



CERTIFIED TECH DEVELOPER

PROGRAMACIÓN ORIENTADA A OBJETOS

PRIMER AÑO
SEGUNDO BIMESTRE

Tabla de contenido

Módulo 1: Introducción a la programación orientada a objetos.....	8
Primer acercamiento a JAVA.....	8
¿Cómo funciona?.....	8
Primer Programa en JAVA	9
Método Main	9
Áreas de Trabajo.....	9
Tipos de datos.....	10
Estructuras de Control.....	11
El bucle For.....	11
String, Integer, Float.....	11
Variables.....	11
Clases.....	11
Paquetes.....	12
String	12
Integer.....	12
Float	13
Ingreso de datos: Scanner.....	13
¿Cómo lo creamos?.....	13
¿Qué métodos tiene?.....	14
Definición del Scanner.....	14
Funciones	15
Desarrollo de la función	16
Objetos y Clases	17
Concepto de Objetos	17
Encapsulamiento (público/privado)	17
Diagrama UML	18
Clases	18
Diagrama de Clases.....	18
Resumen	19
Implementación de clases en Java	19
Nombres en Java	19

Crear una Clase.....	20
Atributos, Constructores y Métodos	20
Atributos.....	20
Constructores.....	21
Métodos	21
Proteger el encapsulamiento	21
Los métodos get.....	22
Los métodos set.....	22
Instancia.....	22
Código.....	23
Variables y Métodos de clase	23
Clase	23
Objetos.....	23
Variables de Clase.....	23
Métodos de clase.....	23
Módulo 2: Programación orientada a objetos en Java.....	24
Relacionemos las clases de objetos	24
Tipos de relaciones	24
Composición.....	24
Diferencia entre la relación de agregación y la relación composición	25
Representación en UML de relaciones	25
Relación de asociación	25
Multiplicidad o cardinalidad	25
Relación de uso	26
Agregación.....	26
Composición	27
Implementar relaciones en Java	27
Relación de Asociación.....	27
Relación de Agregación.....	28
Relación de Composición.....	28
Herencia.....	29
Utilidad de la herencia	29
Herencia múltiple	30
Generalización y especialización	30

Herencia en Java (Clase 10)	30
Encapsulamiento y la Herencia.....	31
Firma de un método.....	32
Sobrecarga de Métodos.....	32
Sobrecarga en Java (Clase 10)	32
Sobreescritura de Métodos.....	33
Sobreescritura en JAVA (Clase 10).....	33
La clase Object.....	34
.toString().....	34
.hashCode().....	35
Equals.....	36
Ejemplo.....	36
instanceof	37
.getClass().....	37
Casting.....	38
Clase Abstracta	38
Buenas prácticas.....	39
Métodos abstractos.....	40
Métodos abstractos en Java	40
Sobrescribir métodos abstractos.....	41
Atributos y métodos en clases abstractas.....	41
Polimorfismo	42
Vinculación Dinámica (Dynamic Binding)	42
Polimorfismo.....	43
Casting.....	43
Clases abstractas vs concretas.....	44
Interface.....	44
Interfaces y la Herencia	45
Interfaces.....	45
Interfaces y la Herencia	46
Interface e Implements.....	47
Comparar objetos.....	48
Método compararCon.....	48
Interface Comparable.....	49

Implementación en Java.....	49
Colecciones.....	50
Interfaz Set.....	50
Interfaz List	50
Map.....	50
Tipos de colecciones	51
Recorrer Colecciones.....	53
for / while	53
Iterator	53
for each.....	54
Operaciones sobre colecciones	54
Crear una colección.....	54
Aregar elementos.....	55
Eliminar elementos.....	57
Obtener o buscar elementos	57
Programación Paramétrica.....	58
Sintaxis y uso.....	58
Colecciones paramétricas	59
Ejemplo.....	60
Solución con Generics	60
Arrays.....	61
Arrays vs. Colecciones	62
Igualdad y ordenamiento en las colecciones	62
Elementos iguales.....	62
Orden entre elementos.....	63
Relaciones 1 a muchos con colecciones	63
Introducción a excepciones.....	64
Cuando usar excepciones	64
Solución con excepciones.....	64
Excepciones en detalle	65
Diferentes excepciones	66
Diferenciando errores	66
El bloque finally	66
Excepciones.....	66

Proteger la integridad de una clase	67
RuntimeException.....	67
Lanzar una excepción.....	68
Crear nuestras propias Excepciones	68
Módulo 3: Patrones de diseño.....	69
Patrones.....	69
Patrones Creacionales.....	70
Patrones Estructurales	70
Patrones de Comportamiento.....	70
Concepto de Patrones de Diseño.....	70
Composición y Herencia.....	71
Singleton	72
Patrón factory.....	73
Factory Method.....	74
Abstract Factory Method.....	74
Patrón State.....	75
Motivación.....	76
Diagrama UML.....	77
Patrón State: ejemplo de un modelo de Auto	78
Patrón Composite.....	82
Características	82
Presentación en UML.....	82
Ejemplo.....	83
Introducción a Patrón Observer	84
Propósito	85
Ventajas y Desventajas.....	86
Diagrama UML.....	86
Conclusiones	87
Ejemplo de Patrón Observer.....	87
Patrón Strategy.....	90
Propósito	90
Diagrama UML.....	90

Módulo 1: Introducción a la programación orientada a objetos

Primer acercamiento a JAVA

Java es un **lenguaje** de programación **de propósito general** esto quiere decir que puede ser utilizado para varios propósitos: puede desarrollar todo un sistema de software para hacer un comercio electrónico como MercadoLibre, un sistema para reserva de canchas de tenis o algo relacionado a rentar películas y series online como Netflix. Con JAVA podemos acceder a bases de datos, comunicarnos entre dispositivos, capturar datos, hacer cálculos matemáticos y muchas cosas más. Todo esto hace que sea un lenguaje de propósito general.

También es un **lenguaje de alto nivel** porque es más parecido al lenguaje humano y más lejano al de las máquinas. Podríamos decir que el lenguaje que entienden las máquinas está compuesto por 0 y 1 pero para nosotros sería muy complejo comunicarle órdenes de esta manera, estos lenguajes cercanos al de las computadoras se llaman lenguajes de bajo nivel.

Algunas ventajas de los lenguajes de programación de alto nivel es que resultan en un código más fácil de leer, escribir y mantener por los humanos. En general permiten emplear menos líneas de código en comparación con los lenguajes de bajo nivel, también permiten escribir código ejecutable en distintos tipos de máquinas y sistemas operativos y, generalmente, emplean paradigmas de programación.

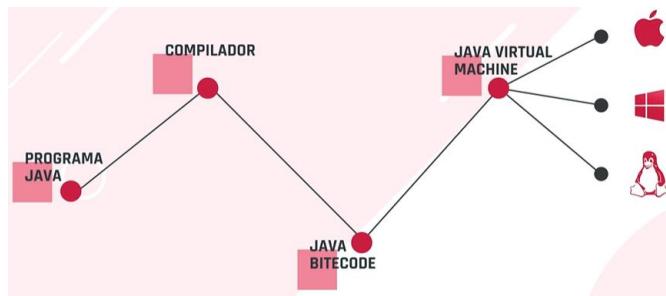
Otra de las características de Java es que es un lenguaje que **utiliza el paradigma de la programación orientada a objetos** esto significa que el código representa abstracciones en las cuales podríamos, por ejemplo, crear un carrito con unas líneas de código, un producto con otras y luego se puede agregar un producto a un carrito. Esto permite desarrollar sistemas complejos de manera rápida, adaptable y que son fáciles de entender.

Es un **lenguaje fuertemente tipado** esto implica que, si establezco que mis datos son de tipo numérico, nunca me permitiría almacenar una letra en ese lugar, ahorrándonos inconvenientes.

¿Cómo funciona?

Java es un lenguaje compilado e interpretado esto es muy importante porque, al trabajar de esta forma, nos brinda independencia de la plataforma. Esta independencia implica que nuestro programa, una vez compilado, va a poder ser ejecutado en cualquier sistema operativo siempre y cuando dispongamos del intérprete.

En primer lugar, tenemos el compilador, este va a traducir nuestro código de alto nivel a un código de más bajo nivel llamado JAVA BITECODE empaquetando nuestro programa listo para ser distribuido. Luego, al momento de querer ejecutarlo, el intérprete de JAVA llamado JAVA Virtual Machine (JVM) permitirá interpretar y ejecutar estas instrucciones en cualquier sistema operativo.



De esta manera, generalmente, los desarrolladores no deben pensar para cual plataforma o sistema operativo específico estas desarrollando el programa. Cualquier aplicación desarrollada en este lenguaje puede ser embebida en una web y ejecutada en un navegador al margen del sistema operativo. JAVA es muy utilizado en todo tipo de aplicaciones de productividad como procesadores de texto u hojas de cálculo. La gran mayoría de las aplicaciones nativas de Android están desarrolladas utilizando lenguaje JAVA.

Primer Programa en JAVA

Método Main

El método Main en JAVA es un estándar utilizado por la JVM o máquina virtual de Java para iniciar la ejecución de cualquier programa hecho con este lenguaje. Es el punto de entrada de la aplicación, esto quiere decir que cuando apretamos el botón de *play*, el código que se escribe en el mail se ejecuta línea por línea.

El método main tiene una firma definida que nos permite reconocerlo, esta es la siguiente: **Public static void main (String [] args) {}**.

Áreas de Trabajo

Cuando creamos nuestro primer proyecto, vamos a disponer de un archivo fuente con el siguiente aspecto:

```

Main.java
1 package com.company;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         // write your code here
7         System.out.println("Mi primer Programa");
8     }
9 }
10

```

Mi código

Prestemos atención al nombre del archivo, en este caso Main.java. A continuación, veamos la segunda línea, “public class Main”, ese es el nombre de la clase. Más adelante veremos qué es una clase en profundidad, pero, para comenzar a trabajar, debemos tener en cuenta que en **Java el nombre de la clase y el nombre del archivo deben coincidir** —incluso en mayúsculas y minúsculas—.

Vemos luego de la clase: **public static void Main (String args [])**, esta función indica el comienzo del programa. Todo lo que escriba dentro de esta función se ejecutará cuando ejecute el programa.

Por último, vemos una línea de código: `System.out.println("Mi primer ejemplo ");`; esta es la forma de mostrar algo por consola.

Tipos de datos

Si pensamos en un programa de computadora vamos a pensar en un conjunto de instrucciones que procesa datos de forma automática y siguiendo lo que nosotros decidimos.

Cuando programamos estos tipos de datos tienen nombres específicos, los datos pueden ser numéricos de tipo *Integer* o *Double*. Pueden ser de tipo texto o sea *String* o pueden ser verdaderos o falsos que son los de tipo *boolean*. También existen tipos de datos que son colecciones o sea conjuntos de datos como los *arrays* o los diccionarios. El primer problema que nos encontramos al programar es cómo hacemos para manejar esos datos y las respuestas son las variables.

Las **variables** son espacios de memoria dentro de nuestro programa donde vamos a almacenar valores, es decir, esos mismos datos. Podemos pensar a las variables como una caja que tiene una **etiqueta** que va a ser su **nombre**. Estas variables van a contener **valores**, es decir, van a ser contenedores de un valor dependiendo del tipo que le asignemos y nos van a permitir hacer operaciones como: sumar números, concatenar textos, imprimir por pantalla la información, entre otros. Son el componente más fundamental a la hora de programar

En JAVA cuando creamos esa caja tenemos que decidir qué tipo de datos vamos a querer guardar. Por ejemplo, si quiero guardar un dato en la variable nombre lo que debería escribir es: `String nombre = "Romi";`

Algunos aspectos importantes:

- En primer lugar, la definición de variables se empieza con el tipo de dato que quiero guardar en este caso, String.
- En segundo lugar, va el nombre de la variable, es importante usar nombres declarativos al momento de etiquetar la variable, o sea, definir un nombre que represente la información que vamos a querer almacenar.
- En tercer lugar, tenemos el operador igual simple qué es un operador de asignación y luego de este operador pondremos el valor que queremos guardar



Por último, pero no menos importante recordar siempre que las líneas en JAVA terminan con ";", sin esto no va a funcionar.

Estructuras de Control¹

El bucle For

Este tiene una estructura muy definida. En principio, dentro de las llaves vamos a escribir todo el código que queremos que se repita. Luego, dentro de los paréntesis, podemos distinguir tres campos divididos entre sí por puntos y comas, estos van a determinar cuántas veces se va a repetir nuestro bucle.

El primer término va a ser el inicio, ahí se va a inicializar una variable que va a ir llevando la cuenta de cuántas veces se repitió el ciclo. En el medio vamos a agregar la condición de permanencia, mientras esta condición se cumpla, es decir, nos dé true, el código entre las llaves se va a seguir ejecutando; cuando la condición sea falsa el ciclo termina y se deja de repetir. Finalmente, en el tercer lugar tenemos lo que se conoce como modificador o paso donde vamos a elegir como contar los ciclos: de uno en uno, de dos en dos, en forma decreciente o de forma decreciente.

```
for(Integer i = 0; i < valorMaximo; i++){
    //código que se ejecuta cada vez
}
```

String, Integer, Float

Variables

En Java encontramos como herramienta para el desarrollo los tipos primitivos, llamamos así a los tipos de datos que solo nos permiten almacenar un valor. Por ejemplo, int, float, double y char. Cuando definimos una variable con estos tipos primitivos, solo podemos almacenar valores.

Clases

En este caso tendremos un elemento que, además de almacenar un valor, nos permite realizar ciertas operaciones que ya vienen programadas, a estas operaciones las llamamos **métodos**.

Por ejemplo, String es una clase, por eso, se la inicializa en mayúscula. Todas las clases las **nombramos con la inicial en mayúscula**, si definimos:

```
String nombre;
```

Estas son funciones que ya vienen resueltas y solo podemos utilizarlas con la clase a la cual le pertenece, es decir, cada clase en Java tiene sus propios métodos.

Para comenzar a conocer cómo funcionan las clases propias del lenguaje, vamos a nombrar 3 clases que nos resultan útiles: String, Integer, Float, notemos que todas comienzan con la inicial en mayúscula.

¹ Ver archivo “Java cheatsheet estructuras”

Las clases Integer y Float son equivalentes a los tipos de datos primitivos, es decir, me permiten almacenar valores de los tipos indicados, pero además me dan ciertas funcionalidades. Se suele decir que envuelven los tipos primitivos.

Algo a tener en cuenta cuando usamos estas clases es que no podemos usar operadores como “==”, para efectuar una comparación por igual usamos `.equals()`. El equals se utiliza para comparar por igual, siempre que estemos trabajando con clases. Si queremos comparar si un valor es mayor o menor que otro debemos usar `.compareTo()`.

Otra cosa a destacar es que una String a la cual no le asignamos nada tiene el `valornull`. Esto sucede con todas las clases, si definimos un elemento (objeto) de una clase inicialmente tendrá el valor null.

Paquetes

Para organizar las clases, existen los paquetes, estos son contenedores donde se pueden agrupar las clases. Más adelante los utilizaremos para nuestras clases, pero por ahora debemos saber que también las clases de Java se encuentran agrupadas en paquetes, o como su nombre en inglés: `package`.

String

Para utilizar datos de tipo texto, vamos a declararlos como String. Las Strings nos permiten utilizar funciones ya programadas, que le pertenecen. Las llamamos métodos.

```
public static void main(String[] args){  
    {}  
    String nombre;  
    {}
```

Si aún no hemos asignado nada a las String, entonces, contiene un valor null, en ese caso no se pueden usar los métodos.

```
String nombre;  
  
if (nombre==null)  
{  
    System.out.println("Cadena con valor nulo");  
}
```

Integer

Integer como clase y no como tipo primitivo se utiliza de una forma distinta. Para comenzar a utilizar un Integer tenemos dos posibilidades:

```
Integer valor=0;
```

En este caso definimos y creamos un Integer, dándole un valor inicial 0.

```
Integer num= new Integer (1);
```

En la segunda forma hacemos algo similar, pero la parte de la izquierda es la definición y la parte de la derecha la creación con un valor inicial 1.

Cuando solo definimos algo de tipo Integer, su valor inicial es null, es necesario darle un valor inicial.

Ejemplo

Comprobamos la relación entre dos números enteros, utilizando clases. Métodos usados: .equal(), .compareTo()

```
Integer valor1=10;
Integer valor2=30;
int comparar;

if (valor1.equals(valor2))
    System.out.println("Son iguales");
else
{   comparar=valor1.compareTo(valor2);
    if (comparar>0)
        System.out.println("valor1 es mayor que valor2");
    else
        System.out.println("valor2 es mayor que valor1");
}
```

Float

Float como clase y no como tipo primitivo se utiliza de una forma distinta. Para comenzar a utilizar un Float tenemos dos posibilidades:

```
Float coeficiente=2.5f;
```

En este caso definimos y creamos un Integer, dándole un valor inicial 2.5f, la f quiere decir float, si no lo ponemos se asume que es algo de tipo Double.

```
Float num= new Float (0.5);
```

En la segunda forma hacemos algo similar, pero la parte de la izquierda es la definición y la parte de la derecha la creación con un valor inicial 0.5.

Al igual que Integer, si no tiene un valor inicial, está en null.

Cuando solo definimos algo de tipo Float, su valor inicial es null, siempre es necesario darle un valor inicial.

Ingreso de datos: Scanner

Scanner es una clase propia de Java, que nos permite ingresar valores. Tiene métodos, funciones ya programadas, que nos permiten ingresar distintos tipos de datos.

¿Cómo lo creamos?

Paso 1: Cuando definimos nuestro elemento de tipo Scanner, nos aparece esta indicación. Esto significa que para poder utilizarlo debemos agregar la clase correspondiente, que se encuentra en java.util

```
public class Main {  
  
    public static void main(String[] args) {  
        // write your code here  
        Scanner|  
    } 
```

Paso 2: Cuando aceptamos la sugerencia que se vio en la pantalla anterior, nos agrega el import, finalizamos la definición dándole un nombre como lo haríamos con cualquier variable.

```
import java.util.Scanner;  
  
public class Main {  
  
    public static void main(String[] args)  
        //Write your code here  
        Scanner lector;  
}
```

Paso 3: Luego de definirlo, es necesario crear el objeto u instanciarlo. Lo creamos asociado a System.in, es decir, todo ingreso de datos será interceptado por el Scanner.

```
Scanner lector;   
lector=new Scanner(System.in); 
```

¿Qué métodos tiene?

- **nextByte()** para leer un dato de tipo byte.
- **nextShort()** para leer un dato de tipo short.
- **nextInt()** para leer un dato de tipo int.
- **nextLong()** para leer un dato de tipo long.
- **nextFloat()** para leer un dato de tipo float.
- **nextDouble()** para leer un dato de tipo double.
- **nextBoolean()** para leer un dato de tipo booleano.
- **nextLine()** para leer un string hasta encontrar un salto de línea.
- **next()** para leer un string hasta el primer delimitador, generalmente hasta un espacio en blanco o hasta un salto de línea.

Definición del Scanner

Definimos el Scanner, para luego ingresar los valores en las variables definidas.

```
Scanner scanner;
scanner = new Scanner(System.in);
int num1;
int num2;
float coeficiente;
String nombre;
char inicial;
```

Valores numéricos

Ingresamos los datos numéricos. Para lograr una interacción más amigable con el usuario, indicar qué se espera.

```
int num1;
int num2;
float coeficiente;

{} System.out.println("Ingrese primer valor");
num1= scanner.nextInt();
System.out.println("Ingrese segundo valor");
num2= scanner.nextInt();

System.out.println("Ingrese el coeficiente");
coeficiente= scanner.nextFloat();
```

Ingresar Texto

Ingresamos un texto, en este caso un nombre y luego se obtiene la inicial. No hay un método para ingreso de caracteres.

```
String nombre;
char inicial;
{} System.out.println("Ingrese su nombre");
nombre= scanner.nextLine();
inicial= nombre.charAt(0);
```

Funciones

Las funciones en Java son similares a las vistas en JavaScript, pero hay algunas cosas a tener en cuenta por ser un lenguaje tipado, vamos a tener que definir más cosas.

Para definirla vamos a considerar 3 cosas:

- Qué devuelve la función
- Qué nombre tiene
- Los parámetros que necesitamos

Cuando decimos qué devuelve nos referimos al tipo de dato que devuelve la función. Entonces la definición sería en forma general algo de este estilo.

```
Tipo devuelto nombre (parámetros)
```

Nombre: El nombre debe ser lo más descriptivo posible, no importa si necesitamos unir dos o más palabras.

Parámetros: No hay muchas diferencias en cuanto a los parámetros, solo que es necesario indicar el tipo de cada uno, entonces para que una función reciba valores lo indicaremos de la siguiente manera.

```
(int num1, int num2)
```

```
(double importe, String descripcion)
```

```
(int cantidad, int posicion, String nombre)
```

Tipo devuelto: Esta es la mayor diferencia con la forma en la que aprendimos en JavaScript.

Las funciones pueden devolver un valor de retorno de algún tipo determinado, por ejemplo, int, double, Integer, String, etc. En realidad, pueden devolver cualquier cosa no solo valores, también estructuras enteras. Pero hay que indicar que tipo tiene lo que devolvemos.

```
int suma(int num1, int num2)
```

```
double calcularTotal(double importe, int cantidad)
```

Pero hay otro tipo de funciones, las que **no devuelven nada** en ese caso en donde indicamos el tipo devuelto colocaremos la palabra reservada void.

```
void mostrarMensaje(String mensaje)
```

Usamos las funciones de tipo void, cuando queremos que nuestra función sólo realice una serie de pasos o acciones y no nos devuelva nada.

Desarrollo de la función

Hasta ahora vimos cómo definir una función, ahora veamos que varía en la implementación, vamos a tener dos situaciones. Que la función tenga valor de retorno o que no devuelva nada.

En el primer caso, debemos incluir un **return** con el valor devuelto, el tipo de este valor tiene que coincidir con el tipo de dato indicado como tipo devuelto.

```
int suma(int num1, num2)
{
    return num1,num2;
}
```

¡El valor returned tiene que ser del tipo indicado!

En caso de tener una función no tenga tipo de retorno, nos quedaría así:

```

void mostrarMensaje(String mensaje)
{
    System.out.println(mensaje);
}

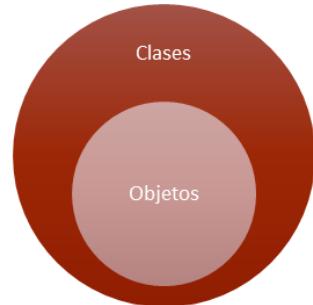
```

Clase 4: Objetos y UML

Objetos y Clases

Una de las cuestiones fundamentales cuando hablamos de programación orientada a objetos, aun antes de empezar a programar nuestro sistema, es modelar los efectos de la vida real que influyen nuestro contexto. Cada uno de los elementos de la vida real que vamos a programar van a traducirse en nuestro sistema en una **clase (abstracción)**.

Las clases son abstracciones, ideas que representan ese elemento de la vida real que modelo en el sistema. Está claro que nuestro programa no solo funciona con ideas, van a existir ciertos elementos individuales que representan a cada uno de los elementos del sistema, estos serán los **objetos o instancias** concretas del molde que representan elementos concretos de nuestro sistema.



Concepto de Objetos

Un objeto es algo que tiene atributos y responsabilidades. Los **atributos** de un objeto son las características y propiedades distintivas que permiten darle significado. Por ejemplo:

- Clase: Veterinaria
- Objeto: Veterinario
- Atributo: nombre, apellido, matrícula

Mientras que las **responsabilidades o comportamientos** son la manera en que actúa o reacciona un objeto (es decir, es lo que representa la actividad visible y comprobable exteriormente), en la programación orientada a objetos vamos a llamar al comportamiento de los objetos: "métodos", los cuales nos van a permitir establecer cómo van a responder los objetos cuando interactuemos con ellos.

¡Atención! Los atributos y comportamientos van a depender del contexto del objeto.

Cada método especifica la operación o comportamiento que a su vez puede acceder a la estructura interna del objeto, como así también interactuar con otros objetos. Por lo general, los encontramos como verbos indicando las acciones que puede realizar el objeto.

Encapsulamiento (público/privado)

El encapsulamiento es una de las propiedades más importante de la programación orientada a objetos.

Cuando hablamos de encapsulamiento, no debemos olvidarnos del origen de la palabra "colocar en cápsulas". ¿Qué serían las cápsulas? Es un envoltorio que protege el contenido en su interior. En la POO, buscamos impedir que cualquier otro objeto pueda tener acceso a la estructura interna de un

objeto. Solamente yo puedo cambiar o mostrar mi estado y con los métodos específicos que van a indicar cómo pedir cambios en dichos atributos desde el exterior del objeto.

De ahora en más, cuando diseñamos nuestros objetos, tenemos que tener en cuenta el encapsulamiento. Por ejemplo, si tuviéramos un objeto Persona, que tiene como atributo su clave de acceso bancario, no sería conveniente que todos los objetos puedan acceder libremente a dicho atributo. El objeto Persona debería establecer un método controlado y seguro para devolver la clave de acceso bancario, por ejemplo, si es que se cumplen medidas de seguridad.

¡Importante!

- Cuando definamos un objeto, dejar sus atributos privados.
- Los métodos que sean públicos serán vistos por los otros objetos.
- Usar siempre métodos públicos para ver o modificar las características de tus objetos.
- Para cambiar el valor de un atributo se usa un método **set**, por ejemplo, para cambiar el nombre será setNombre(String)
- Para obtener el valor de un atributo se usa un método **get**, por ejemplo, para saber el nombre será getNombre():String

Los métodos para ver o cambiar atributos se los denomina getters y setters respectivamente.

Diagrama UML

UML son las siglas para Unified Modeling Language, que en castellano significan: Lenguaje de modelado unificado. Es un lenguaje de modelado, de propósito general, usado para la visualización, especificación, construcción y documentación de sistemas orientados a objetos.

U (Unificado): Unifica varias técnicas de modelado en una única.

M (Modelo): Mediante su sintaxis se modelan distintos aspectos del mundo que permiten una mejor interpretación.

L (Lenguaje): Al ser un lenguaje, UML cuenta con una sintaxis y una semántica, es decir, existen reglas sobre cómo deben agruparse los elementos del lenguaje y el significado de esta agrupación.

Clases

Diagrama de Clases

El lenguaje de modelado unificado (UML) contiene distintos diagramas de estructura, comportamiento e interacción. En este caso, vamos a ver un diagrama de estructura conocido como **diagrama de clases**, que muestra una vista estática de la estructura del sistema, o de una parte de este, describiendo qué atributos y comportamiento debe desarrollar con los métodos necesarios para llevar a cabo las operaciones del sistema.

El objetivo del diagrama de clases es mostrar qué clases (tipos de objetos) forman el sistema y las relaciones entre ellas.

Resumen

En programación, vamos a entender a cada clase como un nuevo tipo de dato y cada uno define, además de atributos, sus responsabilidades (¿Qué puedo hacer con ese objeto?).

Cuando estemos definiendo clases vamos a agregar las responsabilidades, que, al estar programando, vamos a llamar métodos. Como estamos haciendo este proceso desde cero tenemos que aclarar en nuestro diagrama todos los métodos, es decir, todas las responsabilidades que van a poder ejecutar los objetos de nuestra clase.

Luego de los atributos, en la parte inferior del rectángulo, se listan todos los métodos que van a hacer propios de la clase. De este modo, vamos a ir agregando los métodos a las clases de nuestro sistema. Anotaremos primero el nombre de la responsabilidad luego, entre paréntesis, agregaremos los parámetros en caso de ser necesario. También podemos definir si el método va a devolver un resultado señalándolo con ":" y agregando el tipo de dato que va a devolver esa responsabilidad al ejecutarse.



Al igual que con los atributos, podríamos agregar un "+" o un "-" del lado izquierdo aclarando si el método es público o privado.

¿Por qué ponemos los atributos por diseño como privados? Esto es lo que se llama **encapsulamiento**. En este importa el **qué** y nos permite armar sistemas complejos pero fáciles de usar. Esto significa que cada una de las responsabilidades de nuestra clase, hasta las responsabilidades más simples de todas, se van modelar como métodos así ocultamos las propiedades y el comportamiento interno de los objetos.

Clase 5: Clases

Implementación de clases en Java

Nombres en Java

Camel case es un estilo de escritura que se aplica a frases o palabras compuestas. El nombre se debe a que las mayúsculas a lo largo de una palabra en CamelCase se asemejan a las jorobas de un camello.

Atributos: Los nombres de los atributos comienzan con minúscula, si necesitamos usar más de una palabra, a partir de la segunda inicializamos en mayúscula (elAtributo)

Métodos: Se nombran de la misma forma que los atributos, la primera palabra en minúscula y si el nombre tuviera más palabras, todas se inicializan en mayúscula. Recomendamos poner nombres lo más descriptivos posibles, aunque esto implique usar varias palabras (calculoSueldoNeto)

Objetos: La primera palabra en minúscula y si tiene más de una palabra, las siguientes se inicializan en mayúscula (nombre, importeTotal)

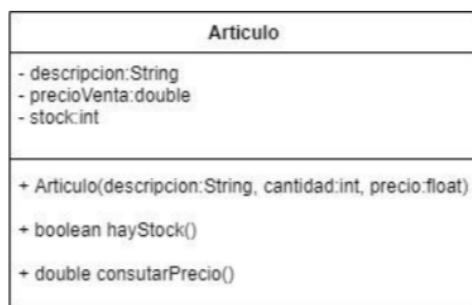
Clase: Los nombres de las clases siempre van con la inicial en mayúscula, si necesitamos usar dos o más palabras para nombrar una clase van pegadas y con todas las iniciales en mayúscula (CamelCase, Empleado)

Paquetes: Todas las letras en minúscula

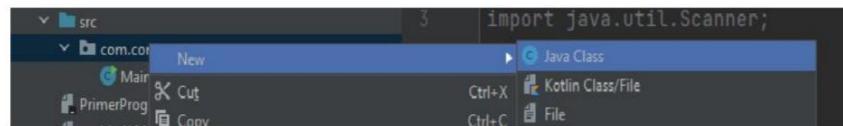
Constantes: Todas las letras en mayúscula y si hay más de una palabra, separadas por guión (IVA, DIAS_SEMANA)

Crear una Clase

Luego de realizar el diseño de la clase en un diagrama, el próximo paso es implementarla:



Para crear una clase en Java, hacemos clic con el botón derecho del mouse, obtenemos un menú contextual, seleccionamos New, luego Java Class y le asignamos un nombre (siempre comenzando con mayúsculas).



Al crear la clase estará vacía. Debemos incorporar los atributos y métodos. Definiéndolos con el alcance correcto para mantener el encapsulamiento, esto sería los atributos como *private* y los métodos como *public*.

No cambiar nunca el nombre de la clase. El nombre del archivo .java debe coincidir con el nombre de la clase.

Atributos, Constructores y Métodos

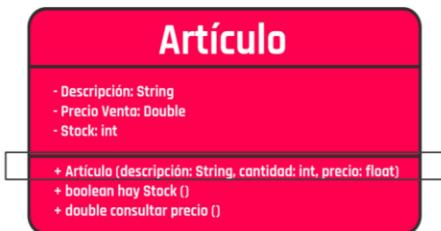
Atributos

Definimos los atributos como privados para preservar el encapsulamiento.

```
public class Articulo{  
  
    private String descripcion;  
    private double precioVenta;  
    private int stock;  
}
```

Constructores

El constructor en el diagrama:



El constructor es un método que no tiene tipo de dato, se llama igual que la clase. Recibe como parámetros los valores que se desea asignar inicialmente a los atributos, es decir, los valores iniciales. Se puede usar para inicializar los atributos.

```
public class Artículo{  
    private String descripcion;  
    private double precioVenta;  
    private int stock;  
  
    public Artículo(String descripción, int cantidad,double precio){  
        this.descripcion=descripción;  
        precioVenta=precio;  
        stock=cantidad;  
    }  
}
```

Métodos

```
public class Artículo{  
    private String descripción;  
    private double precioVenta;  
    private int stock;  
  
    +   public Artículo(String descripción, int cantidad,double precio)  
  
    public boolean hayStock(){  
        return stock>0;  
    }  
    public double consultarPrecio(){  
        return precioVenta;  
    }  
}
```

Proteger el encapsulamiento

Los atributos de una clase deben ser privados, para garantizar el ocultamiento, sin embargo, en algún momento podemos necesitar consultar o cambiar el valor de un atributo.

Los métodos de acceso son métodos públicos que nos permiten acceder al valor de los atributos privados del objeto. Los métodos modificadores nos permiten cambiar el valor de un atributo y los métodos consultores u observadores nos devuelven el valor guardado en un atributo.

Para nombrarlos usamos dos prefijos: **get** y **set**. El primero de estos es para los consultores, por ejemplo, `getNombre`, `getValor`, `getSueldo`, etc., mientras que el segundo, para los métodos modificadores: `setNombre`, `setValor`, `setSueldo`, etcétera. Debido a estos prefijos se los suele llamar métodos getters y setters.

```

public class Articulo{
    private String descripcion;
    private double precioVenta;
    private int stock;

    public String getDescripcion(){
        return descripcion;
    }
    public double getPrecioVenta(){
        return precioVenta;
    }
    public int getStock(){
        return stock;
    }
    public void setDescripcion(String descripcion){
        this.descripcion= descripcion;
    }
    public void setPrecioVenta(double precio){
        precioVenta=precio;
    }
    public void setStock(int stock){
        this.stock=stock;
    }
}

```

Los métodos get

Estos métodos siempre devuelven algo del mismo tipo que el atributo al que acceden, no tienen parámetros porque solo acceden al valor guardado en el atributo, sin cambiarlo.

```

public String getDescripcion(){
    return descripcion;
}
public double getPrecioVenta(){
    return precioVenta;
}
public int getStock(){
    return stock;
}

```

Los métodos get permiten acceder al valor de un atributo para una consulta o para usar ese valor en otra operación.

Los métodos set

Los métodos son de tipo void y tienen un parámetro del mismo tipo que el atributo al que acceden, el valor que recibe en este parámetro es el que se asigna al atributo al que acceden.

```

public void setDescripcion(String descripcion){
    this.descripcion= descripcion;
}
public void setPrecioVenta(double precio){
    precioVenta=precio;
}
public void setStock(int stock){
    this.stock=stock;
}

```

Los métodos set permiten cambiar el valor de un atributo, reciben por parámetro el nuevo valor y lo asignan al atributo correspondiente.

Instancia

Una instancia es un elemento tangible (ocupa memoria durante la ejecución del programa) generado a partir de una definición de clase. Todos los objetos empleados en un programa han de pertenecer a una clase determinada.

Aunque el término a veces se emplea de una forma imprecisa, un objeto es una instancia de una clase predefinida en Java o declarada por el usuario y referenciada por una variable que almacena su dirección de memoria. Cuando se dice que Java no tiene punteros simplemente se indica que Java no tiene punteros que el programador pueda ver, ya que todas las referencias a objeto son de hecho punteros en la representación interna.

En general, el acceso a los atributos se realiza a través del operador punto, que separa al identificador de la referencia del identificador del atributo (`idReferencia.idAtributo`). Las llamadas a los métodos para realizar las distintas acciones se llevan a cabo separando los identificadores de la

referencia y del método correspondiente con el operador punto
(idReferencia.idMetodo(parámetros)).

Código

```
public class Main{  
  
    public static void main(String[] args){  
  
        Articulo articulo=new Articulo("Artículo 1",100,1100.  
  
        if(articulo.hayStock())  
        {  
            System.out.println("Hay stock disponible");  
        }  
        System.out.println("El precio de venta es:"+articulo.consultarPrecio());  
  
    }  
}
```

Creamos un objeto o instancia de la clase Articulo.

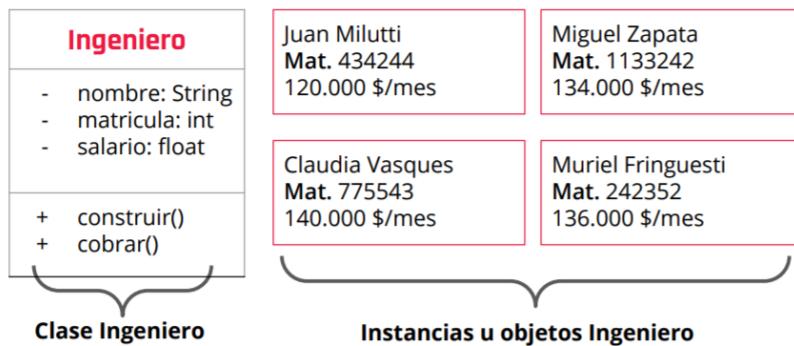
Variables y Métodos de clase

Clase

Vimos que el enfoque de la programación orientada a objetos se basa en identificar objetos con sus atributos y responsabilidades. Entonces, encontramos que hay grupos de objetos que, aunque tienen diferentes estados (valores de los atributos), tienen en común cuáles son los atributos y cuáles son sus responsabilidades. Entonces este "molde" es lo que llamamos clases.

Objetos

Todos los objetos de una clase tienen la misma estructura: los mismos atributos y el mismo comportamiento, es decir, pueden hacer lo mismo. Pero cada objeto tiene sus propios atributos, puede tener distintos valores en sus atributos, tiene un estado propio.



Variables de Clase

Vamos a llamar variables de clase a aquellas variables (atributos) que guardan valores comunes a todos los objetos.

En nuestro diagrama vamos a subrayar el nombre de la variable para identificar que es una variable de clase.

Métodos de clase

Un método de clase se puede utilizar, sin necesidad de instanciar o crear un objeto, directamente con la clase. Para indicar que es un método de clase también debemos subrayarlo.

Módulo 2: Programación orientada a objetos en Java

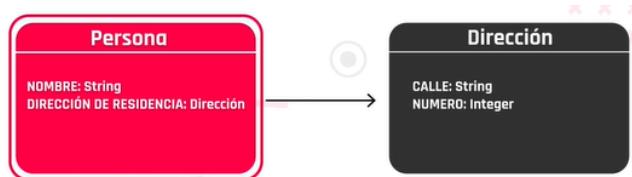
Relacionemos las clases de objetos

Las relaciones existentes entre clases nos indican como se comunican los objetos de esas clases entre sí y de qué manera sus mensajes se dirigen según las relaciones establecidas por eso, existen distintos tipos de relaciones.

Tipos de relaciones

Asociación

Se trata de una relación estructural que describe una conexión entre clases. Una relación de asociación es unilateral, va en un solo sentido. Esta puede ser identificada rápidamente con el uso de la palabra “tiene”, “conoce” o alguna otra que refiere a una conexión.

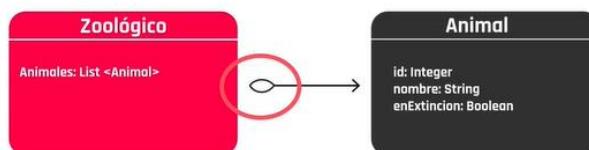


Cómo podemos ver la relación se establece nuestro diagrama UML una flecha desde la persona hacia la dirección luego, en los atributos de persona, agregamos el atributo dirección de residencia que va a ser del tipo dirección.

Agregación

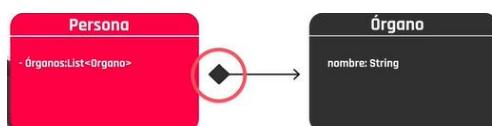
Cuando una clase no solo tiene o conoce a otra clase, sino que además **es parte** estamos frente una relación de agregación.

¿Cómo nos damos cuenta que estamos frente a una relación de agregación? Esto se indica con un rombo vacío partiendo de la clase que contiene a la otra.



Composición

Se define cuando una clase está compuesta por otra clase. A diferencia de la reacción de agregación podemos notar cuando se trata una relación de composición porque se indica con un rombo relleno.



Diferencia entre la relación de agregación y la relación composición

En la relación de composición una clase está compuesta por otra por lo tanto una de las clases no tiene sentido por su cuenta y va a depender de la otra. Mientras que una relación de agregación ambas clases pueden seguir existiendo independientemente.

Representación en UML de relaciones

Las relaciones es la forma en la que los objetos interactúan entre sí.

Relación de asociación

La relación de asociación se dibuja con una flecha que une a ambas clases, esta flecha está abierta y eso es muy importante. Otro aspecto relevante es de dónde a dónde va la flecha, esta tiene que ir desde el objeto que tiene al otro objeto.



Cuando una asociación lleva una flecha indica una dirección de recorrido (de navegación). Implica que es posible para un objeto en un extremo acceder al objeto del otro extremo porque el primero contiene referencias específicas a este último (al que apunta la flecha), no siendo cierto en el sentido contrario.

La forma en que entendemos nuestro contexto nos puede engañar haciendo que modelar el sistema dependa de las decisiones que tomamos, esto hace que el modelado no sea sencillo ya que no hay una sola respuesta válida.

¿En qué impacta la relación de asociación? Dependiendo de cómo este establecida la relación, se agrega un atributo nuevo del lado del objeto que contiene al otro. Este atributo va a ser de tipo "objeto B".

Multiplicidad o cardinalidad

Cuando la clase A no tiene solo una instancia de la clase B, en las flechas que los une se agrega el detalle de la cardinalidad. Por ahora este atributo lo indicaremos con la palabra list. (Ejemplo – mascota: list <mascota>).

La multiplicidad también llamada cardinalidad especifica el número de instancias de una clase que puede estar relacionadas con una única instancia de una clase asociada. La multiplicidad limita el número de objetos relacionados.

Para establecer las multiplicidades, primero, nos paramos en una de las clases, por ejemplo, la clase Persona y paso siguiente debemos hacernos la siguiente pregunta: ¿Para una instancia de esa clase, en este caso de la clase Persona, cuántas posibles instancias podría tener de la clase a la que está asociada, en este caso Trabajo?

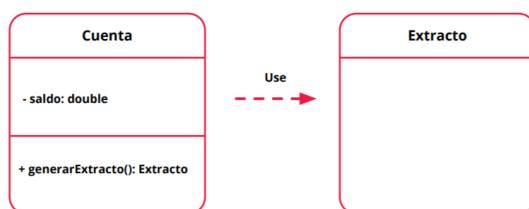
Luego nos paramos en la otra clase, en nuestro caso la clase Trabajo y nos debemos hacer la misma pregunta: ¿Para una instancia de esa clase, o sea, para un trabajo, cuántas posibles personas podrían tener ese trabajo?



Relación de uso

Una relación de uso es un tipo de asociación que como lo indica su nombre es una relación del tipo “usa un”. La particularidad frente al otro tipo de asociación “tiene un” es que no hay una referencia de una clase a la otra, sino que, en este caso, la relación se da porque hay algún método que devuelve o recibe como parámetro una variable que es del tipo de la otra clase.

En el ejemplo a continuación la clase Cuenta tiene un método que devuelve un Extracto, pero no necesita tener una instancia Extracto dentro de la Cuenta

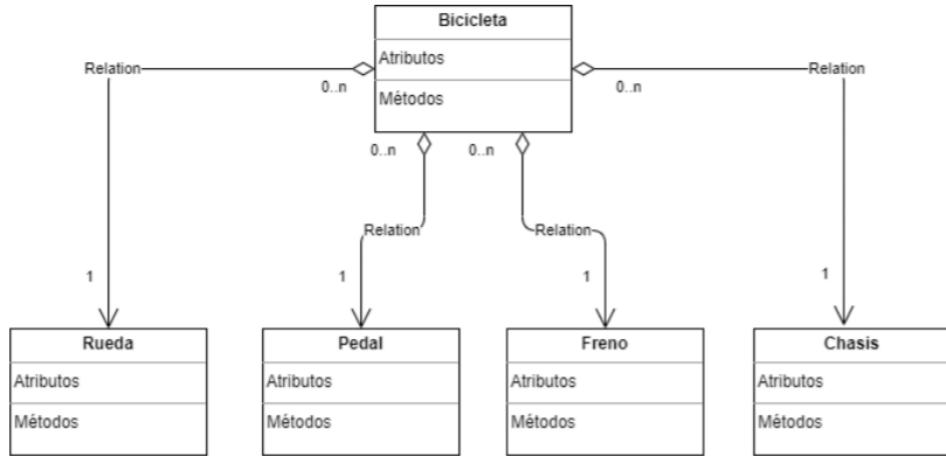


Una forma fácil de reconocer esta relación es cuando una clase en vez de ser atributo de otra aparece como **parámetro** en uno de sus **métodos**.

De asociación	De uso
<ul style="list-style-type: none"> •Colaboración continua •Flecha completa •Atributo 	<ul style="list-style-type: none"> •Colaboración temporal •Flecha punteada •Parámetro

Agregación

Un caso muy común de relaciones entre clases es la llamada agregación, donde existe una relación entre los agregados y el todo, pero los componentes pueden existir, aunque el todo fuese destruido. Dicho en otras palabras, es una relación que indica que una clase forma parte de otra/s clase/s con una relación débil, de tal forma que existe una independencia respecto a su existencia. Decimos también que una agregación es una relación de tipo “es parte de”.



Composición

La composición es un tipo de agregación que es más fuerte, donde todas las partes (clases) solamente pueden pertenecer a un todo y lo representamos con un rombo relleno en lugar de vacío como en la agregación. Es el caso en el que una clase de objeto A “es dueño de” una clase de objeto B, y B no tiene razón de existir sin A. Como mencionamos anteriormente, a diferencia de la agregación, en este caso, la parte no tiene sentido sin el todo.

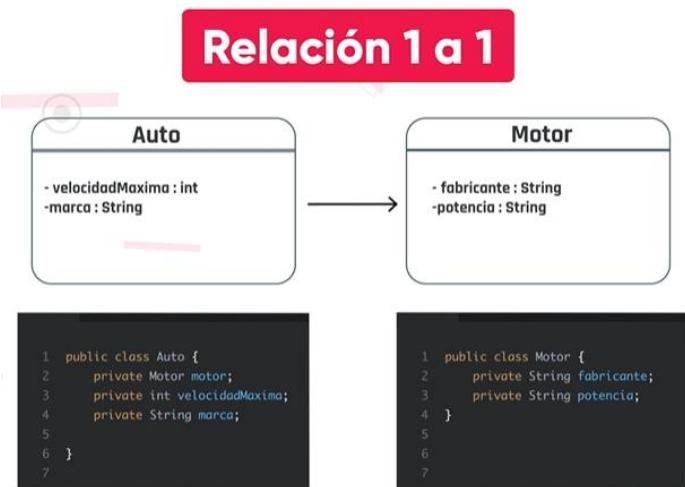


Implementar relaciones en Java

Las relaciones existentes entre clases nos indican como se comunican los objetos de esas clases entre sí, y la manera en que sus mensajes se dirigen según las relaciones establecidas. Es por esa razón que existen diferentes tipos de relaciones.

Relación de Asociación

En este caso tenemos la relación de asociación donde un auto tiene un motor, cuando lo ejecutamos en JAVA las relaciones se implementan como atributos. En UML tratamos de no ponerlo como atributo porque se entiende a través de la relación y sería información redundante.



En el caso de que exista una relación de uno a muchos, por ejemplo, la clase auto y la clase rueda, se utiliza un array donde guardamos muchos objetos del mismo tipo.



Relación de Agregación

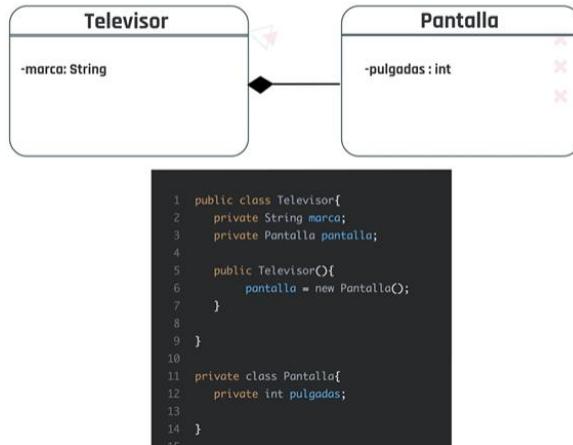
Hay tipos especiales de asociaciones como la de agregación, donde un objeto usa a otro para poder funcionar, en JAVA lo ejecutamos de la misma manera que antes.



Relación de Composición

Por último, tenemos la relación de composición donde una parte no tiene sentido sin el todo. Podemos utilizar el constructor para obligar a que cada vez que se crea un objeto del tipo A si o si venga con un objeto del tipo B.

Relación de composición



La manera en la que se implementan los diferentes tipos de relaciones son iguales, lo que varía en su implementación es la **cardinalidad** con lo cual, no debemos preocuparnos si no logramos distinguir en una instancia si una relación es de agregación o de composición, pero sí tenemos que prestar atención a la cardinalidad para poder implementarlo en JAVA ya sea con una propiedad o con un array.

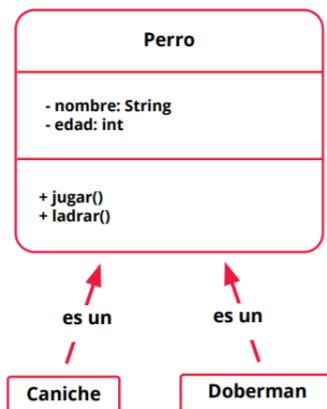
Clase 8: Herencia en UML

Herencia

La herencia es uno de los pilares del paradigma orientado a objetos, también conocida como una relación del tipo “es un”.

Podemos decir que un caniche es un Perro de la misma manera que podríamos decir que Profesores es un Empleado, y más aún: un Empleado es una Persona, por lo tanto, un Profesor es una Persona. Nuevamente, al observar la realidad y pasar por el proceso de abstracción, obtuvimos una serie de entidades que se ordenan naturalmente, y la herencia responde a ello.

Podemos decir entonces que la herencia es un ordenamiento entre clases que define una relación “es un”.



Utilidad de la herencia

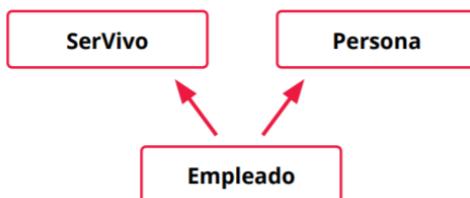
Esta es una pregunta interesante, ya que la herencia es uno de los pilares de la orientación a objetos. Si analizamos el esquema anterior, tanto Caniche como Doberman hacen lo mismo que hace el perro.

¿Si hacen lo mismo que el perro, para qué escribir el código de lo que hacen? ¿No sería más conveniente escribirlo una sola vez en la clase Perro y que Doberman o Caniche, “obtengan” este comportamiento desde Perro?

De hacer esto, decimos que Caniche y Dóberman “heredan” el comportamiento de un perro, es decir, la clase Dóberman hereda de la clase Perro, todos sus atributos y responsabilidades favoreciendo la reutilización.

Herencia múltiple

Se establece cuando una clase hereda de varias otras clases, en este caso, la clase hija hereda atributos y responsabilidades de los diferentes padres.



En Java no está permitida la herencia múltiple y al no es considerada una buena práctica de diseño.

Generalización y especialización

Nos encontramos en el modelo que estamos realizando un conjunto de clases, por ejemplo, Caniche y Doberman. Nos damos cuenta que ambas tienen algunos atributos y/o responsabilidades comunes. En dicho caso, creamos una clase de la cual ambas heredarán ambas y transportaremos todos los atributos y/o responsabilidades que eran comunes a esta nueva clase que, en este ejemplo, llamaremos Perro. Este proceso mental de abstracción lo llamamos **generalización**.

Nos encontramos en el modelo que estamos realizando con que modelamos una clase Perro y, analizando mejor el contexto, nos dimos cuenta que hay perros como el dóberman que tienen además de los atributos y/o responsabilidades que describimos otros diferentes que no tienen todos los perros, como, por ejemplo, cuidar(), ya que los caniches no cuidan. En este caso, creamos una clase y le colocamos estos atributos y/o responsabilidades que únicamente tiene ese tipo de perro, en este ejemplo, esas clases son Doberman y Caniche. Este proceso mental de abstracción lo llamamos **especialización**.

Herencia en Java (Clase 10)

Un empleado tiene ciertas características y responsabilidades. Estas características se agregan a la que ya tenía por ser persona y lo mismo sucede con las responsabilidades, es decir, el empleado hereda todo lo que tenía como persona.

En la implementación de la clase empleado indicamos la herencia mediante la palabra **extends**, tanto la superclase como la subclase tienen constructor, pero cada uno debe hacerse cargo de sus atributos entonces, en persona, tendríamos algo así:

```

1 public class Persona {
2     private String nombre;
3     private String dni;
4
5     public Persona(String nombre, String dni){
6         this.nombre=nombre;
7         this.dni=dni;
8     }

```

```

1 public class Empleado extends Persona {
2     private double sueldo;
3     private double descuento;
4     private String legajo;
5
6 }

```

Empleado es una persona y por lo tanto tiene nombre y DNI. Antes de trabajar con los atributos propios es necesario invocar al constructor de la superclase. Siempre lo primero que hacemos en el constructor de una subclase es invocar al constructor de la superclase respetando si tiene parámetros o no.

```

1 public class Persona {
2     private String nombre;
3     private String dni;
4
5     public Persona(String nombre, String dni){
6         this.nombre=nombre;
7         this.dni=dni;
8     }

```

```

1 public class Empleado extends Persona {
2     private double sueldo;
3     private double descuento;
4     private String legajo;
5
6     public Empleado(String nombre, String dni,
7     String legajo){
8         super(nombre, dni);
9         this.legajo=legajo;
10        sueldo=30000;
11    }
12 }

```

Digital House ~

Un empleado es una persona y hereda todas sus características y responsabilidades, por esto, es necesario que antes de crear sus atributos se creen los atributos de persona. Esto lo logramos utilizando la superclase que como método hace referencia a su constructor.

Encapsulamiento y la Herencia

Recordemos que cuando una propiedad es **pública** significa que es accesible desde cualquier clase. Es decir, en el momento en que un objeto quiera acceder a un valor público puede obtenerlo y modificarlo sin ninguna operación de por medio. Esto sería el equivalente a no ocultar información y, por lo tanto, “romper” el encapsulamiento.

Por el contrario, si declaramos un atributo **privado** limitamos completamente el acceso al dato. Nadie que no sea la propia clase puede acceder a ese dato. Siempre que se quiera acceder o modificar el dato, se debe hacer una operación para tal fin, por ejemplo, a través de getters o setters.

Con la herencia aparece un modificador de visibilidad nuevo llamado **protegido**, que en los diagramas UML se especifica con el “#”, donde nos permite tener una visibilidad intermedia del atributo o método al que declaramos como tal. Es decir, es privado para otras clases, pero público para las clases

hijas. El uso de este modificador de visibilidad “rompe” el encapsulamiento y evitaremos en lo posible su uso como buena práctica.

Firma de un método

La firma de un método en la programación orientada a objetos, es ni más ni menos que la definición completa de un método, es decir, su nombre, sus parámetros y sus tipos y el orden de aparición de dichos parámetros.

No podrán en una misma clase existir dos métodos con la misma firma, es decir, con el mismo nombre y cantidad de parámetros con sus respectivos tipos en el mismo orden. Decimos, entonces, que los siguientes métodos tienen diferentes firmas, son métodos diferentes porque, si bien se llaman igual, tienen diferente cantidad de parámetros o difiere alguno de sus tipos:

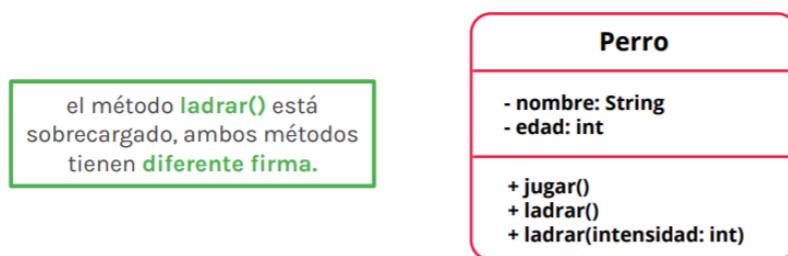
```
+ sumar(numero1: double, numero2: double): double  
+ sumar(numero1: double, numero2: double, numero3: double): double  
+ sumar(numero1: int, numero2: int): int
```

Sobrecarga de Métodos

La sobrecarga de métodos está relacionada con la firma de los mismos. Es posible en el paradigma orientado a objetos tener en una misma clase dos o más métodos con el mismo nombre y cuyo comportamiento sea diferente. Esto es factible porque al momento de invocar dicho método se puede saber a cuál de todos invocar siempre que su firma sea diferente.

El nombre y cantidad, tipo y orden de parámetros forman parte de la firma de un método y deben ser diferentes para poder sobrecargarlo.

El valor que devuelve un método y los modificadores de visibilidad no forman parte de la firma.



Dado que lo que devuelve un método no forma parte de la firma, los métodos sobrecargados pueden devolver cosas diferentes o lo mismo.

Sobrecarga en Java (Clase 10)

Recordemos que solo podemos sobrecargar un método si varía su firma. Como vemos en este caso:

```

public class Empleado{
    private String nombre;
    private String legajo;
    protected double sueldo;
    protected double descuentos;

    public double calcularSueldo(){
        return sueldo-descuentos;
    }
    public double calcularSueldo(double premio){
        return sueldo-descuentos + premio;
    }
}

```

Cuando se utilizan los métodos, según los parámetros que se pasan, actuará el método cuya firma coincida con los parámetros utilizados.

```

public class Main {
    public static void main(String[] args) {
        Empleado miEmpleado = new Empleado("Juan", "1111");
        System.out.println("Sueldo a cobrar: " + miEmpleado.calcularSueldo());
        System.out.println("Sueldo a cobrar: " + miEmpleado.calcularSueldo(5000));
    }
}

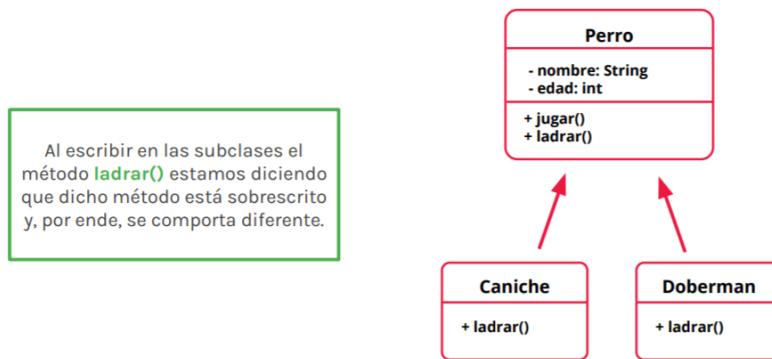
```

Sobreescritura de Métodos

Cuando leemos la palabra sobreescritura nos viene como primera idea la de volver a escribir algo existente. Esta primera idea es muy similar a la que podemos encontrar en la programación orientada a objetos y que resultará clave para ciertos escenarios a resolver.

Para poder sobreescibir métodos necesitamos una relación de herencia, ya que lo que vamos a sobreescibir es un método de la superclase para que se comporte diferente en la subclase.

A diferencia de la sobrecarga donde los métodos tienen que tener diferente firma, en este caso, los métodos deben tener la misma firma.



Sobreescritura en JAVA (Clase 10)

A la clase Vendedor, con los métodos sobreescritos, es necesario darles otro comportamiento para un Vendedor. Por eso, surge la necesidad de sobreescibirlos

```

public class Vendedor extends Empleado{
    private int comision;
    private double importeVentas;

    @Override
    public double calcularSueldo(){
        return sueldo-descuentos + importeVentas/100*comision;
    }
    @Override
    public double calcularSueldo(double premio){
        return sueldo-descuentos + premio+ importeVentas/100*comision;
    }
}

@Override
public double calcularSueldo(){
    return sueldo-descuentos + importeVentas/100*comision;
}
@Override
public double calcularSueldo(double premio){
    return sueldo-descuentos + premio+ importeVentas/100*comision;
}

```

`@Override` nos indica que se anula el comportamiento anterior del método, lo estamos sobrescribiendo para darle una forma distinta de resolver. Para los objetos Vendedor, el método a ejecutar es este que sobrescribe el anterior.

Clase 10: Herencia en JAVA

La clase Object

Todas las clases que creamos en Java derivan de la clase Object, aunque no esté escrito explícitamente. Por eso, cuando creamos una clase nueva, tiene ciertos métodos que hereda. De estos métodos vamos a tomar algunos y para que funcionen correctamente, debemos sobrescribirlos. El comportamiento que tienen por defecto puede causar errores o no ser el más adecuado.

Métodos heredados:

- `String toString()`
- `boolean equals(Object o)`
- `int hashCode()`

.`toString()`²

Toda clase hereda de Object el método `toString()`, es decir, si no lo implementamos, los objetos que instanciamos tendrán este método. Por ejemplo, en nuestra clase `Empleado`, con nombre, legajo, sueldo y descuentos como atributos, qué pasaría si uso el método `toString()`:

```

public class Empleado{
    private String nombre;
    private String legajo;
    protected double sueldo;
    protected double descuentos;
}

```

Al usar el método, no tendríamos un error, pero la información mostrada no sería algo comprensible:

² https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Object/toString

```
public static void main(String[] args) {  
    Empleado nuevoEmpleado=new Empleado("Juan","1111");  
    System.out.println(nuevoEmpleado.toString());  
}
```

Esta es la salida que obtenemos: `com.company.Empleado@1540e19d`

El método `.toString()`, intenta representar con texto el objeto, pero como no lo sobrescribimos, vamos a obtener ese tipo de salida. La solución es sobrescribir el método mostrando solo la información que deseamos mostrar y dándole a la cadena de salida el formato más adecuado.

Recordemos que es importante no cambiar la firma del método, sino estaremos sobrecargando. Agregamos el método `toString()` y devolvemos la cadena con la información del objeto que queremos devolver.

```
public class Empleado{  
    private String nombre;  
    private String legajo;  
    protected double sueldo;  
    protected double descuentos;  
  
    @Override  
    public String toString(){  
        return "Nombre: " + nombre + "\n" +  
               "Legajo: " + legajo;  
    }  
}
```

La salida que obtenemos es la que nosotros programamos, en este caso, el nombre y el legajo.

```
public static void main(String[] args) {  
    Empleado nuevoEmpleado=new Empleado("Juan","1111");  
    System.out.println(nuevoEmpleado.toString());  
}
```

```
Nombre: Juan  
Legajo: 1111
```

.hashCode()

Este es otro de los métodos heredados de `Object`. Cuando se utiliza este método nos devuelve un número único que identifica al objeto, es decir, si tengo dos objetos de la misma clase, el `hashCode()` generaría un número distinto para cada uno y ese número me va a servir para identificarlo.

Para sobrescribir este método hacemos lo siguiente:

```

public class Empleado{
    private String nombre;
    private String legajo;
    protected double sueldo;
    protected double descuentos;

    @Override
    public int hashCode(){
        int hash=31;
        hash= hash* nombre.hashCode();
        hash= hash* legajo.hashCode();
        return hash;
    }
}

```

Para generar un número único se trabaja con números primos. Puede ser cualquier número primo, en este caso se usó el 31. Como nombre y legajo son strings, o sea, también son objetos, tienen su propio hashCode(). Multiplicamos todos los números y obtenemos el hashCode del objeto. En una string, el hashCode se genera a partir de los caracteres. Por ejemplo, el número de legajo es siempre distinto.

```

@Override
public int hashCode(){
    int hash=31;
    hash= hash* nombre.hashCode();
    hash= hash* legajo.hashCode();
    return hash;
}

```

Con la sobrecarga que hicimos, obtenemos el valor que se muestra:

```

public static void main(String[] args) {
    Empleado nuevoEmpleado=new Empleado("Juan","1111");
    System.out.println(nuevoEmpleado.hashCode());
}

```

-1480218112

Equals

Cuando creamos un objeto o instancia, lo que tenemos es una referencia a la memoria (RAM), es decir, no se almacenan datos directamente en la variable de tipo objeto, solo la referencia al lugar donde están los valores de los atributos del objeto. Es por eso que no podemos utilizar el operador “==” para comparar la igualdad entre dos objetos porque estaríamos comparando referencias.

Para comparar correctamente debemos usar el método **equals()**, el cual lo heredamos de Object, pero no siempre funciona correctamente el equals que obtenemos por defecto, por eso, es necesario sobreescibirlo. El método equals() recibe como parámetro un objeto Object, esto nos dará una dificultad adicional a la hora de sobreescibirlo.

Cuando escribimos una clase, una de las cosas que debemos determinar es cómo vamos a comprobar la igualdad de dos instancias de esa clase.

Ejemplo

Como ejemplo vamos a trabajar con la clase empleado y, tal como mencionamos, dos empleados son iguales si sus legajos también lo son.

```

public class Empleado{
    private String nombre;
    private String legajo;
    protected double sueldo;
    protected double descuentos;

}

```

Vamos a sobrescribir el `.equals(Object o)`. Nuestro primer paso es recordar cómo es la firma de este método, debemos respetar la firma del `equals()` heredado de `Object`

```

public class Empleado{
    private String nombre;
    private String legajo;
    private double sueldo;
    private double descuentos;

    @Override
    public boolean equals(Object o){
    }
}

```

Para comenzar a escribir nuestro `equals`, debemos considerar que el parámetro que me está llegando es un `Object`, no dice que sea un `Empleado`, entonces, lo primero a verificar es si realmente es un `Empleado`, si no lo fuera ya podemos decir que no son iguales. Vamos a ver dos formas de comprobarlo: `instanceof` y `getClass()`.

`.instanceof`

Una forma de comparar dos instancias. `Instanceof`

```

@Override
public boolean equals(Object o){
    if (o==null)
        return false;
    if (!(o instanceof Empleado))
        return false;
    else{

    }
}

@Override
public boolean equals(Object o)
{
    if (o == null)
        return false;
    if (!(o instanceof Empleado))
        return false;
    else{
        instanceof es un operador que nos
        permite comprobar si o es una
        instancia de Empleado. En este caso,
        si no lo es devolvemos falso, los
        objetos no pueden ser iguales.
    }
}

```

`.getClass()`

Con `getClass()` también podemos comparar la clase a la que pertenecen los objetos.

```

@Override
public boolean equals(Object o){
    if (o==null)
        return false;
    if (this.getClass()==o.getClass())
        return false;
    else{
        }
}

```

```

@Override
public boolean equals(Object o)
{
    if (o==null)
        return false;
    if (this.getClass()==o.getClass())
        return false;
    else{
        }
}

```

Comprobamos si la clase de la instancia es la misma clase del objeto recibido como parámetro.

Casting

Ahora nos restaría comprobar la igualdad (tener el mismo legajo). Para hacer esta comprobación, vamos a necesitar pedirle a “o”, el legajo para compararlo con el de la instancia. Pero “o” es un Object, o sea, no “sabe” que tiene legajo.

Así quedaría el método terminado, pero para mayor comodidad usamos un casteo. Si bien no es necesario, crear un nuevo objeto, es más cómodo para una posterior lectura.

```

@Override
public boolean equals(Object o){
    if (o==null)
        return false;
    if (!(o instanceof Empleado))
        return false;
    else{
        Empleado empleadoAux=(Empleado)o;
        return
            this.getLegajo().equals(empleadoAux.getLegajo());
    }
}

```

```

@Override
public boolean equals(Object o)
{
    if (o==null)
        return false;
    if (this.getClass()==o.getClass())
        return false;
    else{
        Empleado empleadoAux=(Empleado)o;

        if(this.getLegajo().equals(empleadoAux.getLegajo()))
            return true;
    }
    return false;
}

```

Casteamos “o” y lo asignamos a un objeto de tipo Empleado. Con el casting lo que logramos es transformar, para poder asignarlo a un objeto de tipo Empleado y de esta forma usar sus métodos.

```

@Override
public boolean equals(Object o)
{
    if (o==null)
        return false;
    if (this.getClass()==o.getClass())
        return false;
    else{
        Empleado empleadoAux=(Empleado)o;
        if ((this.getLegajo()).equals(empleadoAux.getLegajo())))
            return true;
        }
    return false;
}

```

Comprobamos que tiene el mismo legajo, lo hacemos con equals. Legajo es un atributo de tipo String, por eso, no podemos usar el operador “==”.

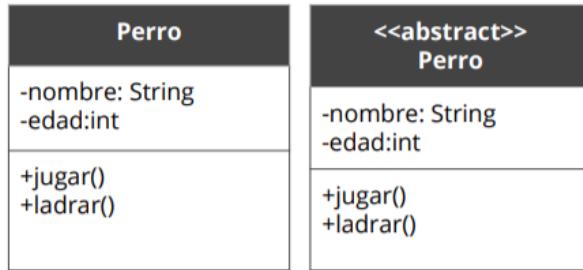
Clase 11: Clases Abstractas

Clase Abstracta

Las clases abstractas son aquellas que por sí mismas no se pueden identificar con algo “concreto” (no existen como tal en el mundo real), pero sí poseen determinadas características que son comunes en otras clases que heredarán de esta.

Estas clases abstractas nos permite declarar métodos, pero que estos no estén implementados, o sea, que no hacen nada en la clase abstracta, y estos métodos que también llamaremos abstractos obligarán a las subclases a sobrescribirlos para darles una implementación.

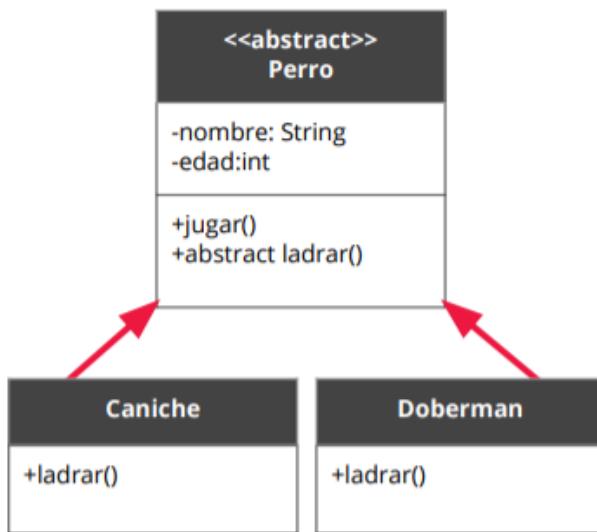
Las clases abstractas en el diagrama UML las representaremos ya sea indicando su nombre en forma cursiva o explicitando arriba de su nombre que es abstracta **<<abstract>>**, podemos optar cualquiera de las dos.



Los métodos abstractos los vamos a especificar en los diagramas UML como se muestra a continuación, el método ladrar será ahora un método abstracto y en los diagramas UML le anteponemos la palabra abstract para identificarlo como tal.

Al convertir en abstracto al método ladrar nos indica que el mismo no está implementado en la clase Perro y deberá implementarlo toda clase que herede de Perro.

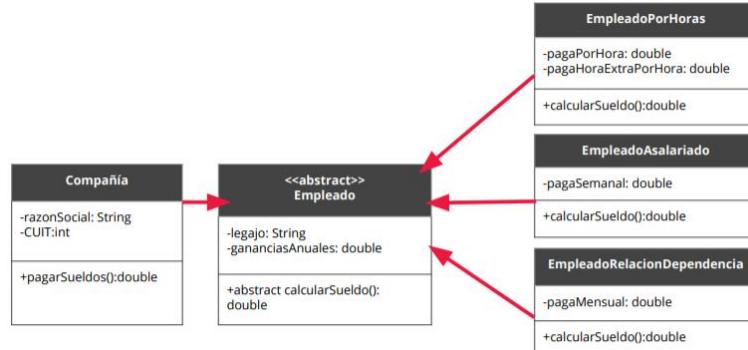
En el ejemplo que vemos a continuación, al contar la clase abstracta Perro con un método abstracto, obliga a la clase Caniche y Dóberman a sobrescribir dicho método implementándolo.



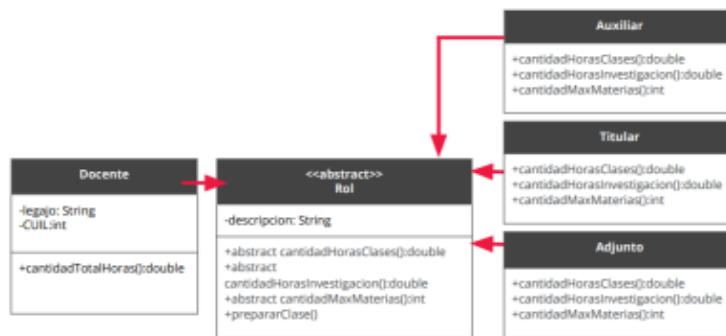
De esta manera una clase abstracta puede tener atributos y métodos que serán heredados por las subclases, pero también puede contener métodos abstractos que actúan como un contrato obligando a estas subclases a implementar dichos métodos.

Buenas prácticas

A continuación, en el siguiente modelo UML, se muestra a una Compañía que tiene varios tipos de empleados y cuyo cálculo de sueldo es diferente en cada caso.



En este otro ejemplo, los diferentes docentes en un instituto tienen diferentes roles y si bien todos los roles preparan la clase, reparten su tiempo de manera diferente según el rol.



Martin Fowler describió a esta problemática de roles “**Role Object**” como un patrón de diseño para modelar roles, siendo una buena práctica a la hora de modelar esta problemática.

Métodos abstractos

Definimos a las clases abstractas y el comportamiento en abstracto con la palabra clave “abstract”. Cómo el comportamiento es abstracto (solo decimos qué hacer), los métodos abstractos no tienen código asociado, no tienen “cuerpo”. Veamos un ejemplo:

abstract class

Especifica que la clase Perro es abstracta.

```

public abstract class Perro{
    public abstract string ladrar();
}

```

Especifica que el método ladrar() es abstracto.

Métodos abstractos en Java

Si Dóberman quiere SER UN Perro, entonces debe respetar el contrato de los Perros: debe implementar un método que se llame ladrar, que devuelva un String y que no reciba parámetros. En pocas palabras, debe sobrescribir todos los métodos abstractos definidos en Perro

Es decir, si Perro dice qué todos los perros deben ladrar(), la hija Doberman, debe “explicar” cómo hacerlo. Llamaremos a esta operación “implementar” el método ladrar().

```
public class Doberman extends Perro{  
  
    public string ladrar() {  
        return "ladro como un dóberman GUAU!!!";  
    }  
  
}
```

Sobrescribir métodos abstractos

Si Caniche quiere SER UN Perro, también debe sobrescribir todos los métodos abstractos definidos en Perro.

```
public class Caniche extends Perro{  
  
    public string ladrar() {  
        return "ladro como un caniche guau...";  
    }  
  
}
```

Cuando implementamos los métodos, estos dejan de ser abstractos, por eso, en Dóberman y Caniche ya no usamos la palabra clave abstract.

Las reglas para la implementación de los métodos abstractos son las de la sobreescritura (de hecho, es lo que estamos haciendo, sobrescribiendo comportamiento abstracto) así que aplican las mismas reglas: respetar tipo, cantidad y orden de los parámetros.

Si no lo hacemos, entonces **no respetamos el contrato**, si no respetamos el contrato, la clase arrojará un **error de compilación**.

Atributos y métodos en clases abstractas

Una clase abstracta es una clase como cualquier otra y, por tanto, puede tener atributos y puede tener métodos concretos. Aun así, tengamos en cuenta que solo los abstractos serán los que definan el contrato.

```
public abstract class Perro{  
    private String nombre;  
  
    public void setNombre(String nombre){  
        this.nombre = nombre;  
    }  
  
    public String getNombre(){  
        return nombre;  
    }  
  
    public abstract string ladrar();  
}
```

¿Por qué tener métodos concretos en una clase que no se puede instanciar? Porque estos métodos son susceptibles de ser reutilizados.

Por otro lado, que una clase abstracta no se pueda instanciar, no significa que no pueda tener constructores, el objetivo es el mismo: podemos definir constructores para reutilizar código cuando heredamos de esa clase abstracta.

```
public class Prueba{  
  
    public static void main(String[] args){  
  
        Doberman perro1 = new Doberman();  
        perro1.ladrar();  
  
        Caniche perro2 = new Caniche();  
        perro2.ladrar();  
  
    }  
}
```

Polimorfismo

Vinculación Dinámica (Dynamic Binding)

La vinculación dinámica de una referencia funcional es igual que un enchufe. En un enchufe se puede conectar diferentes cosas: un TV, una heladera, una notebook.

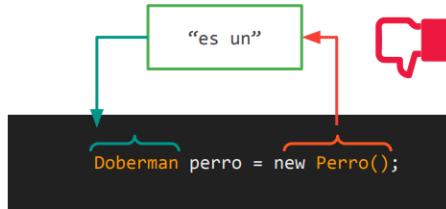
Aquí, tanto la **referencia** como el **objeto** referenciado son del mismo tipo: Dóberman. Sin embargo, es posible que la referencia y el objeto referenciado sean de distinto tipo.



En los lenguajes que no son tipados la referencia y el objeto pueden ser de cualquier tipo, en los **fuertemente tipados** como Java el **objeto** debe ser de una clase que tenga una **relación del tipo “es un”** respecto de la referencia.



Pero la inversa no es posible, es decir, un perro no siempre es un Dóberman.



En Java todas las clases por definición heredan de Object con lo cual un Doberman es un Object.

Polimorfismo

Es la capacidad de un mismo objeto de comportarse como otro. En otras palabras, es la capacidad de un objeto de funcionar de diversas formas. Veamos con los ejemplos anteriores:

```
Perro p;
p = new Doberman();
p.ladrar();

p = new Caniche();
p.ladrar();
```

La referencia p se puede comportar y ladrar como un Doberman.
Y la misma referencia p al vincularse dinámicamente (dynamic binding) con un Caniche.
Se comporta de forma diferente y ladra como un Caniche.

Si utilizamos polimorfismo, podemos estar seguros de que modificaciones futuras, que agreguen nuevas subclases, no deberían afectar el código ni su funcionamiento. Si el código usa Perros (es decir, cualquier objeto que "es un" Perro) siempre que nuevas razas de perros introducidas al sistema hereden de Perro funcionarán correctamente.

Casting

Supongamos que Dóberman tiene un método llamado morderComoDoberman(), pero la referencia o sea la variable es del tipo Perro. Para forzar a un perro a que sea un Dóberman utilizaremos el casteo. De esta manera podremos invocar los métodos propios de Dóberman.

```
Perro perro = new Doberman();
perro.ladrar();

((Doberman)perro).morderComoDoberman()
Casteo
```

Lo mismo sucede si nuestro objeto referencia es del tipo Object. En este caso como la clase Object no tiene tampoco el método ladrar() debemos castearlo ya sea a Perro que tiene dicho método o a Doberman.

```

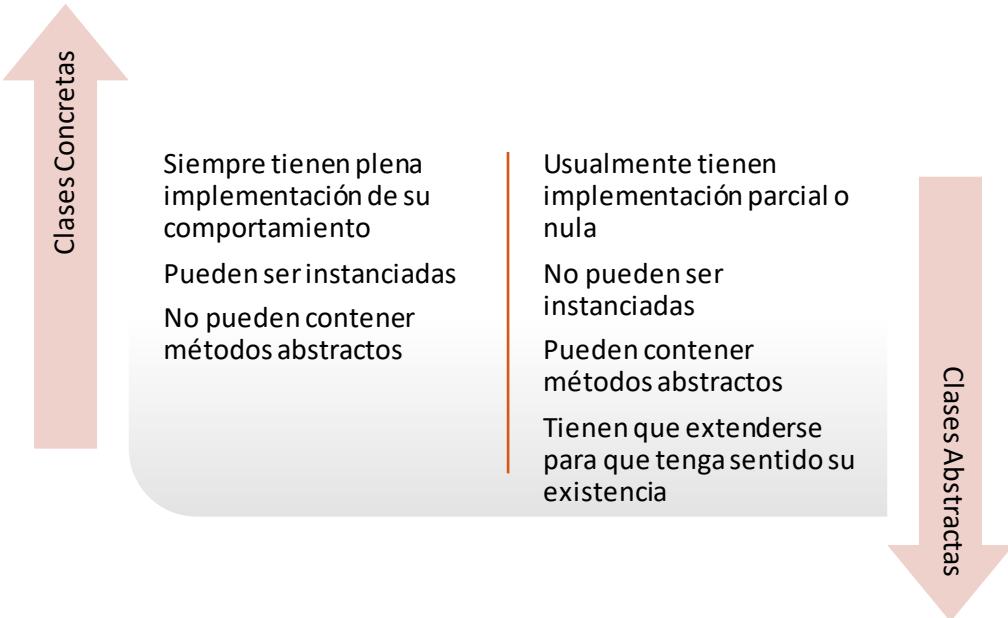
Object perro = new Doberman();
((Perro)perro).ladrar()

((Doberman)perro).morderComoDoberman()

Casteo

```

Clases abstractas vs concretas



Clase 13: Interface

Interface

A veces cuando estamos modelando sistemas necesitamos una forma de poder agrupar nuestras clases a partir de comportamientos. Las **interfaces** son, justamente, el concepto clave para poder hacer eso.

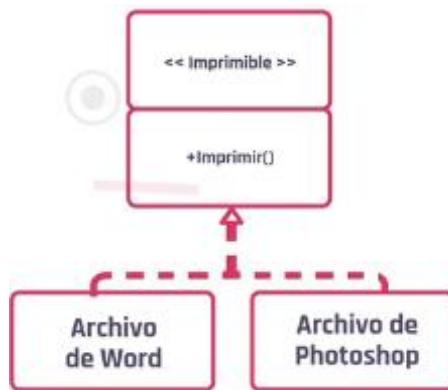
¿Qué quiere decir esto de los comportamientos? Supongamos que estamos modelando una impresora, vamos a tener las clases archivo de Photoshop y archivo de Word. Tendremos, también, una clase impresora que va a tener un método imprimir que va a recibir archivos por parámetro.

¿Cuántos métodos debería tener la impresora? Si queremos poder imprimir archivos de Photoshop y archivos de Word tendría sentido que dos responsabilidades distintas variando el objeto recibido por parámetro. El problema surge cuando queremos poder imprimir también otro tipo de archivos, deberíamos agregar una nueva responsabilidad por cada tipo de archivo nuevo que quisiéramos imprimir lo que no tiene mucho sentido si queremos que nuestro sistema crezca.

De hecho, podríamos usar herencia para pensar genéricamente en el concepto de archivo, pero ¿Qué pasaría si después queremos imprimir algo que no es un archivo? como un contrato que herede de documentoLegal. Todos estos pensamientos nos están llevando a que, en realidad, una impresora imprime cosas que se pueden imprimir sin importar su tipo, es decir nos interesan sólo aquellos objetos que son de clases que se comportan de una forma particular. O sea, aquellos que son imprimibles. Este concepto lo vamos a definir con las **interfaces**, podemos agregar entonces en

nuestro sistema una interfase llamada `imprimible` que define la responsabilidad imprimir. Esto no significa que vamos a definir cómo se hace para imprimir algo solo estamos indicando que puede haber objetos que se comporten como imprimibles.

En UML las interfaces no tienen atributos y se distinguen del resto de las clases poniendo el nombre entre símbolos dobles de menor y mayor. ¿De qué nos sirve todo esto? Podemos decir que nuestras clases se comportan como imprimibles relacionándola con la interfase mediante una flecha como la herencia, pero punteada.



Esto significa que, cada clase, ahora debe implementar el método `imprimible` porque debe cumplir con el contrato definido por la interfaz. De este modo, finalmente, vamos a modificar nuestra clase impresora para que tenga un solo método *imprimir*, pero cuyo parámetro será algo de tipo imprimible.

A medida que nuestro sistema crezca, cada cosa que queramos imprimir deberá relacionarse con interfase e implementar el método apropiado. Hay que tener en cuenta que muchas veces las interfaces son un poco más sencillas de identificar a través de los **sufijos -able** e **-ible** y representan distintos comportamientos que van a tener las **clases concretas**.

Es importante entender que las interfaces, al fin y al cabo, no nos permiten definir instancias no vamos a instanciar un objeto de tipo imprimible. Las interfaces nos dan características compartidas por las clases, un contrato que deben cumplir y que nos permiten entenderlas de forma genérica según su comportamiento.

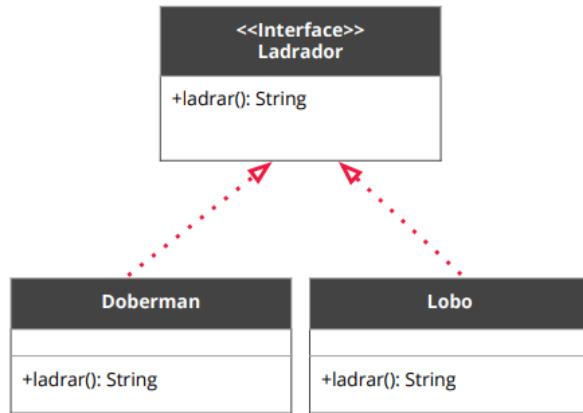
Interfaces y la Herencia

Interfaces

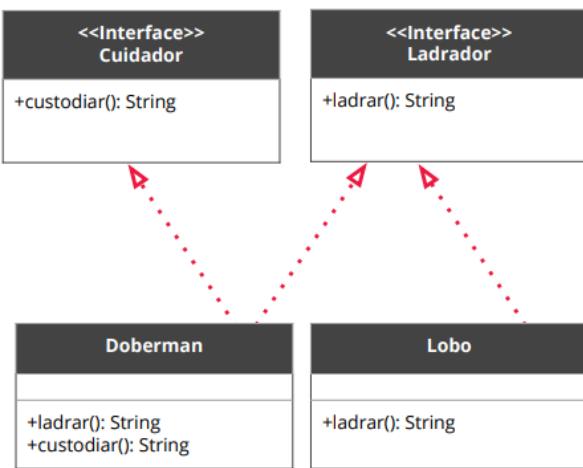
Las interfaces son también relaciones del tipo “es un”, muy similares a las clases abstractas: se definen con la palabra clave “interface” en vez de “class”. Todos sus métodos son abstractos, por lo cual, no es necesaria la palabra “abstract” y, al igual que las clases abstractas, los métodos no definen un cuerpo.

Una interface establece un contrato. Toda clase que implemente una interface está **obligada a implementar todos los métodos de esa interface**.

Por ejemplo, todas aquellas clases que implementen `Ladrador` deben implementar el método `ladrar`.



Una clase solo puede heredar de una sola clase, pero puede implementar múltiples interfaces.

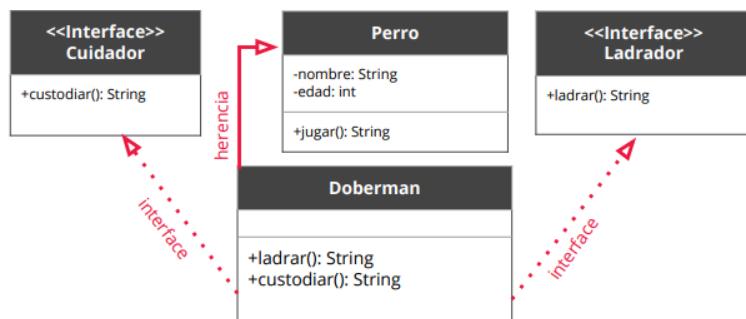


Interfaces y la Herencia

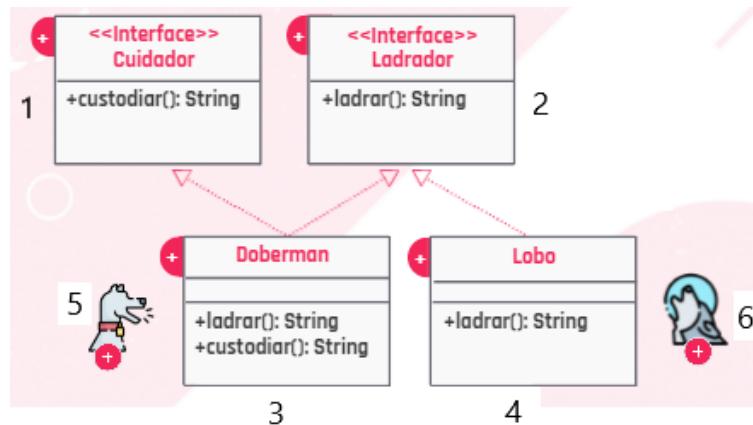
Cuando heredamos de una clase sumamos atributos y comportamientos de la clase padre, mientras que cuando implementamos una interface solo obligamos a la clase que la implementa a sobrescribir, es decir, implementar, los métodos de la misma.

Tienen en común que son relaciones del tipo “es un”, con lo cual las interfaces también nos permiten realizar vinculación dinámica y, por ende, polimorfismo.

Lo que permiten las interfaces es independizarse de una jerarquía, permiten agregar comportamiento a una clase que no se obtenga desde un nivel superior en la jerarquía, se “enchufa” lateralmente a la jerarquía. Incluso podríamos hasta mezclar ambos mecanismos.



Interface e Implements



Paso 1:

```
public interface Cuidador{  
    public void String custodiar();  
}
```

Paso 2:

```
public interface Ladrador{  
    public void String ladrar();  
}
```

Paso 3:

```
public  
class Doberman implements Cuidador, Ladrador{  
  
    public void String custodiar(){  
        return "estoy atento custodiando la  
casa";  
    }  
  
    public void String ladrar(){  
        return "Guau! Guau!";  
    }  
}
```

Paso 4:

```
public class Lobo implements Ladrador{  
  
    public void String ladrar(){  
        return "guau! los lobos también  
ladramos";  
    }  
}
```

Paso 5:

```
/*Dada una referencia ladrador del tipo Ladrador (Ladrador ladrador)*/  
  
ladrador = new Doberman(); //ladrador es ahora del tipo Doberman()  
System.out.println(ladrador.ladrar()); //Polimorfismo
```

Paso 6:

```
/*Dada una referencia ladrador del tipo Ladrador (Ladrador ladrador)*/  
  
ladrador = new Lobo(); //ladrador es ahora del tipo Lobo()  
System.out.println(ladrador.ladrar()); //Polimorfismo
```

Comparar objetos

A la hora de comparar tipos primitivos lo hacemos con los operadores “==”, “>”, “<”, “>=”, “<=”, “!”, “!=”, pero, ¿cómo hacemos si queremos comparar dos objetos? Porejemplo, dos morrones

Para poder comparar dos objetos lo primero que tendremos que saber es por cuál o cuáles de sus atributos los vamos a comparar. Es decir, cómo responderíamos a la pregunta: ¿estos morrones son iguales?

La primer duda que nos surgirá es si debemos considerar el color, tipo, peso o el tamaño y sobrescribir el método equals().

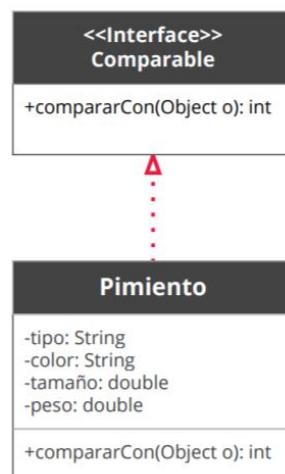
Pero qué sucede ahora si lo que queremos saber es ¿cuál de los pimientos es mayor que el otro? En este caso no usamos el método equals() porque no nos dice cual es mayor o menor.

Método compararCon

Una solución a la problemática planteada es lograr que todos los objetos que necesite comparar tengan por ejemplo un método compararCon que reciban como parámetro al otro objeto con el que se desea hacer la comparación y nos devuelva, por ejemplo:

- Cero: si son iguales.
- Mayor a cero: si el objeto que invoca el método es mayor al recibido como parámetro.
- Menor a cero: si el objeto que invoca el método es menor al recibido como parámetro.

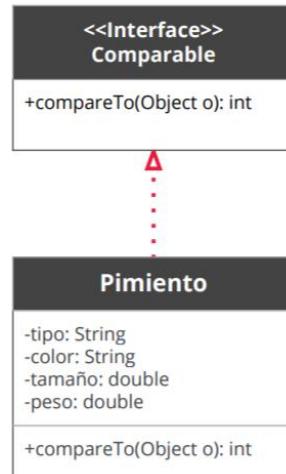
¿Cómo hacemos para obligar a todos los objetos que queremos comparar para que tengan un método compararCon? Con las interfaces podemos hacer que quien la implemente posea sí o sí un método compararCon y pueda establecer su propia implementación.



Interface Comparable

No necesitamos crear una interface para comparar objetos porque Java tiene la suya, es la interface Comparable y es necesaria utilizarla en otras circunstancias para comparar objetos, por ejemplo, para ordenarlos en las colecciones.

El método que obliga a implementar la interface Comparable de Java es el método **compareTo**. Para utilizar la interface Comparable de Java debemos importar el paquete *java.lang*.



Implementación en Java

```
import java.lang.*;

public class Pimiento implements Comparable{

    private String tipo;
    private String color;
    private double tamano;
    private double peso;

    public Pimiento(){
    }

    public int compareTo(Object obj){

        Pimiento p2 = (Pimiento) obj;
        int respuesta = 0;

        if(this.getPeso() > p2.getPeso())
            respuesta = 1;
        if(this.getPeso() < p2.getPeso())
            respuesta = -1;

        return respuesta;
    }

    public void setTipo(String tipo){
        this.tipo = tipo;
    }

    public void setColor(String color){
        this.color = color;
    }

    public void setTamano(double tamano){
        this.tamano = tamano;
    }

    public void setPeso(double peso){
        this.peso = peso;
    }

    public String getTipo(){
        return tipo;
    }

    public String getColor(){
        return color;
    }

    public double getTamano(){
        return tamano;
    }

    public double getPeso(){
        return peso;
    }
}
```

El método compareTo debe devolver: Si son iguales: 0. Si es mayor: un número mayor a cero. Si es menor: un número menor a cero.

```

public class Prueba{

    public void main(String args[]){
        Pimiento p1 = new Pimiento();
        p1.setPeso(200);
        p1.setColor("amarillo");
        Pimiento p2 = new Pimiento();
        p2.setColor("rojo");
        p2.setPeso(150);

        if(p1.compareTo(p2) > 0){
            System.out.println("Pimiento amarillo es mayor al rojo");
        }else if(p1.compareTo(p2) < 0){
            System.out.println("Pimiento rojo es mayor al amarillo");
        }else{
            System.out.println("Pimiento rojo es igual al amarillo");
        }
    }
}

```

Clase 16: Colecciones

Colecciones

Una colección representa a un grupo de objetos que son conocidos como elementos. Cuando queremos trabajar con un conjunto de objetos, por ejemplo, en una relación uno a muchos o muchos a muchos, necesitamos la manera de poder guardarlos.

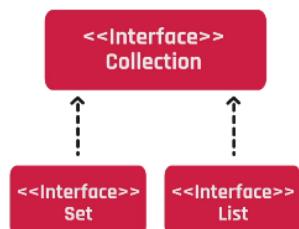
En JAVA se emplea la interfaz Collection para este propósito, gracias a esta interfaz podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes que pueden ser:

- Añadir elementos: add(Object o)
- Eliminar elementos: remove
- Obtener el tamaño: size()
- Recorrer todos los objetos almacenados en la colección: iterator()

Partiendo de la interfaz genérica Collection se extienden otra serie de interfaces genéricas estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.

Interfaz Set

La interfaz set contiene únicamente los métodos heredados de collection, añadiendo la descripción de que los elementos duplicados están prohibidos. Para comprobar si los elementos están duplicados o no es necesario que tengan implementados, de forma correcta, los métodos equals y hashCode.



Interfaz List

Esta interfaz define una sucesión de elementos. A diferencia de la interfaz Set, esta admite elementos duplicados.

Map

Es la única colección que no extiende de Collection. Esta asocia claves a valores, es decir, que podemos almacenar elementos y luego acceder a los mismos a través de una clave. Esta interfaz no puede contener clave duplicadas y cada una ellas solo pueden tener asociado un valor como máximo.

Tipos de colecciones

ArrayList

Implementa la interface List. Almacena los elementos en forma contigua y, por eso, tiene acceso secuencial a los elementos. Es muy eficiente cuando tenemos que almacenar y acceder a los elementos directamente a través de su posición (pos). Sus métodos más importantes son:

`.add(Object o)`: Agrega un elemento.

`.add(Object o, int pos)`: Agrega un elemento en una posición determinada.

`.remove(Object o)`: Quita un elemento.

`.remove(int pos)`: Quita un elemento en una posición determinada.

`.get(int pos)`: Obtiene un elemento en una posición determinada.

`.size()`: Permite conocer la cantidad de elementos de la lista.

LinkedList

Implementa la interface List. Esta implementación es más performante cuando necesitamos hacer inserciones en lugares próximos a la mitad de la lista, es decir, cuando estamos manipulando sus elementos, pero es poco eficiente cuando solo necesitamos agregar o acceder a los elementos, ya que en estos casos el ArrayList es una mejor opción. Sus métodos más importantes son:

`.add(Object o)`: Agrega un elemento.

`.add(Object o, int pos)`: Agrega un elemento en una posición determinada.

`.remove(Object o)`: Quita un elemento.

`.remove(int pos)`: Quita un elemento en una posición determinada.

`.get(int pos)`: Obtiene un elemento en una posición determinada.

`.size()`: Permite conocer la cantidad de elementos de la lista

HashSet

Implementa la interface Set. A diferencia de ArrayList y LinkedList, las HashSet no pueden almacenar valores duplicados ni nulos. Es la implementación con mayor rendimiento de todas, pero no garantiza ningún orden a la hora de recorrerla. Es decir que al recorrerla los elementos no se encuentran en el orden en que los hemos insertado y por esto mismo no cuenta con un método get. Sus métodos más importantes:

`.add(Object o)`: Agrega un elemento.

`.remove(Object o)`: Quita un elemento.

`.size()`: Permite conocer la cantidad de elementos de la lista.

LinkedHashSet

Implementa la interface Set. No puede almacenar valores duplicados ni nulos, pero los elementos son almacenados en el mismo orden de inserción. Con lo cual, al recorrerla, veremos que los elementos se encontrarán en el mismo orden en que fueron insertados. Es un poco menos performante que HashSet y no cuenta con un método get. Sus métodos más importantes son:

`.add (Object o)`: Agrega un elemento.

`.remove (Object o)`: Quita un elemento.

`.size ()`: Permite conocer la cantidad de elementos de la lista.

TreeSet

Implementa la interface Set, pero también hereda de una clase llamada SortedSet y esto Permite que TreeSet almacene sus elementos de acuerdo al valor de dichos elementos. Sus métodos más importantes son:

`.add (Object o)`: Agrega un elemento.

`.remove (Object o)`: Quita un elemento.

`.size ()`: Permite conocer la cantidad de elementos de la lista.

HashMap

Implementa la interface Map. Los mapas son conjunto de duplas (clave - valor). Es razonable pensar que las claves no se pueden repetir y cada clave corresponde solo a un valor. Tanto en las HashMap como en las HashSet los elementos no se almacenan en el mismo orden de inserción. Sus métodos más importantes son:

`.put (Object key, Object value)`: Agrega un elemento.

`.get (Object key)`: Permite obtener un elemento según una clave determinada.

`.remove (Object key)`: Quita un elemento según una clave determinada.

`.size ()`: Permite conocer la cantidad de elementos de la lista.

LinkedHashMap

Implementa la interface Map. A diferencia de HashMap los elementos se almacenan en función del orden de inserción. Sus métodos más importantes son:

`.put (Object key, Object value)`: Agrega un elemento.

`.get (Object key)`: Obtiene un elemento según una clave determinada.

`.remove (Object key)`: Quita un elemento según una clave determinada.

`.size ()`: Permite conocer la cantidad de elementos de la lista.

TreeMap

Implementa la interface Map. Los elementos se almacenan ordenadamente según la clave. Es importante aclarar que se ordenan según la clave y no según el valor del objeto que almacenan.

`.put (Object key, Object value)`: Agrega un elemento.

`.get (Object key)`: Obtiene un elemento según una clave determinada.

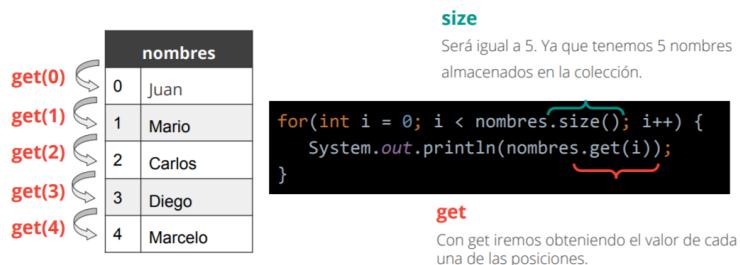
`.remove (Object key)`: Quita un elemento según una clave determinada.

`.size ()`: Permite conocer la cantidad de elementos de la lista.

Recorrer Colecciones

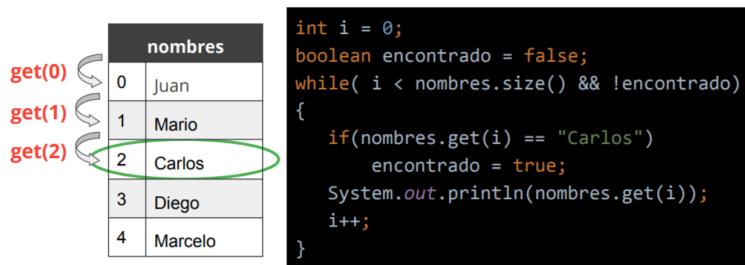
for / while

Una manera de recorrer una colección es a través de un **ciclo for**



Otra manera muy similar es hacerlo con un **ciclo while**. Es muy útil cuando necesitamos cortar el ciclo antes de recorrer todos sus elementos.

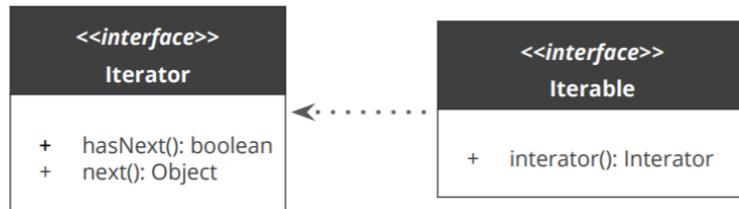
En el siguiente ejemplo, necesitamos encontrar a "Carlos", con lo cual, una vez hallado, podemos salir del bucle para no seguir recorriendo innecesariamente.



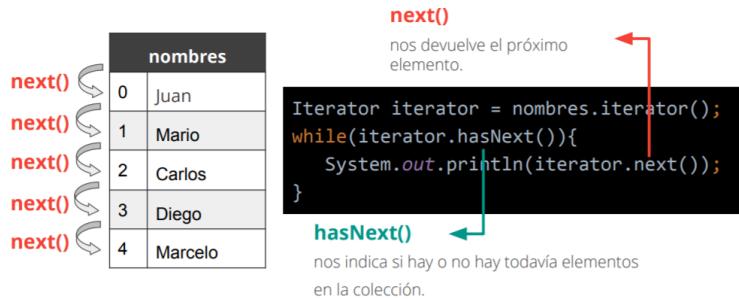
Para poder recorrer una colección con un ciclo for o while, podemos observar que necesitamos de los métodos `size()` y `get()`. Como no todas las colecciones poseen estos métodos no podremos utilizar estas opciones en algunas colecciones. Solo podemos utilizar estas opciones de recorrido con las List, es decir, con ArrayList y LinkedList.

Iterator

En Java las colecciones implementan la interface Iterable, lo que obliga a implementar el método `iterator()`. Este devolverá un objeto del tipo Iterator, donde mediante los métodos `hasNext()` y `next()` podremos recorrer la colección.



Utilizar el método `iterator()` es otra manera de recorrer las colecciones y podremos hacerlo en todos los tipos de colecciones.



for each

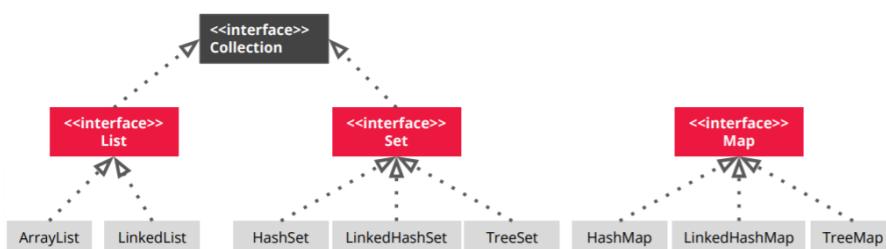
Muchos lenguajes poseen una manera simple y elegante de recorrer una colección a través de los ciclos `for each`. Desde la versión 1.5 de Java se incluyó esta sencilla forma de recorrer las colecciones.



Operaciones sobre colecciones

Crear una colección

Las colecciones en Java están implementadas a través de esta familia de clases e interfaces. Conocerla nos permitirá crear las colecciones de la manera más genérica posible.



Al momento de crear una colección o cualquier tipo de objeto, es una buena práctica que el tipo de referencia sea lo más genérico posible.



Dado que **ArrayList**, y **LinkedList** implementan la interface **List**, trataremos a estas colecciones siempre como una **List**, ya que las operaciones que necesitamos hacer sobre estas colecciones se encuentran establecidas en esta interface.

```
List nombres = new ArrayList();
```



```
List nombres = new LinkedList();
```

Por el contrario, **HashSet**, **LinkedHashSet** y **TreeSet** implementan la interface **Set**, por ende, trataremos a estas colecciones siempre como una **Set**.

```
Set nombres = new HashSet();
```



```
Set nombres = new LinkedHashSet();
```



```
Set nombres = new TreeSet();
```

HashMap, **LinkedHashMap** y **TreeMap** implementan la interface **Map**, por ende, trataremos a estas colecciones siempre como una **Map**.

```
Map nombres = new HashMap();
```



```
Map nombres = new LinkedHashMap();
```



```
Map nombres = new TreeMap();
```

Agregar elementos

Tanto la interface **List** como **Set** nos proporcionan el método **add** que recibe como parámetro un **Object** y, como toda clase hereda de **Object**, podemos almacenar cualquier tipo de objeto en ellas. Comencemos con **ArrayList**.

nombres	
add	0 Juan
add	1 Mario
add	2 Carlos
add	3 Marcelo
add	4 Marcelo

```
List nombres = new ArrayList();
nombres.add("Juan");
nombres.add("Mario");
nombres.add("Carlos");
nombres.add("Marcelo");
nombres.add("Marcelo");
```

En el caso de las **Set**, si bien tienen el mismo método **add**, se comportan muy diferente. No almacenan los valores repetidos ni nulos y, en el caso de las **HashSet** no respeta el orden de inserción.

	nombres
add	Marcelo
add	Juan
add	Carlos
add	Mario
add	Marcelo

```
Set nombres = new HashSet();
nombres.add("Juan");
nombres.add("Mario");
nombres.add("Carlos");
nombres.add("Marcelo");
nombres.add("Marcelo");
```

Las **LinkedHashSet**, como toda Set, no almacenan valores repetidos ni nulos, pero, a diferencia de la HashSet, sí respetan el orden de inserción.

	nombres
add	Juan
add	Mario
add	Carlos
add	Marcelo
add	Marcelo

```
Set nombres = new LinkedHashSet();
nombres.add("Juan");
nombres.add("Mario");
nombres.add("Carlos");
nombres.add("Marcelo");
nombres.add("Marcelo");
```

Las **TreeSet** como toda Set no almacenan valores repetidos ni nulos y los inserta ordenadamente. En el siguiente ejemplo, al ser elementos String los inserta alfabéticamente.

	nombres
add	Carlos
add	Juan
add	Marcelo
add	Mario
add	Marcelo

```
Set nombres = new TreeSet();
nombres.add("Juan");
nombres.add("Mario");
nombres.add("Carlos");
nombres.add("Marcelo");
nombres.add("Marcelo");
```

Las Map no poseen un método add, en su lugar, poseen un método llamado put que recibe dos parámetros: una key y un valor. Permiten valores duplicados, pero no keys duplicadas. Las **HashMap**, además, no respetan el orden de inserción.

	nombres
put	30888999 Mario
put	40888999 Marcelo
put	27888999 Carlos
put	29888999 Juan
put	50888999 Marcelo

```
Map nombres = new HashMap();
nombres.put(29888999, "Juan");
nombres.put(30888999, "Mario");
nombres.put(27888999, "Carlos");
nombres.put(40888999, "Marcelo");
nombres.put(50888999, "Marcelo");
```

Las **LinkedHashMap** tienen el mismo comportamiento que una Map, pero a diferencia de las HashMap respetan el orden de inserción.

	nombres
put	29888999 Juan
put	30888999 Mario
put	27888999 Carlos
put	40888999 Marcelo
put	50888999 Marcelo

```
Map nombres = new LinkedHashMap();
nombres.put(29888999, "Juan");
nombres.put(30888999, "Mario");
nombres.put(27888999, "Carlos");
nombres.put(40888999, "Marcelo");
nombres.put(50888999, "Marcelo");
```

Las **TreeMap** tienen el mismo comportamiento que una Map, pero a diferencia del resto los inserta ordenadamente según la key. En este caso, la Key es un entero, por lo tanto, los ordena de menor a mayor.

The diagram illustrates the insertion of five key-value pairs into a `TreeMap` named `nombres`. The keys are phone numbers and the values are names. The `put` method is shown five times with arrows pointing from the method call to the corresponding row in the table.

nombres	
27888999	Carlos
29888999	Juan
30888999	Mario
40888999	Marcelo
50888999	Marcelo

```
Map nombres = new TreeMap();
nombres.put(29888999,"Juan");
nombres.put(30888999,"Mario");
nombres.put(27888999,"Carlos");
nombres.put(40888999,"Marcelo");
nombres.put(50888999,"Marcelo");
```

Eliminar elementos

Todas las colecciones poseen un método `remove`. En el caso de las `List`, como `ArrayList` y `LinkedList`, se pueden eliminar por índice o por valor.

The diagram shows a `List` named `nombres` containing five elements: Juan, Mario, Carlos, Marcelo, and Marcelo. A red arrow points from the `remove` method to the third element, "Carlos". Two code snippets are shown to the right: `nombres.remove("Carlos");` and `nombres.remove(2);`.

nombres	
0	Juan
1	Mario
2	Carlos
3	Marcelo
4	Marcelo

```
nombres.remove("Carlos");
nombres.remove(2);
```

En el caso de todas las implementaciones de `Set` solo se pueden eliminar elementos pasando como parámetro al método `remove` el valor almacenado.

The diagram shows a `Set` named `nombres` containing four elements: Marcelo, Juan, Carlos, and Mario. A red arrow points from the `remove` method to the element "Carlos". A code snippet to the right shows `nombres.remove("Carlos");`

nombres	
	Marcelo
	Juan
remove	Carlos
	Mario

```
nombres.remove("Carlos");
```

En el caso de las `Map`, los elementos se eliminan por Key. Es decir, `remove` recibe como parámetro la Key del elemento que queremos eliminar.

The diagram shows a `Map` named `nombres` with five entries. A red arrow points from the `remove` method to the entry with key 27888999 and value Carlos. A code snippet to the right shows `nombres.remove(27888999);`

nombres	
27888999	Carlos
29888999	Juan
30888999	Mario
40888999	Marcelo
50888999	Marcelo

```
nombres.remove(27888999);
```

Obtener o buscar elementos

En el caso de las `List`, como `ArrayList` y `LinkedList`, si queremos obtener un valor y conocemos el índice, podemos utilizar el método `get` que recibe como parámetro el índice de la posición.

The diagram shows a `List` named `nombres` with five elements. A red arrow points from the `get(2)` method to the third element, "Carlos". A code snippet to the right shows `System.out.println(nombres.get(2));`

nombres	
0	Juan
1	Mario
get(2)	Carlos
2	Diego
3	Marcelo

```
System.out.println(nombres.get(2));
```

En el caso de las `Set`, para obtener un elemento debemos buscarlo recorriendo la colección, ya que las `Set` no tienen índice.

nombres	
Juan	
Mario	
Carlos	
Diego	
Marcelo	

```

boolean encontrado = false;
String nombre = null;
Iterator it = nombres.iterator();
while(it.hasNext() && !encontrado) {
    nombre = (String) it.next();
    if(nombre == "carlos")
        encontrado = true;
}
System.out.println("Encontramos a " + nombre);

```

En el caso de las **Map**, para obtener un elemento, podemos hacerlo a través de su Key con el método `get`.

nombres	
27888999	Carlos
29888999	Juan
30888999	Mario
40888999	Marcelo
50888999	Marcelo

```
nombres.get(30888999);
```

Programación Paramétrica

La programación paramétrica no es exclusiva de Java. De hecho, el lenguaje tardó bastante en incorporarla. Ada, un lenguaje orientado a objetos, lo contempla desde los años ochenta. Hay otros lenguajes que incorporan esta característica, como C++ (a través de templates), D, Eiffel, Delphi, TypeScript, etc. Hay muchos otros lenguajes que utilizan los “tipos paramétricos de datos”. Algunos, lo llaman genericity dado que se establece un tipo de dato genérico para la estructura de dato o clase que se está definiendo, que se establecerá al momento de su uso. En Java, se lo denomina de manera similar: **Generics**.

En pocas palabras, Generics consiste en diferir la pregunta “¿de qué tipo es este objeto?”. Entonces, el “tipo” del objeto se deja como un parámetro que el programador establecerá al momento de trabajar con ese objeto.

Sintaxis y uso

En Java, para definir un tipo paramétrico de datos, usamos los “`< >`”. Supongamos un ejemplo sencillo: un balde. Este puede contener muchas cosas, tierra, arena, agua, combustible, etc.

```

public class Balde<T> {
    private T contenido;

    public Balde() {
    }

    public llenar(T contenido) {
        this.contenido = contenido;
    }

    public T obtenerContenido() {
        return contenido;
    }
}

```

Aquí, cuando definimos la clase balde, no establecimos el tipo de su contenido (el tipo de su atributo “contenido”), sino que “diferimos” esa decisión hasta el momento de usarlo. Es decir, dejamos el tipo como parámetro. Ese parámetro está definido por la letra T (puede ser cualquier letra del abecedario, menos Ñ, por supuesto). Normalmente se usa T por type. Esto también ocurre con las operaciones

sobre el contenido: limitamos su uso de acuerdo con el tipo del atributo contenido. Ahora, suponiendo que tenemos las clases agua, arena, combustible, etc., podríamos hacer:

```
Agua a = new Agua();
Combustible c = new Combustible();
Balde<Agua> b = new Balde<>();
b.setContenido(a);

// NO COMPILA! No puedo poner combustible en un balde de agua!
//b.setContenido(c);

// si el balde es de agua, siempre obtendré agua de él
System.out.println("Voy a tomar" + b.obtenerContenido());
```

Como podemos ver, no hubo casteos para operar con el contenido, y por tanto no hay peligros de errores al momento de la ejecución. Adicionalmente, no podemos poner algo que no corresponda dentro del balde.

¿Por qué no hacemos, entonces, el tipo del contenido “Object”? Eso nos permitiría poner agua, combustible o arena en el balde...

```
public class Balde {
    private Object contenido;

    public Balde() {
    }

    public llenar(Object contenido) {
        this.contenido = contenido;
    }

    public Object obtenerContenido() {
        return contenido;
    }
}
```

Es totalmente válido, pero no solo debemos castear, sino que podríamos mezclar el contenido del balde:

```
Agua a = new Agua();
Combustible c = new Combustible();
Balde b = new Balde ();
b.setContenido(a);
b.setContenido(c);

//dudo que el contenido tenga un sabor agradable...
System.out.println("Voy a tomar" + b.obtenerContenido());
```

Es por esto por lo que el uso de tipos paramétricos, es decir, de Generics se vuelve interesante.

Colecciones paramétricas

Para comenzar, recordemos que en todas las operaciones que podemos hacer sobre las colecciones, el tipo utilizado es Object.

- add(**Object** o) : void

- `get(int i): Object`
- `iterator(): iterator`
 - `hasNext(): boolean`
- `next(): Object`

Como en Java, todas nuestras clases heredan de `Object`, entonces, podemos mezclar objetos de diferentes tipos en la misma colección.

Ejemplo

Para analizar en detalle el problema, asumamos que tenemos una serie de vehículos donde nuestra empresa debe administrar lo que transportan. Podríamos hacer una lista con los vehículos de nuestra empresa.

```
List vehiculos = new ArrayList();

Moto moto = new Moto();
Camion camion = new Camion();

vehiculos.add(moto);
vehiculos.add(camion);
```

Si en algún momento quisiéramos obtener algún vehículo de la lista, debemos castear.

```
Moto moto = (Moto) vehiculos.get(0);
Camion camion = (Camion) vehiculos.get(1);
```

The diagram shows two lines of code: `Moto moto = (Moto) vehiculos.get(0);` and `Camion camion = (Camion) vehiculos.get(1);`. Below these lines, two green arrows point upwards from a green box labeled "casting".

Si quisiéramos recorrer la lista de vehículos y ver si están disponibles, podríamos hacer esto.

```
for(Object o :vehiculos) {
    System.out.println(((Vehiculo)o).estaDisponible());
}
```

Pero, si quisiéramos agregar carga al camión, es necesario castear solo a aquellos elementos de la lista que sean camiones. De otra manera, tendríamos un error de ejecución.

```
for(Object o :vehiculos) {
    if( o instanceof Camion)
        ((Camion)o).cargar("papas");
}
```

Solución con Generics

Para evitar mezclar objetos de diferente tipo en una colección, a partir de la versión 1.5 de Java todas las colecciones pueden recibir como parámetro el tipo, es decir, soportan Generics.

```
List<Camion> vehiculos = new ArrayList<Camion>();
```

Si usamos las colecciones parametrizando su tipo, tendremos un control en tiempo de compilación sobre los tipos de los objetos que agregamos a la colección, de forma tal que, al momento de correr la colección, no hace falta chequear por el instanceof dado que no deberemos castear, porque no podríamos “mezclar” tipos de objetos.

```
for(Camion o :vehiculos) {  
    o.cargar("papas");  
}
```

Arrays

Los arrays son estructuras de datos estáticas que permiten guardar elementos del mismo tipo en forma contigua.

Permiten el acceso a sus elementos de forma aleatoria a través de un índice que comienza desde 0 (cero). La colección ArrayList tiene este mismo comportamiento y, por ello, su nombre.

En Java, un array es un objeto y, como tal, debe usarse el operador new para crear una instancia, pero a diferencia de las colecciones, los arrays son de longitud fija, la cual debe definirse en la creación, siendo inmutable.

Con los corchetes “[]” se indica que es un array.

```
String[] nombres = new String[5];
```

El tipo puede ser o un tipo primitivo o cualquier clase de objeto.

Al instanciarlo se debe definir dentro del [] el tamaño de la estructura.

Establecemos valores a un array a través de su índice. Dado que es una estructura fija, no se pueden eliminar elementos.

nombres	
0	Juan
1	Mario
2	null
3	Marcelo
4	null

```
nombres[0] = "Juan";  
nombres[1] = "Mario";  
nombres[3] = "Marcelo";
```

Podemos recorrer un array a través de un ciclo for, while o for each y también utilizar la propiedad length que nos indica el tamaño del array.

```

for(int i = 0; i < nombres.length; i++)
    System.out.println(nombres[i]);

int i = 0;
while(i < nombres.length) {
    System.out.println(nombres[i]);
    i++;
}

for(String nombre : nombres)
    System.out.println(nombre);

```

nombres	
0	Juan
1	Mario
2	null
3	Marcelo
4	null

Arrays vs. Colecciones

Array		Colecciones
Estructura	Estática	Dinámica
Tipos Primitivos	Usa tipos primitivos	Hay que utilizar las clases Integer, Float, Double
Longitud	nombres.length	nombres.size()
Obtener un valor	nombres[2]	nombres.get(2)
Establecer un valor	nombres[2] = "Carlos"	nombres.set(2, "Carlos")
Agregar un elemento	No se puede	nombres.add("Juan")
Quitar un elemento	No se pude	nombres.remove("Juan")
Acceso a una posición fuera de rango	Error Excepción	No arroja error
Ordenamiento	No posee	Posee

Tabla 1: Arrays vs Colecciones

Igualdad y ordenamiento en las colecciones

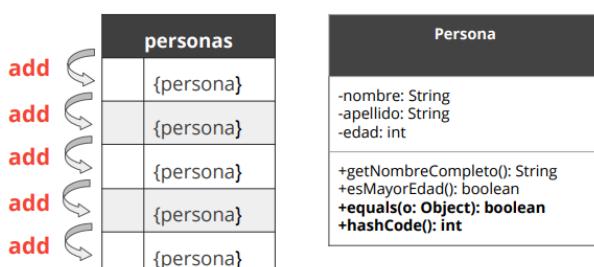
Elementos iguales

Las colecciones Set (HashSet, LinkedHashSet, TreeSet) no aceptan valores repetidos o sea iguales ni nulos, lo mismo sucede con las Key de las Map. Pero, ¿cómo hacemos, por ejemplo, en objetos de una clase Persona para determinar si una persona en la colección es igual a otra?

Debemos establecer el criterio por el cual una persona es igual a otra, si ese criterio no se cumple, diremos que son distintas.

En Java para determinar si dos objetos son iguales se deben sobreescribir los métodos equals() y hashCode().

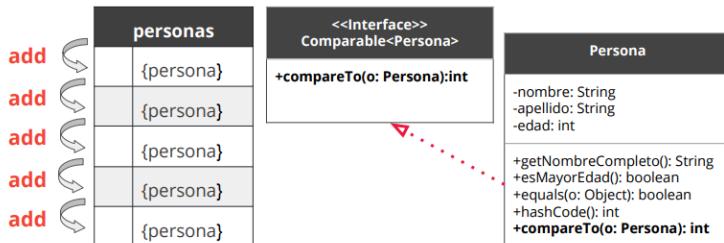
De esta manera, al sobreescribir los métodos equals y hashCode, las colecciones pueden determinar si el elemento que almacenan es igual a otro. En el caso de las Set servirán para no permitir su inserción.



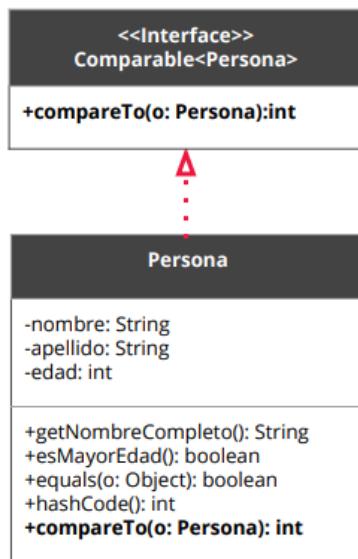
Orden entre elementos

En el caso de las colecciones TreeSet y TreeMap los elementos se almacenan en forma ordenada. No nos alcanza con determinar la igualdad, en este caso, además debemos compararlos, evaluando cuál es mayor, menor o igual a otro.

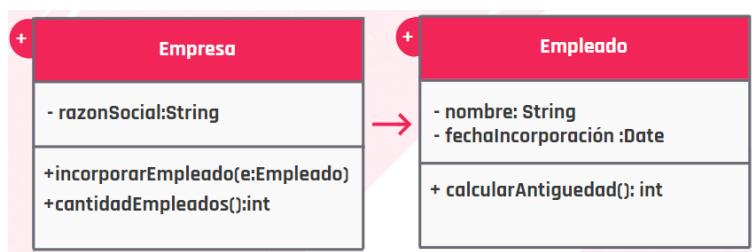
En el caso de las colecciones TreeSet y TreeMap los elementos se almacenan en forma ordenada y para ello debemos implementar la interface de Java Comparable.



Podemos utilizar la interface Comparable para ordenar los elementos de una List (ArrayList o LinkedList) invocando su método sort().



Relaciones 1 a muchos con colecciones



```

import java.util.*;

public class Empresa {
    private String razonSocial;
    private List<Empleado> empleados = new ArrayList<>();

    public void incorporarEmpleado(Empleado empleado) {
        empleados.add(empleado);
    }

    public int cantidadEmpleados() {
        return empleados.size();
    }

    public String getRazonSocial() {
        return razonSocial;
    }

    public void setRazonSocial(String razonSocial) {
        this.razonSocial = razonSocial;
    }
}

import java.util.*;

public class Empleado {
    private String nombre;
    private Date fechaIncorporacion;

    public int calcularAntiguedad() {
        Date fechaActual = new Date();
        return fechaActual.getYear() - fechaIncorporacion.getYear();
    }

    public void setFechaIncorporacion(Date fechaIncorporacion) {
        this.fechaIncorporacion = fechaIncorporacion;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}

```

Clase 17: Manejo de Excepciones

Introducción a excepciones

Cuando en nuestro código se produce un error por una situación excepcional, la forma de prevenirlo es usando excepciones. Para usar excepciones disponemos de los bloques try / catch. Lo que podemos hacer es literalmente lo que nos dicen:

- Intentar (lo que podría darnos problema)
- Atrapar (el problema)

Cuando usar excepciones

```

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int num1, num2, division;

    System.out.println("Primer número, debe ser un valor entero ");
    num1=scanner.nextInt();
    System.out.println(" Divisor, un valor entero ");
    num2=scanner.nextInt();
    division= num1/num2;
    System.out.println(division);
}

```

Si ejecutamos este código, aparentemente es correcto, pero si en el segundo número se ingresa 0, se generará una excepción.

```

Exception in thread "main" java.lang.ArithmetricException: / by zero
at com.company.Main.main(Main.java:16)

```

Solución con excepciones

Para mostrar el error, utilizamos `System.err.println`, esto hará que el mensaje salga en otro color

```

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int num1, num2, division;

    System.out.println("Primer número, debe ser un valor entero ");
    num1=scanner.nextInt();
    System.out.println(" Divisor, un valor entero ");
    num2=scanner.nextInt();
    try{
        division= num1/num2;
        System.out.println(division);
    } catch(ArithmetricException excepcion){
        System.err.println("Se intentó dividir por cero");
    }
}

```

```

try{
    division= num1/num2;
    System.out.println(division);
} catch(ArithmetricException excepcion){
    System.err.println("Se intentó dividir por cero");
}

```

En el bloque try están las instrucciones que podrían generar un problema, en este caso, la división (si el divisor es 0).

El bloque catch “atrapa” la excepción, si se intenta dividir por cero, entonces, se captura esa excepción y, en este caso, se muestra el mensaje. Si se efectúa una división que no tenga inconvenientes, entonces, el catch no actúa. ArithmetricException es el tipo de excepción que ocurrió, cuando ocurre, se crea un objeto en este caso exception.

Excepciones en detalle

```

System.out.println("Primer número, debe ser un valor entero ");
num1=scanner.nextInt();
System.out.println(" Divisor, un valor entero ");
num2=scanner.nextInt();
try{
    division= num1/num2;
    System.out.println(division);
} catch(ArithmetricException excepcion){
    System.err.println("Se intentó dividir por cero");
}

```

Estamos protegiendo el código de la división por cero, pero aún puede haber errores inesperados. Estamos solicitando el ingreso de números enteros, pero podrían ingresar valores con decimales. Para proteger el código vamos a modificarlo.

```

try{
    System.out.println("Primer número, debe ser un valor entero ");
    num1=scanner.nextInt();
    System.out.println(" Divisor, un valor entero ");
    num2=scanner.nextInt();
    division= num1/num2;
    System.out.println(division);
} catch(ArithmetricException excepcion){
    System.err.println("Se intentó dividir por cero");
}

```

Ahora estamos protegiendo el código, pero no analizamos el error. Si se ingresa un número con decimales, nos saldrá otra excepción.

```
Exception in thread "main" java.util.InputMismatchException
```

Diferentes excepciones

Los bloques try / catch nos permiten utilizar más de un catch. De esa forma podemos tratar excepciones específicas primero y luego las más generales. En el ejemplo anterior se pueden generar:

```
Exception in thread "main" java.lang.ArithmetricException: / by zero
```

```
Exception in thread "main" java.util.InputMismatchException
```

Entonces, vamos a adaptar el código para diferenciar la ocurrencia.

Diferenciando errores

El primer catch captura la excepción que ocurriría por un ingreso incorrecto y el segundo, la que ocurriría al intentar dividir por cero.

```
try{
    System.out.println("Primer número, debe ser un valor entero ");
    num1=scanner.nextInt();
    System.out.println(" Divisor, un valor entero ");
    num2=scanner.nextInt();
    division= num1/num2;
    System.out.println(division);

    catch(InputMismatchException excepcion){
        System.err.println("Se ingresó un tipo de dato incorrecto");
    }
    catch(ArithmetricException excepcion){
        System.err.println("Se intentó dividir por cero");
    }
}
```

El bloque finally

A los bloques try / catch se le puede agregar el bloque finally que es opcional, es decir, no es obligatorio utilizarlo. Si no ocurre una excepción y no entra al catch, se ejecuta el finally. Si ocurriera una excepción y el catch la “atrapa”, también se ejecuta el finally. El finally se ejecuta siempre.

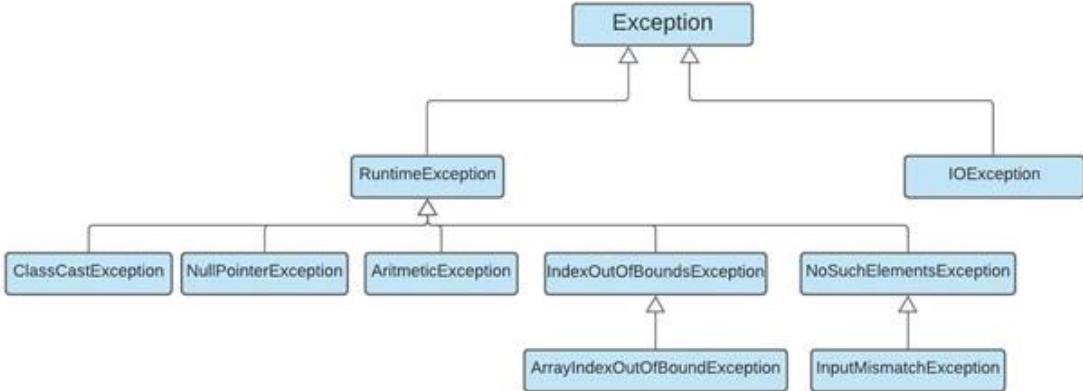
```
try{
    System.out.println("Primer número, debe ser un valor entero ");
    num1=scanner.nextInt();
    System.out.println(" Divisor, un valor entero ");
    num2=scanner.nextInt();
    division= num1/num2;
    System.out.println(division);

    catch(InputMismatchException excepcion){
        System.err.println("Se ingresó un tipo de dato incorrecto");
    }
    catch(ArithmetricException excepcion){
        System.err.println("Se intentó dividir por cero");
    }
    finally{
        System.out.println("Ha finalizado el ejemplo");
    }
}
```

Excepciones

Como todos los elementos en Java, las excepciones son clases, y tienen cierta jerarquía.

Todas las excepciones heredan de Exception, en el gráfico vemos solo algunas. Muchas de las RuntimeException son por errores cometidos al escribir el código. Las IOException son las que no dependen del código, por ejemplo, si en mi programa quiero abrir un archivo o guardar algo en un archivo y el archivo no está porque fue borrado o simplemente si está dañado y no se puede usar, ocurrirá este tipo de excepción, que no es responsabilidad directamente del programador, pero cuando programamos este tipo de acciones, debemos prever que pueden ocurrir. Para ello vamos a usar los bloques try / catch / finally.



Proteger la integridad de una clase

Cuando creamos una clase, estamos tratando de representar algo que tiene un cierto comportamiento. Los valores que se guardan en sus atributos pueden tener que respetar un rango de valores, en ese caso, tenemos que proteger la integridad de la clase, veamos un ejemplo.

Una fecha es algo bien conocido por todos, pero si la clase representa algo no tan habitual, quien tiene que utilizar la clase no tiene por qué saber con qué rango de valores se debe trabajar. A fines prácticos, vamos a establecer que los días pueden estar entre 1 y 31 sin importar el mes y los meses entre 1 y 12.

Fecha
- int day - int month - int year
+ Fecha(int d, int m, int y)

RuntimeException

Al usar la clase Fecha, para crear una nueva fecha, es necesario pasar 3 valores enteros. Esto se está cumpliendo en el ejemplo, pero quien usa una clase no necesariamente entiende el comportamiento de esa clase. La clase se debe proteger a sí misma y evitar que le lleguen valores que no estén en el rango esperado.

```

class Fecha{
    private int day;
    private int month;
    private int year;

    public Fecha(int d, int m, int y){
        if (d<1||d>31||m<1||m>12)
            throw new RuntimeException("Los valores no son válidos");
        day=d;
        month=m;
        year=y;
    }
}
  
```

Con throw lanzamos una excepción en ejecución, para hacerlo la creamos con el new la nueva excepción.

```

public static void main(String[] args) {
    Fecha fecha= new Fecha(100,-100,1000);
}

Exception in thread "main" java.lang.RuntimeException: Los valores no son válidos
  
```

Ahora, si intentamos crear con valores inválidos, nos genera una excepción. Las excepciones de tipo RuntimeException no es obligatorio protegerlas con los bloques try / catch.

Lanzar una excepción

Veamos ahora otra forma de proteger este código. Ahora vamos a hacerlo de forma que quien utilice el método esté forzado a protegerlo con try / catch.

```
class Fecha{  
    private int day;  
    private int month;  
    private int year;  
  
    public Fecha(int d, int m, int y) throws Exception{  
        if (d<1||d>31||m<1||m>12)  
            throw new Exception("Los valores no son válidos");  
        day=d;  
        month=m;  
        year=y;  
    }  
}
```

De la misma forma que antes lanzamos la excepción con throw, pero en este caso es de tipo Exception, ya que no hay definida una excepción que diga fuera del rango esperado. Se agrega que debemos avisar que el método puede generar una excepción, agregamos, a continuación de la firma throws Exception. Es un método **throwable**.

```
public static void main(String[] args) {  
  
    try{  
  
        Fecha fecha= new Fecha(100,-100,1000);  
        catch (Exception e) {  
            System.out.println("No son valores válidos para una fecha");  
        }  
    }  
}
```

Por ser un método throwable, nos obliga a protegerlo con try / catch.

Crear nuestras propias Excepciones

La clase que queremos proteger

Vamos a tomar nuevamente la clase fecha, pero queremos diferenciar el tipo de error que ocurrió. Tenemos dos posibles errores, que el día esté fuera de rango o que el mes esté fuera de rango. Por fines prácticos tomamos días válidos de 1 a 31.

Extender Exception

Extendemos Exception y creamos dos constructores: uno por defecto que no tiene parámetros y el otro con parámetros y sobrescribimos el `toString()`. En el constructor con parámetros puedo recibir un mensaje que es el que luego me va a mostrar el error en detalle.

```
public class FechaException extends Exception{  
  
    public FechaException(){  
        super();  
    }  
    public FechaException(String mensaje){  
        super(mensaje);  
    }  
    public String toString(){  
        return "Se produjo la siguiente Excepción " + this.getClass().getName() + "\n" +  
               " Mensaje: " + this.getMessage() + "\n";  
    }  
}
```

En este ejemplo extendemos de Exception, pero se puede extender de cualquier Excepción definida en el API de Java. Siempre es conveniente utilizar la más relacionada con la condición que se quiere proteger.

Usar nuestras propias excepciones

```
class Fecha{  
    private int day;  
    private int month;  
    private int year;  
  
    public Fecha(int d, int m, int y){  
        day=d;  
        month=m;  
        year=y;  
    }  
  
    public static void main(String[] args) {  
  
        Fecha fecha= new Fecha(100,-100,1000);  
    }  
}
```

Ilustración 1: La clase fecha sin proteger la integridad de los datos

```
class Fecha{  
    private int day;  
    private int month;  
    private int year;  
  
    public Fecha(int d, int m, int y) throws FechaException{  
        if (d<1||d>31)  
            throw new FechaException("Error en el dia");  
        day=d;  
        if (m<1||m>12)  
            throw new FechaException("Error en el mes");  
        month=m;  
        year=y;  
    }  
}
```

El método puede lanzar una excepción de tipo FechaException. Si el día está fuera de rango, se lanza la excepción con el mensaje que informa error en el día. Si el mes está fuera de rango, se lanza la excepción con el mensaje que informa error en el mes. Si se genera la excepción, el código no se sigue ejecutando.

```
public static void main(String[] args) {  
  
    try{  
        Fecha fecha= new Fecha(-1,10,2000);  
    catch(FechaException excepcion){  
        System.err.println(exception.getMessage());  
    }  
}  
  
Error en el dia
```

Obtendremos el mensaje que programamos en nuestra clase, si cometemos error en el mes, el mensaje lo indicaría, ya que generamos la excepción indicando qué error ocurrió.

Clase 19: Introducción a patrones de diseño

Módulo 3: Patrones de diseño

Patrones

Los patrones de diseño son sugerencias de uso frente a una eventual aplicación. Sirven para brindar soluciones rápidas a problemas simples y recurrentes permitiendo agilizar el desarrollo de un diseño.

Estos patrones fueron definidos en un famoso libro del 2003 llamado “Patrones de Diseño” y se dividen en tres partes.



Patrones Creacionales

Tienen por objetivo abstraer el proceso de cómo los objetos son creados en una aplicación. Estos nos proporcionan interfaces para crear y copiar objetos y producir familias de objetos relacionados sin tener que especificar sus clases concretas. Además, nos permiten producir diferentes tipos de representaciones usando el mismo código.

Patrones Estructurales

Lidian con la composición de una clase y objeto. Estos patrones habilitan la colaboración de los objetos con interfaces incompatibles, agregando nuevos comportamientos a objetos. De este modo, podemos dividir una clase o un conjunto de clases estrechamente vinculadas en dos jerarquías separadas, abstracción e implementación, que pueden ser desarrolladas independientemente una de la otra.

Otra función que nos ayuda es la posibilidad de componer objetos en una estructura de árbol y luego trabajar con esas estructuras como si fueran objetos individuales.

Patrones de Comportamiento

Se encargan de las relaciones entre objetos y clases y las distribuciones de responsabilidades en una aplicación. Algunos patrones de comportamiento se basan en utilizar la herencia para distribuir el comportamiento entre las clases, mientras que otros se basan en utilizar la composición para distribuir ese comportamiento.

Es muy importante mantener un modelo balanceado y estos patrones nos pueden ayudar en ese sentido. Un buen diseño es primordial para el desarrollo de software.

Concepto de Patrones de Diseño

Prácticamente en todas las áreas se desarrollan estándares para realizar algún procedimiento. En informática no es diferente. Como su nombre indica, utilizar un patrón de diseño es correspondiente a utilizar una **estructura de programación ya conocida** y consolidada en el mercado, cuya función está previamente definida.

Un patrón de diseño, también conocido por el término original en inglés design pattern, describe una **solución general reutilizable** para un problema recurrente en el desarrollo de sistemas de software orientados a objetos. No es un código prefabricado, una versión final, sino un modelo de cómo resolver un determinado problema. Los patrones de diseño definen las relaciones e interacciones entre clases u objetos, sin especificar los detalles de los involucrados.

Conceptualmente, un patrón de diseño debe definir: un nombre, el problema, la solución, cuándo aplicar esa solución y las consecuencias de hacerlo.

Composición y Herencia

De manera general, una definición para composición y herencia sería que son dos mecanismos para reutilizar la funcionalidad, es decir, no ser repetitivo ni escribir código innecesario.

Herencia

La herencia siempre se ha considerado una herramienta básica para extender y reutilizar la funcionalidad basada en los atributos y métodos de una clase. Por ejemplo:

```
public class Animal{  
}  
public class Cachorro extends Animal{  
}  
public class Gato extends Animal{  
}
```

Ventajas	Desventajas
Captura lo común y lo aísla de lo diferente. La herencia se ve directamente en el código, incluso debido a su naturaleza estática.	Encapsulación débil y acoplamiento estrecho, donde el cambio de una superclase puede afectar a todas las subclases.
Permite crear una estructura jerárquica de clases cada vez más especializada. Por lo tanto, no hay que empezar desde cero para especializar una clase existente.	A veces un objeto debe ser de una clase diferente en diferentes momentos, lo que no es posible con la herencia, ya que tiene una relación estática.

Tabla 2: Ventajas y Desventajas de la Herencia

Composición

En la composición, en lugar de codificar un comportamiento de forma estática, como se hace con la herencia, definimos pequeños comportamientos predeterminados y la usamos para declarar comportamientos más complejos. Por ejemplo:

```
public class Sistema {  
  
    Persona persona = new Persona();  
  
}  
public class Persona{  
}
```

Ventajas	Desventajas
El comportamiento se puede elegir en tiempo de ejecución en lugar de estar vinculado en tiempo de compilación.	El software es muy dinámico y parametrizado y es más difícil de entender que el software más estático.
Los objetos que fueron instanciados y que están contenidos en la clase que los instancia, se accede sólo a través de su interfaz, siguiendo así el principio de programación para una interfaz y no para una implementación.	

Tabla 3: Ventajas y Desventajas de la Composición

¿Cuándo usarlos?

En general, siempre se prefiere usar la composición sobre la herencia, sin embargo, podemos definir algunas reglas para identificar cuándo podemos usar la herencia para que no tengamos los problemas que conlleva.

1. La herencia se usa si una instancia de una clase Hija nunca necesitará convertirse en un objeto de otra clase.
2. Si la jerarquía de herencia representa una relación "Es una" y no una relación "Tiene una"
3. Si desea o necesita realizar cambios globales en sus clases secundarias cambiando una clase principal.
4. Cuando la clase secundaria se extiende en lugar de reemplazar total o parcialmente las responsabilidades de la clase principal.

Singleton

Singleton es un patrón de diseño de creación que garantiza que una clase tenga una sola instancia y define un punto de acceso global para ella. En el patrón Singleton, una clase administra su propia instancia y evita que cualquier otra clase cree una instancia de ella.

Para crear la instancia usando el patrón Singleton, debe pasar obligatoriamente por la clase, ninguna otra clase puede instanciarla. El patrón Singleton también proporciona un punto de acceso global a su instancia.

La clase en sí siempre ofrecerá su propia instancia y, si aún no tiene una, crea y devuelve esta instancia recién creada.



Para crear una clase con el patrón Singleton es necesario realizar los siguientes pasos:

1. Crear un atributo estático del mismo tipo que la clase con el nombre de instancia.
2. Todos los constructores de la clase deben usar el modificador private.
3. Crear un método `getInstance()` estático que devuelva el atributo de instancia.



```

public class SingletonEjemplo {
    //Atributo con mismo nombre de la clase
    private static SingletonEjemplo instance = new
    SingletonEjemplo();

    //Constructores privados
    private SingletonEjemplo(){
    }

    //Método getInstance() estático
    public static SingletonEjemplo getInstance(){

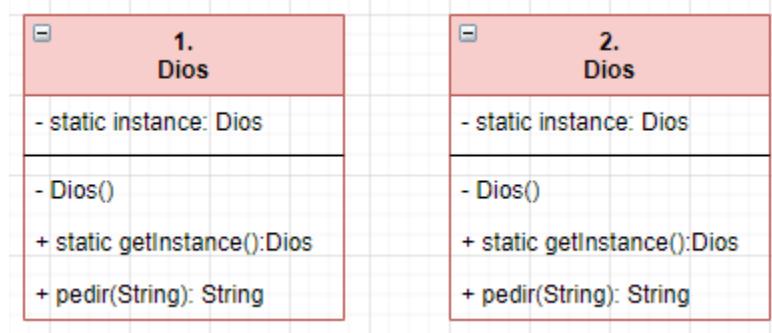
        return instance;
    }
}

```

Código de la clase
SingletonEjemplo

El patrón Singleton se usa cuando necesita un solo punto para crear una instancia de clase y cuando solo necesita una instancia de una clase.

Dos implementaciones de singleton



```

public class Dios{
    private static Dios instancia;

    private Dios(){
    }

    /*Esta técnica se llama inicialización tardía
    hasta que no se invoque al método getInstance
    no se crea ningún objeto en memoria.*/

    public static Dios getInstance(){
        if(instancia == null)
            instancia = new Dios();
        return instancia;
    }

    public String pedir(String pedido){
        return "tu pedido fue escuchado: " + pedido;
    }
}

```

```

public class Dios{
    private static Dios instancia = new Dios();

    private Dios(){}

    public static Dios getInstance(){
        return instancia;
    }

    public String pedir(String pedido){
        return "tu pedido fue escuchado: " + pedido;
    }
}

```

Ilustración 3: 2. Dios

Ilustración 2: 1. Dios

Patrón factory

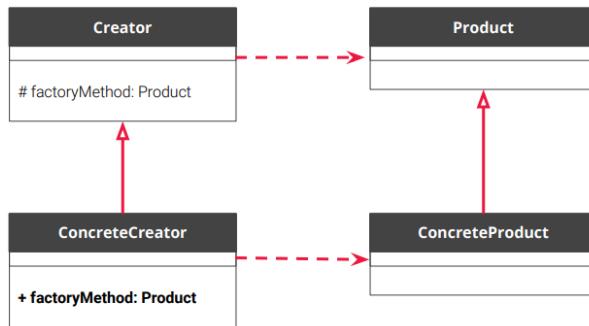
El patrón de diseño Factory es uno de los principales patrones de diseño y uno de los más utilizados en la mayoría de los lenguajes de programación en la actualidad.

Tiene dos variaciones:

- Factory Method
- Abstract Factory

Factory Method

El patrón Factory Method define una interfaz para crear un objeto, pero permite que las subclases decidan qué clase instanciar. Estas fábricas de construcción minimizan el uso de la palabra clave "new", encapsulan el proceso de inicialización y diferentes implementaciones concretas. Además, esta centralización minimiza el efecto de agregar y eliminar clases concretas en el sistema y los efectos de las dependencias de clases concretas.

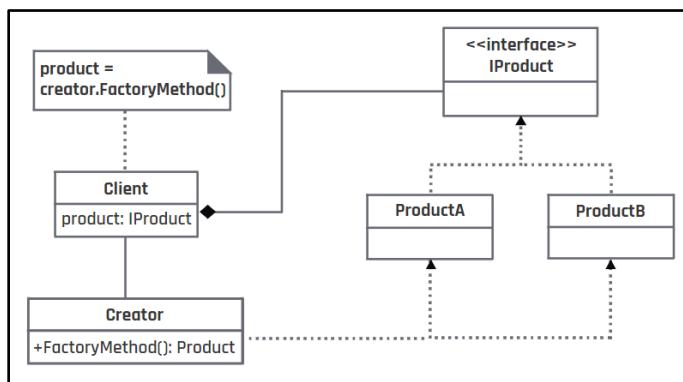


Creator (Creador abstracto): declara el Factory Method (método de fabricación) que retorna el método de la clase Producto. Este elemento también puede definir una implementación base que devuelve un objeto de la clase ConcreteProduct.

ConcreteCreator (Creador concreto): sobrescribe el método que fabrica el Product y nos permite instanciar el ConcreteProduct sin hacer referencia directa al mismo.

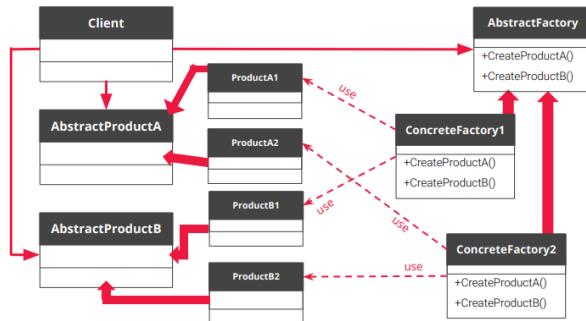
Product (Producto abstracto): define una interfaz para los objetos creados por el Factory method.

ConcreteProduct (Producto concreto): representa una implementación para la interfaz del producto.



Abstract Factory Method

Es un patrón que proporciona una interfaz para crear familias de objetos dependientes o relacionados sin especificar sus clases concretas. Por lo tanto, Abstract Factory ofrece encapsulación de un grupo de fábricas y controla cómo el cliente accede a estas fábricas.

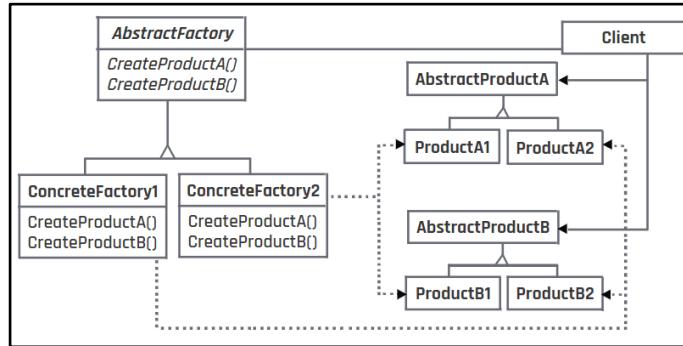


AbstractFactory: Su propósito es declarar métodos de creación de tipo AbstractProduct, que son implementados por una clase de tipo ConcreteFactory, que hereda o implementa a AbstractFactory.

AbstractProduct: Declara métodos implementados por clases de tipo ConcreteProduct. ConcreteFactory crea internamente un objeto de tipo ConcreteProduct, pero este objeto se devuelve como AbstractProduct.

ConcreteFactory: Implementa los métodos declarados en AbstractFactory, creando un objeto de tipo ConcreteProduct y devolviéndolo como AbstractProduct.

ConcreteProduct: Es la clase que especifica la instancia correcta a crear. Implementa los métodos declarados en AbstractProduct.



Conclusión

El propósito del patrón Factory es crear objetos, por lo que se considera un patrón de creación.

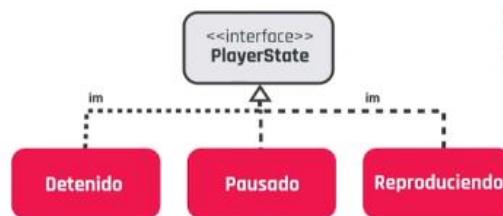
Básicamente, la lógica de creación está encapsulada dentro de la fábrica (FactoryMethod) y se proporciona un método que devuelve un objeto (Método Factory predeterminado) o la creación del objeto se delega a una subclase (método Abstract Factory predeterminado).

Clase 20: Patrón State

Patrón State

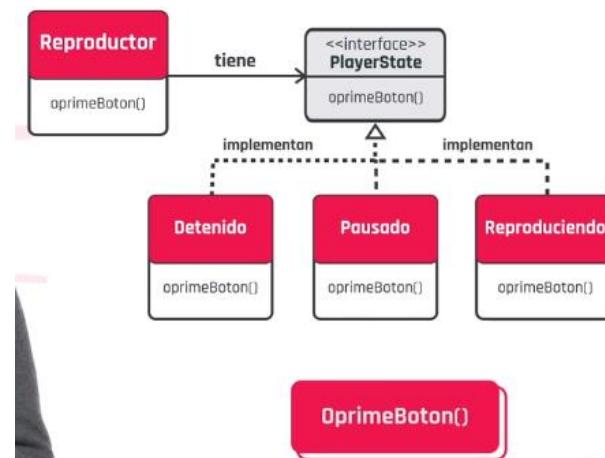
Un reproductor de música puede tener diferentes estados: detenido, reproduciendo, pausado, entre otros. Cuando se oprime el botón de reproducción pueden pasar diferentes cosas. Si comenzamos con el reproductor detenido va a escucharse una canción, en cambio, si ya estábamos escuchando una canción se detiene. Para que esto suceda debemos tener una clase que represente cada estado y, por supuesto, todas deben tener la posibilidad de responder a los mismos métodos. Para esto vamos a

implementar una interfaz que contiene estos métodos la cual llamaremos PlayerState, de este modo las clases implementan esta interfaz.



Ahora vemos que nuestro reproductor tiene una variable llamada estado que va a ser del tipo PlayerState, es decir, siempre nuestro reproductor va a tener un estado. Si cambio el valor de la variable estado cambio el estado de mi reproductor.

Cuando el reproductor recibe un OprimeBotón() le pide que su variable de estado haga oprime botón entonces haría cosas distintas de acuerdo a que tiene esta variable . Cuando ese estado contiene un objeto de la clase pausado y el reproductor recibe OprimeBotón() este ejecuta el método de pausado y cambia el contenido de la variable para que ahora contenga un objeto de estado reproduciendo, cambiando así completamente el estado de nuestro reproductor.



Lo mejor de todo es que podemos usar este patrón siempre que tengamos que **cambiar el comportamiento de un objeto de acuerdo a un estado** así cada estado será una clase y todo se implementará la interfaz que define Cuáles son las acciones que pueden depender del estado.

Motivación

Cuando se requiere que un objeto tenga diferentes comportamientos según el estado en que se encuentra, resulta complicado poder manejar el cambio de comportamientos y los estados de dicho objeto. El patrón State propone una buena solución a esta complicación, creando básicamente un **objeto por cada estado posible del objeto que lo invoca**.

Se implementa una clase para cada estado diferente del objeto y cada clase implementará los métodos cuyo comportamiento varía según ese estado. Así, siempre se tendrá una referencia a un estado concreto y se comunicará con este para resolver sus responsabilidades.

Ventajas y desventajas

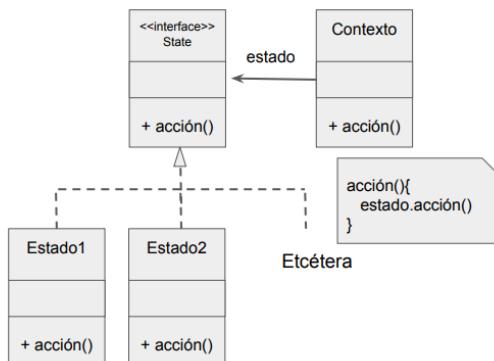
VENTAJAS

- Se localizan fácilmente las responsabilidades de los estados específicos ya que se encuentran en las clases que corresponden a cada estado. Esto brinda una mayor claridad en el desarrollo y el mantenimiento posterior.
- Hace los cambios de estado explícitos al estar representado cada estado en una clase.
- Facilita la ampliación de estados.
- Permite a un objeto cambiar de clase en tiempo de ejecución dado que al cambiar sus responsabilidades por las de otro objeto de otra clase, la herencia y responsabilidades del primero han cambiado por las del segundo.

DESVENTAJA

- Se incrementa el número de subclases.

Diagrama UML



- **Clase contexto:** define la interfaz con el cliente. La instancia de contexto es la que define su estado actual.
- **Interface State (estado):** interface para el encapsulamiento de las responsabilidades asociadas con un estado particular de contexto. Define las responsabilidades de cada estado.
- **Clase estado:** cada una implementa el comportamiento o responsabilidad de contexto.

¿Cómo funciona?

La clase contexto envía mensajes al objeto dentro de su código que contiene una instancia de estado para brindarle a estos la responsabilidad que debe cumplir el objeto contexto. Así, el objeto contexto va cambiando las responsabilidades según el estado en que se encuentre, puesto que también cambia instancia de estado al hacer un cambio de estado.

Dicho en pocas palabras: Contexto le dice a la instancia de estado que haga la acción... Pero cuando cambia la instancia de la clase estado (Estado1, Estado2, etc) la acción se realiza de forma diferente según este.

Conclusiones

El patrón no indica exactamente dónde definir las transiciones de un estado a otro. Existen dos formas de solucionar esto:

1. Definiendo estas transiciones dentro de la clase contexto.
2. Definiendo estas transiciones en las subclases de State.

Es más conveniente utilizar la primera solución cuando el criterio a aplicar es fijo, es decir, no se modificará. En cambio, la segunda resulta conveniente cuando este criterio es dinámico. Este se presenta en la dependencia de código entre las subclases.

También hay que evaluar en la implementación cuándo crear instancias de estado concreto distintas o utilizar la misma instancia compartida. Esto dependerá si el cambio de estado es menos o más frecuente respectivamente.

Patrón State: ejemplo de un modelo de Auto

Tenemos un auto que puede sencillamente tener varios estados.

Puede estar:

- **Apagado**: sin la llave de contacto y el motor detenido.
- **Parado**: el motor en marcha, pero sin moverse.
- **EnMarcha**: avanzando en el camino.
- **SinNafta**: se le terminó el combustible.

El auto deberá tener atributos para conocer su velocidad y la cantidad de combustible que tiene cargado. Las responsabilidades podrán ser:

- **Acelerar**: incrementa la velocidad y consume nafta.
- **Contacto**: sirve para encender el auto.
- **Frenar**: le permite detenerse.

Las transiciones de estado se encuentran, cuando, por ejemplo:

Situación	Nuevo Estado
El auto está apagado y ejecuto contacto	Parado
El auto se le termina el combustible	Sin Nafta
El auto está parado y ejecuto acelera	En Marcha

Tabla 4: Transiciones

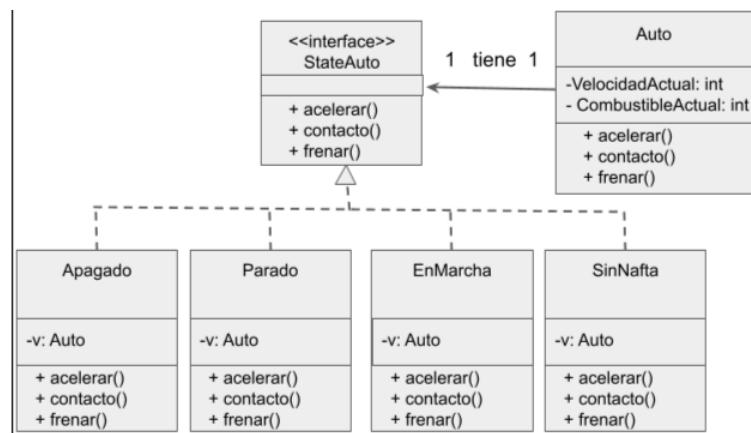
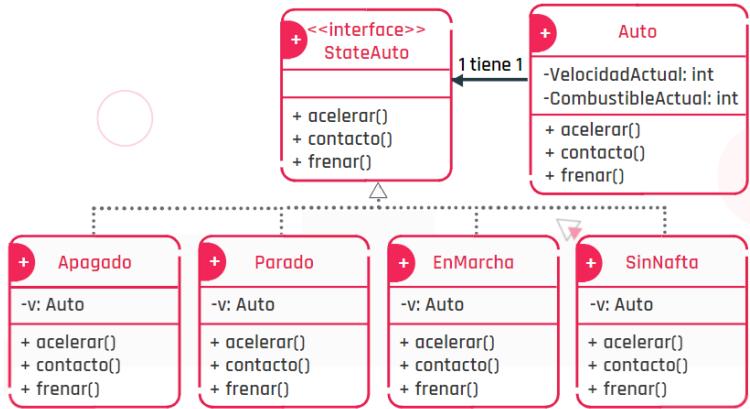


Ilustración 4: Diagrama UML

Implementación Típica en Java



Interfaz StateAuto

```

public interface StateAuto {
    void acelerar();
    void frenar();
    void contacto();
}
  
```

Clase Auto

```

public class Auto {
    private StateAuto estado;
    // Estado del auto (apagado, parado, en marcha, sin combustible)
    private int velocidadActual = 0; // Velocidad actual del auto
    private int combustibleActual = 0; // combustible restante

    public Auto(int combustible) {
        this.setCombustibleActual(combustible);
        // Estado inicial (Apagado)
        this.setEstado(new Apagado(this));
    }
    public void acelerar() {
        getEstado().acelerar();
        System.out.println("Velocidad actual: " + getVelocidadActual() + ". Combustible restante: " +
getCombustibleActual());
    }
    public void frenar() {
        getEstado().frenar();
    }
    public void contacto() {
        getEstado().contacto();
    }
    public void modificarVelocidad(int kmh) {
        setVelocidadActual(getVelocidadActual() + kmh);
    }
    public void modificarCombustible(int decilitros) {
        setCombustibleActual(getCombustibleActual() + decilitros);
    }
    public int getVelocidadActual() {
        return velocidadActual;
    }
    public void setVelocidadActual(int velocidadActual) {
        this.velocidadActual = velocidadActual;
    }
    public int getCombustibleActual() {
        return combustibleActual;
    }
    public void setCombustibleActual(int combustibleActual) {
        this.combustibleActual = combustibleActual;
    }
    public StateAuto getEstado() {
        return estado;
    }
    public void setEstado(StateAuto estado) {
        this.estado = estado;
    }
}
  
```

Clase Apagado

```
public class Apagado implements StateAuto{

    // Referencia a la clase de contexto
    private Auto v;
    // Constructor que inyecta la dependencia en la clase actual
    public Apagado(Auto v)
    {
        this.v = v;
    }

    @Override
    public void acelerar() {
        System.out.println("ERROR: El auto esta apagado. Efectue el contacto para iniciar");
    }

    @Override
    public void frenar() {
        // Frenar con el auto parado tampoco sirve de mucho...
        System.out.println("ERROR: El auto esta apagado. Efectue el contacto para iniciar");
    }

    @Override
    public void contacto() {
        // Comprobamos que el auto disponga de combustible
        if (v.getCombustibleActual() > 0)
        {
            // El auto arranca -> Cambio de estado
            //           estado = PARADO;
            v.setEstado(new Parado(v));
            System.out.println("El auto se encuentra ahora PARADO");
            v.setVelocidadActual(0);
        }
        else
        {
            // El auto no arranca -> Sin combustible
            //estado = SIN COMBUSTIBLE
            v.setEstado(new SinNafta(v));
            System.out.println("El auto se encuentra sin combustible");
        }
    }
}
```

Clase Parado

```
public class Parado implements StateAuto{
    private Auto v;

    // Constructor que inyecta la dependencia
    public Parado(Auto v) {
        this.v = v;
    }

    @Override
    public void frenar() {
        // No ocurre nada. Si ya se encuentra detenido, no habrá efecto alguno
        System.out.println("ERROR: El auto ya se encuentra detenido");
    }

    @Override
    public void contacto()
    {
        // El auto se apaga           estado = APAGADO;
        v.setEstado(new Apagado(v));
        System.out.println("El auto se encuentra ahora APAGADO");
    }

    @Override
    public void acelerar() {
        // Comprobamos que el auto disponga de combustible
        if (v.getCombustibleActual() > 0)
        {
            // El auto se pone en marcha. Aumenta la velocidad y cambiamos de estado
=> EN_MARCHA;
            v.setEstado(new EnMarcha(v));
            System.out.println("El auto se encuentra ahora EN MARCHA");
            v.modificarVelocidad(10);
            v.modificarCombustible(-10);
        }
    }
}
```

```

        }
    else
    {
        //estado = SIN COMBUSTIBLE
        v.setEstado(new SinNafta(v));
        System.out.println("El auto se encuentra ahora SIN COMBUSTIBLE");
    }
}

}

```

Clase EnMarcha

```

public class EnMarcha implements StateAuto{
    private final int VELOCIDAD_MAXIMA = 200;

    // Referencia a la clase de contexto
    private Auto v;

    // Constructor que inyecta la dependencia en la clase actual
    public EnMarcha(Auto v) {
        this.v = v;
    }

    @Override
    public void acelerar() {
        if (v.getCombustibleActual() > 0)
            {
                // Aumentamos la velocidad, en el mismo estado
                if (v.getVelocidadActual() >= VELOCIDAD_MAXIMA)
                    {
                        System.out.println("ERROR: El auto alcanzó su velocidad máxima");
                        v.modificarCombustible(-10);
                    }
                else
                {
                    v.modificarVelocidad(10);
                    v.modificarCombustible(-10);
                }
            }
        else
        {
            //estado = SIN COMBUSTIBLE
            v.setEstado(new SinNafta(v));
            System.out.println("El auto se quedó sin combustible");
        }
    }

    @Override
    public void frenar() {
        // Reducimos la velocidad. Si esta llega a 0, cambiaremos a estado "PARADO"
        v.modificarVelocidad(-10);
        if (v.getVelocidadActual() <= 0)
        {
            //estado = PARADO;
            v.setEstado(new Parado(v));
            System.out.println("El auto se encuentra ahora PARADO");
        }
    }

    @Override
    public void contacto() {
        // No se puede detener el auto en marcha!
        System.out.println("ERROR: No se puede cortar el contacto en marcha!");
    }
}

```

Clase SinNafta

```

public class SinNafta implements StateAuto{
    // Referencia a la clase de contexto
    private Auto v;

    // Constructor que inyecta la dependencia en la clase actual
    public SinNafta(Auto v)
    {
        this.v = v;
    }

    @Override
    public void acelerar() {
        System.out.println("ERROR: El auto se encuentra sin combustible");
    }
}

```

```

@Override
public void frenar() {
    System.out.println("ERROR: El auto se encuentra sin combustible");
}

```

Clase 22: Patrón Composite

Patrón Composite

El Patrón Composite es un patrón de la categoría estructural que se enfoca en la manera en que los objetos están compuestos para formar estructuras más complejas aún. El objetivo este patrón es componer objetos en **estructuras de árbol para representar jerarquías**. Composite permite tratar de manera uniforme objetos individuales y composiciones de objetos.

Características

Es de mayor utilidad empleado en estructuras que pueden ser tratadas jerárquicamente, el ejemplo clásico es la estructura de un árbol. También será una solución para estructuras complejas que pueden ser tratadas de manera uniforme. Además, **prioriza la composición sobre la herencia**.

¿CUÁNDO USAR COMPOSITE? Cuando la estructura de objetos puede ser representada jerárquicamente o cuando queremos que el código cliente trate los objetos compuestos y los objetos simples de la misma.

Ventajas	Desventajas
Es más fácil crear objetos por composición	Dependiendo de la estructura, puede romper el principio de segregación de la interfaz
Resulta sencillo: <ul style="list-style-type: none"> • Crear una jerarquía de objetos • Usar polimorfismo y recursión • Agregar nuevos tipos de elementos a la estructura 	Los objetos del tipo “Leaf” tienden a tener métodos que no utilizan

Tabla 5: Ventajas y Desventajas de Patrón Composite

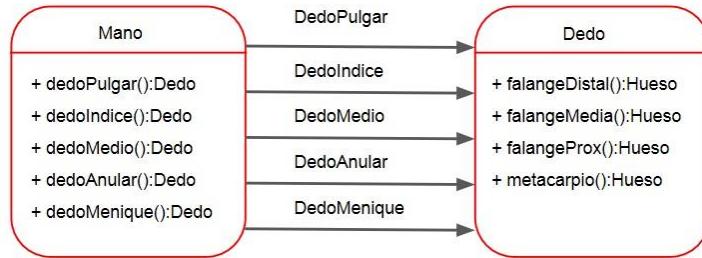
Presentación en UML

Vamos entender la relación de composición entre clases a través de una analogía con el cuerpo humano: ¿Qué es el cuerpo humano? Es un sistema.

Como si fuese un software, podemos interpretarlo compuesto por módulos, dominios, funcionalidades, requisitos funcionales, requisitos no funcionales y reglas de negocio.

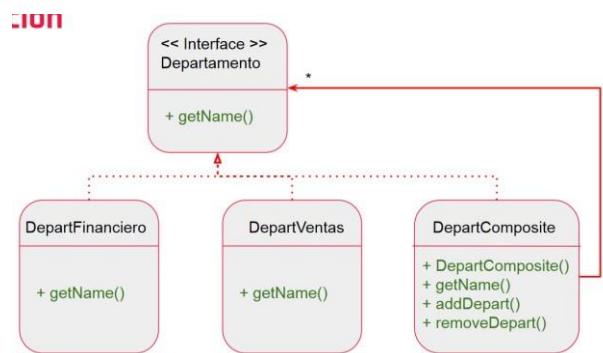
Desde el punto de vista estructural, también como un software, podemos interpretarlo como un espacio de nombres o paquetes, compuesto de clases, clases compuestas por otras clases, todas con sus métodos, etc.

Una mano está compuesta por dedos. Podemos entender a la mano como una clase, parte del brazo, y la clase mano posee cinco composiciones de la clase dedo. Una mano está compuesta por cinco dedos: pulgar, índice, medio, anular, meñique.



Ejemplo

Ahora, vamos a imaginar una implementación. Supongamos que deseamos construir una estructura jerárquica de departamentos en una empresa. Veamos el diagrama UML a seguir



Implementación de las clases del diagrama UML

```

public interface Departamento {
    void getName();
}

```

Código de la interfaz Departamento

Para los componentes hoja vamos a definir Clases para los departamentos financiero y de ventas:

```

public class DepartFinanciero implements Departamento {
    {
        private int id;
        private String name;
        public void getName() {
            System.out.println(getClass().getSimpleName());
        }
    }
}

```

Código de la clase DepartFinanciero

```

public class DepartVentas implements Departamento {
    {
        private int id;
        private String name;
        public void getName() {
            System.out.println(getClass().getSimpleName());
        }
    }
}

```

Código de la clase DepartVentas

```

public class DepartComposite implements Departamento {
    private int id;
    private String name;

    private List<Departamento> childDepartments;

    public DepartComposite(int id, String name) {
        this.id = id;
        this.name = name;
        this.childDepartments = new ArrayList<>();
    }

    public void getName() {
        childDepartments.forEach(Departamento::getName);
    }

    public void addDepart(Departamento department) {
        childDepartments.add(department);
    }

    public void removeDepart(Departamento department) {
        childDepartments.remove(department);
    }
}

```

Código de la clase DepartComposite

Esta es una clase compuesta que contiene una colección de componentes de clase Departamento, y también métodos para adicionar o remover elementos de esta lista.

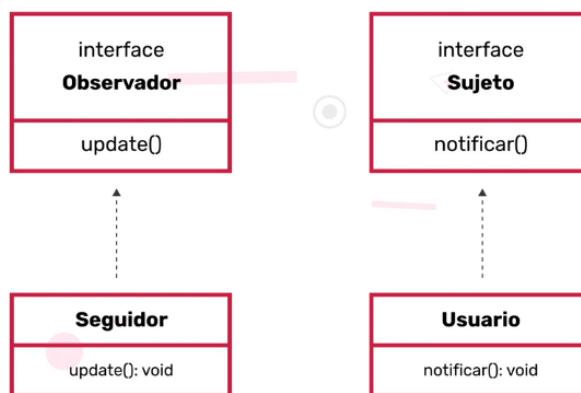
El método compuesto getName() está implementado iterando sobre la lista de elementos hoja e invocando el método apropiado para cada uno.

Clase 25: Patrón Observer

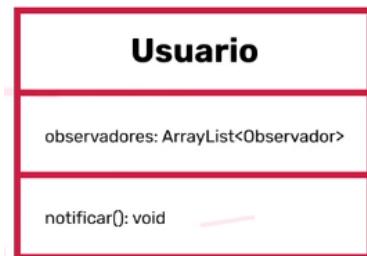
Introducción a Patrón Observer

En las redes sociales podemos tener amistades o seguidores que reciben constantemente actualizaciones de nuestra actividad. Cuando hacemos una publicación, damos un like o compartimos información, es común que cada uno de esos reciba la misma notificación. Si llevamos esto al patrón observer, cada amigo o seguidor es un observador y nosotros somos el sujeto también conocido como observable por lo que, se establece implícitamente una relación de uno a muchos.

Para implementar el patrón observer debemos tener **dos interfaces**: la primera, **observador** que va a tener un **método update()** que se va actualizar cuando el sujeto lo notifique o dispare un evento. La segunda interfaz es **sujeto** que va a tener el **método notificar()** que va a avisar algún cambio o evento. Ahora necesitamos clases concretas que implementen las interfaces que acabamos de crear, para eso usaremos el ejemplo anterior en el cual, usuario es el sujeto y los seguidores son observadores.



Como habíamos comentado anteriormente el patrón observer establece una relación de uno a muchos es por eso que cada vez que el usuario tenga un nuevo seguidor esta lo almacenará en un ArrayList de tipo observador llamado observadores.



Hasta acá se puede deducir que cada usuario de esa red social tiene un `ArrayList` de observadores donde se va agregando uno a uno cada seguidor nuevo. Cada seguidor tiene que declarar el método `update()` de la interfaz `Observador` añadiendo la lógica que le corresponda.

```

1 @Override
2 public void update()
3 {
4     System.out.println("¡Nueva notificación!");
5 }
6

```

El **método `notificar()`** de la interfaz sujeto es dónde sucede **gran parte de la magia del patrón observer**. Este contiene un `for each` que va a recorrer el `ArrayList` `observadores` notificando así a cada seguidor sobre el cambio realizado haciendo uso del método `update()`.

```

1 @Override
2 public void notificar() {
3     for(Observador o : observadores)
4         o.update();
5 }
6

```

Esta lógica del método `notificar` se asemeja mucho a cuando reenviamos un mensaje a todos nuestros contactos en WhatsApp al mismo tiempo.

Podemos decir que este patrón lo podemos usar cuando necesitamos que un objeto tenga que notificar a muchos otros objetos cuando se genera algún cambio específico.

El patrón observer define una dependencia de uno a muchos entre objetos

Propósito

Un determinado objeto puede tener otros dependientes de este. Estos otros objetos tal vez necesiten actualizarse según cambio de estado en el objeto del que dependen. Al intentar implementar esta lógica surgen muchas dificultades.

El patrón Observer propone una solución creando una interfaz que, cuando un objeto cambie de estado, se notifique y se actualicen automáticamente todos los objetos que dependen de él.

Se dispone de una interfaz para el **SujetoObservable** (`Observable`) y una para los **Observadores** (`Observador`). La clase concreta que será observada implementa la interface `Observable` y los observadores concretos implementan `Observador`. Estas dos interfaces tienen un método que deben

declarar las clases concretas, y por medio de estos métodos es que se actualizarán los **observadores** con cada cambio de estado en el **sujetoObservable**.

Así, siempre se obtendrán las actualizaciones de estado independientemente del tipo de objeto que sean tanto los observadores como el sujeto observable.

Ventajas y Desventajas

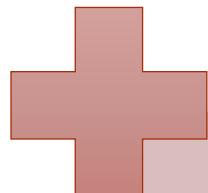
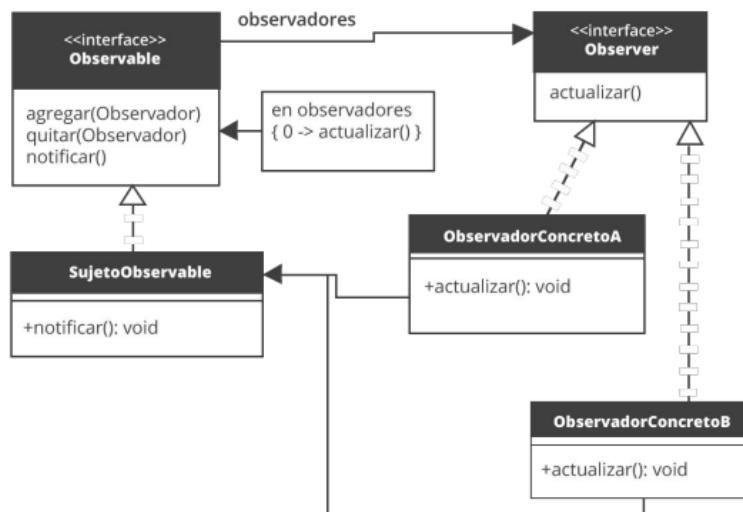
	
<p>Ventajas</p> <ul style="list-style-type: none"> • Permite modificar los sujetos y observadores de forma independiente. Es posible reutilizar objetos sin volver a utilizar sus observadores o viceversa. Esto añade escalabilidad al permitir añadir observadores sin modificar el sujeto u otros observadores. • A diferencia de una petición ordinaria, la notificación enviada por un sujeto no necesita especificar su receptor. Esto genera una difusión entre todos los observadores interesados que haya. • Gracias a que sujeto y observador no están fuertemente acoplados, pueden pertenecer a diferentes capas de abstracción de un sistema. 	<p>Desventajas</p> <ul style="list-style-type: none"> • Una actualización aparentemente inofensiva sobre el sujeto puede generar una serie de actualizaciones en cascada de los observadores y sus objetos dependientes. Esto puede generar falsas actualizaciones muy difíciles de localizar.

Diagrama UML



- **Interfaz observable (Sujeto):** cada implementación (sujeto) conoce a sus observadores y puede ser observado por cualquier número de observadores. También proporciona una interfaz para agregar o quitar observadores.
- **Observador:** define una interfaz para que cada implementación (ObservadorConcreto) pueda actualizar los objetos que deben ser notificados de cambios en el sujeto.

- **SujetoConcreto**: envía una notificación a sus observadores cuando cambia su estado.
- **ObservadorConcreto**: mantiene referencia a un objeto SujetoConcreto. Guarda un estado que debería ser consistente con el del Sujeto. Implementa la interfaz de actualización del Observador para mantener su estado sincronizado con el del sujeto.

¿CÓMO FUNCIONA? Cuando el SujetoObservable sufre algún cambio de estado, se ejecuta el método “notificar()”, el cual recorre una lista que contiene todos los objetos que observan al SujetoObservable y llama a su método “actualizar()”. De esta manera se mantienen todos los observadores actualizados ante cualquier cambio, sin necesidad de que estén consultando constantemente si existen actualizaciones de estado.

Conclusiones

El patrón crea una dependencia directa de cada observador hacia el sujeto. Si bien esto puede llegar a traer complicaciones, es la manera en la que está estructurado el patrón.

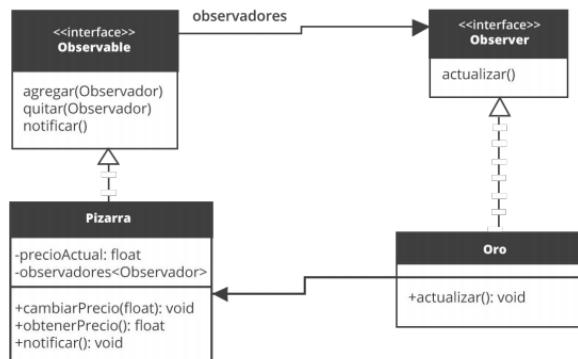
Algo que no debe suceder es que el sujeto dependa de algún observador. Este comportamiento vulneraría la ignorancia que debe tener el sujeto sobre sus observadores y podría generar dependencias cíclicas.

Es conveniente especificar las modificaciones de interés explícitamente. Se puede mejorar la eficiencia extendiendo la interfaz de registro del sujeto para permitir que los observadores registren solo aquellos los eventos concretos que le interesen.

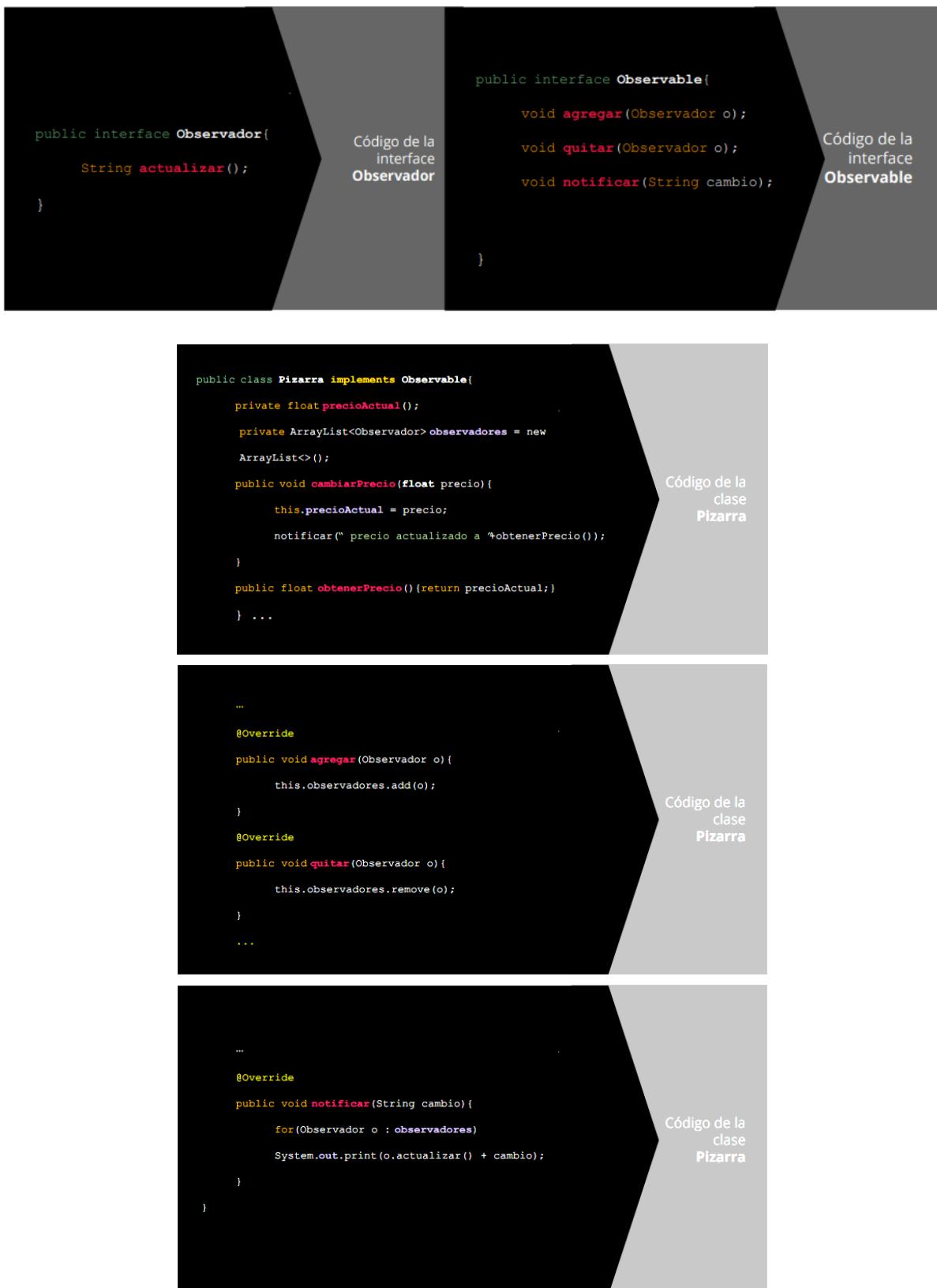
Ejemplo de Patrón Observer

Ahora imaginemos una implementación. Supongamos que tenemos un sistema que muestra en una pizarra el precio del oro y que al actualizarse se sincroniza con cada instancia suscrita informando el cambio del precio.

Veamos el diagrama UML a seguir.



En JAVA



```

public class Oro implements Observador{
    @Override
    public String actualizar() {
        return this + "> Cambio de estado: ";
    }
}

```

Código de la clase Oro

```

public class Main{
    public static void main(String[] args){
        Pizarra pizarra = new Pizarra();
        Observador obs1 = new Oro();
        Observador obs2 = new Oro();

        pizarra.agregar(obs1);
        pizarra.agregar(obs2);

        pizarra.cambiarPrecio(2.5f);
        pizarra.cambiarPrecio(4.3f);
    }
}

```

Código de la clase Main

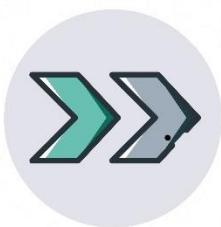
```

Oro@568db2f2> Cambio de estado:  precio actualizado a 42.5
Oro@378bf509> Cambio de estado:  precio actualizado a 42.5
Oro@568db2f2> Cambio de estado:  precio actualizado a 44.3
Oro@378bf509> Cambio de estado:  precio actualizado a 44.3

Process finished with exit code 0

```

Patrón Observer



Dependencia unilateral

Los observadores dependen del sujeto y no al revés. Este desacople permite mayor escalabilidad cuando se trabaja con aplicaciones que tienen muchas entidades.

Acoplamiento abstracto

Debido a que todas las clases que observan al **sujeto** implementan la interfaz **Observador**, este último no conoce la clase concreta de ningún **Observador**. Por lo tanto, el acoplamiento entre sujetos y observadores es mínimo.

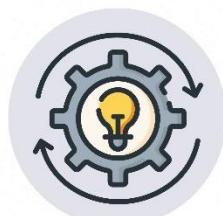


Capacidad de difusión

Todos los observadores recibirán las actualizaciones automáticamente, permitiendo así una poderosa capacidad de notificar cambios de estado a otros objetos sin modificarlos.

Gestor de cambios

Cuando la relación dependencia entre sujetos y observadores es compleja se puede prescindir de un objeto que mantenga estas relaciones. Llamado **GestorDeCambios**, su propósito es minimizar el trabajo necesario para que los cambios sean reflejados en los observadores.



Patrón Strategy

Cuando en la programación orientada objetos tenemos que cambiar la estrategia o modo de actuar utilizamos el Patrón Strategy.

Propósito

Un determinado objeto va a tener un comportamiento que puede ser simple y siempre el mismo, pero a veces este comportamiento se vuelve más complejo y de acuerdo con las necesidades, cambiante.

El patrón Strategy hace que los algoritmos varíen independientemente del cliente que los esté usando. Propone una solución simple basada en un objeto que cambia y cuyo comportamiento es el que se adapta a la circunstancia.

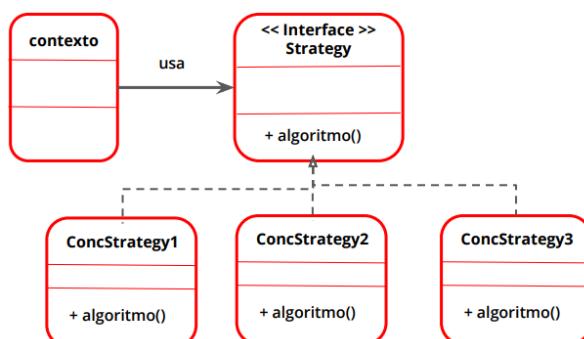
Se dispone de una interface, Estrategia (Strategy), que sirve para establecer las firmas de los métodos que vamos a hacer que cambien y las diferentes clases que implementan esta interface, Estrategias Concretas (ConcreteStrategy), que son las que tienen los diferentes algoritmos para realizar la tarea.

Podremos ver algún caso donde queremos que la Estrategia también tenga algún atributo que sea común a cada Estrategia concreta. En este caso la Estrategia será una clase abstracta y las Estrategias concretas serán heredadas.

Ventajas	Desventajas
El uso del patrón proporciona una alternativa a la herencia de clases, ya que puede realizarse un cambio dinámico de estrategia.	Aumenta el número de objetos creados, por lo que se produce un aumento en el intercambio de objetos entre estrategia y contexto
Si un algoritmo utiliza información que no deberían conocer los clientes, la utilización del patrón Strategy evita la exposición de dichas estructuras.	La clase que elige la estrategia debe elegir cuál es la mejor para ese momento
Este patrón de diseño nos sirve para intercambiar un sin número de estrategias posibles.	
Evita las sentencias switch o if	

Tabla 6: Ventajas y Desventajas del Patrón Strategy

Diagrama UML



- **Contexto:** es el elemento que usa los algoritmos, delega en la jerarquía de estrategias. Configura una estrategia concreta mediante una referencia a la estrategia necesaria y puede, por lo tanto, cambiar la estrategia de acuerdo a sus necesidades.
- **Interface Strategy:** declara una interface común para todos los algoritmos soportados. Esta interface será usada por el contexto para invocar a la estrategia concreta. También puede usarse una clase de tipo Abstract en caso de precisar variables o métodos comunes a las diferentes estrategias.
- **ConcStrategy:** implementa el algoritmo utilizando la interfaz definida por la estrategia. Sirve para tener en esta clase una lógica que se puede reutilizar y aislada del contexto.

¿CÓMO FUNCIONA? Cuando el contexto quiere utilizar algún algoritmo concreto primero setea la variable (que tendrá un objeto que implemente la interface Strategy) con un valor adecuado. Luego invoca el método que desea ejecutar. El que recibe esta invocación será un objeto EstrategiaConcreta que lo ejecuta con el algoritmo propio