**ORIENTACIÓN A OBJETOS**

Surge a partir de los problemas que tenemos y necesitamos plasmar en código

**PARADIGMA**

Teoría que suministra la base y modelo para resolver problemas

**SE COMPONE DE ESTOS 4 ELEMENTOS:**

CLASES - PROPIEDADE - MÉTODOS - OBJETOS

**PILARES DE LA PROGRAMACIÓN A OBJETOS:**

ENCAPSULAMIENTO - ABSTRACCIÓN - HERENCIA - POLIMORFISMO

**UML (UNIFIED MODELING LANGUAGE)**

LENGUAJE DE MODELADO UNIFICADO

* CLASES
* CASOS DE USO
* OBJETOS
* ACTIVIDADES
* ITERACION
* ESTADOS
* IMPLEMENTACION

**OBJETOS**

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

**PROPIEDADES**

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.

**COMPORTAMIENTO O MÉTODOS**

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**

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.



**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).

**UML**

* + NOMBRE CLASE
  + Atributo 1 Identidad
  + Atributo 2
  + Atributo 3
  + Atributo n Estado
  + Operacion 1
  + Operacion 2 Comportamiento
  + Operacion 3
  + Operacion n

Ejemplo:

* Person
  + Name
  + 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.

**CLASE**

* Modularidad
* Divide el programa en diferentes partes o módulos / clases
* Separar las clases en archivos

**DECLARANDO UN OBJETO**

Clase / Nuevo nombre de variable a instanciar en representación de la clase = new Clase ();

Tipo de objeto Nombre del objeto Instanciando

Creando el objeto

**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.

* Crea nuevas instancias de una clase
* Tiene el mismo nombre que la clase que inicializa
* Usa la palabra reservada new para invocarlo.

Método constructor usa cero o más argumentos contenidos dentro de los paréntesis que siguen al nombre. No regresa un valor.

**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.

**METODOS STATIC**

Puede ser accesado indicando el nombre de la clase, la anotación punto y el nombre del método. Se invoca en una clase que no tiene instancias de la clase.

EJEMPLOS:

Public class Calculadora {

Public static int suma (int a, int b) {

return a+b;

}

}

Calculadora.suma(5, 2);

Public class Calculadora {

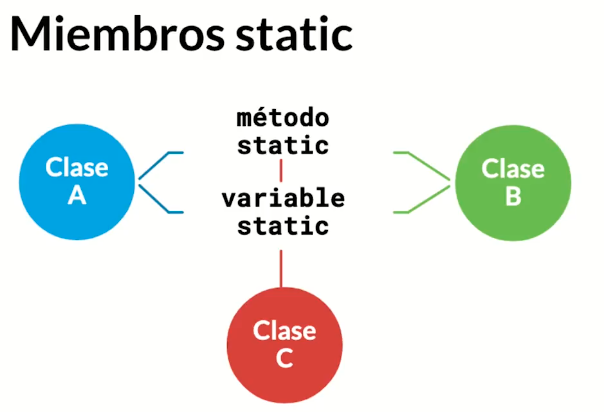
Public static final double PI = 3.1415926

public static int valor = 0;

}

Calculadora.PI;

Calculadora.valor;

****

**¿Por qué utilizar variables estáticas y constantes?**

Las variables estáticas y las constantes son herramientas valiosas en la programación, ya que ofrecen flexibilidad y control. Una variable estática nos permite compartir datos a lo largo de todo el programa, siendo accesibles desde cualquier clase, mientras que las constantes nos garantizan que ciertos valores no cambiarán.

**DEFINICIÓN DE CONSTANTES EN JAVA**

Para declarar una constante, utilizamos la palabra clave final, lo cual asegura que una variable no podrá ser reasignada una vez inicializada. Es común en Java que las constantes también se declaren como estáticas mediante static final, aunque no es una regla obligatoria.

**¿CUÁNDO USAR LA PALABRA CLAVE FINAL?**

La palabra final es especialmente útil en situaciones donde garantizar la inmutabilidad es necesario. Es preferido en casos donde:

* **Seguridad**: Evita que los valores críticos se cambien por accidente.
* **Consistencia**: Proporciona un único punto de verdad para valores invariables.
* **Optimización**: Permite al compilador hacer optimizaciones, dado que el valor no cambiará.

En la práctica, decidir cuándo usar static y final depende de las necesidades específicas del programa. Habiendo dominado estos conceptos, se estará mejor preparado para escribir código más eficiente y organizado. ¡Continúa explorando y aprendiendo, y verás cómo tus habilidades en Java se consolidan!

**Miembros final**

**Eje:**

Public class calculadora {

Public static final double PI = 3.1416

}

calculadora.PI;

**FINAL**

Cuando se declara una variable como final, significa que una vez que se le asigna un valor, ese valor no puede ser cambiado. Es útil cuando se desea crear constantes o asegurar que una variable no sea modificada después de su inicialización.

**SOBRECARGA**

A veces necesitamos que dos o más métodos tengan el mismo nombre pero con diferentes argumentos.

EJE:

Public class calculadora{

Public int suma (int a, int b){

return a+b;

}

Public float suma (float a, float b){

return a+b;

}

Public float suma (int a, float b){

return a+b;

}

}

**SOBRECARGA DE CONSTRUCTORES**

La sobrecarga de constructores se usa para inicializar objetos.

La sobrecarga de constructores permite que una clase tenga más de un constructor, siempre y cuando tengan diferentes firmas (es decir, diferente número o tipo de parámetros). Esto proporciona flexibilidad al crear objetos de la clase, ya que puedes instanciarlos de distintas maneras según las necesidades.

**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.

En Java, los modificadores de acceso son palabras clave que determinan el nivel de visibilidad o acceso que tienen las clases, métodos, variables y constructores dentro de tu programa. Estos modificadores controlan desde qué otros lugares en el código (como clases y paquetes) se puede acceder a un miembro de una clase

**PUBLIC**

Cuando se declara un miembro de una clase como public, significa que puede ser accedido desde cualquier otra clase, sin importar en qué paquete se encuentre.

**PROTECTED**

El modificador protected permite que un miembro de una clase sea accesible desde:

* La propia clase.
* Cualquier clase que esté en el mismo paquete.
* Subclases, incluso si están en otros paquetes.

**PRIVATE**

Un miembro declarado como private solo puede ser accedido dentro de la propia clase. No es accesible desde ninguna otra clase, ni siquiera desde clases que hereden de la clase que tiene el miembro private.

**DEFAULT**

Cuando un miembro de una clase no tiene ningún modificador de acceso (es decir, se deja sin especificar), se le asigna el modificador de acceso por defecto, que permite que el miembro sea accesible solo desde clases en el mismo paquete.

**GETTERS Y SETTERS**

En Java, los getters y setters son métodos que se utilizan para acceder y modificar los valores de los atributos de una clase. Estos métodos son parte de una técnica común llamada encapsulamiento, que es uno de los pilares de la programación orientada a objetos. La idea principal del encapsulamiento es ocultar el acceso directo a los atributos de una clase y proporcionar métodos controlados para interactuar con ellos.

**GETTER:**

Es un método que se utiliza para obtener el valor de un atributo privado de una clase

**SETTER:**

Es un método que se utiliza para modificar el valor de un atributo privado de una clase.

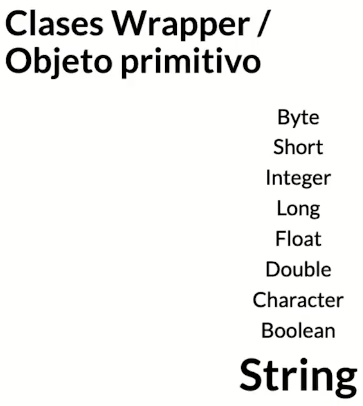
**VARIABLE**

Una variable es un espacio de almacenamiento en memoria que está asociado con un nombre y tiene un tipo de datos específico. Las variables se utilizan para almacenar datos que pueden cambiar durante la ejecución del programa. En Java, las variables pueden ser de tipos primitivos o de tipos de referencia.

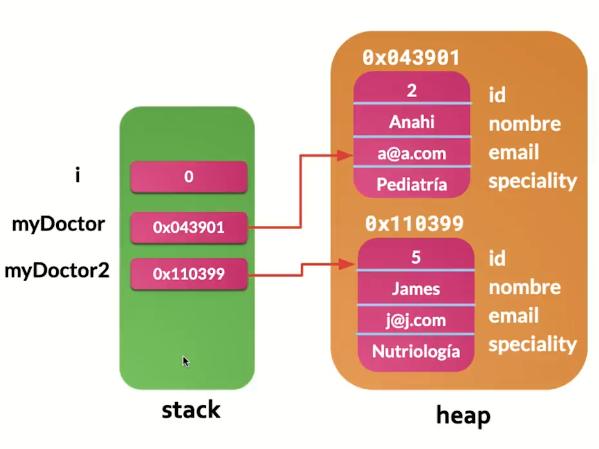
* Un numero
* Un caracter
* Un valor verdadero falso

**OBJETOS**

Un objeto es una instancia de una clase. Es una entidad en el programa que tiene atributos (variables) y comportamientos (métodos) definidos por su clase. Los objetos se crean utilizando la palabra clave new en Java y se almacenan en la memoria heap (montículo).



**MEMORIA**



Al hacer una igualacion entre objetos, no estamos asignando el mismo valor de obj1 a obj2, estamos asignando en verdad las direcciones de memoria

**CLASES ANIDADA**

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

**CLASES ANIDADAS ESTÁTICAS**

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.

AMAR LAS CLASES ANIDADAS ESTATICAS

**POR SI LAS MOSCAS:**

2. Clases Anidadas No Estáticas (Inner Classes)

**Las clases anidadas no estáticas (también conocidas como inner classes) son clases que tienen acceso tanto a los miembros estáticos como no estáticos de la clase externa. Se instancian a través de una instancia de la clase externa, ya que están asociadas con una instancia específica de la clase externa.**

Características:

* **Necesita una instancia de la clase externa para ser instanciada.**
* **Puede acceder a los miembros de la clase externa, tanto estáticos como no estáticos.**

Ejemplo:

**java**

**Copiar**

**public class Externa {**

**private String mensaje = "Mensaje no estático desde la clase externa";**

**// Clase anidada no estática**

**public class Anidada {**

**public void mostrarMensaje() {**

**System.out.println(mensaje); // Accede a los miembros no estáticos de la clase externa**

**}**

**}**

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

**Externa externa = new Externa(); // Se crea una instancia de la clase externa**

**Externa.Anidada anidada = externa.new Anidada(); // Instanciación de la clase anidada**

**anidada.mostrarMensaje();**

**}**

**}**

**En este ejemplo, la clase Anidada es una clase anidada no estática (o inner class). Para crear una instancia de Anidada, necesitas primero una instancia de la clase Externa.**

3. Clases Locales (Local Classes)

**Las clases locales son clases definidas dentro de métodos, constructores o bloques de inicialización. Solo son accesibles dentro del bloque donde se definen y tienen acceso a las variables locales y parámetros del método, pero solo si estas son finales o efectivamente finales.**

Características:

* **Se definen dentro de métodos o bloques de código.**
* **Tienen acceso a las variables locales y parámetros del método, pero solo si esas variables son final o efectivamente finales (es decir, no se modifican después de su inicialización).**

Ejemplo:

**java**

**Copiar**

**public class Externa {**

**public void metodo() {**

**final String mensaje = "Mensaje desde la clase local";**

**// Clase local dentro de un método**

**class Local {**

**public void mostrarMensaje() {**

**System.out.println(mensaje); // Accede a la variable local 'mensaje'**

**}**

**}**

**// Instanciando la clase local dentro del método**

**Local local = new Local();**

**local.mostrarMensaje();**

**}**

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

**Externa externa = new Externa();**

**externa.metodo();**

**}**

**}**

**Aquí, la clase Local está definida dentro del método metodo(). La clase Local tiene acceso a la variable local mensaje, pero esta variable debe ser final o efectivamente final.**

4. Clases Anónimas (Anonymous Classes)

**Las clases anónimas son una forma especial de clase local que no tiene un nombre. Se utilizan generalmente para crear implementaciones rápidas de interfaces o clases abstractas, especialmente cuando necesitas pasar un comportamiento específico a un método o constructor.**

Características:

* **No tienen un nombre, por eso se llaman anónimas.**
* **Son útiles para instanciar interfaces o clases abstractas con una implementación concreta en el mismo lugar donde se crean.**
* **Se definen generalmente en el momento de la instancia de un objeto.**

Ejemplo:

**java**

**Copiar**

**public class Externa {**

**public void ejecutar() {**

**// Clase anónima que implementa la interfaz Runnable**

**Runnable tarea = new Runnable() {**

**@Override**

**public void run() {**

**System.out.println("Ejecutando tarea en un hilo.");**

**}**

**};**

**// Ejecutando la tarea en un nuevo hilo**

**new Thread(tarea).start();**

**}**

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

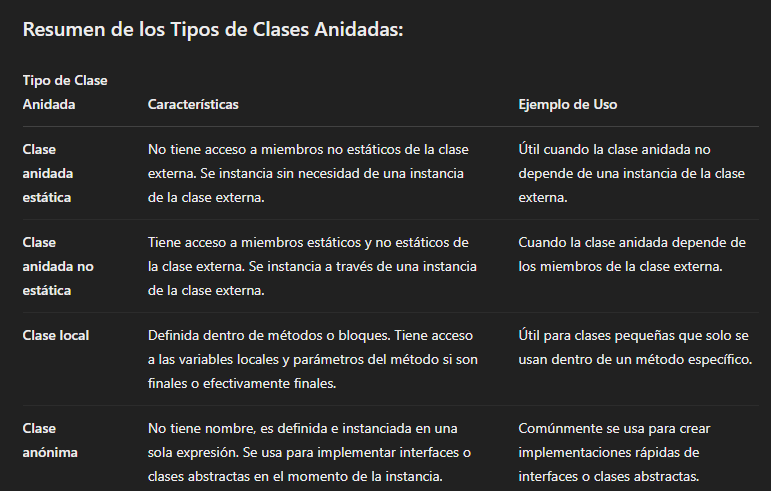
**Externa externa = new Externa();**

**externa.ejecutar();**

**}**

**}**

**En este ejemplo, se define una clase anónima que implementa la interfaz Runnable directamente en el momento de su instanciación. Esta clase no tiene un nombre y es utilizada para pasar el comportamiento requerido a la clase Thread.**

****

**DRY**

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.

**HERENCIA**

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.

**SUPER**

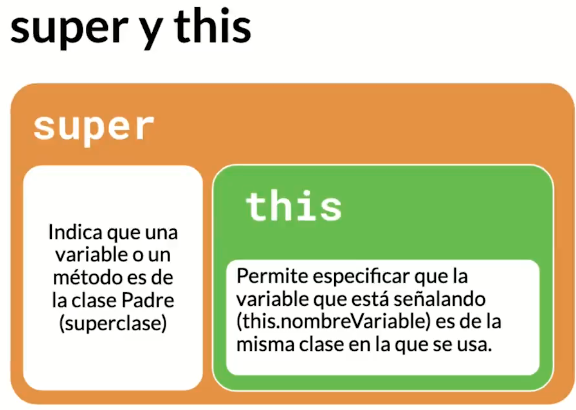
La palabra clave super se utiliza para referirse a la clase padre (superclase) de la clase actual (subclase). super tiene varias aplicaciones que explicaremos a continuación.

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.

**THIS**

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.

****

**SOBREESCRITURA**

Cuando una clase hereda de otra y en esta clase hija se redefine un método con una implementación distinta a la de la clase padre. La sobreescritura de constructores consiste en usar los miembros heredados de una superclase pero con argumentos diferentes. Los métodos marcados como final o static no se pueden sobrescribir.

**SOBREESCRITURA DE CONSTRUCTORES**

Un constructor en una subclase usando los miembros heredados de la superclase con argumentos diferentes.

**@Override**

La anotación @Override en Java es una anotación (o annotation) que se utiliza para indicar que un método está sobrescribiendo un método de una clase padre o superclase. Este es un concepto clave en la herencia y la programación orientada a objetos, ya que permite a las subclases proporcionar una implementación específica de un método que ya está definido en su clase base.

Cuando una subclase sobrescribe un método de la clase padre, debe coincidir exactamente en términos de:

* Nombre del método.
* Tipo y número de parámetros.
* Tipo de valor de retorno (o void si no retorna nada).

EJE::

class Animal {

// Método en la clase padre (superclase)

public void hacerSonido() {

System.out.println("El animal hace un sonido");

}

}

class Perro extends Animal {

// Método sobrescrito en la clase hija (subclase)

@Override

public void hacerSonido() {

System.out.println("El perro dice: ¡Guau!");

}

}

public class Main {

public static void main(String[] args) {

Animal animal = new Perro();

animal.hacerSonido(); // Salida: El perro dice: ¡Guau!

}

}

Salto de linea en java: \n

**POLIMORFISMO**

Posibilidad de sobreescribir un método con comportamientos diferentes. El concepto de polimorfismo es central en la programación orientada a objetos, permitiendo a los métodos realizar diferentes acciones según el contexto. Sobrescribir toString es un excelente ejemplo de polimorfismo. Al cambiar el comportamiento predefinido del método para dar una salida diferente o más completa, muestras cómo el polimorfismo permite darle múltiples formas a un solo método, contestando a la noción de "muchas formas" que el nombre sugiere.

* HERENCIA CLASES
* METODOS SOBREESCRITOS
* MUCHAS FORMAS

**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.

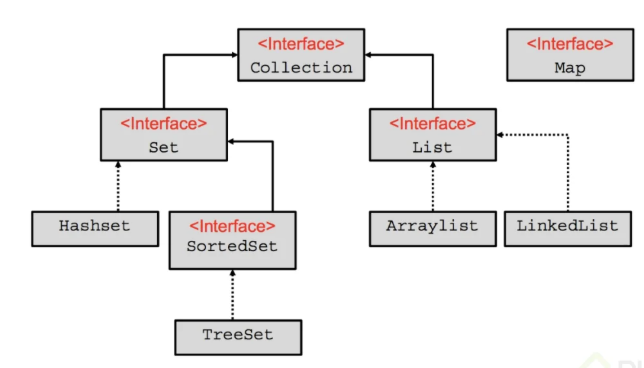
Es un tipo de referencia similar a una clase que podría contener solo constantes y definiciones de métodos. Se establece la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno, pero no bloques de código).

**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.



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.

**SET**

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.

**LIST**

La interface List tiene éstas características:

Puede almacenar objetos repetidos.

Los objetos se almacenan en orden secuencial.

Tenemos acceso al índice.

**HASHSET**

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

**TREESET**

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

**ARRAYLIST**

Clase ArrayList puede tener duplicados, no está sincronizada por lo tanto es más rápida.

**VECTOR**

Clase Vector es sincronizada, los datos están más seguros pero es más lento.

**LINKEDLIST**

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.

**MAP**

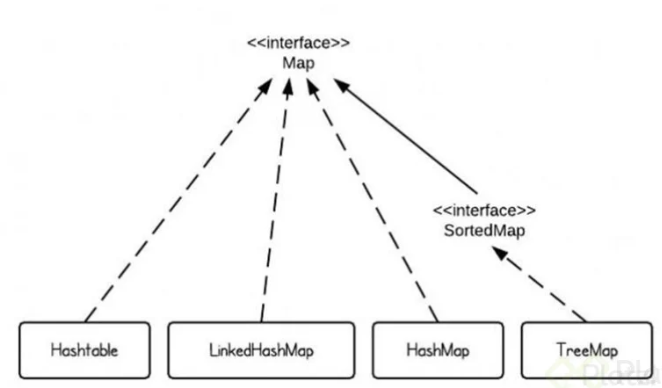
Lo primero que debes saber es que tiene tres implementaciones:

HashTable

LinkedHashMap

HashMap

SortedMap ➡️ TreeMap



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>();

**HASHMAP**

HashMap: Los elementos no se ordenan. No aceptan claves duplicadas ni valores nulos.

**EJE:**

**// 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));**

**}**

****

**LINKEDHASHMAP**

LinkedHashMap: Ordena los elementos conforme se van insertando; provocando que las búsquedas sean más lentas que las demás clases.

**EJE:**

**// 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));**

**}**

****

**TREEMAP**

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.

**EJE:**

**// 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));**

**}**

****

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

**CLASES ABSTRACTAS**

No se implementa todos lo métodos y no se crea instancias. Una clase abstracta en Java es una clase que no puede ser instanciada directamente. Es decir, no puedes crear objetos directamente de una clase abstracta. Las clases abstractas se utilizan como una plantilla para otras clases que heredan de ellas. La principal función de una clase abstracta es proporcionar una estructura común a todas las clases que la extienden, y puede contener tanto métodos abstractos como métodos con implementación.

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.

**EJE:**

**// Clase abstracta Animal**

**abstract class Animal {**

**// Atributo común**

**String nombre;**

**// Constructor**

**public Animal(String nombre) {**

**this.nombre = nombre;**

**}**

**// Método normal con implementación**

**void dormir() {**

**System.out.println(nombre + " está durmiendo.");**

**}**

**// Método abstracto que debe ser implementado por las clases hijas**

**abstract void hacerSonido();**

**}**

**// Clase Perro que extiende de Animal**

**class Perro extends Animal {**

**public Perro(String nombre) {**

**super(nombre); // Llamada al constructor de la clase abstracta**

**}**

**@Override**

**void hacerSonido() {**

**System.out.println(nombre + " hace: ¡Guau!");**

**}**

**}**

**// Clase Gato que extiende de Animal**

**class Gato extends Animal {**

**public Gato(String nombre) {**

**super(nombre); // Llamada al constructor de la clase abstracta**

**}**

**@Override**

**void hacerSonido() {**

**System.out.println(nombre + " hace: ¡Miau!");**

**}**

**}**

**Usando las clases:**

**public class Main {**

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

**// No se puede crear un objeto directamente de Animal**

**// Animal animal = new Animal("Animal"); // Esto daría error**

**Perro perro = new Perro("Rex");**

**Gato gato = new Gato("Mimi");**

**perro.dormir(); // El perro usa el método dormir heredado**

**perro.hacerSonido(); // El perro implementa su propio sonido**

**gato.dormir(); // El gato también usa el método dormir heredado**

**gato.hacerSonido(); // El gato implementa su propio sonido**

**}**

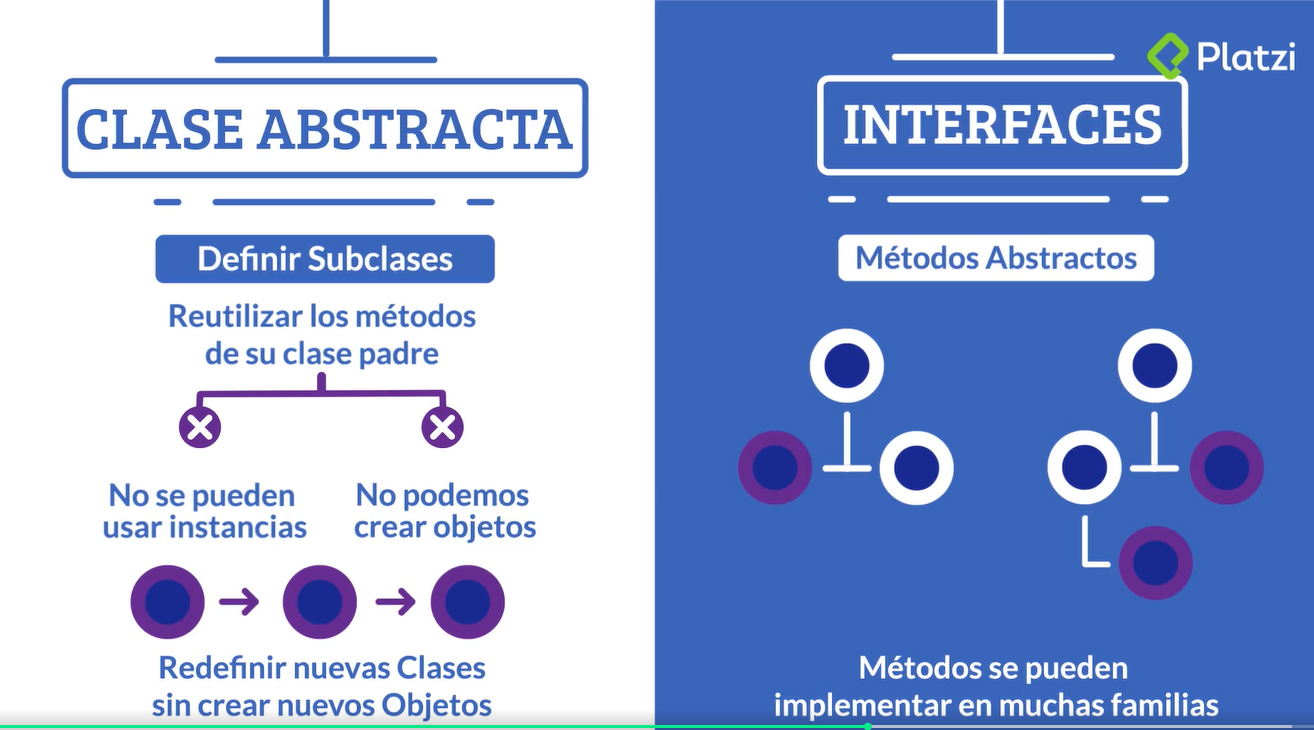
**}**

**—-------------------------------///////////////////////////////////////////—---------------------------------------------**

**CLASES ANONIMAS**

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. Las clases anónimas en Java son una forma especial de definir e implementar clases de manera rápida y sin necesidad de escribir una clase completa con nombre. Son clases internas que no tienen nombre y generalmente se utilizan para implementar interfaces o extender clases en situaciones donde no se necesita reutilizar el código de la clase.

**DIFERENCIAS ENTRE LAS CLASES ABSTRACTAS E INTERFACES**



**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:

**EJE:**

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

}

}



**ESTRUCTURAS DE ÁRBOL**

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.