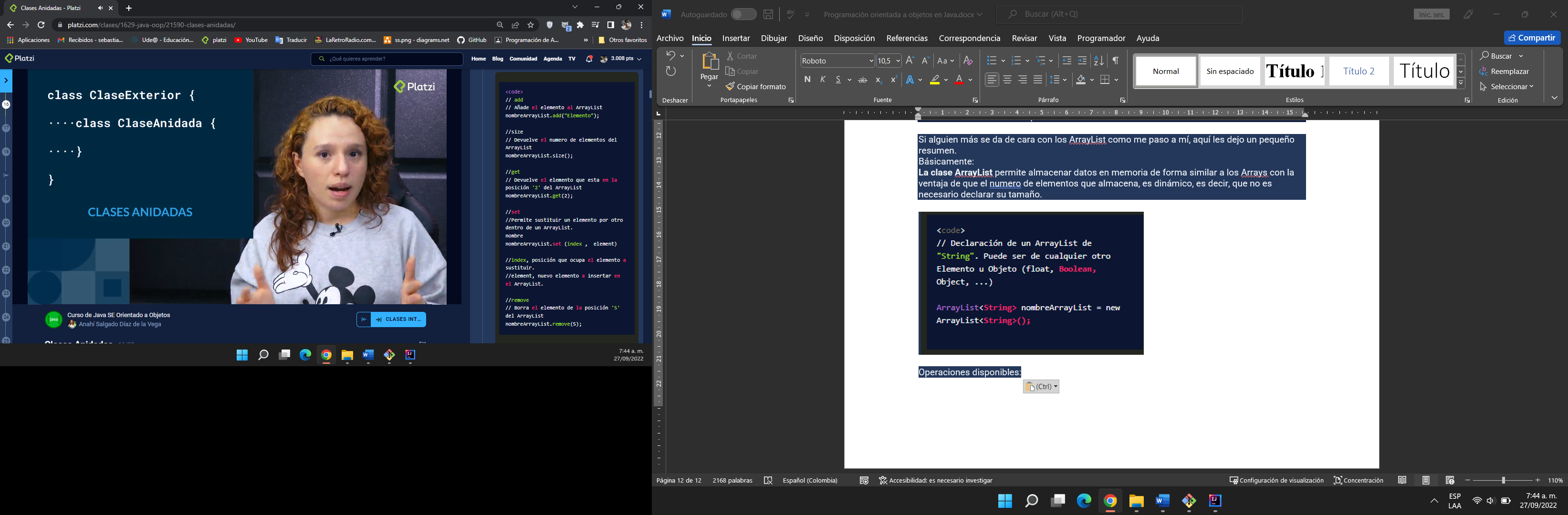
Las **Clases Estáticas** no necesitan ser instanciadas para poder ser llamadas y ejecutadas, aunque debes recordar que solo permiten llamar a los métodos estáticos de sus clases padre.

Si alguien más se da de cara con los ArrayList como me paso a mí, aquí les dejo un pequeño resumen.

Básicamente:  
**La clase ArrayList** permite almacenar datos en memoria de forma similar a los Arrays con la ventaja de que el numero de elementos que almacena, es dinámico, es decir, que no es necesario declarar su tamaño.

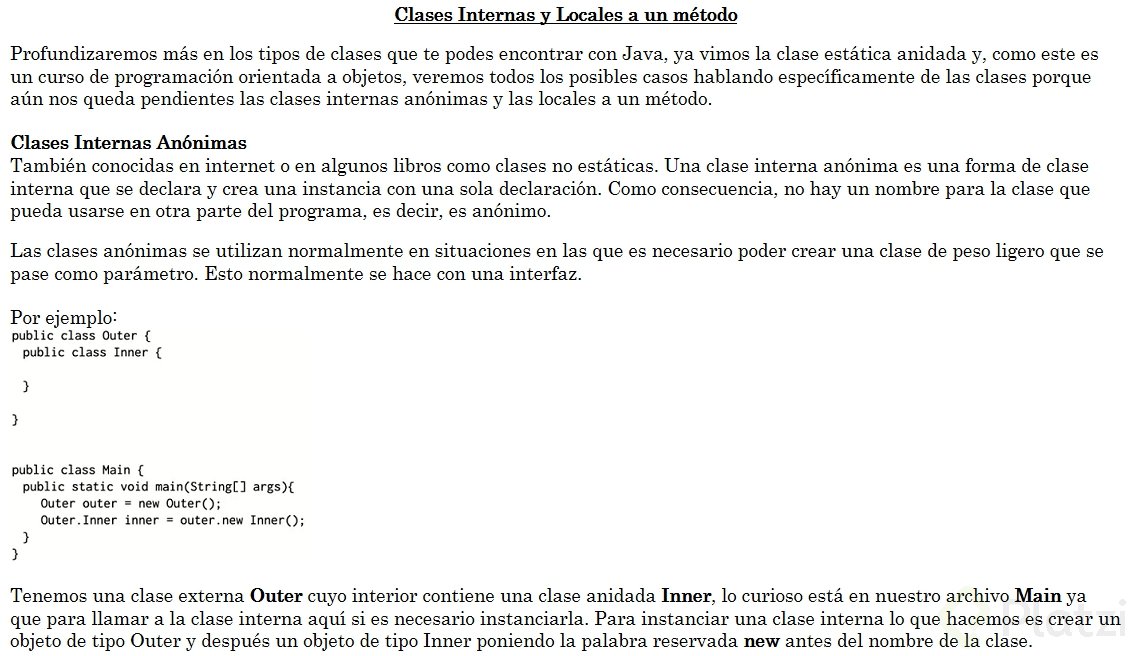


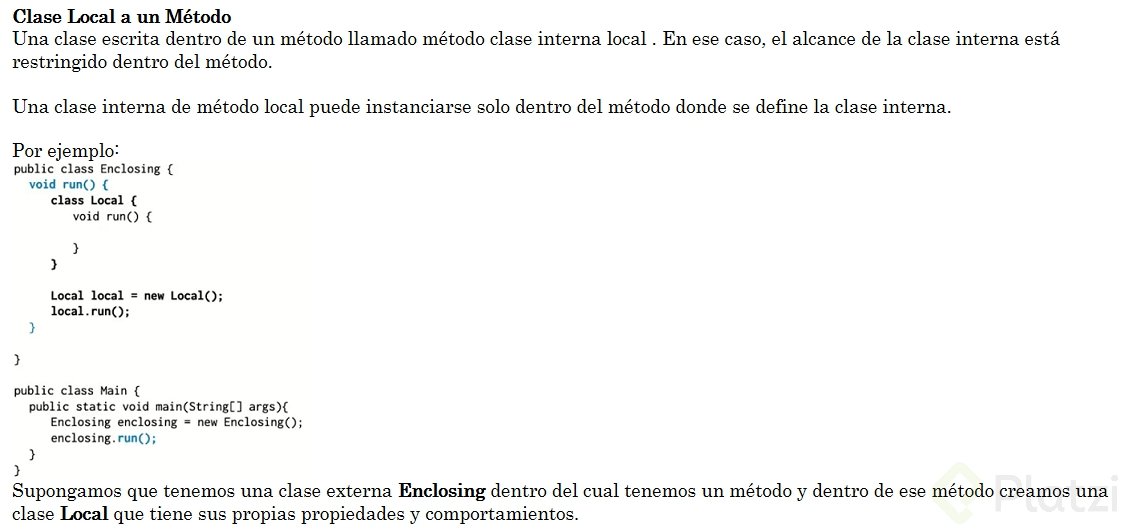
Operaciones disponibles:

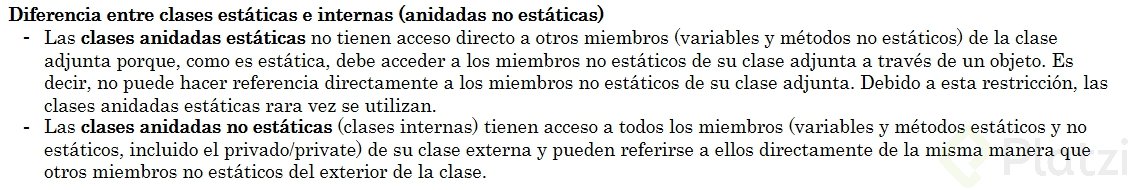


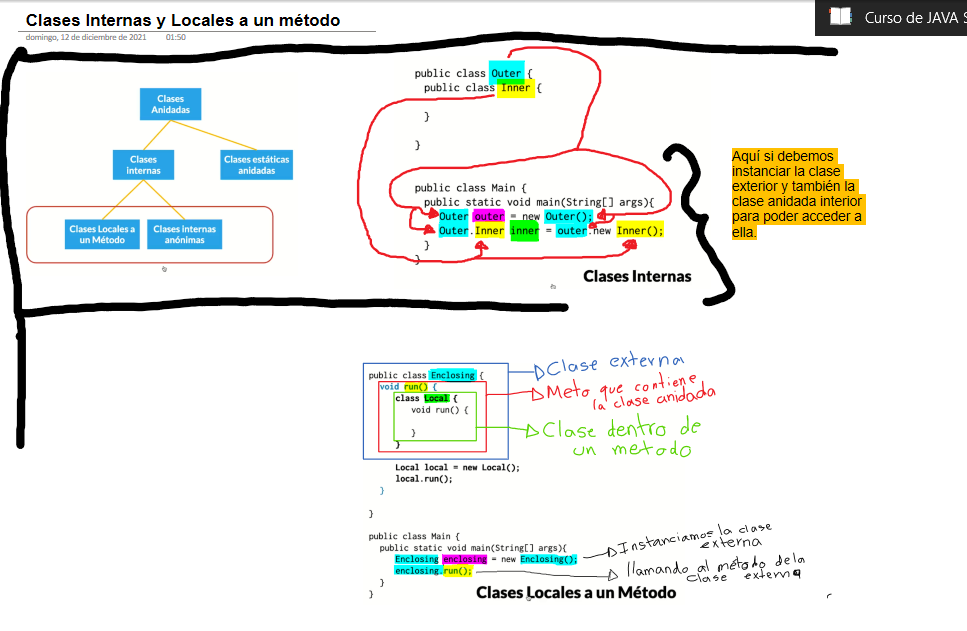
Obviamente hay mas comandos pero, estos son los más comunes.  
Espero que esto le sirva a alguien.

# Clases Internas y Locales a un método









# Enumerations

Los enumerations son tipos de datos muy especiales pues este, es el único en su tipo que sirve para declarar una colección de constantes, al ser así estaremos obligados a escribirlos con mayúsculas.

Usaremos enum cada vez que necesitemos representar un conjunto fijo de constantes. Por ejemplo los días de la semana.

Así podemos declarar un enumeration usando la palabra reservada **enum.**

public **enum** **Day** {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY,

THURSDAY, FRIDAY, SATURDAY

}

Puedo crear referencias de enumerations de la siguiente forma:

Day day;

**switch** (day) {

**case** MONDAY:

**System**.out.println(“Mondays are good.”);

**break**;

**case** FRIDAY:

**System**.out.println(“Fridays are nice”);

**break**;

**case** SATURDAY: **case**: SUNDAY:

**System**.out.println(“Weekends are the best”);

**break**;

**default**:

**System**.out.println(“Midweek are so-so”);

**break**;

}

Y puedo llamar un valor del enumeration así:

**Day**.MONDAY;

**Day**.FRIDAY;

**Day**.SATURDAY

Los enumerations pueden tener atributos, métodos y constructores, como se muestra:

public **enum** **Day** {

MONDAY("Lunes");

TUESDAY("Jueves");

FRIDAY("Viernes");

SATURDAY("Sábado");

SUNDAY("Domingo");

**private** String spanish;

**private** Day(String s) {

spanish = s;

}

**private** String getSpanish() {

**return** spanish;

}

}

Y para utilizarlo lo podemos hacer así:

**System**.out.println(**Day**.MONDAY);

Imprimirá: **MONDAY**

**System**.out.println(**Day**.MONDAY.getSpanish());

Imprimirá: **Lunes**

# ¿Qué es la Herencia? Don't repeat Yourself

**Don’t repeat yourself** (DRY) consiste en detectar cuando estamos repitiendo el mismo código una y otra vez para crear algún método o función que nos ayude a evitar estos repetidos.

Esta es una de las bases de la programación que siempre debemos tener en cuenta, ya que nos ayuda a reducir la dificultad de nuestro código para implementar cambios y/o mejoras en nuestra aplicación.

La **Herencia** consiste en crear nuevas clases a partir de otras clases, establecemos una relación padre e hijo entre nuestras clases. Es diferente a las clases anidadas, ya que, en vez de crear clases dentro de clases, le indicamos a nuestras subclases de qué superclase pueden heredar (extends) para reutilizar el código de algunos de sus métodos.

Recuerda que nuestras clases no pueden heredar de más de una clase.

**public** **class** **SuperClass** {

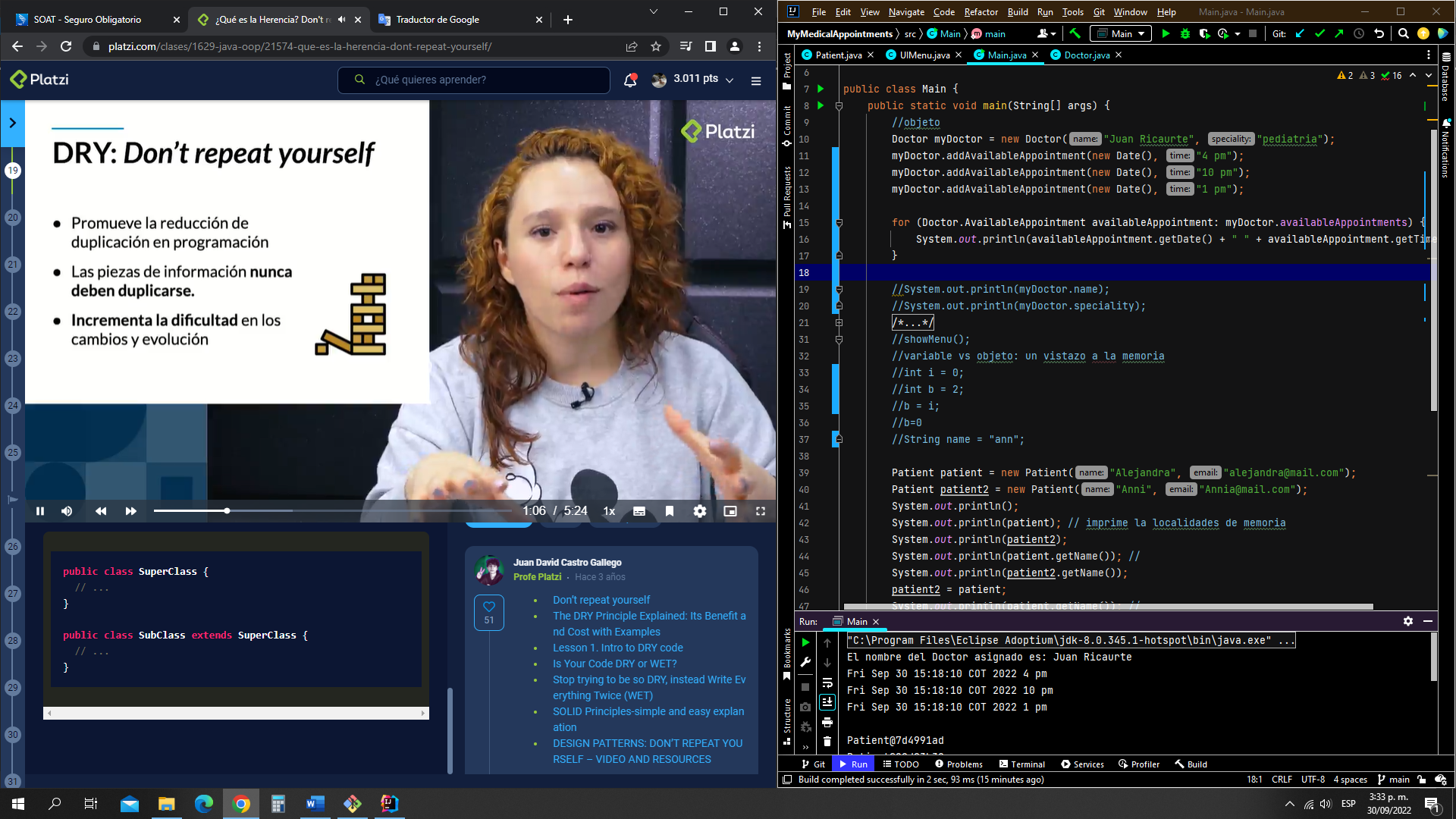
// ...

}

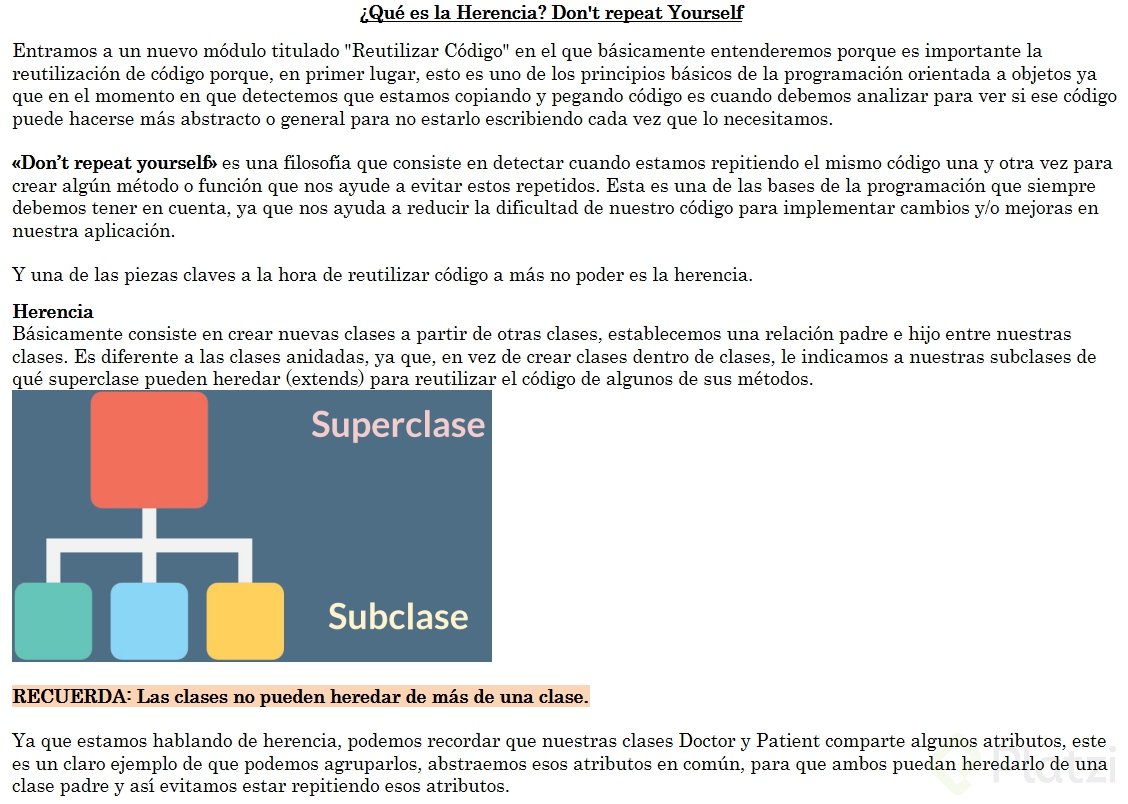
**public** **class** **SubClass** **extends** **SuperClass** {

// ...

}



La **herencia** nos permite **evitar** el **código repetitivo**.



# Super y This

**Super** indica que una variable o método es de la clase padre, la superclase de cual heredan nuestras subclases, solo la usamos cuando aplicamos herencia.

Además, podemos llamar al constructor de la clase padre desde sus diferentes subclases usando super(); y enviando los argumentos que sean necesarios.

Por otro lado, **this** nos permite especificar que nuestras variables están señalando a la misma clase donde estamos trabajando, ya sea una clase normal, anidada, subclase o superclase.

**public** **class** **User** {

**int** age = 1;

**public** **int** **getAge**() {

**return** **this**.age;

}

}

**public** **class** **Doctor** **extends** **User** {

String speciality = "Dentist";

Doctor() {

**super**.getAge(); // 1

**this**.getSpeciality(); // Dentist

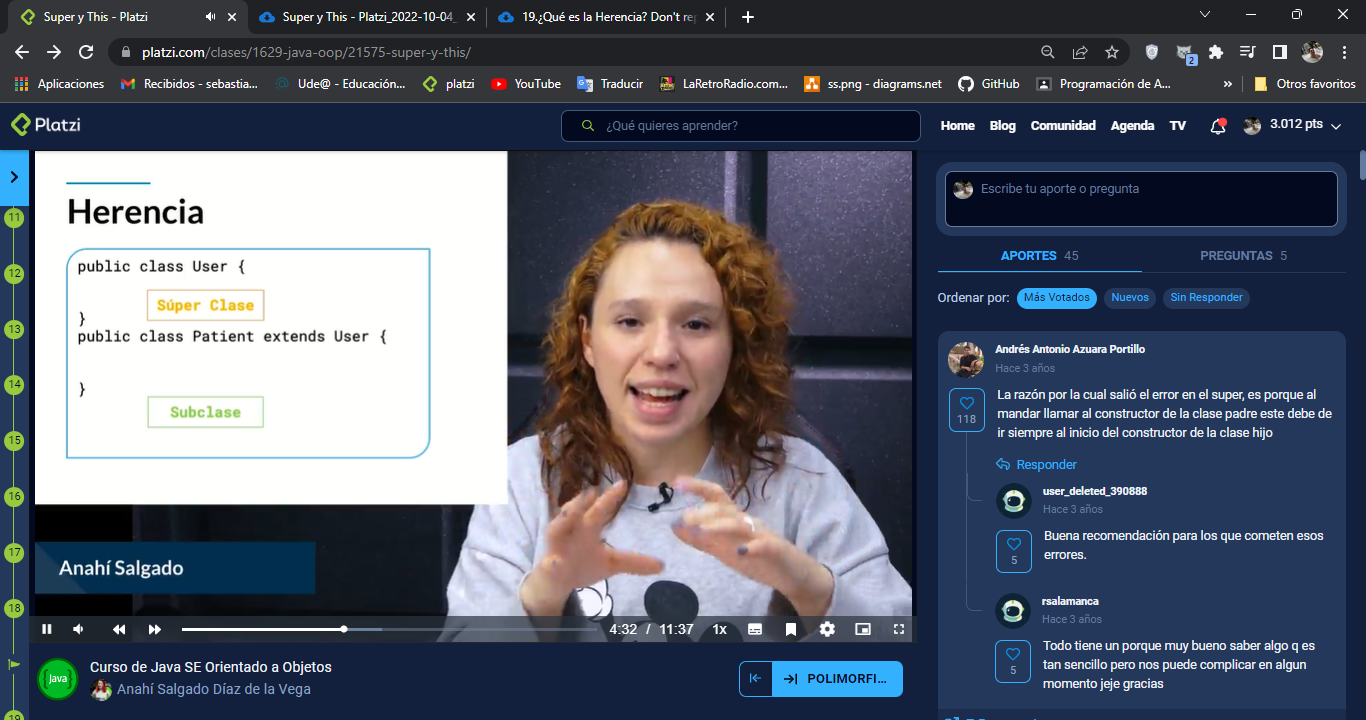
}

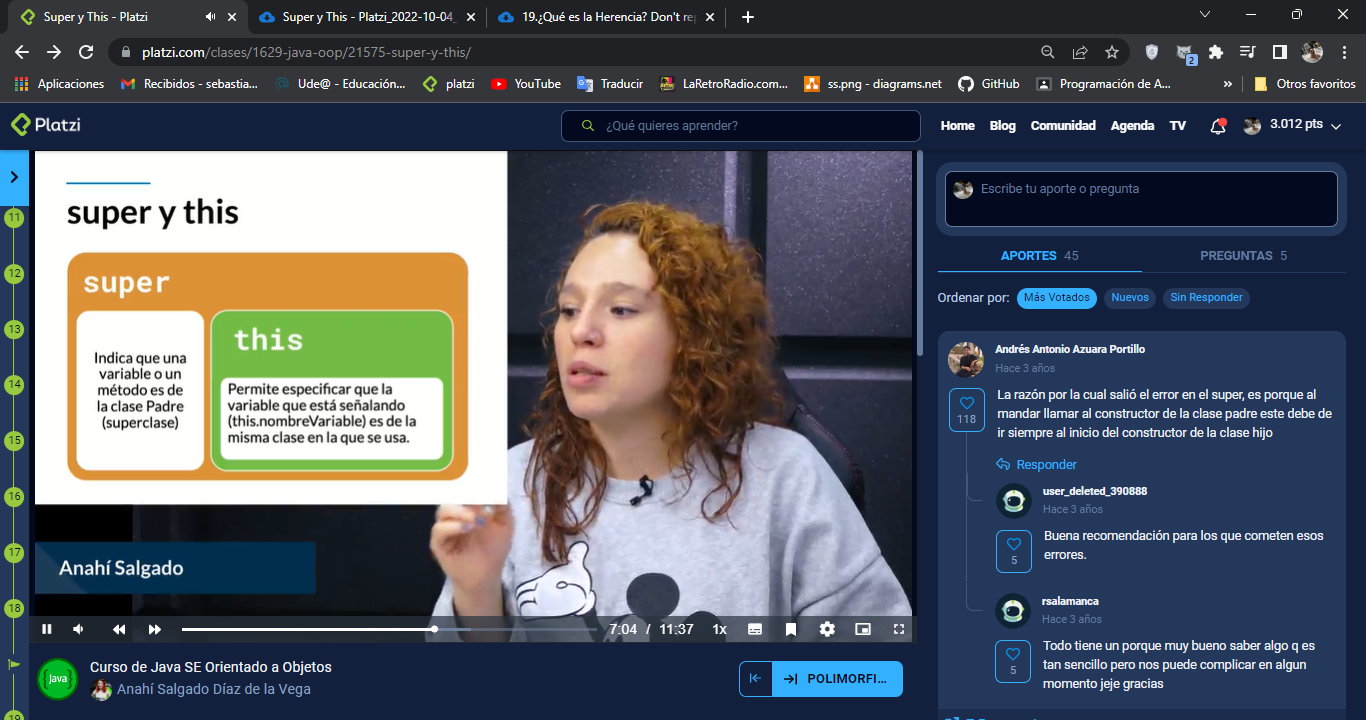
**public** **int** **getSpeciality**() {

**return** **this**.speciality;

}

}





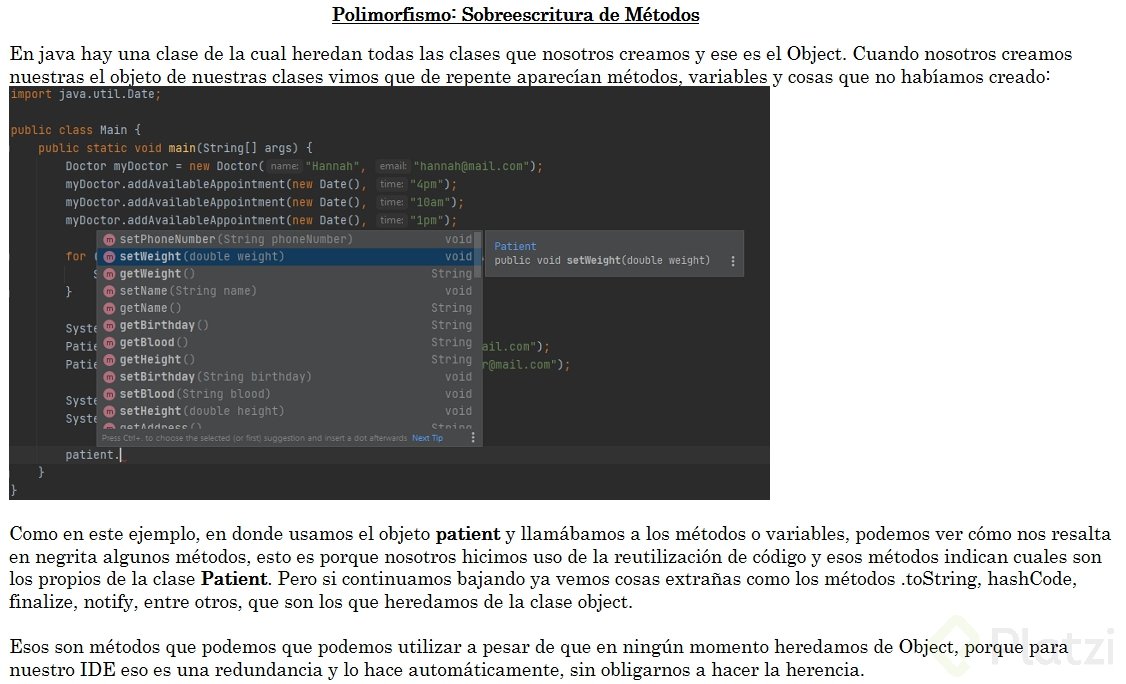
# Polimorfismo: Sobreescritura de Métodos

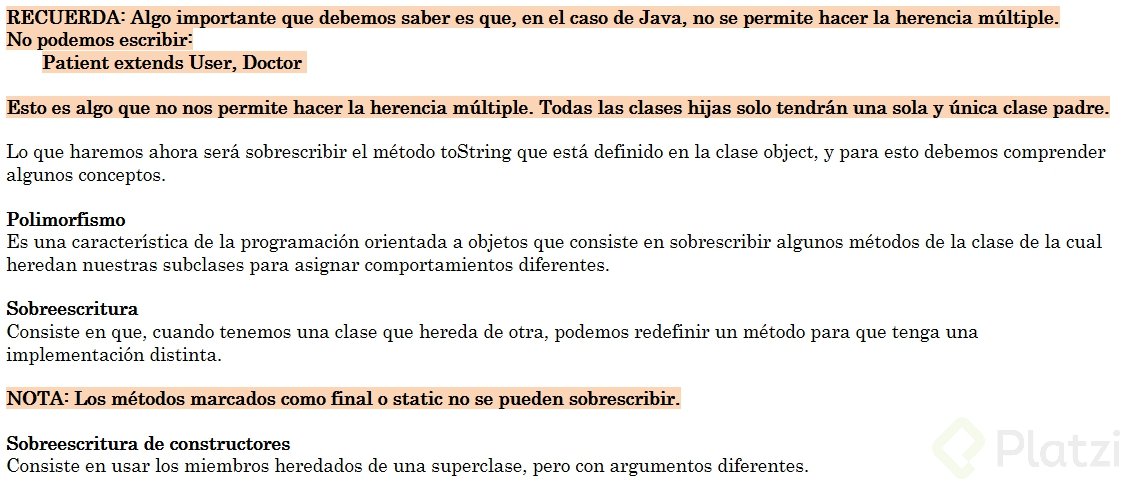
El **Polimorfismo** es una característica de la programación orientada a objetos que consiste en sobrescribir algunos métodos de la clase de la cual heredan nuestras subclases para asignar comportamientos diferentes.

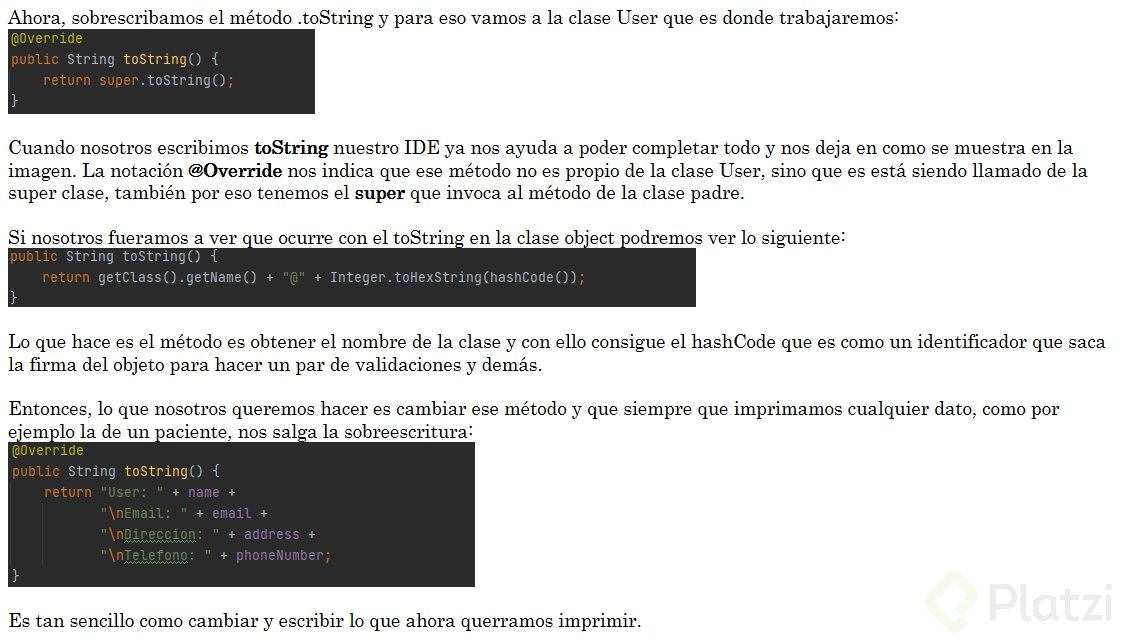
Además de los métodos de las superclases, también podemos redefinir el comportamiento de los métodos que “heredan” todos nuestros objetos, así como .toString, hashCode, finalize, notify, entre otros.

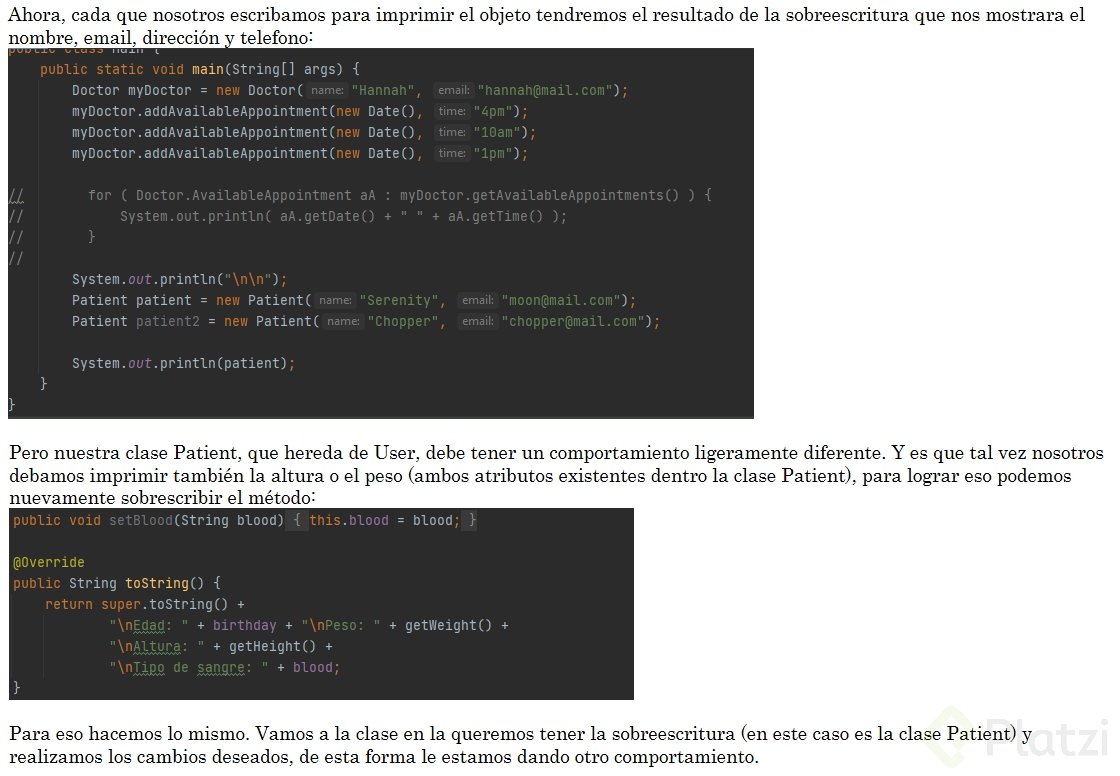
La sobreescritura de constructores consiste en usar los miembros heredados de una supreclase pero con argumentos diferentes.

Recuerda que no podemos sobrescribir los métodos marcados como final o static.









# Polimorfismo: Sobreescribiendo el método toString

Cabe aclarar que en POO tenemos 3 clases de polimorfismo de las cuales ya hemos revisado 2 tipos:

* **Sobrecarga**: Podemos escribir un método con el mismo nombre varias veces cambiando los parámetros que recibe ya sea en cantidad o en tipos, ademas podemos retornar otro tipo de dato.

**public** **int** suma (**int** num1, **int** num2) {

return num1 + num2;

}

**public** **String** suma (**String** text1, **String** text2) {

return num1 + " " + num2;

}

* **Sobreescritura:** Podemos escribir un método de la clase padre y cambiar su contenido y asi realizar comportamiento mas especifico:

<!-- Clase Padre User -->

public String toString () {

return name + " " + email;

}

<!-- Clase Hija Patient -->

public String toString () {

return super.toString() + " " + blood + " " + weight;

}

* **Ligadura Dinámica:** Podemos ejemplificar a las clases hijas pero tratarlas como su clase padre, puede ser util por ejemplo si queremos un arreglo que contenga objetos tipo User pero que implícitamente también va a contener doctores y pacientes:

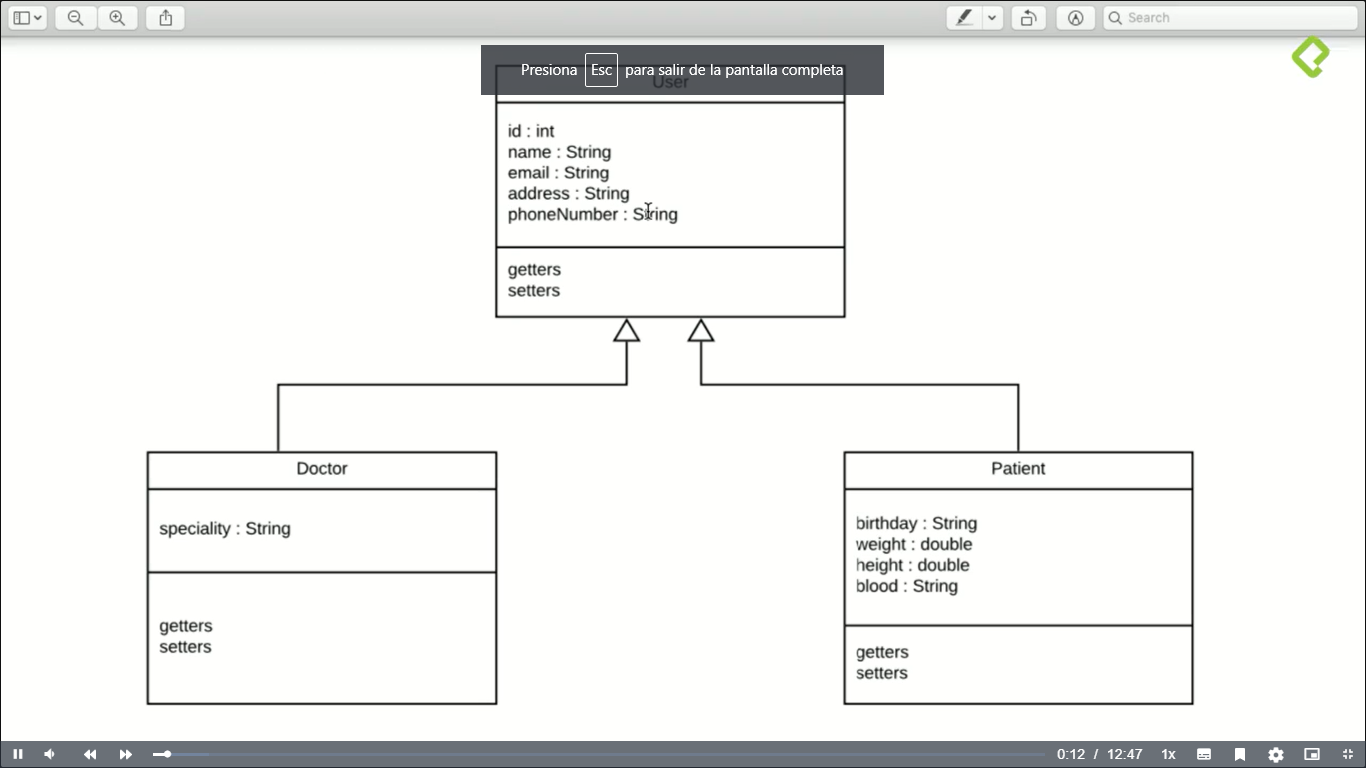
User doctor = **new** Doctor ("Cristian", "criss.ud@gmail.com");

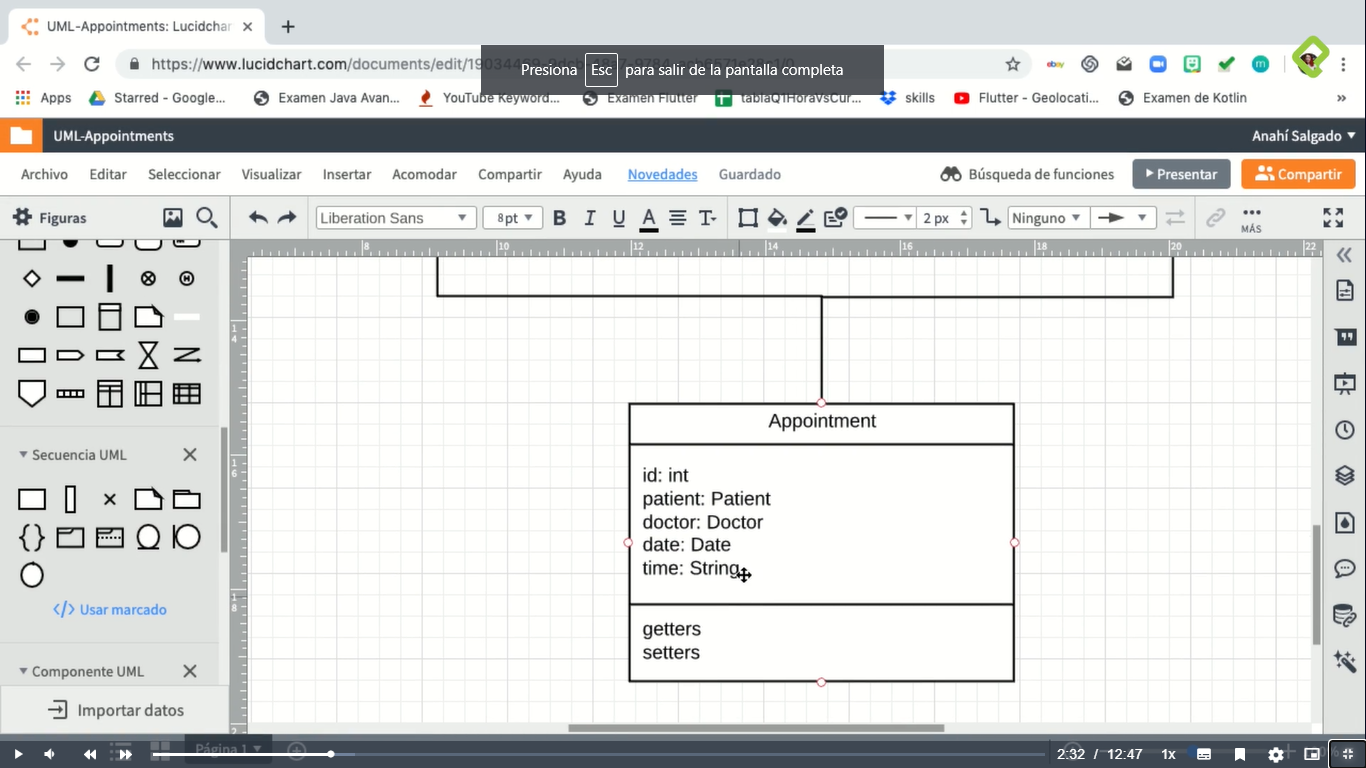
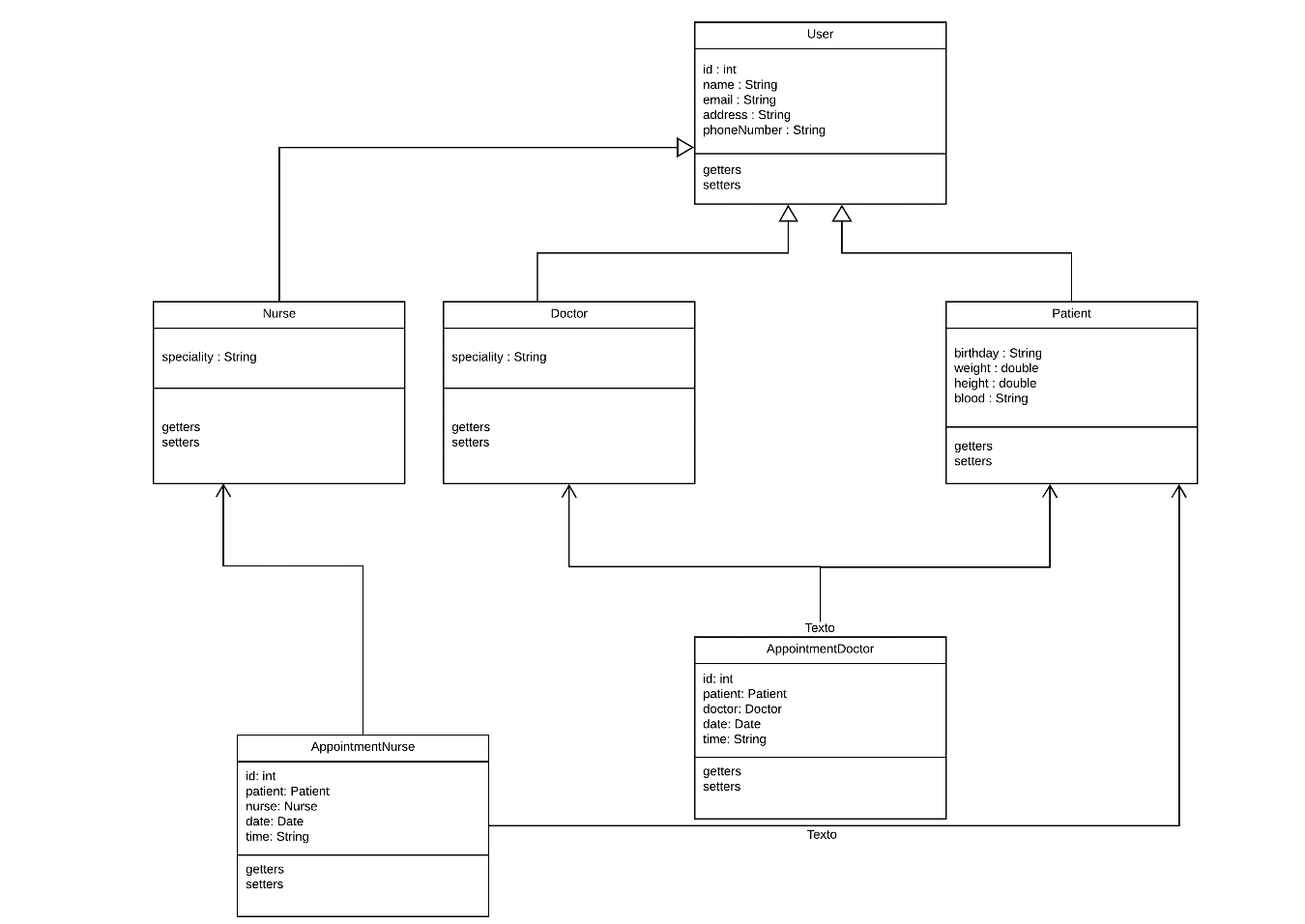
User paciente = **new** Patient ("Carlos", "carlos96@hotmail.com");

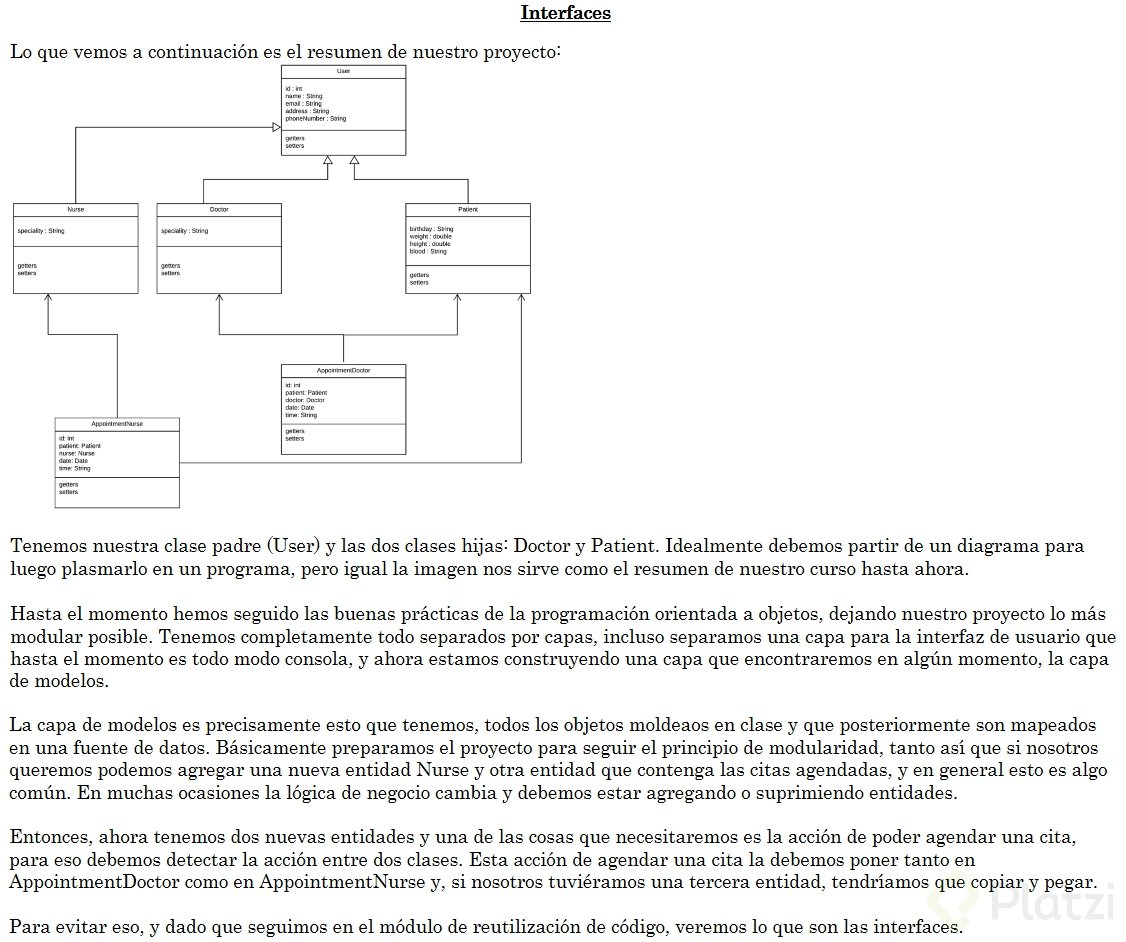
# Interfaces

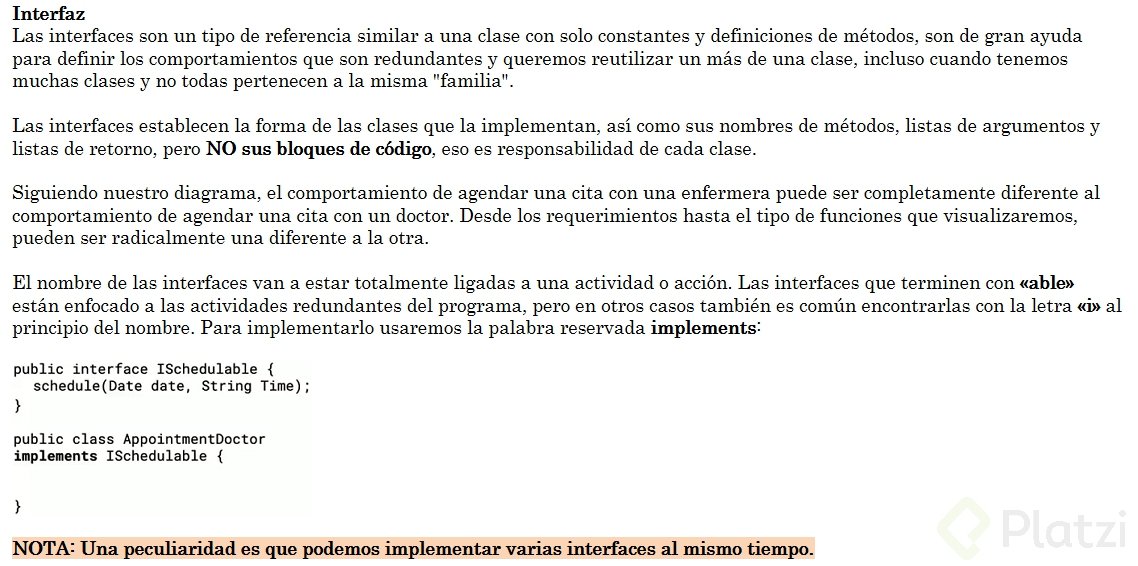
Las **Interfaces** son un tipo de referencia similar a una clase con solo constantes y definiciones de métodos, son de gran ayuda para definir los comportamientos que son redundantes y queremos reutilizar más de una clase, incluso cuando tenemos muchas clases y no todas pertenecen a la misma “familia”.

Las interfaces establecen la forma de las clases que la implementan, así como sus nombres de métodos, listas de argumentos y listas de retorno, pero NO sus bloques de código, eso es responsabilidad de cada clase.







# Creando una interfaz para definir si una fecha es agendable

**Composición de Interfaces en Clases**: abstraer todos los métodos/comportamientos de una clase para modularizarlos (comprimirlos, encapsularlos) en una interfaz y reutilizar su código en diferentes clases.

Las interfaces se crean utilizando la palabra reservada interface y se implementan en nuestras clases con implements.

Recuerda que podemos heredar (implementar) más de una interfaz, pero no podemos hacerlo de las clases padres o superclases.

**public** **interface** **ISchedulabe** {

**void** **schedule**(Date date, String Time);

}

**public** **class** **AppointmentDoctor** **implements** **ISchedulable** {

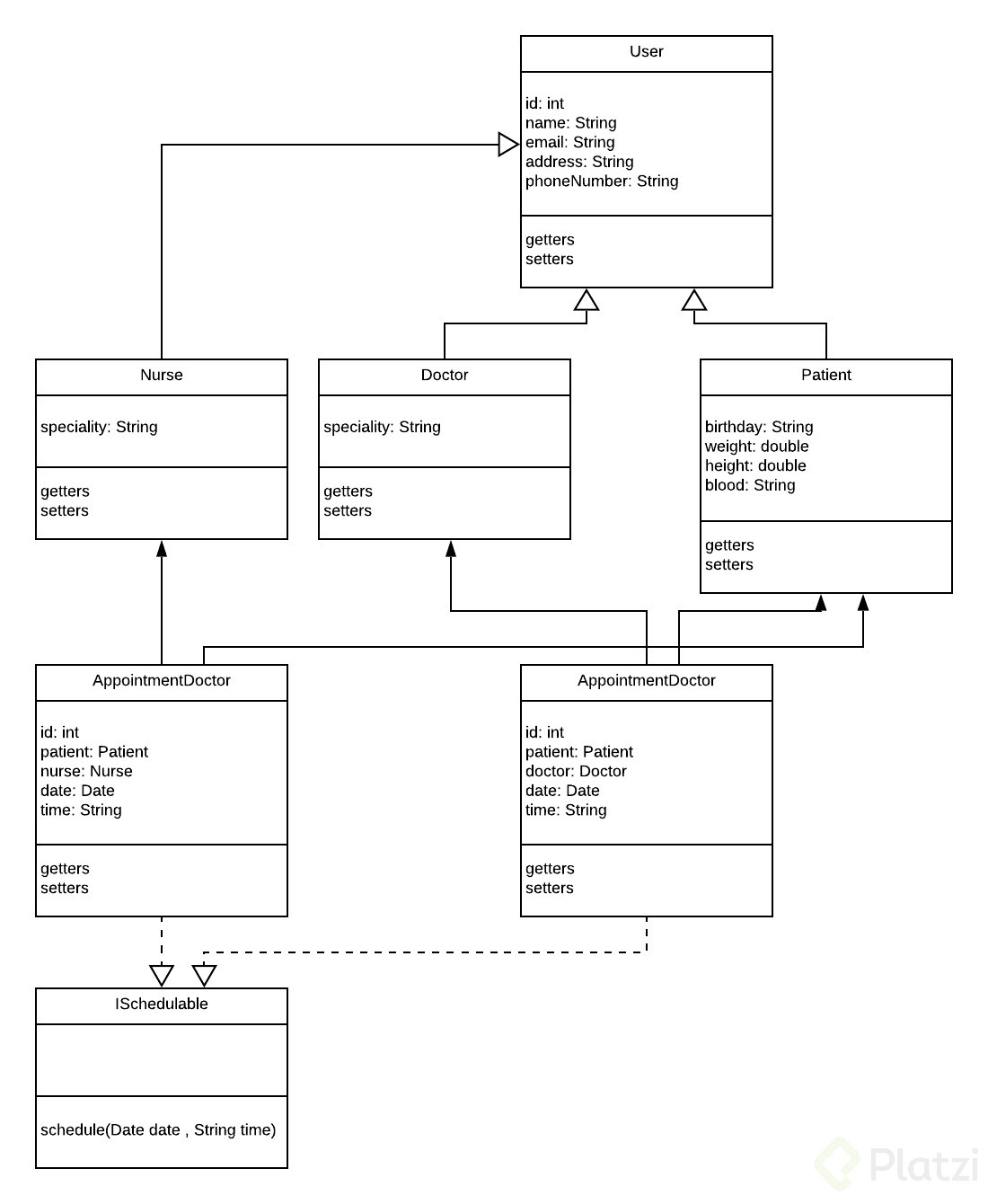
@Override

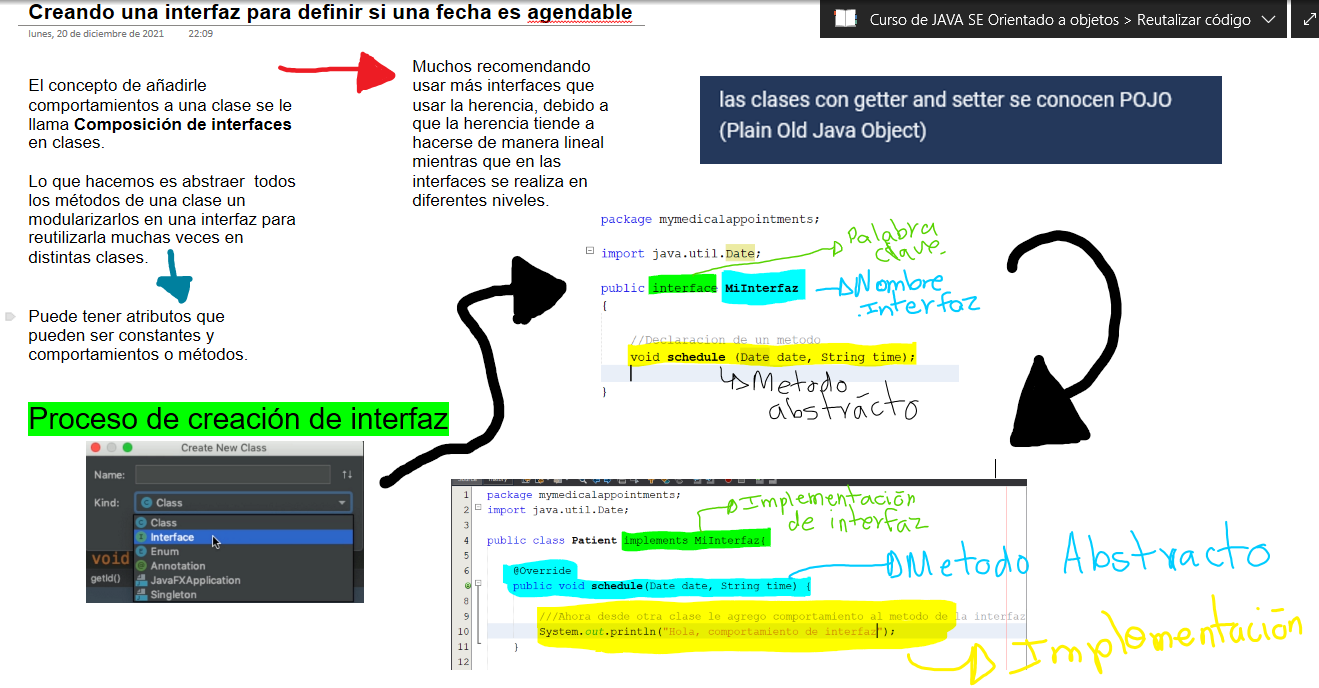
**public** **void** **schedule**(Date date, String Time) {

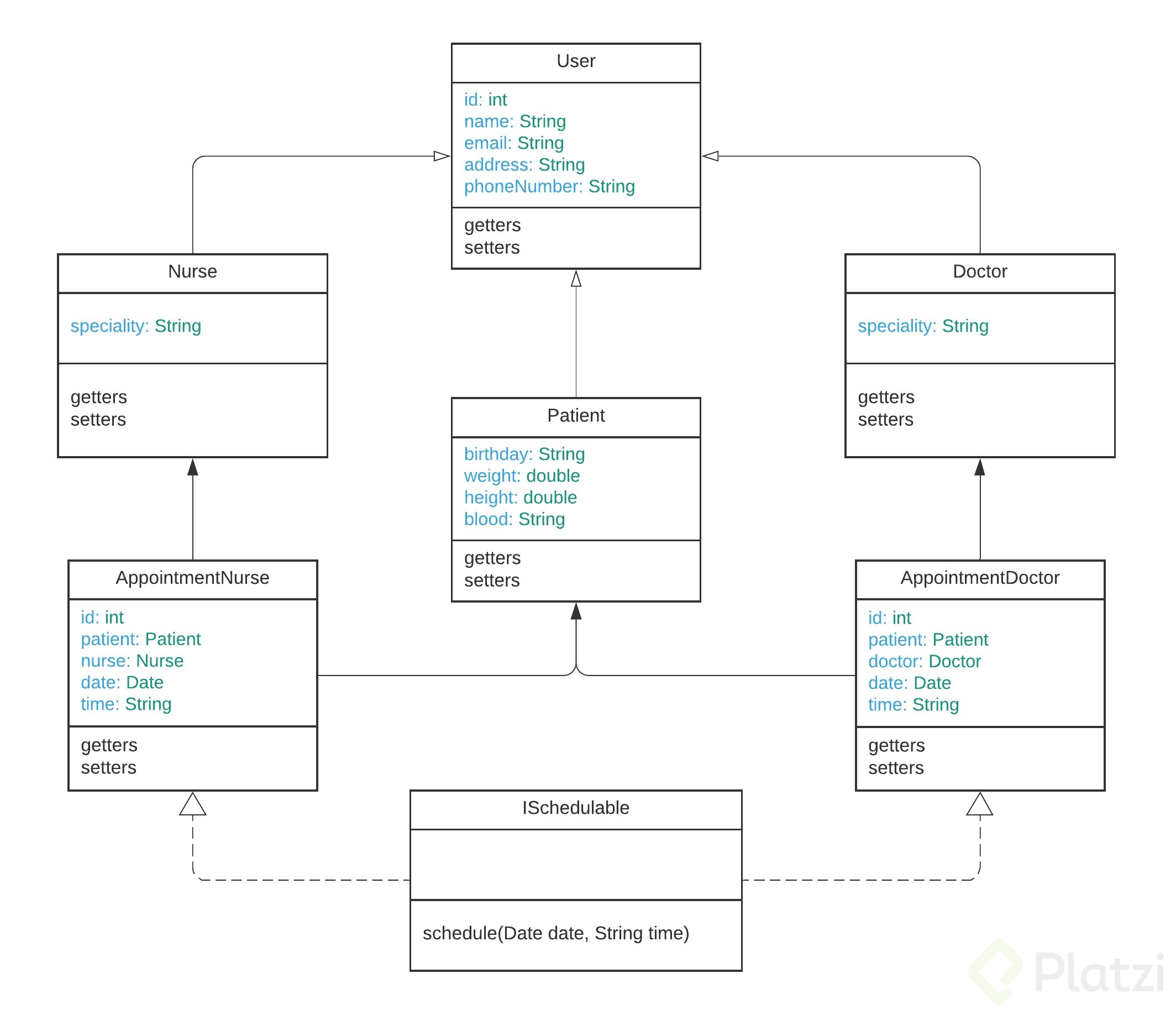
// ...

}

}







las clases con getter and setter se conocen POJO (Plain Old Java Object)

# Collections

Otras interfaces que son muy importantes en Java son los llamados **Collections**

Los Collections nos van a servir para trabajar con colecciones de datos, específicamente y **solamente con objetos**, para esto recuerda que tenemos disponibles nuestras clases Wrapper que nos ayudan a convertir datos primitivos a objetos.

Los collections se diferencian de los arrays en que su tamaño no es fijo y por el contrario es dinámico.

A continuación te muestro un diagrama de su composición:

Diagrama

Descripción generada automáticamente

Como podemos observar el elemento más alto es la interfaz **Collection**, para lo cual, partiendo de su naturalidad de interface, entendemos que tiene una serie de métodos “básicos” dónde su comportamiento será definido a medida que se vaya implementando en más elementos. De ella se desprenden principalmente las interfaces **Set** y **List.**

La interface **Set** tendrá las siguientes características:

Almacena objetos únicos, no repetidos.  
La mayoría de las veces los objetos se almacenarán en desorden.  
No tenemos índice.

La interface **List** tiene éstas características:

Puede almacenar objetos repetidos.  
Los objetos se almacenan en orden secuencial.  
Tenemos acceso al índice.

## Si seguimos analizando las familias tenemos que de **Set** se desprenden:

Clase HashSet  
Interfaz SortedSet y de ella la clase TreeSet.

**HashSet** los elementos se guardan en **desorden** y gracias al mecanismo llamado hashing (obtiene un identificador del objeto) **permite almacenar objetos únicos.**

**TreeSet** almacena **objetos únicos**, y gracias a su estructura de árbol el \*acceso es sumamente **rápido.**

## Ahora si analizamos la familia List, de ella se desprenden:

Clase **ArrayList** puede tener duplicados, no está sincronizada por lo tanto es más rápida  
Clase **Vector** es sincronizada, los datos están más seguros pero es más lento.  
Clase **LinkedList**, puede contener elementos duplicados, no está sincronizada (es más rápida) al ser una estructura de datos doblemente ligada podemos añadir datos por encima de la pila o por debajo.

Diagrama

Descripción generada automáticamente

## Sigamos con Map

Lo primero que debes saber es que tiene tres implementaciones:

HashTable  
LinkedHashMap  
HashMap  
SortedMap ➡️ TreeMap

Diagrama

Descripción generada automáticamente

La interfaz **Map** no hereda de la interfaz Collection porque representa una estructura de datos de Mapeo y no de colección simple de objetos. Esta estructura es más compleja, pues cada elemento deberá venir en pareja con otro dato que funcionará como la llave del elemento.

**Map**

Donde K es el key o clave  
Donde V es el value o valor

Podemos declarar un map de la siguiente forma:

Map<Integer, String> map = **new** HashMap<Integer, String>();

Map<Integer, String> treeMap = **new** TreeMap<Integer, String>();

Map<Integer, String> linkedHashMap = **new** LinkedHashMap<Integer, String>();

Como observas solo se puede construir el objeto con tres elementos que implementan de ella: **HashMap**, **TreeMap** y **LinkedHashMap** dejando fuera HashTable y SortedMap. SortedMap estará fuera pues es una interfaz y HashTable ha quedado deprecada pues tiene métodos redundantes en otras clases. Mira la funcionalidad de cada uno.

Como te conté hace un momento Map tiene implementaciones:

**HashMap:** Los elementos no se ordenan. No aceptan claves duplicadas ni valores nulos.  
**LinkedHashMap:** Ordena los elementos conforme se van insertando; provocando que las búsquedas sean más lentas que las demás clases.  
**TreeMap:** El Mapa lo ordena de forma “natural”. Por ejemplo, si la clave son valores enteros (como luego veremos), los ordena de menos a mayor.

Para iterar alguno de estos será necesario utilizar la interface **Iterator** y para recorrerlo lo haremos un bucle while así como se muestra:

## Para HashMap

// **Imprimimos** el **Map** con un **Iterador**

**Iterator** it = map.keySet().**iterator**();

**while**(it.hasNext()){

**Integer** key = it.next();

**System**.**out**.println("Clave: " + key + " -> Valor: " + map.get(key));

}

## Para LinkedHashMap

// **Imprimimos** el **Map** con un **Iterador**

**Iterator** it = linkedHashMap.keySet().**iterator**();

**while**(it.hasNext()){

**Integer** key = it.next();

**System**.**out**.println("Clave: " + key + " -> Valor: " + linkedHashMap.get(key));

}

## Para TreeMap

// **Imprimimos** el **Map** con un **Iterador**

**Iterator** it = treeMap.keySet().**iterator**();

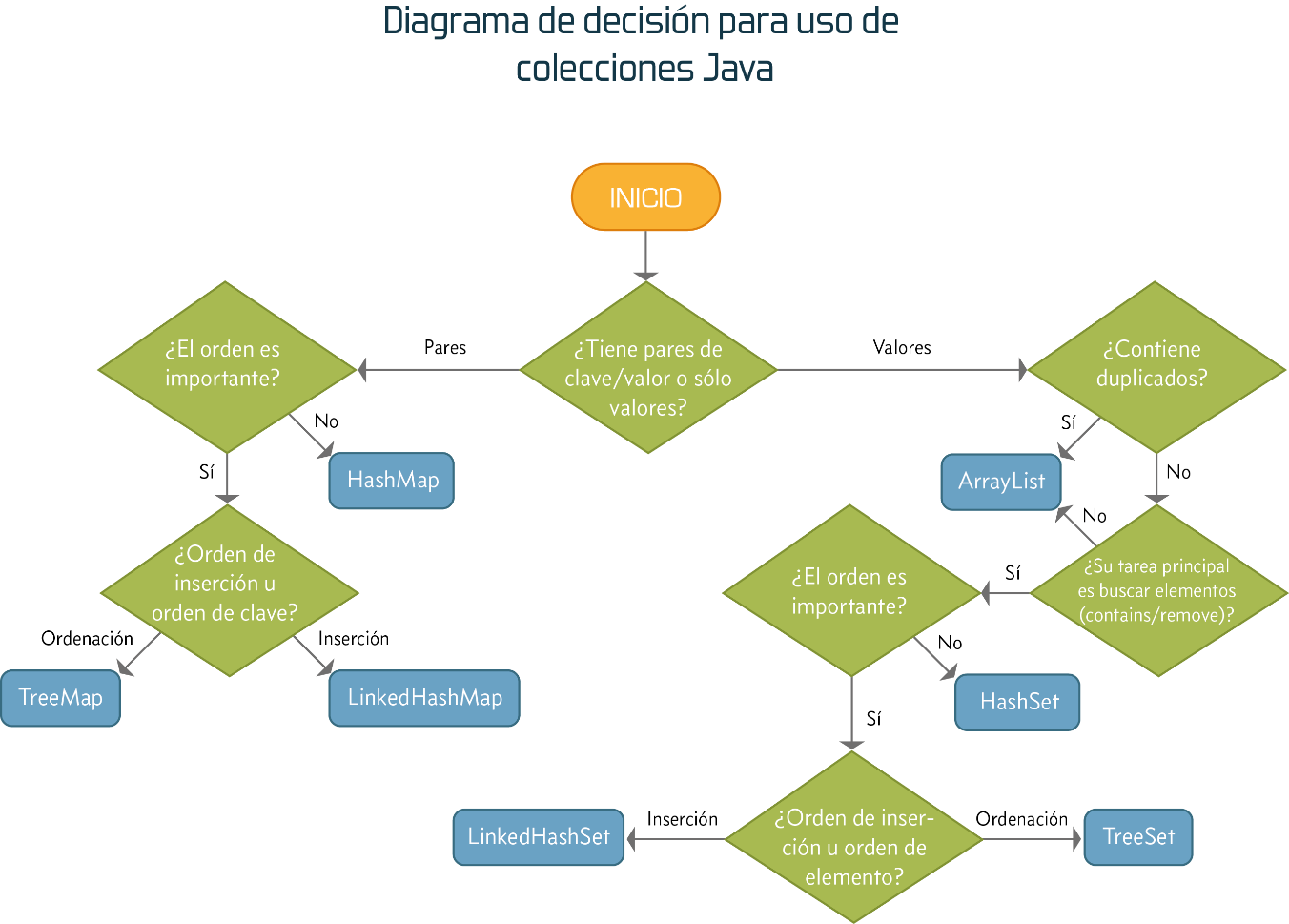
**while**(it.hasNext()){

**Integer** key = it.next();

**System**.**out**.println("Clave: " + key + " -> Valor: " + treeMap.get(key));

}

Ahora [lee esta lectura](https://docs.oracle.com/javase/tutorial/collections/interfaces/deque.html) y en la sección de tutoriales cuéntanos en tus palabras cómo funciona **Deque.**



<https://www.youtube.com/watch?v=bTu-fz1JmWQ>

# Clases Abstractas

A veces NO necesitamos implementar todos los métodos de una clase heredada o interfaz. No siempre necesitamos crear instancias o implementar todos los métodos heredados de una clase padre, así como tampoco podremos necesitamos algún método de nuestras interfaces, pero estas nos obligan a escribir el código de todos los métodos que definimos genéricamente.

Afortunadamente, las **Clases Abstractas** resuelven todos estos problemas. Son una combinación entre interfaces y herencia donde no implementaremos todos los métodos ni tampoco crearemos instancias.

**public** **abstract** **class** **Figura** {

// ...

}

**class** **Triangulo** **extends** **Figura** {

// ...

}

La herencia desprende, entre algunos, dos conceptos particulares, que son las CLASES ABTRACTAS y las INTERFACES.

Las CLASES ABSTRACTAS son aquellas clases que tienen por lo menos un método abstracto, es decir, un método no definido. Y además no pueden ser instanciadas, únicamente pueden ser heredadas (extends). Para definir una clase abstracta usamos:

**abstract** **class** **Ejemplo** {

//código

}

Las INTERFACES son siempre una clase abstracta pública, tanto así, que todos sus métodos son abstractos públicos implícitamente, es decir, no necesitamos usar las palabras reservadas public abstract al momento de declararlos.  
Las INTERFACES tampoco pueden ser instanciadas, únicamente pueden ser **implementadas** (implements).  
Y únicamente pueden contener constantes públicas.  
Para definir una INTERFAZ usamos:

**interface** **Ejemplo** {

//constantes y/o métodos

}

Xxxxxxxxxxxxxxxxxxxxx

<https://www.youtube.com/watch?v=J2aXYUA1h20>

xxxxxxxxxxxxxxxxxxxxxxxxxx

# Miembros abstractos

Los **Métodos Abstractos** son los métodos que debemos implementar obligatoriamente cada vez que usemos nuestras clases abstractas, mientras que los métodos que no sean abstractos van a ser opcionales.

**public** **abstract** **class** **Figura** {

**abstract** **void** **dibujar**(); // obligatorio

**void** **dibujar3D**(); // no es obligatorio

}

**class** **Triangulo** **extends** **Figura** {

**void** **dibujar**() {

// Instrucciones para dibujar el triángulo...

}

}

Recuerda los métodos abstractos solo se pueden implementar en clases abstractas. Y las clases abstractas no necesitan ser instanciadas para ser implementadas.

# Clases Anónimas

Las **Clases Anónimas** son una forma de instanciar clases abstractas sin necesidad de usar sus clases hijas. Pero este tipo de instanciación tiene algunas restricciones: el ciclo de vida de estas instancias NO es duradero, no las tendremos disponibles durante toda la ejecución del programa.

// Clase Abstracta:

**public** **abstract** **class** **Figura** {

**abstract** **void** **dibujar**();

}

// Clase Anónima:

User user = **new** User() {

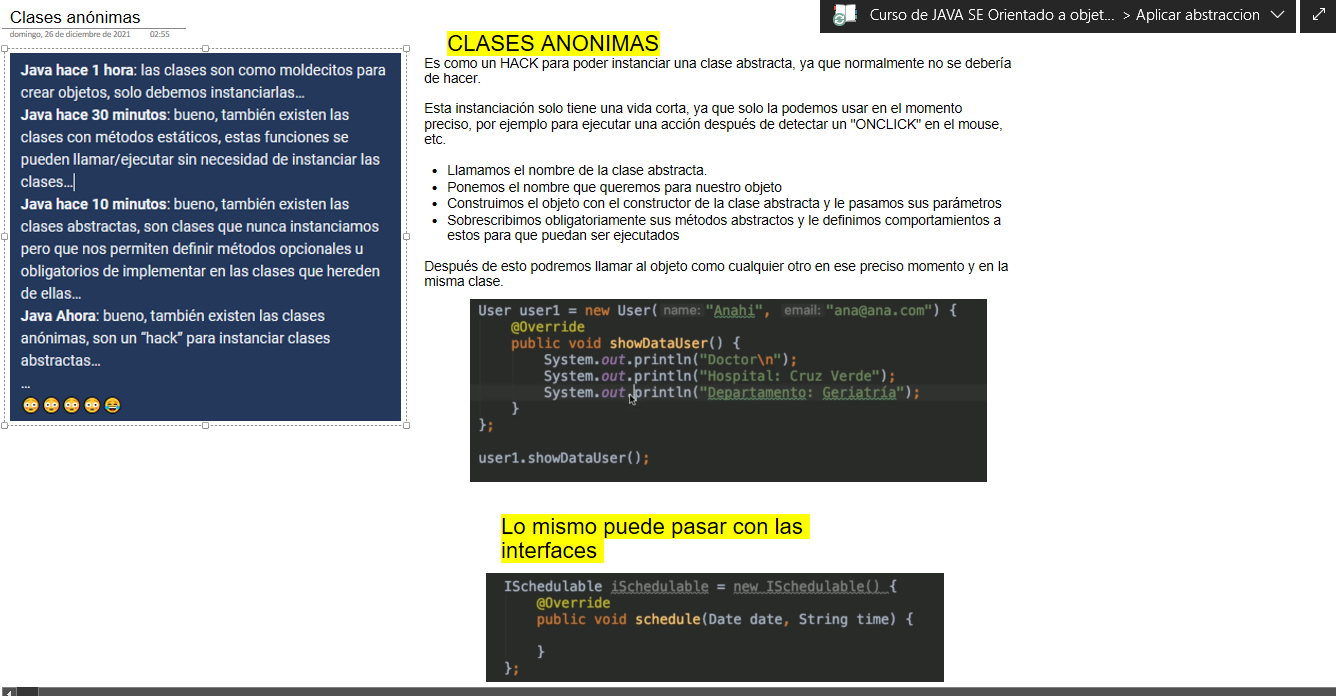
@Override

**public** **void** **showDataUser**() {

// Instrucciones...

}

};



# Interfaces en Java 8 y 9

Las **Interfaces** nos permiten usar métodos abstractos y campos constantes para implementar herencia/polimorfismo de forma muy similar a las clases abstractas.

A partir de Java 8 podemos tener implementación en métodos para heredar y reutilizar diferentes comportamientos. No todos los métodos de nuestras interfaces deben ser abstractos, ahora podemos usar el modificador de acceso default y desde Java 9 también private.

Recuerda que el nivel de acceso de default y private son los mismos que estudiamos en clases anteriores.

**public** **interface** **MyInterface** {

// Métodos default: nos permite heredar la definición

// de la función y también su implementación...

**default** **void** **defaultMethod**() {

privateMethod("Hello from the default method!");

}

// Métodos private: nos permiten definir comportamiento,

// pero solo se puede usar desde otras clases de esta

// interfaz, no se hereda a la clase hija....

**private** **void** **privateMethod**(final String message) {

System.**out**.println(message);

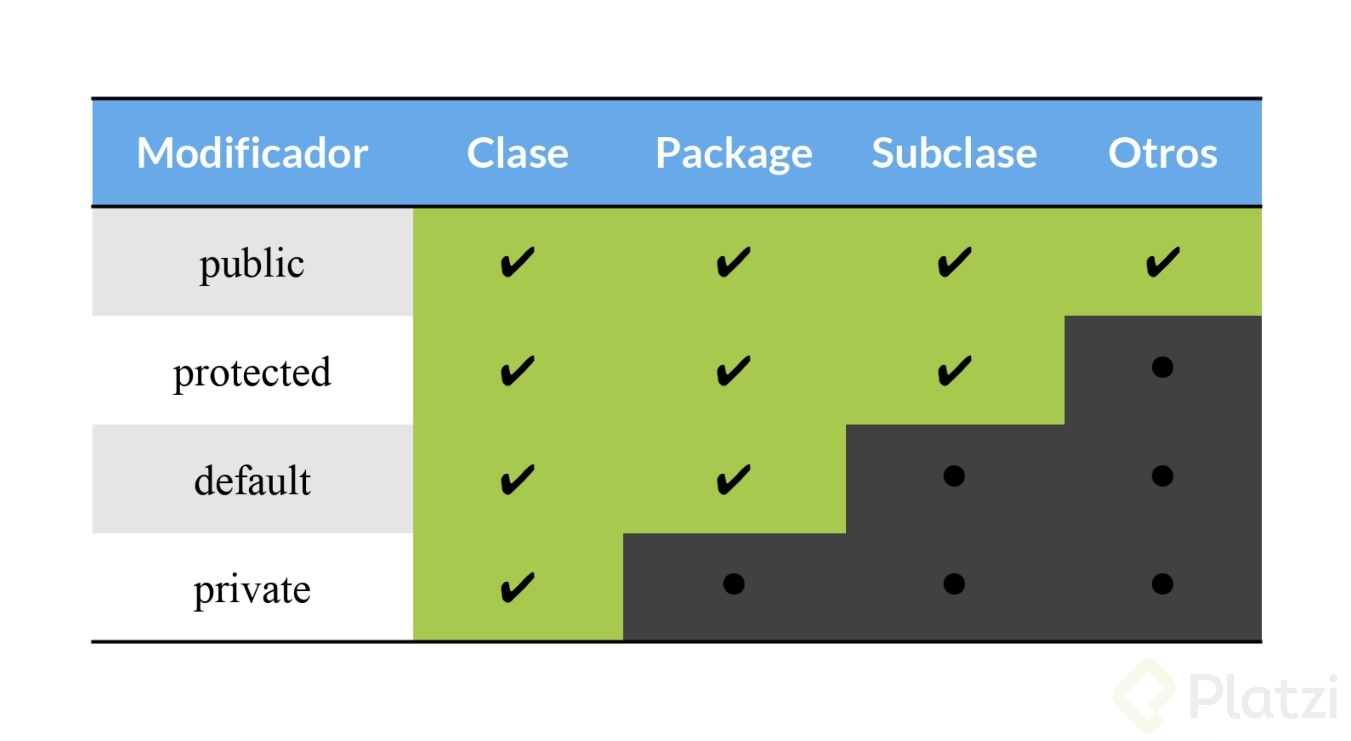
}

// Métodos abstractos: recuerda que todos los métodos

// son abstractos por defecto...

**void** **normalMethod**();

}



Con este cambio, la interfaz y la clase abstracta parecen ser practicamente lo mismo. Sin embargo, la interfaz sigue siendo una abstracción completa (diseñada para que se implementen todos los métodos). La posibilidad de crear métodos default parace obedecer únicamente a facilitar la actualización de las interfaces sin que esto rompa las clases que ya la implementaban con anterioridad.

Les comparto mis notas de esta Clase:

**Interfaces en Java 8 y 9**

La sintaxis de las interfaces cambio un poco a partir de Java 8 y 9.

* **Java 8**  
  Ya no todos los métodos tiene que ser abstractos pues se cuenta con un nuevo modificador de acceso para los métodos, default. Esto permite que los métodos dentro de una interfaz puedan tener implementación, y a su vez dicha implementación se puede reutilizar en cualquier Clase sin importar a que familia pertenezca. Este modificador default en las interfaces tiene las mismas reglas y restricciones que en en caso de los métodos convencionales, es decir puede ser accedido a nivel de la Clase y por Clases/Interfaces que estén dentro de otro paquete.
* **Java 9**

Se puede añadir el modificador de acceso private a los métodos de una Interfaz. Este modificador tiene las mismas restricciones que en el caso de los métodos convencionales, es decir, sólo puede ser accedido a nivel de la Clase. También permite hacer una implementación en el método de una Interfaz.

**public** **interface** **MyInterface** {

**default** **void** **defaultMethod**() {

privateMethod("Hello from the default method!")

}

**private** **void** **privateMethod**(**final** String string) {

System.out.println(string);

}

**void** **normalMethod**();

}

Este tipo de interfaces son muy útiles al trabajar con bases de datos.

# Herencia en interfaces

Las interfaces pueden heredar de otras interfaces utilizando la palabra clave extends, el concepto de herencia se aplicará como naturalmente se practica en clases, es decir, la interfaz heredará y adquirirá los métodos de la interfaz padre.

Una cosa interesante que sucede en caso de herencia con interfaces es que, aquí sí es permitido la herencia múltiple como ves a continuación:

**public** **interface** **IReadable** {

**public** **void** **read**();

}

**public** **interface** **Visualizable** **extends** **IReadable**, **Serializable** {

**public** **void** **setViewed**();

**public** Boolean **isViewed**();

**public** String **timeViewed**();

}

Además siguiendo las implementaciones de métodos default y private de las versiones Java 8 y 9 respectivamente podemos sobreescribir métodos y añadirles comportamiento, si es el caso.

**public** **interface** **Visualizable** **extends** **IReadable**, **Serializable** {

**public** **void** **setViewed**();

**public** Boolean **isViewed**();

**public** String **timeViewed**();

@Override

**default** **void** **read**() {

// TODO Auto-generated method stub

}

}

# Definiendo las citas disponibles

Algunas veces necesitamos trabajar las fechas como tipo de dato Date y otras veces como String. Para resolver esto podemos usar SimpleDateFormat.

SimpleDateFormat format = **new** SimpleDateFormat(pattern: "dd/MM/yyyy");

// Transformar fechas de formato String a Date:

**this**.date = format.parse(dateAsString);

// Transformar fechas de formato Date a String:

**this**.date = format.format(dateAsDate);

# Recorriendo estructuras de árbol en Java

Las **estructuras de árbol** pertenecen al grupo de estructuras de datos no lineales, es decir, donde toda la información es almacenada con un orden específico. En estas estructuras tenemos “troncos” principales con diferentes ramificaciones que surgen a partir de ellos. Son muy útiles para trabajar con grandes cantidades de datos organizados de forma jerárquica.

La forma de implementarlos en Java es usando un Map de tipo TreeMap. Recuerda que también podemos guardar Maps dentro de otros Maps. De esta forma podemos definir una lista ordenada de doctores y sus fechas disponibles para agendar citas médicas.

// 1. Doctor#1

// - - - Fecha#1

// - - - Fecha#2

// 2. Doctor#2

// - - - Fecha#1

// - - - Fecha#2

// 3. Doctor#3

// - - - Fecha#1

// - - - Fecha#2

Map<Integer, Map<Integer, Doctor>> doctors = **new** TreeMap<>();

# Cierre del curso: Continúa con Programación Funcional en Java

Felicitaciones por terminar el Curso de Introducción a Java SE!

En este curso aprendimos a implementar los 4 pilares de la programación orientada a objetos en Java con un proyecto para agendar citas médicas.

Recuerda resolver los ejercicios, completar el examen, darle 5 estrellas a la profesora y compartir tu proyecto, notas, dudas y comentarios en la sección de discusiones.

No olvides que puedes continuar tu ruta de aprendizaje de Java con los siguientes cursos:

* Curso de Java SE: Programación Funcional
* Curso de Java SE: Persistencia de Datos