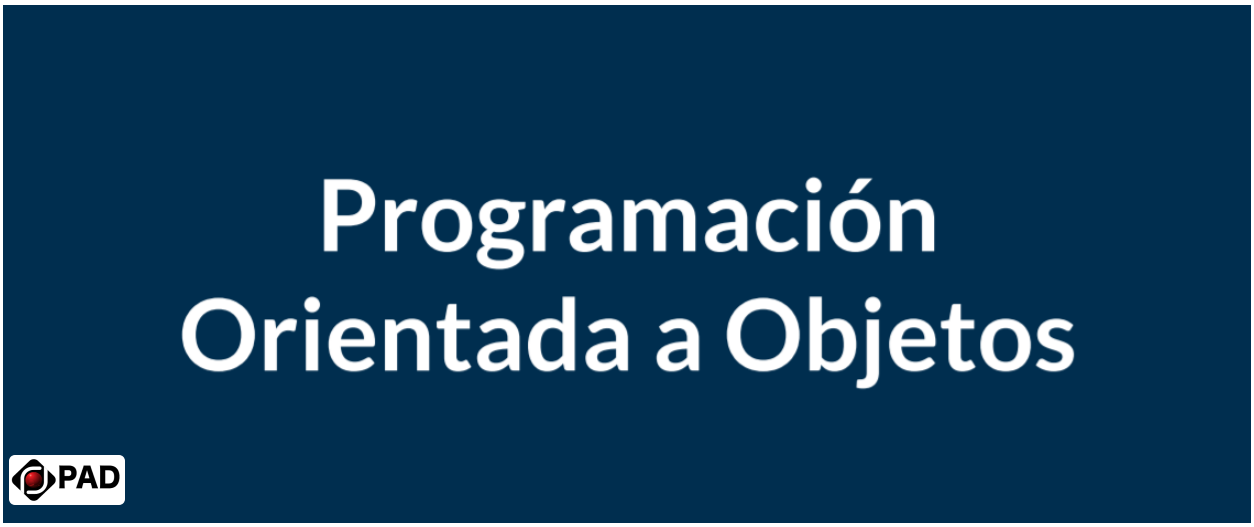


POO - Programación Orientada a Objetos - PADSA

Los 4 principios de la programación orientada a objetos.

- **Abstracción**

Definir las propiedades y métodos de un objeto.

- **Encapsulamiento**

Encerrar-Limitar (Encapsular) objetos, variables, métodos a un un alcance definido para tener interactividad o acceder a ellas según sea necesario.

- **Herencia**

Se heredan propiedades y métodos de una super clase llamada clase padre a varias subclases que tienen dichas propiedades en común y ciertas otras que diferencian cada subclase. Con el fin de reducir la redundancia de código, simplificando así nuestro código y volviéndolo escalable.

- **Polimorfismo**

Multi-forma, sobre escritura. Es la capacidad de las subclases de sobre escribir o modificar sus propiedades o métodos partiendo de los de la clase padre.

La POO se basa en: Objetos, Clases, Propiedades y Métodos.

¿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.

Abstracción: ¿Qué es una Clase?

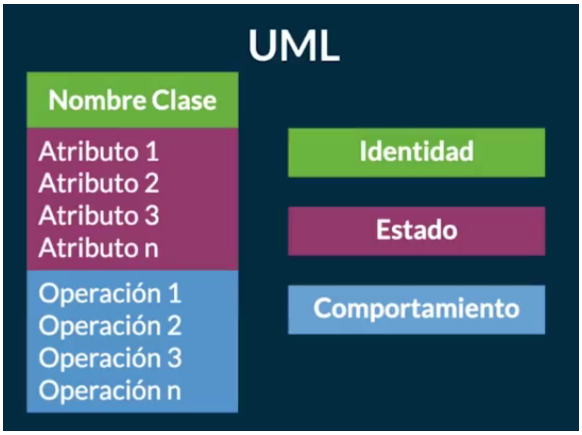
La **Abstracción** se trata de analizar objetos de forma independiente, sus propiedades, características y comportamientos, para *abstraer* su composición y generar un modelo, lo que traducimos a código como clases.

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), 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(el código no se rompe, solo por módulos)
- Que nuestro código sea mantenible.
- Mejorar la legibilidad.
- Resolución rápida de problemas.
- La modularidad es delegar el código en diferentes módulos(clases).

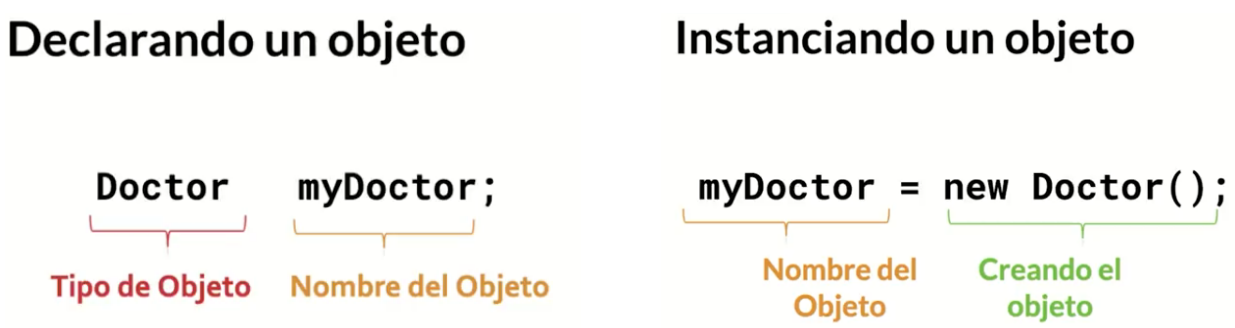
Ejemplo de clase

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...
    System.out.println(name);
  }
}
```



Declarar un Objeto:

Instanciar un Objeto:

```
// Tipo de Objeto ---- Nombre del Objeto
Doctor myDoctor;
```

```
// Otro objeto del mismo tipo Doctor:
Doctor anotherDoctor;
```

```
// Nombre del Objeto ---- Clase base para crear algún tipo de objetos
myDoctor = new Doctor();
```

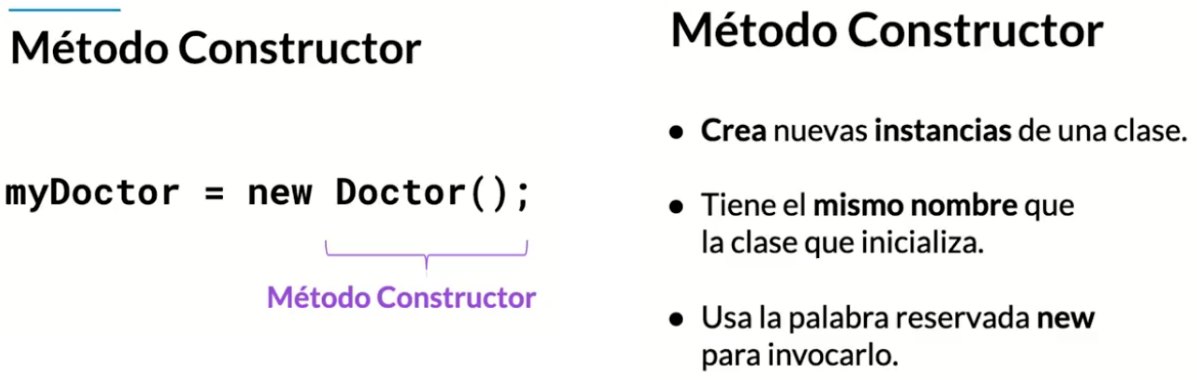
```
// Otro objeto
anotherDoctor =new Doctor();
```

Utilizar el objeto:

```
// Declarar el objeto ---- Instanciar el objeto
Doctor myDoctor =new Doctor();
myDoctor.name = "Anahí Salgado";
myDoctor.showName();
```

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.



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 usa cero o mas argumentos contenidos dentro de los paréntesis, *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.

```
//Método constructor sin argumentos
Doctor(){
System.out.println("-->Construyendo al objeto Doctor<--");
}
//Método Constructor con argumentos
Doctor(String name){
System.out.println("El nombre del Doctor asignado es: " + name);
//Llamado al método
Doctor anotherDoctor = new Doctor("Anahí Salgado");
```

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 {
```

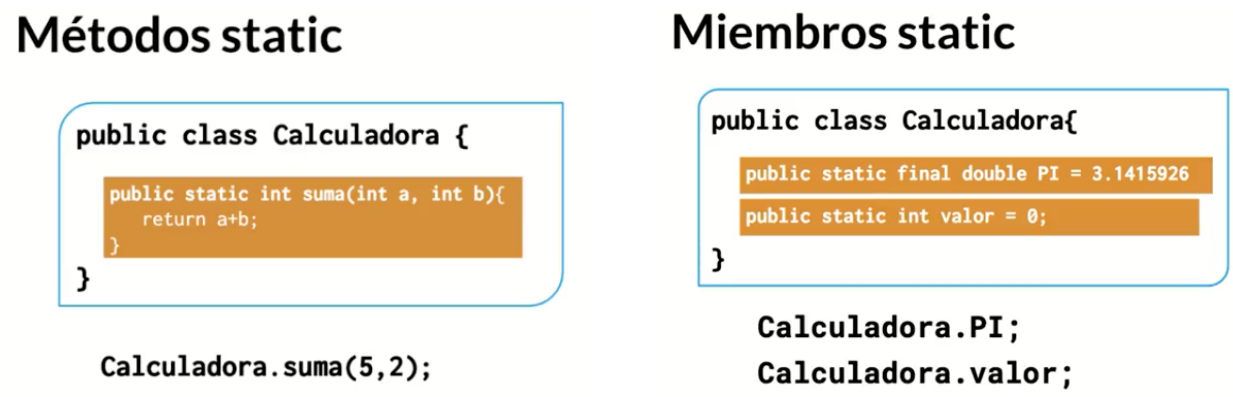
```
public static void (String[] args) {
int number = suma(3, 5);
    System.out.println(number + PI);
  }
}
```

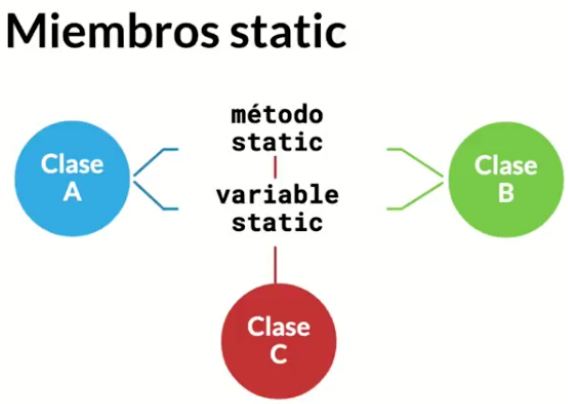
Se utiliza ***static*** para llamar una variable sin utilizar un objeto.

Puede ser accesado indicando el nombre de la clase, la notación punto y el nombre del método.

Se invoca en una clase que no tiene instancias de la la clase.

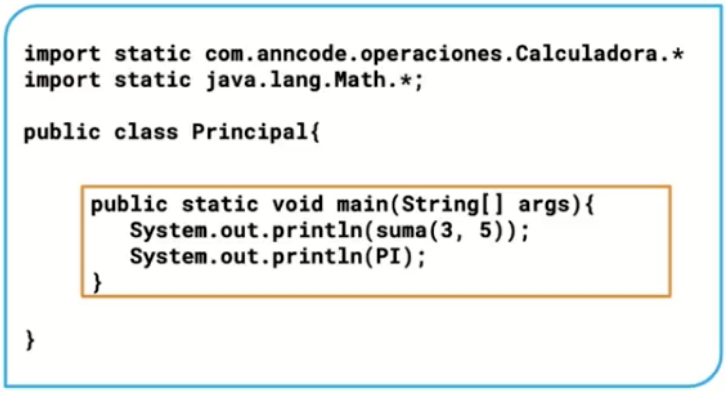
Para los métodos estáticos o las variables estáticos no es necesario crear un objeto para llamarlas, si no simplemente se puede acceder a ellos con el nombre de la clase.





La variables y métodos estáticos pueden ser modificados desde diferentes clases. ¿Qué 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.

Miembros static



```
import static com.anncode.operaciones.Calculadora.*
import static java.lang.Math.*;

public class Principal{

    public static void main(String[] args){
        System.out.println(suma(3, 5));
        System.out.println(PI);
    }

}
```

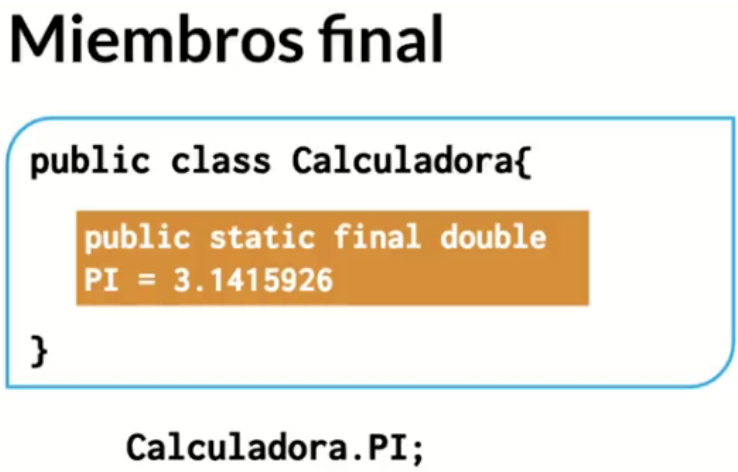
- las variables static no pertenecen al objeto sino a la clase, se se modifica la variable static esto se verá reflejado en todos los objetos que hayan sido creados con esa clase
- una variable de este tipo, mantiene su ciclo de vida en todo el programa mientras este corriendo el programa
- variable static puede mutarse(cambiar de valor), variable final es inmutable (no puede cambiar de valor)
- Un método estático solo puede acceder a datos estáticos

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



La Convención de JAVA es Nombrar los valores constantes en mayúsculas

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;
  }
}
```

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**.

Modificador	Clase	Package	Subclase	Otros
public	✔	✔	✔	✔
protected	✔	✔	✔	●
default	✔	✔	●	●
private	✔	●	●	●

Encapsulamiento: Nivel de acceso de una variables especifica. Los datos solo se podrán modificar a la limitación de la capa de acceso.

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;
  }
}
```

```
  private double weigth;

public void setWeigth(double weigth) {
    this.weigth = weigth;
  }

  public String getWeigth() {
    return this.weigth + "Kg.";
  }
```

Los getters y setters son métodos públicos que nos ayudan a leer y/o modificar nuestras variables privadas.

```
public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
```

```
        if (phoneNumber.length() > 8){
            System.out.println("El número telefónico debe ser de 8 dígitos máximo");
        }else if(phoneNumber.length() == 8){
            this.phoneNumber = phoneNumber;
        }
    }
```

Se puede condicionar la aceptación de los valores que se registran o reciben de argumentos, por ejemplo para solo aceptar números telefónicos de 8 dígitos.

```
public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        if (phoneNumber.length() > 8){
            System.out.println("El número telefónico debe ser de 8 dígitos máximo");
        }else if(phoneNumber.length() == 8){
            this.phoneNumber = phoneNumber;
        }
    }
```

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`.

*Los **Objetos** son entidades complejas formadas por la agrupación de diferentes variables y/o métodos. Los **Arrays** son agrupaciones de una cantidad fija de elementos de un mismo tipo de dato.*

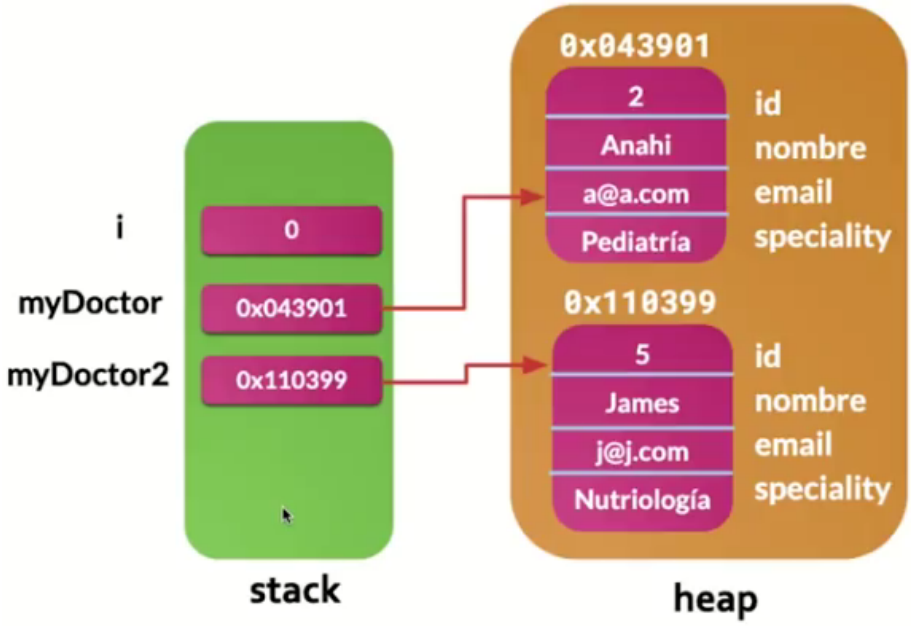
Variable vs. Objeto: Un vistazo a la memoria

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.



Clases Anidadas

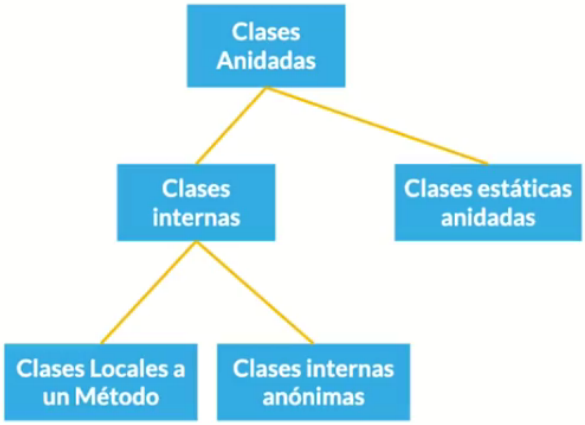
Las **Clases Anidadas** o **Clases Helper** son clases dentro de otras clases que agrupamos por su **lógica** y/o características en común.

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.

```
class ClaseExterior {
····class ClaseAnidada {
····}
}
```

CLASES ANIDADAS



Clases Estáticas: No necesitan crear instancias para llamarlas, solo pueden llamar a los métodos estáticos.

Clases Internas y Locales a un método

```
public class Outer {
  public class Inner {

  }

}


public class Main {
  public static void main(String[] args){
     Outer outer = new Outer();
     Outer.Inner inner = outer.new Inner();
  }
}
```

Clases Internas

```
public class Enclosing {
  void run() {
     class Local {
        void run() {

        }
     }

     Local local = new Local();
     local.run();
  }

}

public class Main {
  public static void main(String[] args){
     Enclosing enclosing = new Enclosing();
     enclosing.run();
  }
}
```

Clases 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**

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`.

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*.

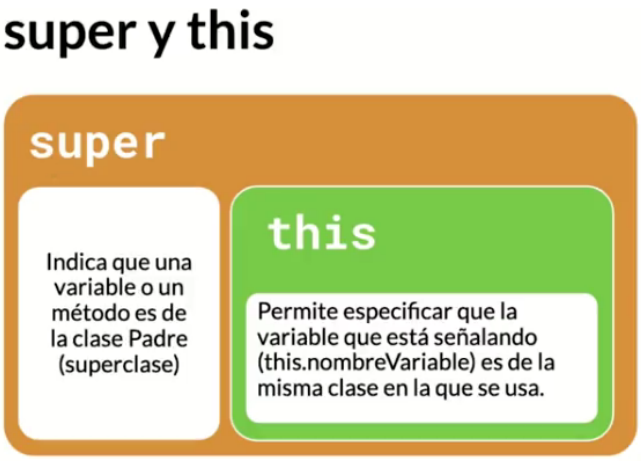
```
public class User {
int age = 1;

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;
  }
}
```



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.

Cuando es necesario crear un constructor? y cuando no?

Cuando necesites tener información desde el momento en que creas un objeto es necesario que utilices un constructor o un builder (sí son muchos atributos ó si hay algunos opcionales).