Apuntes de Repaso:

El constructores son funciones especiales que nos permiten construír objetos.

Por convención son públicos(public). Lo puedo ver desde fuera de la clase. El main por ejemplo.

El nombre del constructor es el mismo que el de la clase, luego (), kuego {}.

Ejemplo Public Vehiculo() { } ; (Constructor vacío).

O

Public Vehiculo(String fabricante){

This.fabricante = fabricante;

}

Sería lo mismo, aunque es confuso, no recomendable hacer

Public Vehiculo (String x){

Fabricante=x; }

Se pueden generar los constructores.

Para crear:

*Clase* Identificador =*new Clase()*

Llamado al constructor

Java Avanzado.

Vamos a crear un mini API REST. Es un proyecto que va a escuchar peticiones http, va a aceptar peticiones y va a generar respuestas.

Vamos a utilizar un frame que se llama Jersey .

Postman se utiliza en el desarrollo web para testear aplicaciones.

Los controladores en java son clases que por convención terminan con “controler”, ejemplo “HelloControler”.

JDVC si quiero buscar datos reales. Recomienda trabajar con algún ARRAY.

Plug in útiles:

RainbowColorear los paréntesis de colores distintos.

Tabnine Completa código.

Gif toolbox

ExtraIcon

AWS toolkit: Si trabajo con AWS

**PROGRAMACIÓN FUNCIONAL.**

*CONC}}EPTO - DIFERENCIAS CON LA POO*

Es un tipo de programación declarativa y no imperativa.

En la programación imperativa programamos que “si pasa esto haz esto”.

En cambio en la programación funcional es del tipo declarativo.

Java es un lenguaje orientado a objetos el cual a heredado cosas de los lenguajes funcionales, pero no es un lenguaje funcional. Victor lo llama pseudofuncional.

La sintaxis de los lenguajes funcionales es totalmente diferente a lo visto en Java.

En java, sus autores, algunos conceptos de la programación funcional los han ido metiendo poquito a poquito en el lenguaje.

*INTRODUCCIÓN A FUNCIONES PURAS. - FUNCIONES PURAS VERSUS IMPURAS.*

Es toda aquella que:

1. dados unos mismos parámetros de entrada siempre produce el mismo resultado de salida.

public static int Suma(int a, - b) {  
 return a+b;

La función suma es una función pura ya que siempre va a devolver el mismo resultado siempre que sus parámetros de entrada no varíen.

1. No puede producir efectos colaterales. Un efecto colateral es todo aquel cambio que se realiza cuando invoco una función. Si una función invoca a otra que podría no ser pura, ya no es una función pura. Cuando una función que queremos sea pura **modifica** una variable global, o local ya no es una función pura.

Función Pura.

public class FuncionPura {  
 public static void main(String[] args) {  
 int result =*Suma*(1, 2);  
 int result2 =*Suma*(1, 2);  
 System.*out*.println(result + " "+ result2);  
 }  
  
 public static int Suma(int a, int b) {  
 return a+b;  
 }  
}

Función Impura

public class FuncionImpura {  
private static int *contador* = 0; // variable global  
  
   
 public static void main(String[] args) {  
 int result =*Suma*(1, 2);  
 int result2 =*Suma*(1, 2);  
 System.*out*.println(result + " "+ result2);  
 }  
  
 public static int Suma(int a, int b) {  
 *contador* +=1;  
 return a+b;  
 }  
 }

En éste caso al llamar a la función suma se modifica la **variable global “contador”,** se produce un daño colateral por lo que la función ya no es pura.

Las variables que pueden modificarse para que una función siga siendo pura son todas aquellas que son locales a la función.

En el momento en que modifico una variable que no es local a la función, por ejemplo, una propiedad de la clase, en ese momento la función ya no es pura.

1. Para que una función sea pura debe hacer exactamente lo que dice. Si la función dice suma lo único que debe hacer es sumar.

public static int Suma(int a, int b) {  
 *contador* +=1;  
 *multiplica*(a,b) //No es pura porque como efecto colateral llama otra función que multiplica.  
 return a+b;  
   
}  
public static int multiplica (int a, int b) { //No es pura porque dice que suma y multiplica.  
 return a+b;  
}

Las funciones puras son muy difíciles de escribir.

Una función que lee de una base de datos suele ser impura.

Una función que lee un fichero de texto suele ser impura porque dicho fichero puede variar.

INTRODUCION A LAS FUNCIONES DE ALTO ORDEN

Una función de alto orden significa que una función puede llamar a otra función o debe devolver una función.

Una función de alto nivel es aquella que debe recibir un parámetro obligatoriamente que sea otra función o que devuelve otra función.

Como argumentos tiene el nombre de otra función que ya exista.

O bien

Una función devuelve en el return otra función.

Ejemplo:

public static String devuelve (String cadena){  
 return cadena.startsWith("a"); // En el retorno devuelve otra función.

Landas:

Son funciones que se definen en funciones. Son funciones Especiales.

|  |  |
| --- | --- |
| public class FuncionConLAnda {     public static void main(String[] args) {  Funcionales f = new Funcionales();  f.pruebas();  } | import java.util.function.Function;  public class Funcionales {  private Function<Integer, Integer> sumador = (x) -> x.*sum*(x, x);  private Function<String, String> toMayus = (x) -> x.toUpperCase();   public void pruebas() {  toMayus.apply("Sergio");  sumador.apply(5);    } } |

Creo tipo de dato Function  
 /\* Los parámetros son que tipo de parámetro acepto y que tipo de parámetro devuelvo  
 (En este caso <String, String> y le doy un nobre (toMayus). luego lo asigno a la variable x y  
 digo me lo devuelva en mayúscula. Esta es una funcion de alto nivel.  
  
 /Debajo tengo un método que hace lo mismo que la función de alto nivel.  
 // public static String toMayuscula(String nombre){  
 // return nombre.toUpperCase();  
 // }  
 }

Cuando invoco toMayus.apply, se ejecuta el cuerpo de la función, la x es el parámetro de entrada, es del tipo String, está hecho de manera funcional.

La misma función de modo imperativo hubiese sido:

Public String toMayus (String x){

Return x.toUppercase();}

Las funciones de alto nivel se utilizan mucho para encadena r la salida de una función con la entrada de otra.

Permite hacer lo mismo que la imperativa pero escrito de manera diferente. A veces con menos líneas.

Luego de crear valor.el String y recorrerlo de la forma tradicional con un For, vemos que nombres tiene una función pura que se llama stream() y que devuelve un array secuencial.

nombres.stream().forEach(x ->System.*out*.println(x));

Para cada valor de x imprime el valor de x.

El valor de x es el que me inyecta la función anterior, que es lo que está en mi arraylist. Cada vez que el for each recibe un valor la x se convierte en éste valor.

La función map es una función de mapeo aplica lo que le indico sobre cada elemento del String. Para poder utilizarlo luego lo guardo en otro Stream

Stream<String> valores =nombres.stream().map(x -> x.toUpperCase());

En la siguiente línea lo recorro y lo imprimo

valores.forEach(x -> System.*out*.println(x));

El Stream tiene la desventaja que una vez que lo utilizo (lo recorro) ya no puedo volver a acceder a él.

Si repitiera nuevamente la última línea (la duplicara), daría error al ejecutar diciendo que el Stream ya se ha utilizado.

Para volver a iterar el array debo crear un nuevo stream.

Pero para esto existe una función ;).

FUNCION REDUCE

var resultado2= stnumeros.filter(x-> x%2 == 0).reduce(0,(x,y)-> x+y);

//Reduce (variable inicial,EXPRESIÓN DE REDUCCIÓN  
//En éte caso la variable inicial es 0 porque trabajo con números.(podría ser 20 y a 20 le sumaría los pares) En cadana sería "", por ejemplo.

Lo que haría en éste caso es sumar los números pares.

El filter deja los pares, reduce va sumado cada número que va quedando.

En la primer iteracciónn el valor inicial es tomado por x (0) y tomará el primer valor (2) la y.

A x le suma el valor de y (2) entonces x ahora vale 2. En la segunda iteracción y valdrá el segundo valor que es 2 y se lo sumara a x y así sucesivamente.

.MAP

.FILTER.

.REDUCE

Se usan mucho. Se puede hacer de la forma vista funcional o también con varios bucles.

import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.function.Function;  
import java.util.stream.IntStream;  
import java.util.stream.Stream;  
  
public class FuncionConLAnda {  
  
 public static void main(String[] args) {  
 Funcionales f = new Funcionales();  
 f.pruebas();  
 ArrayList<String> nombres = new ArrayList();  
 nombres.add("Paco");  
 nombres.add("Pepe");  
 nombres.add("Juan");  
  
 for (String nombre : nombres) {  
 System.*out*.println(nombre);  
 }  
  
 //Sería lo mismo funcionalmente:  
 nombres.stream().forEach(x -> System.*out*.println(x));  
  
 //Para poder ver lo que hace la siguiente línea debo crear una variable antes.  
 nombres.stream().map(x -> x.toUpperCase());  
  
 Stream<String> valores = nombres.stream().map(x -> x.toUpperCase());  
 valores.forEach(x -> System.*out*.println(x));  
//En la líneaa siguiente vuelvo a crear el Stream para volver a iterar  
 Stream<String> valores2 = nombres.stream().map(x -> "Hola " + x.toUpperCase());  
 valores2.forEach(x -> System.*out*.println(x));  
// En la siguiente línea digo que los convierta en mayúscula pero solo guarden en mi stream nuevo los que comiencen con la letra "p"  
  
 Stream<String> valores3 = nombres.stream()  
 .map((x) -> x.toUpperCase())  
 .filter((x) -> x.startsWith("P"));  
 valores3.forEach(x -> System.*out*.println(x));  
  
 //Debajo tengo un array de enteros y lo voy a convertir a String para operar.  
 int[] numeros = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
//Creo un Stream del tipo Integer que llamo stnumeros y uso Arrays.stream (nombre de la variable a convertir)  
  
 var stnumeros = Arrays.*stream*(numeros);  
  
 var resultado = stnumeros.filter(x -> x % 2 == 0); //Guardamos los números pares.  
 resultado.forEach(x -> System.*out*.println(x));  
  
// Funcion REDUCe: toma los datos que me quedan y los reduce a una expresió.  
  
 var stnumeros2 = Arrays.*stream*(numeros);  
  
 var resultado2 = stnumeros2  
 .map((x) -> x \* 2) //Multiplico los números x 2  
 .filter(x -> x % 2 == 0) //Filtro los números  
 .reduce(0, (x, y) -> { //Sumo los números  
  
 System.*out*.println("x es " + x + " y " + y);  
 return x + y; //Si uso llaves puedo seguir escribiendo código y debo terminar con return (forma larga).  
 }); //  
  
  
 System.*out*.println("Mi suma de numeros pares es " + resultado2);  
  
  
 //Con bucles:  
 int sumaTotal = 0;  
 for (int numero : numeros) {  
 numero = numero \* 2;  
 if (numero % 2 != 0) {  
 continue;  
 }  
 sumaTotal += numero;  
  
 }  
 System.*out*.println("Suma total con bucles " + sumaTotal);  
 }}

}

Recursión.

La programación funcional utiliza mucho la recursión.

La Recursión consiste en que una función se llama a si misma.

Cuando la recursión funciona mejor para un algoritmo podríamos utilizarlo. Ejemplos típicos Arboles binarios, suma consecutiva de números, factoriales, etc.

La recursión es una forma de hacerlo, pero se puede hacer también con iteracción utilizando bucles for, while o Do while.

**Tipos de Recursión**

**Recursividad de cola**

Lo último que se ejecuta es la llamada de la función a si misma. Por ejemplo la función suma de la parte práctica.

Como el último paso es invocar la función hablamos de recursividad de cola.

public static void taiRecursion(int valor){  
 if (valor == 0 ){  
 return;  
 }  
 System.*out*.println(valor); //Primero muetro valor  
 *taiRecursion*(valor-1); //Por ultimo llamo la funcion.  
}

**5**

**4**

**3**

**2**

**1**

**Recursividad por cabeza**

Cuando lo primero que hacemos es invocar a la función recursiva.

public static void headRecursion(int valor){  
  
 if (valor==0){  
 return;  
 }  
 *headRecursion*(valor-1); //Primero invoco la funcion  
 System.*out*.println(valor); //Luego muestro valor  
}

1

2

3

4

5

En el segundo caso recién entra al System.out.println luego de salir de la función que se llama a si misma varias veces.

En la pila quedan los números que irá imprimiendo

SUMA FUNCIONAL.

public static int sumaFuncional (int numero){  
 return IntStream.*rangeClosed*(1,5) //rangeClosed genera números del 1 al 5  
 .reduce(0,(a,b)-> a+b);  
}

Tenemos 3 formas de hacer la suma:

Forma Iterativa.

Forma Recursiva.

Forma funcional.

Hacemos Factorial de las 3 formas:

import java.util.stream.IntStream;  
  
public class recursionFactorial {  
 public static void main(String[] args) {  
 System.*out*.println("Factorial iterativo " + *factorial*(5));  
 System.*out*.println("Factorial recursivo " + *factorialRecursivo*(5));  
 System.*out*.println("Factorial Funcional " + *factorialFuncional*(5));  
 }  
 public static int factorial (int numero){  
 int resultado = 1;  
 for (int i = 1; i <=(numero); i++){  
 resultado = resultado \* i ;  
 }  
 return resultado;  
 }  
 public static int factorialRecursivo (int numero){  
 if (numero == 0) {  
 return 1;  
 }return numero\**factorialRecursivo*(numero-1);  
 }  
 public static int factorialFuncional (int numero){  
 return IntStream.*rangeClosed*(1,numero)  
 .reduce(1,(a,b) -> a\*b);  
 }  
}

Ninguna de las 3 formas es mejor o peor.

Ventajas de la recursividad:

1. En algunos casos la complejidad de tiempo. Pero para esto debe utilizarse la técnica de MEMOIZE.
2. Añade claridad al escribir y depurar código.
3. Para recorrido de árboles lo mejor es utilizar la recursividad.

Desventajas de la recursividad:

1. El consumo de memoria. Cada vez que se llama a si mismo crea una copia de todas las variables utilizadas previamente. Si depuramos podemos verlo.
2. Puede ser muy lenta si no se implementa correctamente.

//Midiendo tiempos:  
  
long tiempoInicial = System.*currentTimeMillis*();  
for (int i = 0; i < 100000; i++) {  
 *sumaFuncional*(5);  
}  
long tiempoFinal = System.*currentTimeMillis*();  
System.*out*.println("Ms en total:"+ (tiempoFinal-tiempoInicial) );

SERVICIOS REST

Un servicio REST es una aplicación WEB que cumple una serie de patrones determinados.

REST utiliza los verbos HTTP. Que son un tipo de petición que enviamos a un servidor utilizando el protocolo HTTP.

Los verbos HTTP mas utilizados son:

GET Y POST.

En aplicaciones tipo Rest se utilizan muchísimo PUT y DELETE.

GET: Obtiene información de un servidor. Un recurso.

POST: Envío datos al servidor.

PUT: Habitualmente se utiliza para actualizar algo que ya exista (ejemplo un usuario).

DELETE: Para eliminar un recurso que ya exista.

PATCH: También actualiza un usuario. EL PUT envía todos los datos del usuario y los que ha cambiado, en cambio PATCH sólo envía los que se han modificado.

Los servicios tipo REST son un CRUD.

No todas las aplicaciones son tipo REST. Ni todas soportan el PATCH.

Los servicios REST deben estar bien definidos. Para los recursos se utilizan nombres en plural. Ejemplo users, cars, pets, etc.

<http://www.dominio.tld/users/sgaraycochea/friends>

En la actualidad hablamos de APIs REST.

Las Apis se versionan

http://www.dominio.tld/api/v1/users

v1: versión.

Proyecto Spring Clase 4.

Luego de generar el proyecto, en éste caso descargado de GitHub, para cambiar el puerto 8080, en el archivo application.propietes escribimos:

server.port=9000

Escribiendo en el navegador:

Localhost:9000 nos va a arrojar

[HTTP Status 404 – Not Found](http://localhost:9000/) . ya estoy dentro de mi aplicación.

Lo primero que debemos hacer es crear un fichero con su versión para Jersey, indicando dónde van a estar nuestros controladores. El controlador recibe las peticiones por HTTP y va a procesarlos. Es el punto de entrada a nuestras llamadas.

En la carpeta java creo el nuevo paquete “Config” , dentro del mismo creo un archivo java class que llamamos JerseyConfig

|  |  |
| --- | --- |
| @ApplicationPath("/")  @Component  public class JerseyConfig extends ResourceConfig {  public JerseyConfig() {  this.packages("com.openbootcamp.demospringrest.controllers");  } } | Indica que nuestra aplicación se ejecuta después del :9000  Indicamos a Springque esto es un componente  Dentro de la clase JerseyConfig debo crear un costructor y dentro el código. Indicamos que la clase hereda de ResourceConfig  Indicamos dónde van a estar nuestros controladores.  Lo hacemos con la función this.packages, heredada de ResourceConfig. |

@ApplicationPath("/")   
@Component  
public class JerseyConfig extends ResourceConfig {  
 public JerseyConfig() {  
 this.packages("com.openbootcamp.demospringrest.controllers");  
 }  
}

Controladores.

Una vez indicado dónde van a estar los controladores. Un controlador es una clase de JAVA.

Lo llamamos BootCampersController

Dentro del mismo indicamos que esU un componente y el path.

Un controlador tiene que tener métodos

Para implementar una función con Jersey que me devuelva algo implemento el método GET

Y luego creo la función

CURL

Cuando estamos programando una aplicación de Back End, no vamos a utilizar el navegador para las peticiones, sino que usaremos peticiones HTTP directamente,

Modelos: Un modelo es lo que da forma o lo que contiene la información de los datos.

Crea la carpeta models.

Creamos la clase BootCamper (es el modelo).

Creamos luego los constructores. El Geter y el seter.

Y un constructor vacío.

Servicio: Comunica el controlador con el modelo.

Lo creamos en la carpeta Service.

El controlador acepta la petición , el modelo tiene la información y el servicio hace de unión.

EL servicio manipula el modelo.

Nomenclatura: “nombre del servicio”Service

bootcamperService en éste caso.

Es una clase pero debo indicar que es un service. @Service

POSTMAN (Clase 5).

Postman: Es una aplicación para realizar pruebas o peticiones contra appis, utilizando el protocolo http. Al obtener las respuestas se pueden ejecutar test.

Se utiliza a la hora de depurar appis.

Para poder enviar y recibir datos debemos crear una colección, que es un conjunto de peticiones.

Con + creamos una nueva colección. La llamamos Open Bootcamp.

Una vez creada la colección crearemos peticiones. Para esto desplego la flecha en la colección creada o la deseada y cliqueo ADD Request que abrirá el formulario.

Le doy un título. “Listar usuarios.

Elijo método, GET, en éste caso.

Completo : <http://localhost:9000>

Devuelve los usuarios.

Con POST puedo enviar datos. En éste caso en modelo JSON..

Para poder hacerlo marco la pestaña body – Selecciono RAW y en text puedo seleccionar JSON para verlo coloreado.

Escribo:

{

“nombre” : Prueba,

“valor” : 1.2345

}

SEND y crea el nuevo usuario.

Con GET puedo verlo.

Haciendo clic en la colección puedo crear variables para no tener que escribir permanentemente las direcciones.

Enviroments: Entornos.

Cuando se crea software se trabaja con entornos de desarrollo.

En la parte izquierda se puede encontrar la opción.

Postman también puede utilizarse para la depuración de las APIs.

En http el código 200 es OK.

En postman puedo crear Test. Postman utiliza JScript.

Con el siguiente código comprobamos que la api está funcionando.

pm.**test**("estado es 200"), **function**(){

    pm.response.to.have.status(200);

}

En result test va a estar en verde. Si no funcionara estaría en rojo.

Test de latencia vemos si responde en meenos de 10 milisegundos.

pm.**test**("respuesta en menos 10 milisegundos",()**=>** {

    pm.expect(pm.response.responseTime).to.be.below(10);

});

Exportar:

Los tres puntitos arriba a la derecha desplegan menú para exportar en un fichero.

Sesión 6. Patrones de diseño.

Un patrón de diseño es una técnica para resolver un problema en el mundo del software.

Es la solución a problema de una forma concreta.

Hay alrededor de 24 patrones.

Los objetivos de los patrones son:

* Proponen un catálogo de elementos reutilizables.
* Permite que nos entendamos entre desarrolladores
* Estandarizan el desarrollo.
* Facilitan que los que están empezando no pierdan tiempo tratando de resolver problemas que ya están resueltos.
* Los patrones no tienen como objetivos ser utilizados a la fuerza.

Antipatrones: Son malas prácticas.

EL patrón Singlton afirma que una clase solo puede tener una instancia.

Clasificación de los parones:

Por nombre: Cada patrón tiene un nombre único.

Por campo. Son 3 campos.

Por finalidad: que problema quiere resolver.

Por Aplicabilidad: En que puede usarse.

Por consecuencias: positivas y negativas.

Hay 3 campos principales:

Creacionales. Estructurales y de comportamiento.

Los creacionales sirven para crear instancias de objetos.

Estructurales: Solucionan diseño de composición. Cohesionan diferentes clases. Las acopla.

Comportamiento: Reacciona ante algo.

SESION 7: PATRONES CREACIONALES.

Patrón Singleton.

Solo puede haber una instancia de una Clase.

Intenta evitar que vuelva a instanciar un objeto. Por lo que debo operar con la instancia que ya había creado.

Sirve para que no pueda duplicar el uso de una aplicación, o en base de datos no estar accediendo duplicadamente al mismo fichero.

Se comparte la instancia previamente creada.

|  |  |
| --- | --- |
| Clase Singleton | Main |
| package Singleton;  import static java.util.Collections.*singleton*;  public class Singleton {  private static Singleton *singleton*;  int contador;   private Singleton(){} // Al poner el constructor private no se puede instanciar Singletos desde afuera.  public static Singleton getInstance() { //Esta función es la única que puede generar un new Singleton  if (*singleton* == null) {  *singleton* = new Singleton();  }  return *singleton*;  }  public int getContador() {  return contador;  }   public void setContador(int contador) {  this.contador = contador;  } } | package Singleton;  public class Main {   public static void main(String[] args) {  Singleton singleton= Singleton.*getInstance*();  singleton.setContador(15);  System.*out*.println("Valor" + singleton.getContador() + " En memoria " + singleton);  Singleton singleton2= Singleton.*getInstance*();  singleton2.setContador(20);    System.*out*.println("Valor" + singleton2.getContador() + " En memoria " + singleton2);  }  } |

Ejemplo del control de APP:

|  |  |
| --- | --- |
| Clase App | Main |
| package Singleton;  public class Aplicacion {  private static Aplicacion *aplicacion*;  boolean isRunning = false;  private Aplicacion(){}  public static Aplicacion getInstance(){  if (*aplicacion* == null ){  *aplicacion* = new Aplicacion();  }return *aplicacion*;  }  public void Run() {  if (!isRunning) {  isRunning = true;   }else System.*out*.println("Ya estaba en ejecucion.....");  } } | package Singleton;  public class MainAPP {  public static void main(String[] args) {  Aplicacion app =Aplicacion.*getInstance*();  Aplicacion app2=Aplicacion.*getInstance*();   app.Run();  app2.Run();  } } |

Escondo constructor

Creo variable privada

En síntesis en la clase que no quiero que se reinstancie :

1. Escondo el constructor declarándolo private.
2. La variable laplicación la declaro como privada t estática.
3. Creo un método estático getInstance() que por medio de un if detecta si la variable ya fue instanciada devolviendo una instancia de la misma si no es así. Caso contrario devuelve la variable ya instanciada.

PATRÓN FACTORY:

Sirve para crear una jerarquía de clases y usar una u otra de forma transparente.

Cohesiona una jerarquía de clases en una única.

En el ejemplo cohesiono las clases precioUSD y precioEUR en una llamada PrecioFactory,

En el ejemplo, según el parámetro que pase, me devuelve una instancia de un tipo o de otra. Una instancia del tipo PrecioEUR en el primer caso, en el segundo una instancia del tipo PrecioUSD.

INTERFAZ:

package Factory;  
  
public interface Precio {  
 double getPrecio();  
}

|  |  |
| --- | --- |
| Clases |  |
| package Factory;  public class PrecioUSD implements Precio{  @Override  public double getPrecio() {  return 0.94;  } } | package Factory;  public class PrecioEUR implements Precio{  @Override  public double getPrecio() {  return 1.3;  } } |

|  |
| --- |
| Clase Factory, donde se cohesionan las anteriores |
| package Factory;  public class PrecioFactory {  Precio precio;  private PrecioFactory() {}//oculto constructor   public PrecioFactory(String pais) {  if (pais.equalsIgnoreCase("España")) {  System.*out*.println("España, Precio en EUR");  this.precio= new PrecioEUR();  } else  System.*out*.println("Otro país, Precio en USD ");  this.precio= new PrecioUSD();  } } |

MAIN

Package Factory;  
  
public class main {  
 public static void main(String[] args) {  
 PrecioFactory precio = new PrecioFactory("españa"); //solo puedo invocar el constructor con argumento  
 System.*out*.println(precio.precio.getPrecio());  
 PrecioFactory precio2 = new PrecioFactory("USA");  
 System.*out*.println(precio2.precio.getPrecio());  
 }  
  
}

En síntesis:

Creo la interfaz que es común a todos los objetos que puede construir la clase creadora.

La clase creadora, Precio Factory declara el método precioFactory que es el que va a devolver los nuevos objetos producidos.

PATRON BUILDER:

Permite construír objetos complejos paso a paso.

Consiste en la programación fluída donde un método devuelve su propia instancia.

CLASE VEHICULO

package Builder;  
  
public class Vehiculo {  
 private String marca;  
 private String tipo;  
 private String motor;  
 private int puertas;  
  
 public String getMarca() {  
 return marca;  
 }  
  
 public void setMarca(String marca) {  
 this.marca = marca;  
 }  
  
 public String getTipo() {  
 return tipo;  
 }  
  
 public void setTipo(String tipo) {  
 this.tipo = tipo;  
 }  
  
 public String getMotor() {  
 return motor;  
 }  
  
 public void setMotor(String motor) {  
 this.motor = motor;  
 }  
  
 public int getPuertas() {  
 return puertas;  
 }  
  
 public void setPuertas(int puertas) {  
 this.puertas = puertas;  
 }  
}

COCHE BUILDER

package Builder;  
  
public class CocheBuilder {  
 Vehiculo vehiculo;  
 public CocheBuilder(String marca) {  
 vehiculo=new Vehiculo();  
 vehiculo.setMarca(marca);  
 }  
 public CocheBuilder setPuertas(int puertas){  
 vehiculo.setPuertas(puertas);  
 return this;  
 } public CocheBuilder setMotor(String motor) {  
 vehiculo.setMotor(motor);  
 return this;  
 }  
 public Vehiculo build(){  
 return this.vehiculo;  
 }  
}

MAIN

package Builder;  
  
public class main {  
 public static void main(String[] args) {  
 // Vehiculo vehiculo= new Vehiculo();  
 Vehiculo coche = new CocheBuilder("Filostropo")  
 .setPuertas(5)  
 .setMotor("Eléctrico")  
 .build();  
 System.*out*.println("Marca" + coche.getMarca());  
 System.*out*.println("Puertas" + coche.getPuertas());  
 System.*out*.println("Motor" + coche.getMotor());

Permite construir objetos complejos paso a paso.

En el ejemplo crea el objeto vehículo.

Luego crea un builder (Coche builder)

Dentro de éste objeto se declara la variable vehículo del tipo Vehiculo, es decir invoca a la otra clase.

Vehiculo vehiculo;

Se crea un constructor de CocheBuilder con un patrámetro(marca) y se instancia el mismo dentro de otra clase, no en el main

Public CocheBuilder(String marca){

vehiculo = new Vehiculo

vehiculo.setMarca(marca);}

Luego, debo implementar los mismos parámetros de mi clase Vehiculo:

public CoccheBuilder setPuetas(int puertas){

vehiculo.setPuertas(puertas);

return this;

Se puede ver en el método anterior que:

Estoy devolviendo un CocheBuilder, que es el mismo nombre de mi clase actual. Es lo que se conoce como el truco de la programación fluida.

El this del return :

This significa la instancia actual de mi clase. Esto quiere decir ésta función se devuelve a si misma a su propia clase, esto es para añadir a los builder lo que se conoce como “propiedad de fluidez” en la cual un método devuelve su propia instancia.

Esto puede explicarse de la siguiente manera:

Al devolver el método, por ejemplo setPuertas un CocheBuilder (que tiene la propiedad setMotor, que devuelve un coche builder) y así sucesivamente. Es el truco de la programación fluida.

Esto se ha utilizado en programación funcional.

Al final de la clase se crea la siguiente función:

public Vehiculo build(){  
 return this.vehiculo;  
 }

Creamos un método build() dónde devolveremos un tipo de dato Vehiculo que es el que he creado mas arriba. El método devuelve this.vehiculo que es la variable que he creado mas arriba y es el que voy a instanciar en el main:

Vehiculo coche = new CocheBuilder("Filostropo")

PATRON PROTOIPE

Es u patrón creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.

Tiene como objetivo cl onarse a si mismo, es decir que una clase se clone a si misma, y que ese clon herede todos los métodos y las propiedades de la versión previa y su estado. Es decir que clona el objeto con los valores que tengan asignadas sus variables en ese momento.

Objeto

package Prototype;  
  
public class Coche implements Cloneable{ //Utilizo interfaz Cloneable  
 public String marca;  
 public String modelo;  
 int puertas;  
  
 public Coche(){}  
  
 public Coche clonar() throws CloneNotSupportedException{  
  
  
 return (Coche) this.clone();  
  
 }  
  
}

MAIN

package Prototype;  
  
public class main {  
 public static void main(String[] args) {  
 Coche coche = new Coche();  
 coche.puertas = 5;  
 coche.marca = "Fiat";  
 coche.modelo = "Coupe";  
  
 System.*out*.println(coche.marca + " " + coche.puertas + " " + coche.modelo);  
try {  
 Coche clonado = coche.clonar();  
 clonado.puertas = 3;  
 System.*out*.println(clonado.marca + " " + clonado.puertas + " " + clonado.modelo);  
 } catch (CloneNotSupportedException e) {  
 System.*out*.println("No se puede clonar." + e.getMessage());  
 }   
 }  
}

PATRONES ESTRUCTURALES.

1-PATRON DECORADOR: Permite añadir funcionalidad a una clase existente sin modificarla.

Ejemplo de los teléfonos.

Victor no lo usa mucho, solo en casos forzados.

Usos reales:

Enviar notificaciones a usuarios por correo, por Facebook, Instagram, etc.

Creamos la clase base de correo y luego el decorador y las siguientes clases.

package Decorator;  
  
public interface Telefono {  
 void crear();  
}

package Decorator;  
  
public class TelefonoBasico implements Telefono{  
 @Overrid

e  
 public void crear() {  
 System.*out*.println("Soy un teléfono báscico. Tengo éstas características: ");  
 this.tengoGSM();  
 this.tengoSMS();  
 }  
 private void tengoGSM(){  
 System.*out*.println("basico.........--> Tengo GSM");  
 }  
 private void tengoSMS(){  
 System.*out*.println("--> Tengo SMS");  
 }  
}

public class TelefonoDecorator implements Telefono {  
 private Telefono telefono;  
 public TelefonoDecorator(Telefono telefono) {  
 this.telefono = telefono;  
 }  
 @Override  
 public void crear(){  
 this.telefono.crear();  
}}

public class TelefonoInteligente extends TelefonoDecorator{  
 public TelefonoInteligente (Telefono telefono){  
 super(telefono);  
 }  
 @Override  
 public void crear(){  
 super.crear();  
 System.*out*.println("Inteligente............Tengo también 4g");  
 }  
}

ackage Decorator;  
  
public class TelefonoNextGen extends TelefonoDecorator{  
 public TelefonoNextGen (Telefono telefono){  
 super(telefono);  
 }  
 @Override  
 public void crear(){  
 super.crear();  
 System.*out*.println("Next Gen ......Tengo también 5g Volte");  
 }  
}

MAIN:

package Decorator;  
  
public class main {  
 public static void main(String[] args) {  
 TelefonoBasico telefonoBasico= new TelefonoBasico();  
 telefonoBasico.crear();  
 TelefonoInteligente ti=new TelefonoInteligente(new TelefonoBasico());  
 ti.crear();  
  
 TelefonoNextGen tng = new TelefonoNextGen(new TelefonoInteligente  
 (new TelefonoInteligente(new TelefonoBasico())));  
 tng.crear();  
 }  
}

2-PATRON ADAPTER:

Es un adaptador. Crea una clase que permite que clases con la misma implementación y clases con distinta implementación funcionen coherentemente.

Se utiliza mucho con código antiguo que estamos mejorando.

Adaptamos sin modificar el código fuente.

public interface Enchufable {  
 void enciende();  
 void apaga();  
 boolean estaEncendido();  
  
}

public class Horno implements Enchufable{  
 boolean encendido = false;  
 @Override  
 public void enciende(){  
 encendido =true;  
 System.*out*.println("Horno encendido");  
 }  
 @Override  
 public void apaga() {  
 encendido = false;  
  
 }  
  
 @Override  
 public boolean estaEncendido() {  
 return encendido;  
 }  
}

ackage Adapter;  
public class Lampara implements Enchufable{  
 boolean encendido = false;  
 @Override  
 public void enciende(){  
 encendido =true;  
 System.*out*.println("Lampara encendida");  
 }  
 @Override  
 public void apaga() {  
 encendido = false;

package Adapter;  
  
public class Lightbulb {  
 boolean poweredOn = false;  
 int voltage=0;  
  
 public void turnOn(int voltage) {  
 poweredOn = true;  
 this.voltage = voltage;  
 System.*out*.println("Lampara inglesa on");  
 }  
 public void turnOff() {  
 poweredOn = false;  
 voltage = 0;  
 }  
  
 public boolean isPoweredOn() {  
 return poweredOn;  
 }  
}

}package Adapter;  
  
public class PowerAdapter implements Enchufable{  
public Lightbulb lightbulb = new Lightbulb();  
  
 @Override  
 public void enciende(){  
 lightbulb.turnOn(110);  
  
 }  
  
  
 @Override  
 public void apaga() {  
 lightbulb.turnOff();  
  
 }  
  
 @Override  
 public boolean estaEncendido() {  
 return lightbulb.isPoweredOn(); }  
}

public class Main {  
 public static void main(String[] args) {  
 Horno horno= new Horno();  
 Lampara lampara = new Lampara();  
 PowerAdapter lightbulb = new PowerAdapter();  
 *enciende*(horno);  
 *enciende*(lampara);  
 *enciende*(lightbulb);  
  
 }  
public static void enciende (Enchufable enchufable) {  
 enchufable.enciende();  
}  
 public static void apaga (Enchufable enchufable) {  
 enchufable.apaga();  
 }  
 public static boolean estaEncendido (Enchufable enchufable) {  
 return enchufable.estaEncendido();  
 }  
}

3-PATRON FACADE

Es un envoltorio que simplifica algo complejo por detrás. Se oculta lo complejo poniendo por delante lo sencillo. Una fachada.

Código Fachada

public class Main {

public static void main(String []args) {

VideoConversionFacade converter = new VideoConversionFacade();

File mp4Video = converter.convertVideo("youtubevideo.ogg", "mp4");

}

}

Por detrás:

public class VideoConversionFacade {

public File convertVideo(String fileName, String format) {

System.out.println("VideoConversionFacade: conversion started.");

VideoFile file = new VideoFile(fileName);

Codec sourceCodec = CodecFactory.extract(file);

Codec destinationCodec;

if (format.equals("mp4")) {

destinationCodec = new OggCompressionCodec();

} else {

destinationCodec = new MPEG4CompressionCodec();

}

VideoFile buffer = BitrateReader.read(file, sourceCodec);

VideoFile intermediateResult = BitrateReader.convert(buffer, destinationCodec);

File result = (new AudioMixer()).fix(intermediateResult);

System.out.println("VideoConversionFacade: conversion completed.");

return result;

}

PATRONES DE COMPORTAMIENTO

Se ocupan de la comunicación entre diferentes objetos.

1-Patrón Iterator:

Implementa una serie de métodos en nuestra clase de forma que podamos recorrer esos datos sin necesidad de saber como están almacenados esos datos.

Interfaz con operaciones:

package Iterator;  
  
public interface UsuarioIterator {  
 boolean hayMas();  
 //Indica si hay mas usuarios en el iterador?  
  
 void reinicia();  
 // Reinicia el contador  
  
 Usuario siguiente();  
 //Obtiene el siguiente

Objeto Usuario.

public class Usuario {  
 private String nombre;  
 private int edad;  
 private Usuario(){};  
 public Usuario(String nombre, int edad) {  
 this.nombre = nombre;  
 this.edad = edad;  
 }  
 public String getNombre() {  
 return nombre;  
 }  
 public int getEdad() {  
 return edad;  
 }  
  
 }

Implementacion Interfaz

import java.util.ArrayList;  
  
public class Usuarios implements UsuarioIterator{  
private ArrayList<Usuario>usuarios = new ArrayList();  
private int posicion = 0;  
 public void crear (Usuario usuario){  
 usuarios.add(usuario);  
 }  
 @Override  
 public Usuario siguiente() {  
 Usuario usuario = usuarios.get(posicion);  
 posicion=posicion+1;  
 return usuario;  
 }  
  
 @Override  
 public boolean hayMas() {  
 return posicion < usuarios.size();  
 }  
  
 @Override  
 public void reinicia() {  
 posicion=0;  
 }  
}

Main:

import Iterator.Usuario;  
import Iterator.Usuarios;  
  
public class Main {  
 public static void main(String[] args) {  
  
 Usuarios usuarios = new Usuarios(); //Creo un Objeto de la clase Usuarios que van a la lista interna al crearlos  
 usuarios.crear(new Usuario("uno",5));  
 usuarios.crear(new Usuario("dos",25));  
 usuarios.crear(new Usuario("tres",10));  
 usuarios.crear(new Usuario("cuatro",8));  
  
 while (usuarios.hayMas()){  
 Usuario usuario = usuarios.siguiente();  
 System.*out*.println("Usuario es :" + usuario.getNombre());  
  
 }  
 usuarios.crear(new Usuario("cinco",18));  
 Usuario usuario = usuarios.siguiente();  
 System.*out*.println("Usuario es :" + usuario.getNombre());  
 usuarios.reinicia();  
 while (usuarios.hayMas()){  
 usuario = usuarios.siguiente(); System.*out*.println("Usuario es :" + usuario.getNombre());  
  
 }  
  
 }  
}

PATRON OBSERVER:

También llamado Public subsripter.

Cuando ocurre un evento los suscriptores reaccionan ante lo que ocurre.

En éste patrón uno habla y muchos escuchan.

public interface Receptor {  
 void recibe();  
}

En el ejemplo lo comenzamos a desarrollar con una interfaz (Receptor). Todos los que reciban el mensaje deberán implementarla.

Comienzo con la interfaz porque debo ejecutar el método de los receptores en mi patrón. Lo implementaremos creando una emisora. La misma contendrá un array list de mis receptores, ya que necesito saber quienes son.

Para notificar a todos mis receptores tengo en cuenta que :

* Todos están en mi arraylist
* Todos implementan el método recibe().

Entonces voy a recorrer mi lista y por cada receptor invocar a su método emite. Lo hacemos con un for each

public class Emisora {  
 //Arraylist dónde contengo los receptores  
 private ArrayList<Receptor> receptor = new ArrayList<Receptor>();  
  
 //Función para crear los receptores  
 public void meteReceptor(Receptor receptor) {  
 this.receptor.add(receptor);  
 }  
 //Metodo para notificar a todos mis receptores  
 //Aqui viene la implementación  
 public void emite() {  
 for (Receptor receptor : receptor){  
 receptor.recibe();  
 }  
 }  
}

Vamos a implementar 3 receptores:

public class ReceptorSatelite implements Receptor{  
 @Override  
 public void recibe() {  
 System.*out*.println("Señal recibida en Satélite");  
 }  
}

El receptor implementa la interfaz Receptor.

En el método recibe() va lo que quiero que ocurra cuando reciba la motificación. En éste caso al ser un ejemplo solo imprimirá “Señal recibida en”.

public class ReceptorRadio implements Receptor {  
 @Override  
 public void recibe() {  
 System.*out*.println("Señal recibida en radio");  
 }  
}

public class ReceptorTV implements Receptor{  
  
 @Override  
 public void recibe() {  
 System.*out*.println("Señal recibida en TV ");  
 }  
}

Para ver como lo utilizaremos implementaremos el siguiente main:

public class Main {  
 public static void main(String[] args) {  
 Emisora emisora = new Emisora();  
 ReceptorTV receptorTV = new ReceptorTV();  
 ReceptorRadio receptorRadio = new ReceptorRadio();  
 ReceptorSatelite receptorSatelite = new ReceptorSatelite();  
  
 emisora.meteReceptor(receptorTV);  
 emisora.meteReceptor(receptorRadio);  
 emisora.meteReceptor(receptorSatelite);  
  
  
 emisora.emite();  
 }  
}

En el main Instanciamos la emisora y los receptores.

Invocamos la función meteReceptor de la clase emisora por cada receptor para generar el arraylist de éstos últimos.

Al llamar al método emite de la clase emisora, éste recorre el arrarylist y activa a cada receptor.

En el mundo real es que tengamos un sistema de registro de usuarios y mediante el patrón observer cuando el usuario se registre mandemos una señal al sistema de base de datos o al se registros.

También podríamos tener un sistema se notificCIONES.

PATRON MEDIATOR

Actúa de intermediario entre la comunicación entre los objetos. El mensaje el objeto A se lo pasa al mediador, y el mediador se lo pasa al objeto B.

Es útil cuando hay muchas clases que se tienen que comunicar entre éllas. Se utiliza muchísimo en aplicaciones de escritorio.

Proceso.

1. Se crea la clase colega como abstracta y se define dentro de ésta una variable del tipo Mediator llamada mediator. Se define el método setMediator. Luego se definen dos funciones abstractas (Envía y recibe)

}abstract public class Colega {  
 public Mediator mediator;  
 public void setMediator(Mediator mediator){  
 this.mediator = mediator;  
 abstract void recibe();  
 abstract void envia();  
 }  
}

1. Creo la clase abstracta Mediator

Tiene las funciones registra: que registra un hablante en el mediador.

Reenvia: Toma el mensaje enviado y lo pasa.

Hay un tipo de dato Colega colega.

abstract public class Mediator {  
 abstract void registra(Colega colega);  
 abstract void reenvia (Colega colega);  
}

3-Para poder instanciar debo extender las clases ya que estas son abstractas.

Se crea el MediadorConcreto con los respectivos métodos.

Se crea un Arraylist del tipo Colega.

El mediador recibe el mensaje de A y lo reenvía a B,C y D. ´Pero no puede mandárselo a A nuevamente.

Por lo tanto debo evitar duplicidades. Por eso uso el if.

public class MediadorConcreto extends Mediator{  
 ArrayList<Colega>colegas= new ArrayList<>()  
 @Override  
 void registra(Colega colega) {  
 if (!colegas.contains(colega)){  
 colegas.add(colega);  
 colega.setMediator(this);  
 }  
 }  
 @Override  
 void reenvia( Colega colega) {  
 for (Colega actual: colegas) {  
 if (!actual.equals(colega)) {  
 actual.recibe();  
 }  
 }  
 }  
}

Luego creo los participantes de la conversación

ColegaConcreto1,2 y 3 que extiende a Colega.

Debo implementar escucha () y habla().

public class ColegaConcreto1 extends Colega{  
 @Override  
 void recibe() {  
 System.*out*.println("He recibido un mensaje, Soy colega concreto1");  
 }  
 @Override  
 void envia() {  
 System.*out*.println("Soy colega concreto 1, envío mensaje");  
 mediator.reenvia(this);  
 }  
}

En el main:

Creo las instancias de colegas.

Los registro

public class main {  
 public static void main(String[] args) {  
 Mediator mediador = new MediadorConcreto();  
 Colega colega1 = new ColegaConcreto1();  
 Colega colega2 = new ColegaConcreto2();  
 Colega colega3 = new ColegaConcreto3();  
 mediador.registra(colega1);  
 mediador.registra(colega2);  
 mediador.registra(colega3);  
  
 colega1.envia();  
 }}

Se usa esencialmente en interfaces de usuario. Pulso el ok y cambian los fondos por ejemplo.

Su objetivo es comunicar clases.

PATRON STATE:

Se utiliza para crear WorkFlows o flujos de trabajo.

Que si no pasa una cosa no pueda pasar otra.

Ejemplo una máquina de refrescos.

Vamos a implementarlo como un teléfono móvil.

Creo la clase abstracta Estado que tiene 3 métodos abstractos.

abstract public class Estado {  
 public Telefono telefono;  
 public Estado(Telefono telefono){  
 this.telefono = telefono;  
 }  
 abstract public String desbloquear();  
 abstract public String abrirCamara();  
 abstract public String hacerFoto();  
 }

Creo los 3 estados del teléfono:

Estado Bloqueado, Estado Desbloqueado, EstadoCamaraAbierta y Estado HacerFoto.

Por cada estado Extiendo la clase base Estado.

ublic class EstadoBloqueado extends Estado{  
public EstadoBloqueado (Telefono telefono){  
 super(telefono);  
}  
 @Override  
 public String desbloquear() {  
 telefono.cambiaEstado(new EstadoDesbloqueado(telefono));  
 return "Desbloquear() : Movil desbloqueado. Procede";  
 }  
  
 @Override  
 public String abrirCamara() {  
 return "abrir camara(): La cámara está bloqueada, desbloquea el movil antes";  
 }  
  
 @Override  
 public String hacerFoto() {  
 return "hacerFoto() : La camara está bloqueada. Desbloquea el movil antes";  
 }  
}

public class EstadoCamaraAbierta extends Estado {  
 public EstadoCamaraAbierta (Telefono telefono){  
 super(telefono);  
 }  
 @Override  
 public String desbloquear() {  
 return "DESBLOQUAEAR,() Ya estaba desbloqueado el movil";  
 }  
  
 @Override  
 public String abrirCamara() {  
 return "ABRIR CAMARA : La camara ya estaba abierta";  
 }  
  
 @Override  
 public String hacerFoto() {  
 telefono.cambiaEstado(new EstadoHacerFoto((telefono)));  
 return "La foto se va a disparar ya.....";  
 }  
}

public class EstadoHacerFoto extends Estado {  
 public EstadoHacerFoto (Telefono telefono){  
 super(telefono);  
 }  
 @Override  
 public String desbloquear() {  
 return "Movil desbloqueado";  
 }  
  
 @Override  
 public String abrirCamara() {  
 return "Camara abierta";  
 }  
  
 @Override  
 public String hacerFoto() {  
 telefono.cambiaEstado(new EstadoDesbloqueado(telefono));  
 return "La foto se ha hecho";  
 }  
}

Creo el objeto Teléfono.

En ésta clase vamos a guardar el estado actual

public class Telefono {  
 public Estado estado;  
 public Telefono(){  
 estado = new EstadoBloqueado(this);  
  
 }  
 public void cambiaEstado (Estado estado){  
 System.*out*.println("Estado Inicial"+ this.estado.getClass().getName());  
 this.estado = estado;  
 System.*out*.println("Estado Final" + this.estado.getClass().getName());  
 }  
  
 public Estado getEstado() {  
 return estado;  
 }  
}

public class main {  
 public static void main(String[] args) {  
 Telefono tel1 = new Telefono();  
 System.*out*.println(tel1.estado.desbloquear());  
 System.*out*.println(tel1.estado.hacerFoto());  
 System.*out*.println(tel1.estado.abrirCamara());  
  
 }  
}

Éste patrón se utiliza mucho en máquinas de Bending.

PATRON STRATEGY.

Se utiliza para crear un envoltorio sobre varias funciones de manera que un mismo código de pueda utizar de manera diferente. La forma de verlo es pensar en un método de pago en una tienda on line. Donde se pueda pagar con Paypal o visa. El método podría ser pagar pero se implementaría de diferente manera según el pago.

Es uno de los patrones mas utilizados por Victor. Sobre todo cuando hay que generar varias implementaciones de lo mismo.

public interface Usuarios {  
 void crear (String nombre);  
 ArrayList<String> listar();  
}

public class UsuariosFichero implements Usuarios{  
 private PrintStream fichero;  
 private String ficheroUsuarios = "usuarios.txt";  
  
 public UsuariosFichero() {  
 try {  
 fichero = new PrintStream(ficheroUsuarios);  
  
 } catch (Exception e) {  
 System.*out*.println("No puedo abrir el fichero" + e.getMessage());  
 }  
 }  
 @Override  
 public void crear(String nombre) {  
 fichero.println(nombre);  
 }  
 @Override  
 public ArrayList<String> listar() {  
 ArrayList<String> usuarios = new ArrayList<>();  
 try {  
 Scanner scanner = new Scanner(new File(ficheroUsuarios));  
 while (scanner.hasNext()) {  
 usuarios.add(scanner.next());  
 }  
 scanner.close();  
 } catch (Exception e) {  
 System.*out*.println("Error leyendo" + e.getMessage());  
 }  
 return usuarios;  
 }  
}

public class UsuariosMemoria implements Usuarios {  
 private ArrayList<String> usuarios = new ArrayList<>();  
  
 @Override  
 public void crear (String nombre){  
 usuarios.add(nombre);  
 }  
  
 @Override  
  
 public ArrayList<String> listar(){  
 return usuarios;  
 }  
}

public class main {  
 public static void main(String[] args) {  
 UsuariosMemoria usuariosM = new UsuariosMemoria();  
 UsuariosFichero usuarios =new UsuariosFichero();  
 *crear*(usuarios,"fulano");  
 *crear*(usuarios,"mengano");  
 *crear*(usuarios,"pepe");  
 for (String usuario: *listar*(usuarios)){  
 System.*out*.println(usuario);  
 }  
 }  
 public static void crear (Usuarios usuarios,String nombre){  
 usuarios.crear(nombre);  
 }  
 public static ArrayList<String> listar (Usuarios usuarios){  
 return usuarios.listar();  
 }  
}

En el main se podría usar un if que instancie el tipo de usuario con el que quiero trabajar (Memoria o fichero de texto)

Refactoring en Java

Clean code (Código limpio).

Quiere decir código limpio. Es aquel que cualquier desarrollador pude leer y cambiar fácilmente. Es muy difícil escribir éste tipo de código.

Tiene 3 propiedades:

* Focalizado:Un componente se escribe para solucionar un problema concreto.
* Simple.
* Comprobable. Fácilmente verificable

La refactorización es el proceso de cambiar la estructura del software haciéndolo mas simple, mas claro sin alterar el comportamiento.

No se añade nueva funcionalidad cuando se refactoriza.

Técnicas habituales de Refactoring

Read Greenl O test drive developmen.

Tiene 3 estados:

Escribo código de prueba. Pruebas que van a fallar.

Si he cumplido con el test y está en verde debo refactorizarlo.

También está la técnica de absracción.

En el ejemplo luego de escribir el código que muestra de un array de 5 números el mayor y el menor, empezamos a utilizar el operador ternario y la forma corta del IF:

mayor = mayor < numeros[i] ? numeros[i] :mayor;

condición que se evalua.

Si mayor es < números[i]

Condicion Verdadera toma éste valor

Condicion Falsa toma éste valor

Si es verdadero mayor toma el primer valor luego del signo de interrogación (números [i]).

Sino toma el valor luego de los dos puntos (mayor)

Ejemplo de Refactorización en 4 pasos:

1-

public class RefactorMain {  
 public static void main(String[] args) {  
 int[] numeros ={12,23,-45,1,9};  
 *imprimeMayorYMenor*(numeros);  
 int mayor=0;  
 int menor = numeros[0];  
 for (int i=1; i<numeros.length; i++){  
 if (mayor<numeros[i]){  
 mayor=numeros[i];  
  
 }  
 if (menor > numeros[i]){  
 menor=numeros[i];  
  
 }  
  
 }  
 System.*out*.println( "Mayor es: " + mayor + " Menor es: " + menor);  
 }  
 public static void imprimeMayorYMenor(int[] numeros) {  
 }  
}

2-

public class RefactorPaso2 {  
  
 //Reemplazo los If por la forma corta  
 public static void main(String[] args) {  
 int[] numeros ={12,23,-45,1,9};  
 *imprimeMayorYMenor*(numeros);  
 int mayor=0;  
 int menor = numeros[0];  
 for (int i=1; i<numeros.length; i++){  
  
 mayor = mayor < numeros[i] ? numeros[i] :mayor;  
 menor = menor > numeros[i] ? numeros[i]: menor;  
  
 }  
 System.*out*.println( "Mayor es: " + mayor + " Menor es: " + menor);  
 }  
 public static void imprimeMayorYMenor(int[] numeros) {  
 }  
}

3-

public class Refactor3 {  
 // Refactorizo el For  
 public static void main(String[] args) {  
 int[] numeros ={12,23,-45,1,9};  
 *imprimeMayorYMenor*(numeros);  
 int mayor=0;  
 int menor = numeros[0];  
 for (int i :numeros ){  
  
 mayor = mayor < i ? i :mayor;  
 menor = menor > i ? i: menor;  
  
 }  
 System.*out*.println( "Mayor es: " + mayor + " Menor es: " + menor);  
 }  
 public static void imprimeMayorYMenor(int[] numeros) {  
 }  
}

4-

/Utilizo programación funcional  
public class Refactor4 {  
 public static void main(String[] args) {  
 int[] numeros ={12,23,-45,1,9};  
 *imprimeMayorYMenor*(numeros);  
 int mayor= Arrays.*stream*(numeros)  
 .reduce(0, (a, b) -> a > b ? a:b);  
 int menor= Arrays.*stream*(numeros)  
 .reduce(numeros[0], (a,b) -> a < b ? a:b);  
 System.*out*.println( "Mayor es: " + mayor + " Menor es: " + menor);  
 }  
 public static void imprimeMayorYMenor(int[] numeros) {  
  
  
 }  
}

Técnica de extracción del valor de retorno:

1-

public class refactorIf {  
 public static void main(String[] args) {  
 *ifAnidado*();  
 }  
 public static int ifAnidado() {  
 int valor1 =10;  
 int valor2 =20;  
  
 if (valor1>valor2){  
 return valor1;  
 } else {  
 return valor2;  
 }  
 }  
}

2-

//EL else no es necesario (Extracción del valor de retorno  
public class Refactorif2 {public static void main(String[] args) {  
 *ifAnidado*();  
}  
 public static int ifAnidado() {  
 int valor1 =10;  
 int valor2 =20;  
  
 if (valor1>valor2){  
 return valor1;  
 }  
 return valor2;  
 }  
}

CON BUILDER

Constructor largo Builder

String motor;  
 int puertas;  
  
 private ConstructorLargoBuilder(){ }  
 public ConstructorLargoBuilder(String marca){  
 this.marca = marca;  
 }  
 public ConstructorLargoBuilder conModelo(String modelo){  
 this.modelo = modelo;  
 return this;  
 }  
 public ConstructorLargoBuilder conMotor(String motor){  
 this.motor = motor;  
 return this;  
 }  
 public ConstructorLargoBuilder conPuertas(int puertas){  
 this.puertas = puertas;  
 return this;  
 }  
}

public class main {  
 public static void main(String[] args) {  
 ConstructorLargoBuilder builder = new ConstructorLargoBuilder("Ford")  
 .conModelo("Focus")  
 .conMotor("Electrico ")  
 .conPuertas(4);  
  
 }  
}

SIN BUILDEER

public class main1 {

public static void main(String[] args) {

ConstructorLargo cosa =new ConstructorLargo("Audi ","Beta","C ",4);

}

}

public class ConstructorLargo {  
 String marca;  
 String modelo;

String motor;  
 int puertas;  
 public ConstructorLargo (String marca,String modelo,String motor,int puertas) {  
 this.marca = marca;  
 this.modelo = modelo;  
 this.motor = motor;  
 this.puertas = puertas;  
  
 }  
}

Utilizando patrones de diseño para aplicar refactoring

In Laining ?

Consiste en mover parte o toda la función a la función padre.

public class main {  
 public static void main(String[] args) {  
   
 }  
 public boolean hoyEsDomingo(){  
 boolean isDomingo = *hoyEsDiaNumero*(7);  
 if(isDomingo){  
 System.*out*.println(isDomingo);  
 }return isDomingo;  
  
 }  
   
 public static boolean hoyEsDiaNumero(int numero){  
 if (numero==7 || numero==7){  
 return true;  
 }  
 return false;  
 }  
}

public class main2 {  
 public static void main(String[] args) {  
 }  
 public void hoyEsDomingo(int numero){  
 boolean isDomingo = numero ==7 ? true: false;  
 if(isDomingo){  
 System.*out*.println(isDomingo);  
 }  
  
 }  
  
 }

Método de extracción de variables:

ublic class main {  
 public static void main(String[] args) {  
  
 }  
 public boolean esJavaEnMacOs(){  
 if (System.*getProperty*("os.name").contains("Oracle")  
 && System.*getProperty*("os.version").compareTo("11.6")== 0){  
 return true;  
 }return false;  
 }  
}

public class MAin2 {  
 public static void main(String[] args) {  
 *esJavaEnMacOs*();  
 }  
 public static boolean esJavaEnMacOs(){  
 boolean runtimeOracle =System.*getProperty*("os.name").contains("Oracle");  
 boolean osVersion = System.*getProperty*("os.version").compareTo("11.6") ==0 ? true : false;  
  
 if (runtimeOracle && osVersion){  
 System.*out*.println("Es Oracle JM\_V en MAC 11.6");  
 return true;  
 }return false;  
 }  
}

NOMBRADO:

Consiste en dar un nombre claro a aquello que se lo otorgamos.

Un buen nombre identifica claramente para que sirve lo que estamos nombrando.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Elemento | Nombre | Características | Ejemplo |  |  |
| Clases | Sustantivo | Comienza con Mayuscula | Class ReverseString{}  Class UserManagger{} |  |  |
| Interfaces | Adjetivos | En inglés que termine en able | interface Iterable{}  interface Serializable {} |  |  |
|  |  | En Castellano terminan en “ador”. Son adjetivos. | interface Iterador{}  interface Serializador{} |  |  |
| Métodos | Comienzan con verbo en letra minúscula indicando para que sirve | Las palabras siguientes en mayúscula | Public void createUser |  |  |
| Constantes | Se pueden usar nemotécnicos | En mayúsculas. Las palabras no se juntan, se separan con guión bajo. | MAX\_USER =15 |  |  |
| Enumeraciones |  | Ídem constantes |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Las variables deben usar nombres autodescriptivos.

Se escriben en minúscula y se capitaliza la inicial de las palabras siguientes.

Dentro de los bucles for se utilizan letras estándar i,j,k,etc. Fuera del bucle no utilizar éstos nombres.

NOTACIÓN HUNGARA:

Se anteponen de 1 a 3 letras antes de una variable para ayudarnos a identificar el tipo de la misma.

|  |  |  |  |
| --- | --- | --- | --- |
| Tipo de Varrible | Letra a anteponer | Ejemplo |  |
| Array de enteros | a | int []aNumeros = {1,2,3,4,5} |  |
| Boolean | b | boolean bEsDeDia=false |  |
| Char | c | char cletra = ´c ´ |  |
| Double | d | dValor = 5.1d |  |
| Hashtable | h | Hashtable <String,Integer> hUsuarios= new HashTable() |  |
| Int | i | iValores = 5 |  |
| Long | i | iValoresLong =5 |  |
| Enum | n | enum nEstado{  INICIAL,  EN\_PROGRESO  } |  |
| Objetos en general | o | User Manager oUserManager= new UserManager() |  |
| Cadenas de texto | s | String sCadena = “Hola” |  |
| Variables genéricas | v | var vVariable=5 |  |
| Byte | y ó by | byte byByte |  |
| Float | f o fl | Float flFloat=1,5f |  |
|  |  |  |  |
| JButton | btn | JButton btnBotonAceptar |  |
| Jlabel | lbl | JLabel lblLabel |  |
|  |  |  |  |
| Array de enteros | ai | int []aiNumeros = {1,2,3,4,5} |  |
| Array de String | as | String [] asnombres = {“Sergio”,”Mari”} |  |
| Array de Flotantes | af |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Un concepto una palabra.

Ejemplo:

Si tengo una clase que se llama casa, no podría tener dentro un constructor de coche.

Un error común es definir una variable y luego reutilizarla para cualquier otra cosa.

FUNCIONES:

Es una agrupación de código que debe realizar una tarea concreta. Las funciones tienen como objetivo la reutilización de código. En la POO, como es en Java, se habla indistintamente de funciones y de métodos, salvo en las “funciones landa”.

Los puristas afirman que cualquier función dentro de una clase debe llamarse método.

Largo de una función:

Una función no debe ser excesivamente larga ni excesivamente corta, deben ser proporcionales a la tarea que se realiza.

Una función excesivamente corta sería aquella que puedo generar en una línea (in line).

Ejemplo de Función Absurda:

public class Funciones {  
 public static void main(String[] args) {  
 boolean mayor = *esMayor*(4,2);  
 }  
 private static boolean esMayor(int a,int b){  
 if (a > b){  
 return true;  
 }return false;  
 }   
}

El cometido de esta función puede refactorizarse en una línea.

private static boolean esMayor(int a,int b) {  
 return a > b ? true : false;

Por lo tanto usamos el in laining (Extraigo el cuerpo de una función y lo sustituyo por el bloque que la llama).

public class Funciones {  
 public static void main(String[] args) {  
 boolean mayor = 4 > 2 ? true:false;  
 }  
}

Largo de las funciones:

No debemos crear funciones tan grandes que no sean gestionables.

Una función grande es aquella que implementa la lógica de una clase en forma repetitiva o difícil de gestionar.

Tamaño ideal:

El número de líneas necesario para que la función cumpla su propósito sin que se pueda sacar el código fuera de la función y sin que tenga que partirla en funciones mas pequeñas.

Generalmente una función larga al refactorizarla la convertimos en varias funciones mas pequeñas.

Una funcionalidad por función

Regla descendente:

Cuando mas niveles por debajo (cuanto mas funciones llamen a funciones) cuanto mas abajo mas compleja debe ser la función.

La primera función que llama de mi main debería ser simple.

La segunda de tipo medio.

La tercera mas compleja…….

Tratar de no caer en :

public class Funciones {  
 public static void main(String[] args) {  
 *a*();  
 }  
 public static void a(){  
 *b*();  
 }  
 public static void b(){  
 *c*();  
 }  
 public static void c(){  
 *d*();  
 }  
 public static void d(){  
 *e*();  
 }  
 public static void e(){  
 System.*out*.println("Estoy en e");  
 }  
  
  
}

ya que en éste tipo de estructura se va llenando la pila y esto insume tiempo computacional.

Abuso de sentencias de control tipo swich.

Ejemplo:

public class Funciones {  
 public static void main(String[] args) {  
 *esLaborable*(3);  
 }  
  
 public static boolean esLaborable(int dia) {  
 switch (dia) {  
 case 1:  
 return true;  
 break;  
 case 2:  
 return true;  
 break;  
 case 3:  
 return true;  
 break;  
  
 case 4:  
 return true;  
 break;  
  
 case 5:  
 return true;  
 break;  
 case 6:  
 return false;  
 break;  
 case 7:  
 return false;  
 break;  
 case 0:  
 return false;  
 break;  
  
 }  
  
 }  
  
}

Lo primero que debo preguntarme es si debo usar un swich y lo segundo es si tiene que ser tan largo.

En éste caso se puede refactorizar.

public class Funciones {  
 public static void main(String[] args) {  
 *esLaborable*(3);  
 }  
  
 public static boolean esLaborable(int dia) {  
 switch (dia) {  
 case 1:   
 case 2:  
 case 3:  
 case 4:  
 case 5:  
 return true;  
 break;  
 case 6:  
 case 7:  
 case 0:  
 return false;  
 break;  
  
¿Es necesario éste swich? NO:

La función puede reemplazarse por:

public static boolean esLaborable(int dia) {  
 if (dia>0 && dia <6){  
 return true;  
 }  
 return false;  
}

A su vez se puede refactorizar en una línea

public static boolean esLaborable(int dia) {  
 return (dia>0 && dia <6){

por lo tanto podría utilizar inlaining.

Cuando usar Swich:

Cuando tenemos 2 o mas else if . Ejemplo:

public static void demoSwich(){  
 String sFruta = "Manzana";  
 if (sFruta == "Pera") {  
 System.*out*.println("Es una pera");  
 }else if (sFruta == "Melocoton") {  
 System.*out*.println("Es un Melocoton");  
 }else if (sFruta == "banana") {  
 System.*out*.println("Es una banana");  
 }else if (sFruta == "Manzana") {  
 System.*out*.println("Es una Manzana");  
 } else {  
 System.*out*.println("Es otra cosa");}  
  
}  
}

En éste caso con un swich hubiese sido mas adecuado.

También se puede usar en el patrón Factory.

Cuando se construye un compilador se utilizan mucho los swich por claridad del código.

Importancia del nombrado de las funciones:

El nombre demoSwich es incorrecto. Sería correcto “determinaFruta”, comienza con un verbo.

Los argumentos de una función también deben ser bien nombfrados.

Ejemplo incorrecto: determinaFruta(Srring cosa).

La forma correcta: daterminarFruta(String sFruta).

Los buenos parámetros deben seguir las convenciones estándar de los lenguajes.

En los argumenos de las funciones Victor no utiliza nomenclatura Húngara.

Las funciones que reciben un solo parámetro de entrada se llaman funciones monádicas.

public static int sumarNumeros(int []numeros){  
 int resultado = 0;  
 for (int numero:numeros){  
 resultado+=numero;  
 }  
 return resultado;  
}

Las funciones monádicas son utilizables a nivel de métodos funcionales.

Poe ejemplo la función anterior podría haberse hecho de la siguiente manera:

int resultado= Arrays.*stream*(numeros).reduce(0,(a,b)->a +b);

Las funciones de dos parámetros se llaman diádicas.

Las funciones de 3 parámetros son triádicas.

Poliádicas: 4 o mas argumentos.

Victor como mucho usa 2 o 3 parámetros.

Cuando tiene que usar 4 o mas valores lo haremos de otra forma:

Argumentos variables: Cuando tenemos varios argumentos de un mismo tipo en una función, podemos utilizar la sintaxis de argumento variable.

Imaginemos un caso en el que tengo que sumar múltiples enteros, pero no se la cantidad de los mismos.

Para esto se antepone “…” delante del argumento. Internamente es como si le pasara un array.

public static int sumarInfinitamente(int ...numeros){  
 int resultado =0;

for (int numero:numeros){  
 resultado+=numero;  
 }  
 return resultado;  
  
}

Puedo llamar a la función con la cantidad de argumentos que quiera.

Supongamos que tengo el siguiente modelo:

public class Usuario {  
 public String nombre;  
 public String apellido;  
 int edad;  
}

Luego: Tengo el objeto Usuarios donde la función añadirUsuario permite añadirlos a un Arraylist

public class Usuarios {  
 private ArrayList<Usuario>usuarios= new ArrayList<>();  
 public void anadirUsuario(String nombre,String apellido, int edad ){  
 Usuario usuario =new Usuario();  
 usuario.nombre= nombre;  
 usuario.apellido = apellido;  
 usuario.edad= edad;  
 usuarios.add(usuario);  
 }  
}

y en el MAin

Usuarios usuarios = new Usuarios();  
usuarios.anadirUsuario("Pepe", "Botella", 14);  
usuarios.anadirUsuario("Juana","laroca",20);

Luego con el tiempo mi modelo cambia con una nueva variable boolean puedeConducir.

Al cambiar el modelo, debo cambiar la clase Usuarios.

Y al cambiar la clase, debo modificar el main.

Para evitar esto en el main podría solo pasar un objeto a la función.

Usuarios usuarios = new Usuarios();  
Usuario usuario = new Usuario();  
usuario.nombre="pepe";  
usuario.apellido ="botella";  
usuarios.anadirUsuario(usuario);

public class Usuarios {  
 private ArrayList<Usuario>usuarios= new ArrayList<>();  
 public void anadirUsuario(Usuario usuario ){  
 usuarios.add(usuario);  
 }  
}

De ésta manera queda una función mas limpia.

Si cambio el modelo, no tengo que cambiar la función porque le estoy pasando el objeto directamente.

Funciones y trows:

Cuando estoy implementando funciones no se debe llenar todo de println. Si algo falla uso el depurador.

Si sigue fallando dejo que el main opere sobre los errores. Por lo tanto utilizaré trows en la función para que arroje la excepción.

En el cuerpo de la función agrego trows y el nombre de la excepción. En éste caso **throws UserExcepcion**

La misma aparece en rojo porque no existe, por lo tanto la crearé utilizando una nueva clase que llamo UserExcption que extiende a su clase padre Exception.

Tiene un constructor con un mensaje que se lo pasa al padre.

De ésta manera genero una excepción personalizada

public class UserException extends Exception{  
 public UserException (String message){  
 super("Ususrio ya existe" + message);  
 }  
}

Ahora gestionamos la excepción:

public class Usuarios {  
 private ArrayList<Usuario>usuarios= new ArrayList<>();  
 public void anadirUsuario(Usuario usuario )throws UserException {  
 if (usuarios.contains(usuario)){  
 throw new UserException(usuario.nombre);  
 }  
 usuarios.add(usuario);  
 }  
}

Luego para que funcione implemento los bloques try catch en el main.

try {usuarios.anadirUsuario(usuario);  
} catch (UserException e ){  
 System.*out*.println("Error al añaddir el objeto de usuario" + e.getMessage());  
}

Se puede generar una jerarquía de excepciones.

Por ejemplo:

public class UserRegisteredException extends UserException{  
 public UserRegisteredException (String message){  
 super("Usuario ya existe" + message);  
 }  
}

public class UserException extends Exception{  
 public UserException (String message){  
 super("User Exception" + message);  
 }  
}

public void anadirUsuario(Usuario usuario )throws UserException {  
 if (usuarios.contains(usuario)){  
 throw new UserRegisteredException(usuario.nombre);  
 }  
 usuarios.add(usuario);  
}

MAIN

try {  
 usuarios.anadirUsuario(usuario);  
 usuarios.anadirUsuario(usuario);  
} catch (UserException e ){  
 System.*out*.println("Error al añaddir el objeto de usuario" + e.getMessage());  
}

En pantalla la excepción sería:

Error al añaddir el objeto de usuario User Exception Usuario ya existe pepe

También puedo modificar la función para que reciba varios usuarios:

public class Usuarios {  
 private ArrayList<Usuario> usuarios = new ArrayList<>();  
  
 public void anadirUsuarios(Usuario... usuarios) throws UserException {  
 for (Usuario usuario : usuarios) {  
 if (this.usuarios.contains(usuario)) {  
 throw new UserRegisteredException(usuario.nombre);  
 }  
 this.usuarios.add(usuario);  
 }  
 }  
}

En el main:

public static void main(String[] args) {  
  
 Usuarios usuarios = new Usuarios();  
 Usuario usuario = new Usuario();  
 usuario.nombre="pepe";  
 usuario.apellido ="botella";  
 usuario.edad = 15;  
 usuario.puedeConducir= false;  
 Usuario usuario2 = new Usuario();  
 usuario2.nombre="Lali";  
 usuario2.apellido ="pil";

usuario2.edad = 22;  
 usuario.puedeConducir= false;  
 try {  
 usuarios.anadirUsuarios(usuario,usuario2);  
 } catch (UserException e ){  
 System.*out*.println("Error al añaddir el objeto de usuario" + e.getMessage());  
 }

Documentación a través de comentarios.

Un comentario de calidad se coloca en el lugar correcto, se explica con claridad, explica lo que viene a continuación y no es redundante no obvio.

Comenterio TODO:

Indica algo pendiente por hacer y que no se a hecho aún.

Ejemplo:

// TODO: mejorar el código con buenos comentarios.

Comentario informativo que documenta el código:

*/\*\*  
La clase User Exception gestiona las excepciones   
generadas por el gestor de usuarios cuando éste ya existe  
\*/*

Comentario: legal.

Por ejemplo la licencia si utilizamos open source.

/\*

\*Copyright ©

\*/

Comentario de intencionalidad: Describen el objetivo de nuestro código.

Ejemplo:

*/\*\* La clase Usuarios implementa diversos métodos ppara la gestión del   
 \* modelo usuario. Provee diversas funciones….*

*\*implementa el patrón Iterator.*

*\*El objetivo de ésta clase es sustituir a la vieja clase UsusarioManager tras una refactorización*

*\*Como consecuencia …..  
 \*/*

Comentario malo.

Comentario Redundante

.

Comentario Obligatorio. Cuando tenfamos una pieza de código que puede dar lugar a confusión.

Marcadores:no usar.

//-------

Muchos lo utilizan para encontrar el código.

Comentario fin llave.

}// Fin if x

No usar. No son buena práctica.

Mención: @SGaraycochea.

Comentario de encabezado de función: describen claramente lo que va a hacer la función, tipo de parámetro que acepta y tipo de parámetro que recibe.

//Añade un nuevo usuario a la lista de usuarios.  
//  
//El parámretro de entrada es variable, del tipo Usuario.  
//  
//Si el usuario existe (la propiedad name del objeto existe), entonces la funcion generará una  
//excepción del tipo UserRegisterException

Pero existen convenciones de como hacerlo.

En java JAvadoc

Éstos bloques en Java comienzan con /\*\*

Al oprimir la tecla enter me rellena alguna cosas

*/\*\*  
 \* Añade un nuevo usuario a la lista  
 \*   
 \** ***@param*** *usuarios Uno o varios objetos del tipo usuario  
 \** ***@throws*** *UserException Si el usuario ya está registrado.  
 \*/*

Comentarios de variables.

*/\*\*Lista de usuarios \*/*private ArrayList<Usuario> usuarios = new ArrayList<>();

@see Indica sitio web

@since : indica fecha

Comenta el código es documetarlo