Orientación a Objetos II

Sesión Conceptual 02

{desafío} latam_

- Objetivo de la sesión
- Activación de conceptos clave

- Conceptualización
- Ejercicios
- Quiz

- Cierre

Inicio

{desafío}
latam_

Objetivo

- Comprender las clases abstractas para generar herencia.
- Implementar polimorfismo para los principios de POO mediante herencia.

{desafío} latam_

Desarrollo

{desafío}
latam_

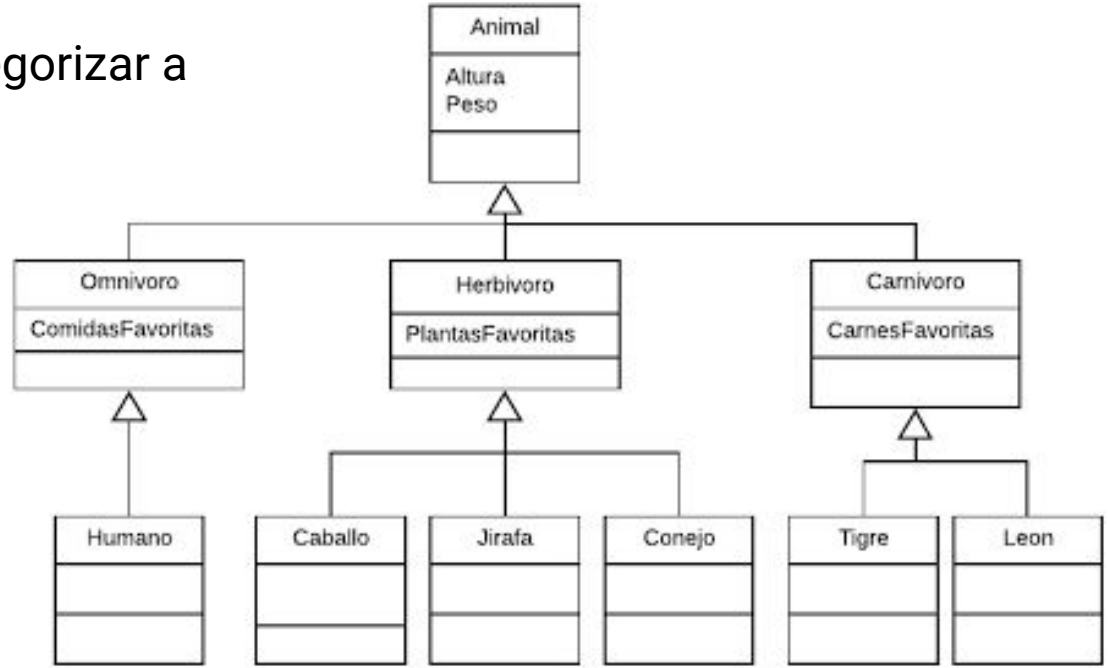
Abstracción II

Las clases abstractas

Según el filósofo José Ortega y Gasset podemos entender por sustantivo abstracto a aquella palabra que nombra un objeto que **no es independiente**, o sea, que siempre necesita otro elemento en el que apoyarse para poder ser. Esto significa que dichos sustantivos, al no referirse a un elemento concreto, hacen referencia a objetos que no se pueden percibir con los sentidos, sino imaginarse.

Las clases abstractas

- Clases que permiten categorizar a otras
- Clases que no pueden instanciarse



Creando clases abstractas

```
public abstract class Animal {
     private int altura;
     private int peso;

     public int getAltura() …
     public void setAltura(int altura) …
     public int getPeso() …
     public void setPeso(int peso) ...
}
```

```
public abstract class Carnivoro extends Animal{
     List<String> carnesFavoritas;

     public List<String> getCarnesFavoritas() {...}

     public void setCarnesFavoritas(List<String> carnesFavoritas) {...}
}
```

Terminando el árbol de herencia

```
public class Tigre extends Carnivoro{

}
```

```
public class Leon extends Carnivoro{

}
```

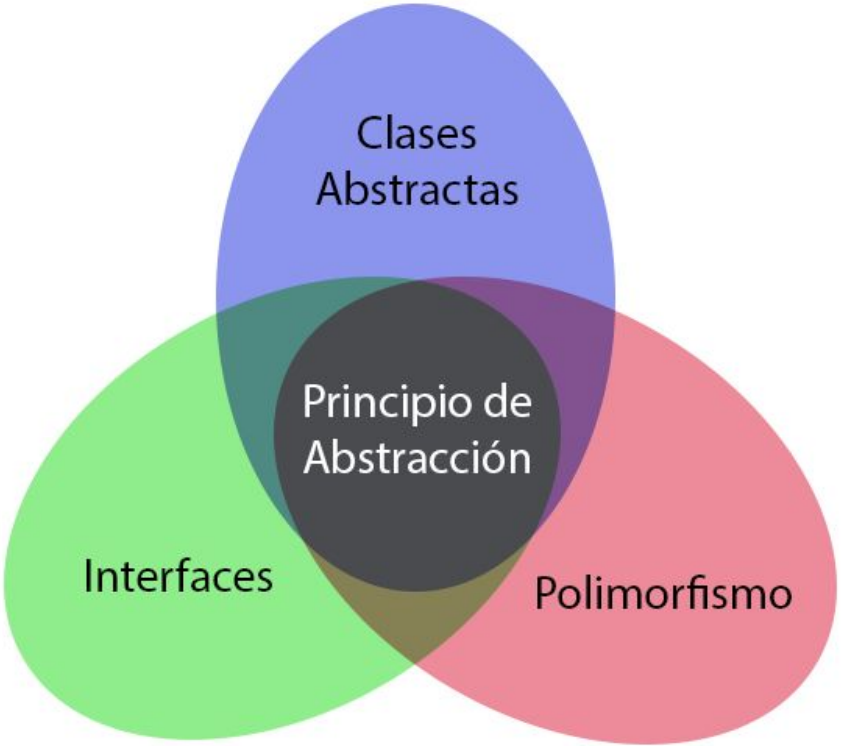
{desafío} latam_

Creando clases abstractas

```
public class Main {
     public static void main(String[] args) {
          Leon leon = new Leon();
          Tigre tigre = new Tigre();
          ArrayList<Animal> listaAnimales = new ArrayList<>();
          ArrayList<Carnivoro> listaCarnivoros = new ArrayList<>();
          listaAnimales.add(leon);
          listaAnimales.add(tigre);
          listaCarnivoros.add(leon);
          listaCarnivoros.add(tigre);
     }
}
```

Principio de abstracción

Principio de abstracción



Mejorando la aplicación de animales

```
public interface Cargable {
     void almacenarCarga(Object carga);
}

public class Camion implements Cargable{

     public List<Object> carga;
     public double cargaMaxima;

     @Override
     public void almacenarCarga(Object carga) {

     }

     //Getters y Setters...
}
```

Mejorando la aplicación de animales

```
public abstract class ServicioCalculo {

     public static boolean esPosibleSubirNuevoAnimal(Camion camion, Animal nuevoAnimal) {
          double peso = 0;
          //Recorremos la lista de animales del camion (carga)
          for(Object objeto : camion.getCarga()) {
               //Casteamos el objeto actual a la clase Animal
               Animal animal = (Animal) objeto;
               //Se suman todos los pesos para obtener la carga actual
               peso = peso + animal.getPeso();
          }

          //Se compara el peso de la carga actual más el peso del nuevoAnimal, versus la carga
máxima
          // Si la carga maxima es inferior a la suma, entonces
          // es posible agregar el nuevo animal y se devuelve true.
          // En el caso contrario, se devolverá false
          return (camion.getCargaMaxima() < (nuevoAnimal.getPeso()+peso));
     }
}
```

Mejorando la aplicación de animales

```
public class Main {

     public static void main(String[] args) {
          Camion camion = new Camion();
          camion.setCargaMaxima(1000);

          Leon leon = new Leon();
          leon.setPeso(200);

          if(ServicioCalculo.esPosibleSubirNuevoAnimal(camion, leon) == true) {
               camion.almacenarCarga(leon);
          }
     }
}
```

Programación con principios

{desafío} latam_

Principio de modularización

- Se deben separar las funcionalidades de un software en módulos (métodos, clases, paquetes, colecciones de paquetes e incluso proyectos) y cada uno de ellos, debe estar encargado de una parte del sistema.
- Para lograr la modularidad, se debe atomizar un problema para obtener sub-problemas y que cada módulo atienda a dar solución a uno de los sub-problemas.

Principio DRY

Don’t Repeat Yourself

- No repetir el código en ninguna instancia

{desafío} latam_

El principio DRY

- Hay casos en que dos porciones de código hacen casi lo mismo, por ejemplo:

```
int indiceNombre;
int indiceApellido;

for(int i = 0; i <= listaNombres; i++){
    if(listaNombres.get(i).equals("Juan")){
        indiceNombre = i;
}

for(int i = 0; i <= listaApellidos; i++){
    if(listaApellidos.get(i).equals("Perez")){
        indiceApellido = i;
}
```

El principio DRY

- En este caso, tenemos dos ciclos que hacen casi lo mismo, podríamos reemplazarlos creando el siguiente método:

```
public int retornarIndice(String elementoBuscado, List<String> lista){
    for(int i = 0; i <= lista; i++){
        if(lista.get(i).equals(elementoBuscado)){
        return i;
    }
}
```

El principio DRY

- Y entonces el primer código quedaría así:

```
int indiceNombre = retornarIndice("Juan", listaNombres);
int indiceApellido = retornarIndice("Perez", listaApellidos);

public int retornarIndice(String elementoBuscado, List<String> lista){
    for(int i = 0; i <= lista; i++){
        if(lista.get(i).equals(elementoBuscado)){
        return i;
    }
}
```

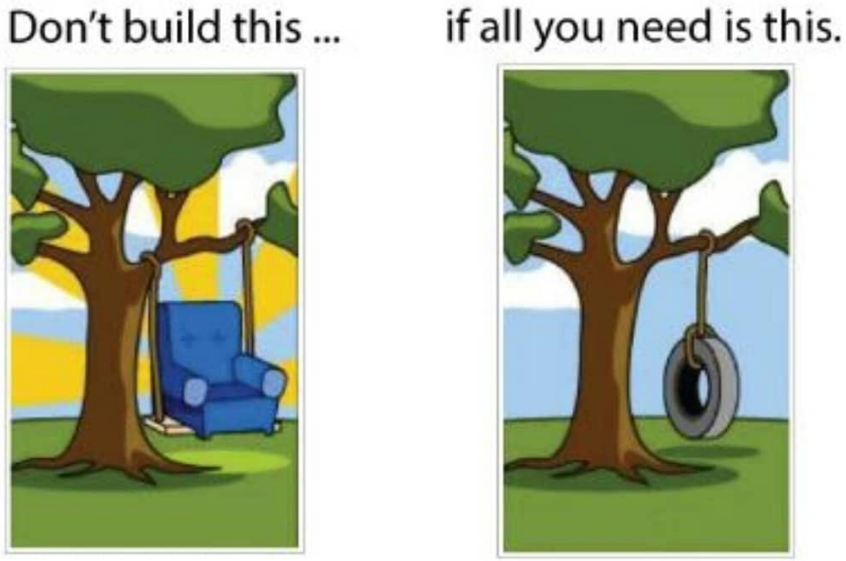
Principio KISS

Keep It Simple Stupid

Crea el software sin hacerlo innecesariamente complejo.
De esta forma, es más fácil de entender y utilizar.

Principio YAGNI

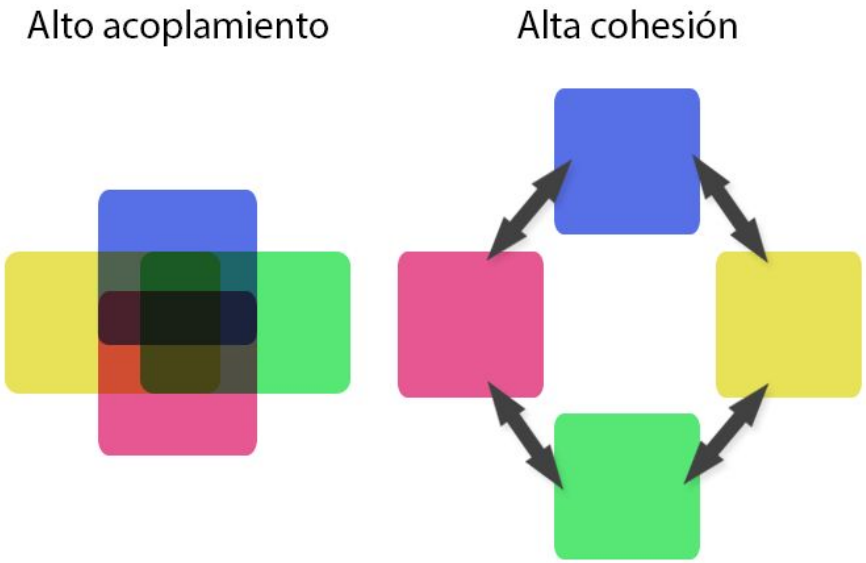
You Are not Going to Need It



Este principio indica que no se deberían agregar piezas que no se van a utilizar

Cohesión y Acoplamiento

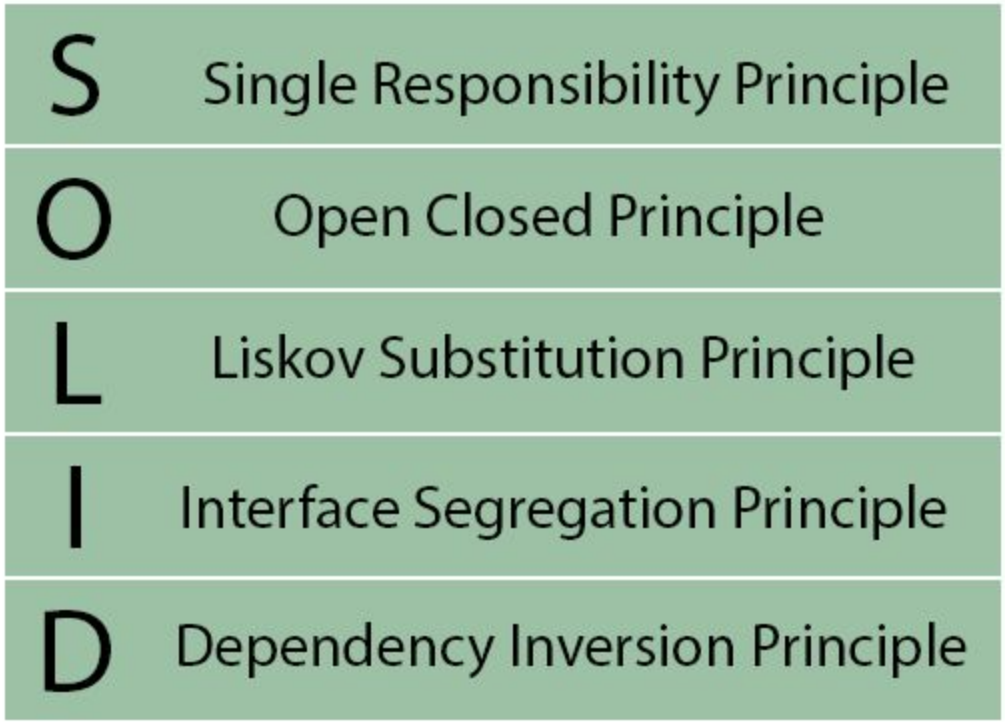
- La cohesión, mide la "fuerza" con que las partes o piezas de un software están conectadas dentro de un módulo. La cohesión puede medirse como alta (fuerte) o baja (débil).
- Un código acoplado, es difícil de entender y mejorar, debido a que muchas cosas dependen de muchas cosas dentro del código.
- Un código con alta cohesión tiene bajo acoplamiento y viceversa.



Programación con principios II

{desafío} latam_

Principios SOLID



Single Responsibility Principle

Principio de responsabilidad única

- Responsabilidad, es la razón por la cual una clase cambia de estado

Se crea una clase PDF y se desea imprimir posteriormente.

La responsabilidad de imprimir debe ser de una clase diferente

```
public class PDF{
    int paginas;
    String titulo;
}

public class Impresora{

    public void imprimir(PDF pdf){
    …
    }
}
```

Open Closed Principle

Principio abierto-cerrado

Un objeto dentro del software debe estar disponible para ser extendido (Abierto), pero no estarlo para modificaciones (Cerrado).

Open Closed Principle

Principio abierto-cerrado

```
private int valor;
private String nombre;

public String getNombre(){...}
public int getValor(){...}
public void setNombre(String nombre){...}
public void setValor(int valor){...}
```

Error:

```
private int valorActual;
private String nombre;

public String getNombre(){...}
public int getValor(){...}
public void setNombre(String nombre){...}
public void setValor(int valor){...}
```

Open Closed Principle

Principio abierto-cerrado

Trabajo de corrección después del cambio:

```
private int valorActual;
private String nombre;

public String getNombre(){...}
public int getValorActual(){...}
public void setNombre(String nombre){...}
public void setValorActual(int valorActual){...}
```

Liskov Substitution Principle

Principio de sustitución de Liskov

Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

{desafío} latam_

Interface Segregation Principle

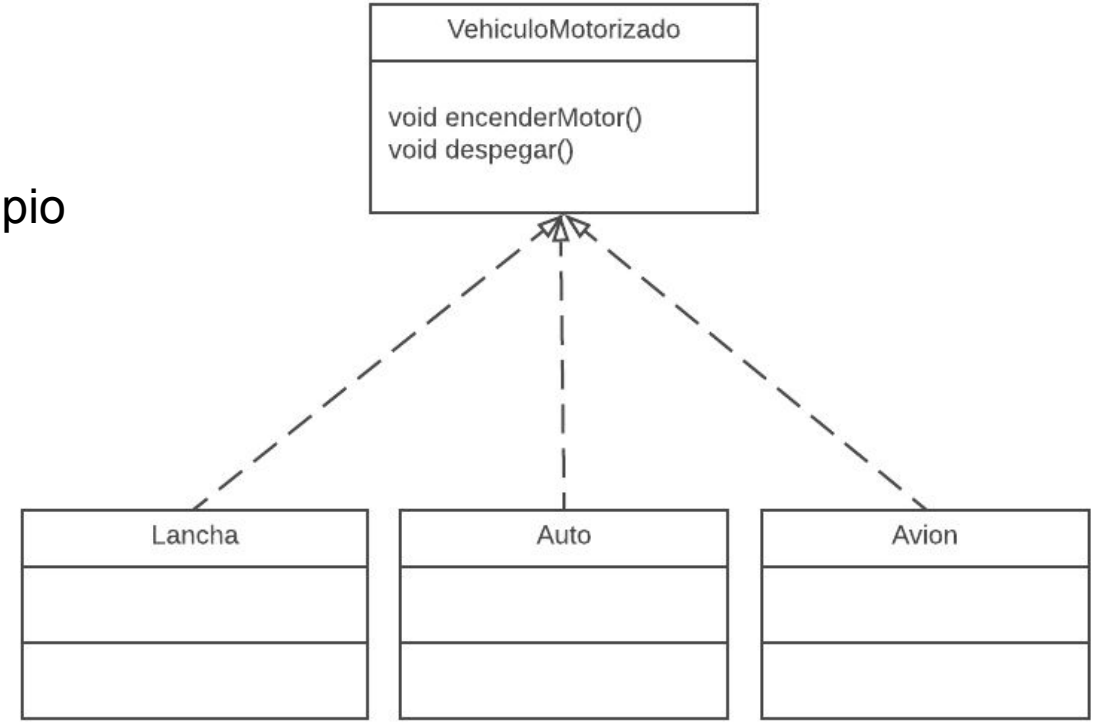
Principio de segregación de interfaces

Los clientes no deberían verse forzados a depender de interfaces que no usan

Interface Segregation Principle

Principio de segregación de interfaces

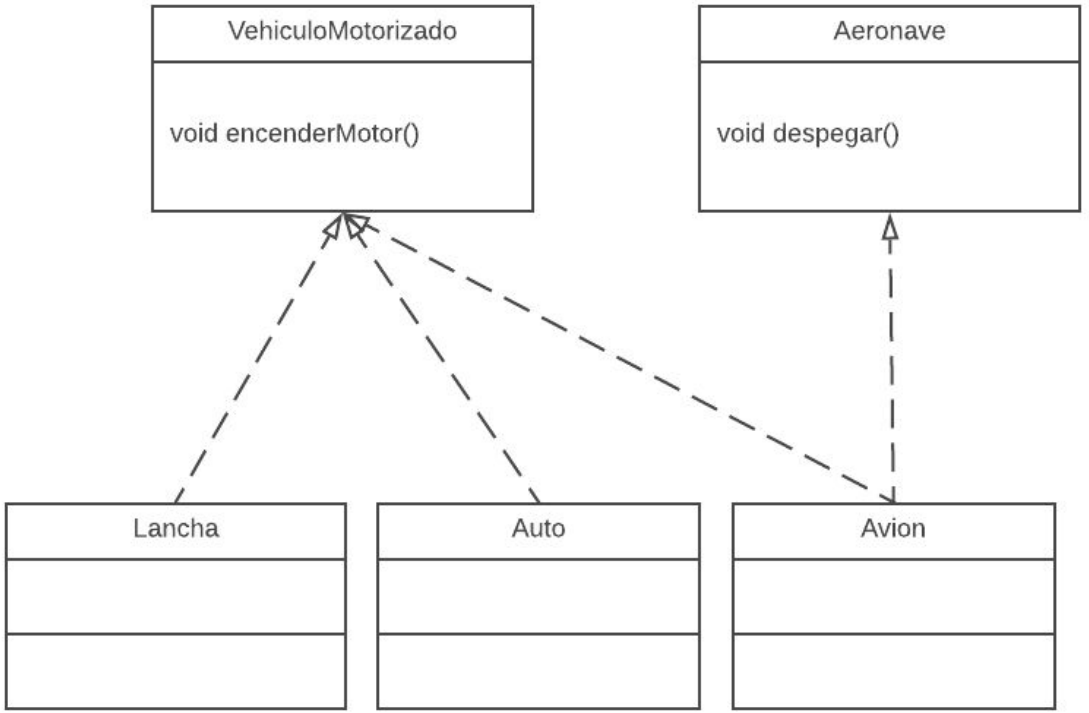
Violación del principio



Interface Segregation Principle

Principio de segregación de interfaces

Solución



Dependency Inversion Principle

Principio de inversión de dependencias

- Los módulos de alto nivel no deben depender de módulos de bajo nivel.

- Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

Dependency Inversion Principle

Principio de inversión de dependencias

```
class Boton{
    Lampara lamp;

    void presionarBoton(Boolean presionado){
        this.lamp.encenderApagar(presionado);
    }
}

class Lampara{
    boolean encendido;

    void encenderApagar(Boolean presionado){
        this.encendido = presionado;
    }
}
```

Dependency Inversion Principle

Principio de inversión de dependencias

Flujo de dependencias

Dependency Inversion Principle

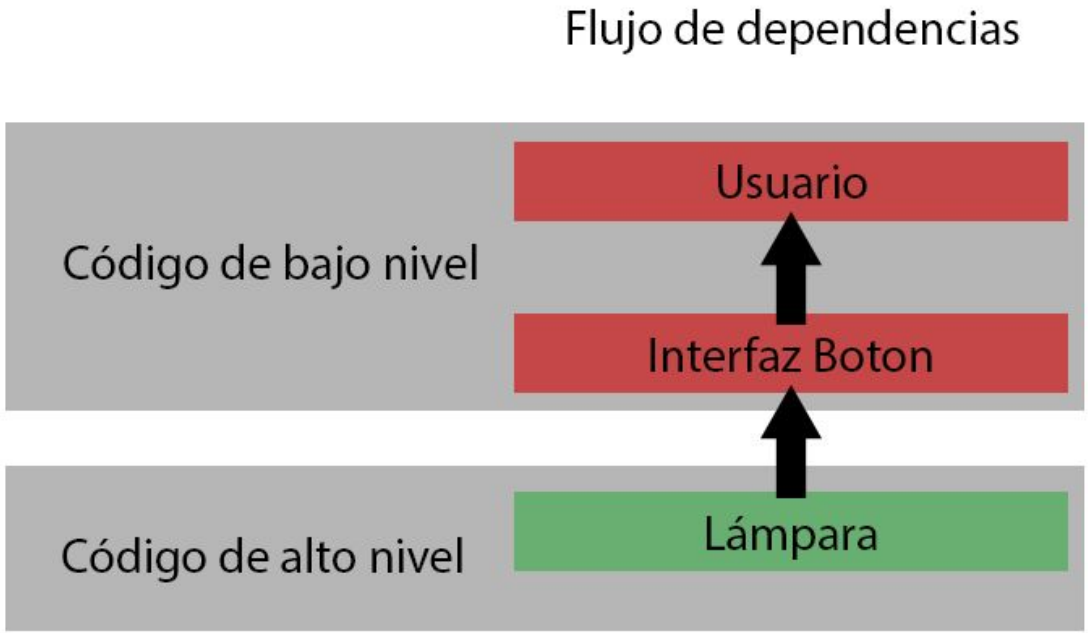
Principio de inversión de dependencias

```
interface Boton{
    void presionarBoton(Boolean presionado);
}

class Lampara implements Boton{
    public boolean encendido;
    @Override
    void presionarBoton(Boolean presionado){
        this.encendido = presionado;
    }
}
```

Dependency Inversion Principle

Principio de inversión de dependencias



Practiquemos polimorfismo

Aplicación para el pago de remuneración

(Polimorfismo con herencia)

Contexto de la aplicación

Una compañía paga a sus empleados en forma semanal y existen tres tipos de empleados:

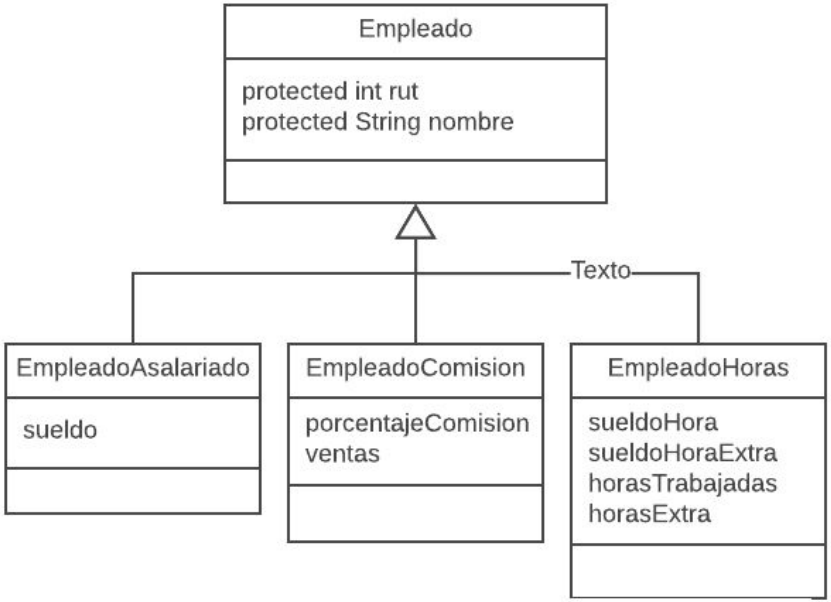
- **Empleados asalariados** que reciben un sueldo semanal fijo, sin importar el número de horas trabajadas;
- **Empleados por hora**, que reciben un sueldo por hora y pago por tiempo extra;
- **Empleados por comisión**, que reciben un porcentaje de sus ventas

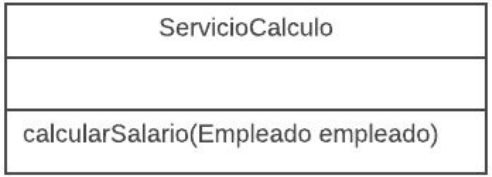
Contexto de la aplicación

Para esto, vamos a crear un servicio estático que permita obtener los datos, los cuales serán:

- Andrés, 16.313.986-3: Empleado asalariado, con 180 dólares semanales.
- Camila, 8.438.298-1: Empleada por hora, que recibe 3,5 dólares por hora y 5,3 por hora extra, que ha trabajado 47 horas esta semana y 7 de esas horas son hora extra.
- Rodrigo, 11.800.835-9: Empleado por comisión, que recibe un 19% de comisión por venta y esta semana ha realizado ventas por un total de 450 dólares.

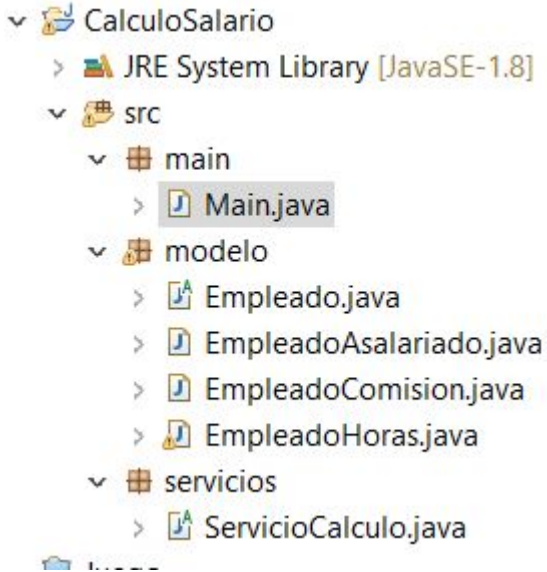
Diagrama de clases





*Los atributos de las subclases de empleado son todos private

Estructura del proyecto



Clase abstracta Empleado

```
public abstract class Empleado {
	protected int rut;
	protected String nombre;
	public Empleado(int rut, String nombre) {
		super();
		this.rut = rut;
		this.nombre = nombre;
	}
	public Empleado() {
		super();
	}
	public int getRut() {
		return rut;
	}
	public void setRut(int rut) {
		this.rut = rut;
	}
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
}
```

Subclase EmpleadoAsalariado

```
public class EmpleadoAsalariado extends Empleado {

	double sueldo;

	public EmpleadoAsalariado(int rut, String nombre, double sueldo) {
		super(rut, nombre);
		this.sueldo = sueldo;
	}

	public EmpleadoAsalariado(int rut, String nombre) {
		super(rut, nombre);
	}

	public double getSueldo() {
		return sueldo;
	}

	public void setSueldo(double sueldo) {
		this.sueldo = sueldo;
	}
}
```

Subclase EmpleadoComision

```
public class EmpleadoComision extends Empleado {
	private double porcentajeComision;
	private double ventas;

	public EmpleadoComision(int rut, String nombre, double porcentajeComision, double ventas) {
		super(rut, nombre);
		this.porcentajeComision = porcentajeComision;
		this.ventas = ventas;
	}
	public EmpleadoComision(int rut, String nombre) {
		super(rut, nombre);
	}
	public double getPorcentajeComision() {
		return porcentajeComision;
	}
	public void setPorcentajeComision(double porcentajeComision) {
		this.porcentajeComision = porcentajeComision;
	}
	public double getVentas() {
		return ventas;
	}
	public void setVentas(double ventas) {
		this.ventas = ventas;
	}
}
```

Subclase EmpleadoHoras

```
public class EmpleadoHoras extends Empleado {

	private double sueldoHora;
	private double sueldoHoraExtra;
	private int horasTrabajadas;
	private int horasExtra;

	public EmpleadoHoras(int rut, String nombre, double sueldoHora, double sueldoHoraExtra, int
horasTrabajadas, int horasExtra) {
		super(rut, nombre);
		this.sueldoHoraExtra = sueldoHoraExtra;
		this.sueldoHora = sueldoHora;
		this.horasTrabajadas = horasTrabajadas;
		this.horasExtra = horasExtra;
	}

	public EmpleadoHoras(int rut, String nombre) {
		super(rut, nombre);
	}

	// Getter y Setter
}
```

ServicioCalculo

```
public abstract class ServicioCalculo {

      public static double calcularSalario(Empleado empleado) throws Exception {
            …
      }
}
```

Método calcularSalario(Empleado empleado) Parte 1

```
if(empleado.getClass().equals(EmpleadoComision.class)) {

     //Casteamos la instancia a EmpleadoComision

     EmpleadoComision empleadoComision = (EmpleadoComision) empleado;

     //Obtenemos el porcentaje en formato 0,19 para multiplicarlo por el total de ventas
y obtener el 19%

     double porcentajeMultiplicable = empleadoComision.getPorcentajeComision() / 100;

     //Devolvemos la siguiente multiplicacion

     return porcentajeMultiplicable * empleadoComision.getVentas();
}
```

Método calcularSalario(Empleado empleado) Parte 2

```
else if(empleado.getClass().equals(EmpleadoHoras.class)) {

     //Casteamos la instancia a EmpleadoHoras

     EmpleadoHoras empleadoHoras = (EmpleadoHoras) empleado;

     double gananciaHoraNormal =
          empleadoHoras.getSueldoHora() * empleadoHoras.getHorasTrabajadas();

     double gananciaHoraExtra =
          empleadoHoras.getSueldoHoraExtra() * empleadoHoras.getHorasExtra();

     return gananciaHoraNormal+gananciaHoraExtra;
}
```

Método calcularSalario(Empleado empleado) Parte 3

```
else if(empleado.getClass().equals(EmpleadoAsalariado.class)) {

     //Devolvemos el sueldo casteando la instancia para poder llegar al atributo sueldo.

     return ((EmpleadoAsalariado) empleado).getSueldo();

}else {

     //Si la instancia no es de ningun tipo de Empleado, devolvemos una excepcion.

     throw new Exception();

}
```

Método main

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

     //Creamos las instancias solicitadas

     //EmpleadoAsalariado(int rut, String nombre, double sueldo)
     EmpleadoAsalariado empAsalariado =
          new EmpleadoAsalariado(16313986, "Andrés", 180);

     //EmpleadoComision(int rut, String nombre, double porcentajeComision, double ventas)
     EmpleadoComision empComision =
          new EmpleadoComision(11800835, "Rodrigo", 19, 450);

     //EmpleadoHoras(int rut, String nombre, double sueldoHora, double sueldoHoraExtra,
     int horasTrabajadas, int horasExtra)
     EmpleadoHoras empHoras =
          new EmpleadoHoras(8438298, "Camila", 3.5, 5.3, 40, 7);

}
```

Método main Parte 1

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

     //Creamos las instancias solicitadas

     //EmpleadoAsalariado(int rut, String nombre, double sueldo)
     EmpleadoAsalariado empAsalariado =
          new EmpleadoAsalariado(16313986, "Andrés", 180);

     //EmpleadoComision(int rut, String nombre, double porcentajeComision, double ventas)
     EmpleadoComision empComision =
          new EmpleadoComision(11800835, "Rodrigo", 19, 450);

     //EmpleadoHoras(int rut, String nombre, double sueldoHora, double sueldoHoraExtra,
     int horasTrabajadas, int horasExtra)
     EmpleadoHoras empHoras =
          new EmpleadoHoras(8438298, "Camila", 3.5, 5.3, 40, 7);

...
```

Método main Parte 2

```
...

     //Creamos la lista y agregamos los empleados
     ArrayList<Empleado> empleados = new ArrayList<>();
     empleados.add(empAsalariado);
     empleados.add(empComision);
     empleados.add(empHoras);

     //Recorremos el arreglo para imprimir el salario correspondiente
     for(Empleado empleado :  empleados) {
          //Utilizamos un try catch por si se arroja la excepción del método
          //calcularSalario.
          //Si algúna excepción se arrojara, el ciclo continuará ejecutandose.
          try {
               System.out.println("El empleado "+empleado.getNombre()+" debe recibir:
               "+ServicioCalculo.calcularSalario(empleado));
          } catch (Exception e) {
               e.printStackTrace();
          }
     }
}
```

Output de la ejecución

```
El empleado Andrés debe recibir: 180.0
El empleado Rodrigo debe recibir: 85.5
El empleado Camila debe recibir: 177.1
```

Módulo para exportar archivos

(Polimorfismo con interfaces)

Contexto de la aplicación

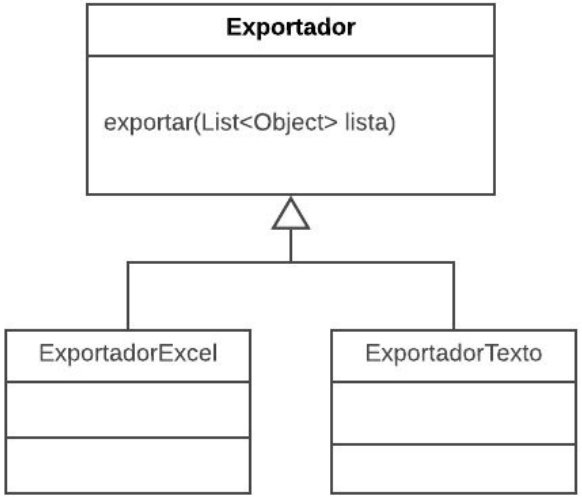
- Una empresa te ha pedido que crees un módulo en java para exportar datos en excel y en archivos txt para una aplicación de la cual no desean exponer el código, por ende, el módulo debe ser una aplicación que sólo reciba datos y los exporte, para que pueda ser agregado a la aplicación padre.
- Los datos siempre vendrán en el mismo formato, una sola linea, con valores separados por coma.
- El módulo debe ser configurable desde un archivo externo, ya que el cliente utilizará el módulo en dos partes diferentes, una que debe exportar en txt y otra que debe exportar en excel, además del formato del archivo, se debe poder especificar la ruta.
- El nombre del archivo siempre debe ser Valores_Sobrantes_(fecha del dia en formato dd-MM-yyyy)_.(xls | txt)
- El archivo txt recibe una lista de palabras y debe separarlas en filas.
- El archivo excel recibe una lista de números y deben estar todos en una misma columna.

Librería Apache POI

central.maven.org/maven2/org/apache/poi/poi/4.1.0/poi-4.1.0.jar

{desafío}
latam_

Diagrama de clases



Estructura del proyecto

- Utilizaremos la carpeta lib para guardar la libreria apache poi (http://central.maven.org/maven2/org/apache/poi/poi/4.1.0/poi-4.1.0.jar)

Creando la interfaz

```
public interface Exportador {
     void exportar(List<Object> lista);
}
```

Implementación ExportadorExcel Parte 1

```
public class ExportadorExcel implements Exportador{

	@Override
	public void exportar(List<Object> lista) {
		String fechaHoyString = new SimpleDateFormat("dd-MM-yyyy").format(new Date());

		// Creamos el libro de trabajo de Excel
		HSSFWorkbook libro = new HSSFWorkbook();
		 // Creamos la hoja de Excel
		HSSFSheet hoja = libro.createSheet();
		// Creamos las filas de Excel
		for(int i = 0; i < lista.size() ; i++) {
			//Casteamos el indice de la lista a String y luego parseamos el resultado a Integer
			Integer numero = Integer.parseInt((String)lista.get(i));
			//creamos la fila
			HSSFRow fila = hoja.createRow(i);
			//creamos la celda en la primera columna
			HSSFCell celda = fila.createCell((short)0);
			//Insertamos el numero en la celda
			celda.setCellValue(numero);
		}


...
```

Implementación ExportadorExcel Parte 2

```
...

            //Exportamos el archivo
            try {
               FileOutputStream elFichero =
                  new FileOutputStream("Valores_Sobrantes_"+fechaHoyString+"_.xls");
               libro.write(elFichero);
               elFichero.close();
               libro.close();
            } catch (Exception e) {
               e.printStackTrace();
            }
            try {
                  libro.close();
            } catch (Exception e) {
               e.printStackTrace();
            }
      }
}
```

Implementación ExportadorTexto

```
public class ExportadorTexto implements Exportador{

	@Override
	public void exportar(List<Object> lista) {
		try {
			String fechaHoyString = new SimpleDateFormat("dd-MM-yyyy").format(new Date());
		 PrintWriter writer = new PrintWriter("Valores_Sobrantes_"+fechaHoyString+"_.txt", "UTF-8");
		 for(Object textObject : lista) {
			String text = (String) textObject;
			writer.println(text);
		 }
		 writer.close();
	   } catch (Exception e) {
		 e.printStackTrace();
	   }
	}
}
```

Clase abstracta LectorConfiguracion

```
public abstract class LectorConfiguracion {

	public static String formatoSalida() throws IOException {
		return buscarConfiguracion("formatoSalida");
	}

	public static String rutaExportacion() throws IOException {
		return buscarConfiguracion("rutaSalida");
	}

	private static String buscarConfiguracion(String configuracion) throws IOException{
	    BufferedReader reader = Files.newBufferedReader(Paths.get("configuracion.txt"));

	   String linea;
	   while ((linea = reader.readLine()) != null) {
	      if(linea.contains(configuracion)) {
			return (linea.substring(linea.indexOf("=")+1, linea.indexOf(";")));
	      }
	   }
	    throw new IOException("No se pudo encontrar el archivo configuracion.txt en la carpeta del
módulo", new Throwable("Archivo de configuraciones no encontrado"));
	}
}
```

Método y clase Main

```
public class Main {

	private static Exportador exportador;

	public static void main(String[] args) {
		if(null == args || args.length == 0) {
			System.out.println("No existen datos para exportar");
		}else {
			try {
				ArrayList<Object> lista = new ArrayList<Object>();
				for(String arg : args[0].split(",")) {
					lista.add(arg);
				}
				if(LectorConfiguracion.formatoSalida().equals("xls")) {
					exportador = new ExportadorExcel();
				}else if(LectorConfiguracion.formatoSalida().equals("txt")) {
					exportador = new ExportadorTexto();
				}
				exportador.exportar(lista);
			} catch (Exception ex) {
				ex.printStackTrace();
			}
		}
	}
}
```

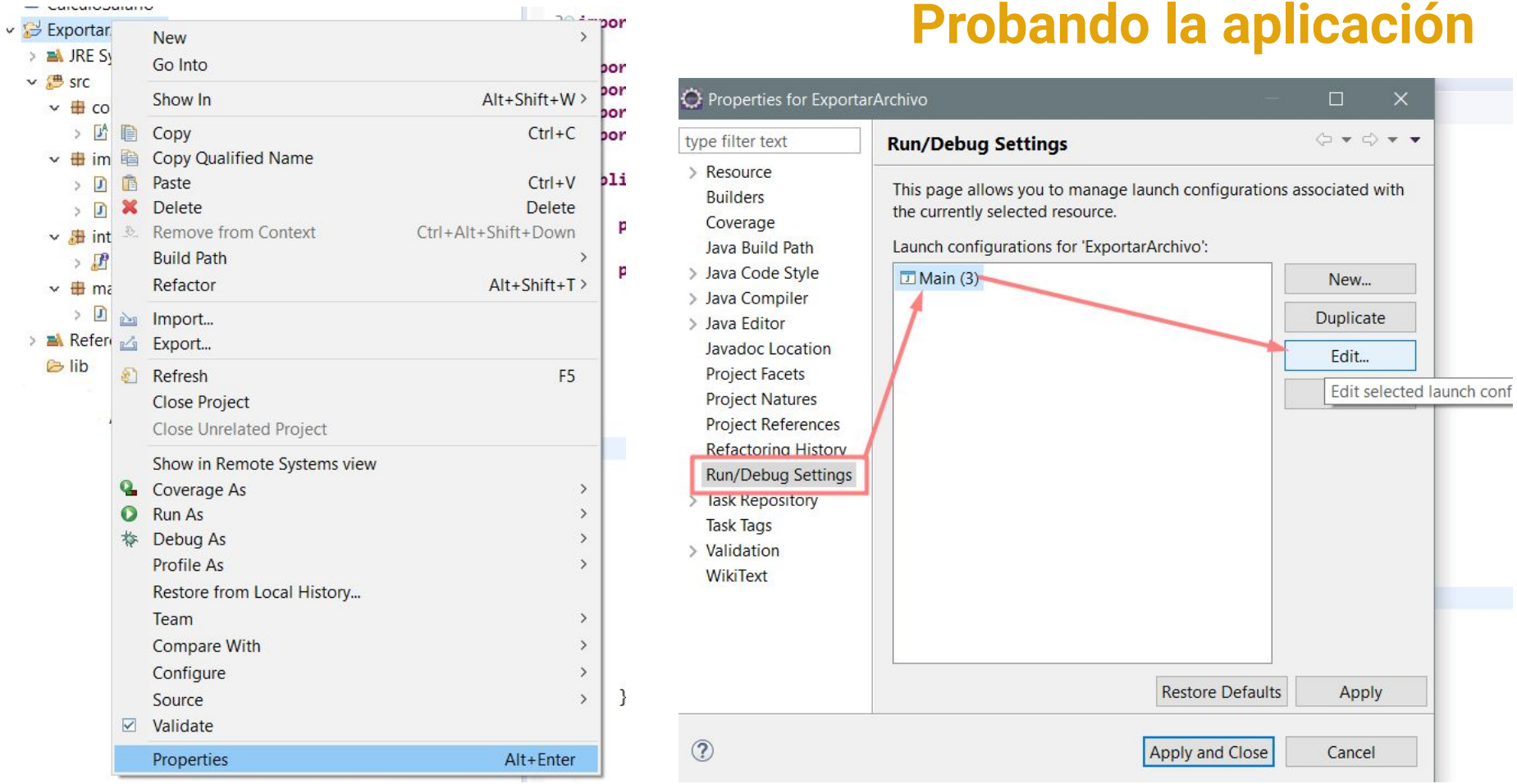
Probando la aplicación

- Al ejecutar se obtiene el siguiente resultado:

```
No existen datos para exportar
```

- Vamos entonces a pasarle como argumento algunos datos al programa

Probando la aplicación

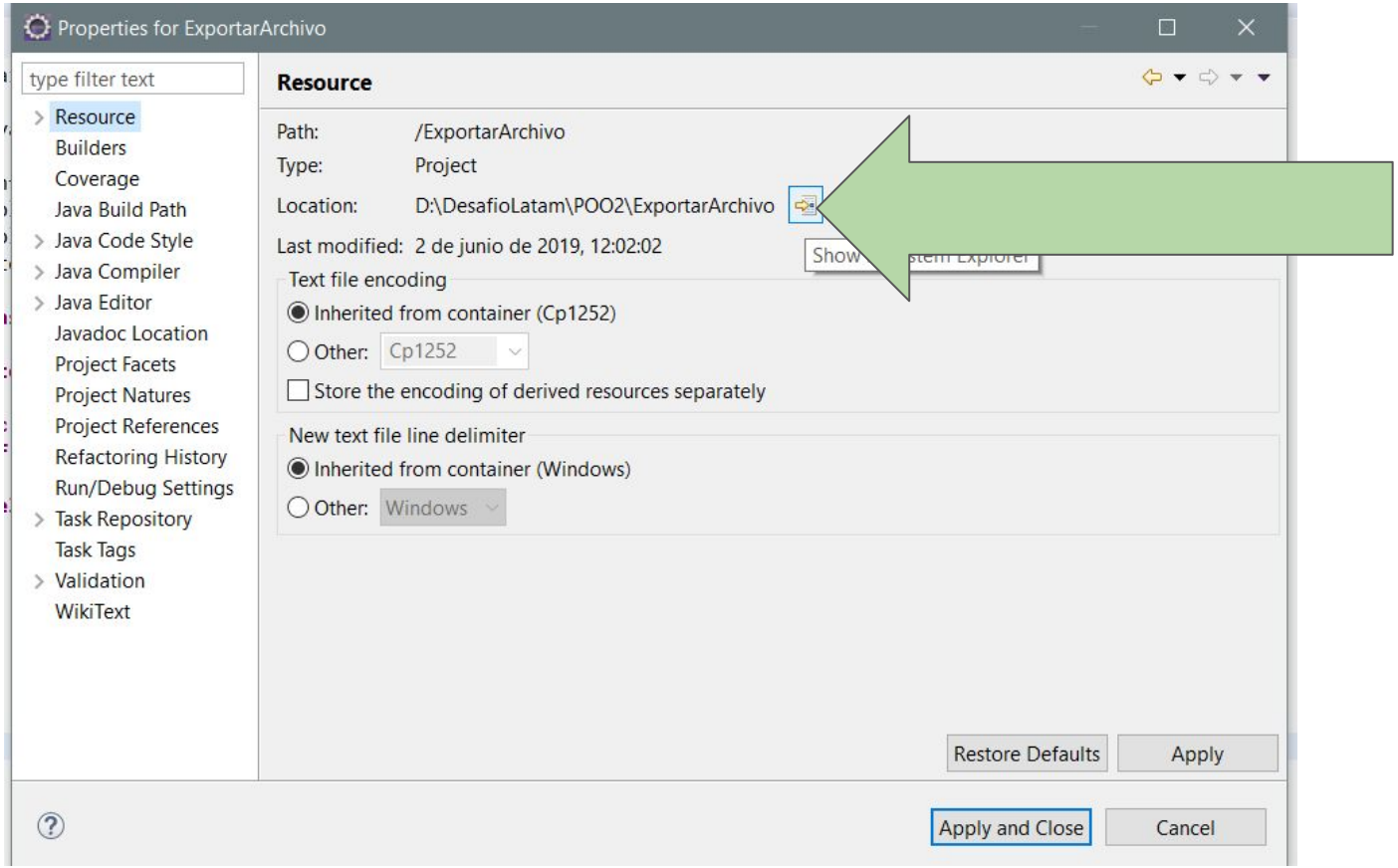


Probando la aplicación

- Al ejecutar se obtiene el siguiente resultado:

```
java.nio.file.NoSuchFileException: configuracion.txt
	at sun.nio.fs.WindowsException.translateToIOException(WindowsException.java:79)
	at sun.nio.fs.WindowsException.rethrowAsIOException(WindowsException.java:97)
	at sun.nio.fs.WindowsException.rethrowAsIOException(WindowsException.java:102)
	at
sun.nio.fs.WindowsFileSystemProvider.newByteChannel(WindowsFileSystemProvider.java:230)
	at java.nio.file.Files.newByteChannel(Files.java:361)
	at java.nio.file.Files.newByteChannel(Files.java:407)
	at java.nio.file.spi.FileSystemProvider.newInputStream(FileSystemProvider.java:384)
	at java.nio.file.Files.newInputStream(Files.java:152)
	at java.nio.file.Files.newBufferedReader(Files.java:2784)
	at java.nio.file.Files.newBufferedReader(Files.java:2816)
	at configuracion.LectorConfiguracion.buscarConfiguracion(LectorConfiguracion.java:19)
	at configuracion.LectorConfiguracion.formatoSalida(LectorConfiguracion.java:11)
	at main.Main.main(Main.java:23)
```

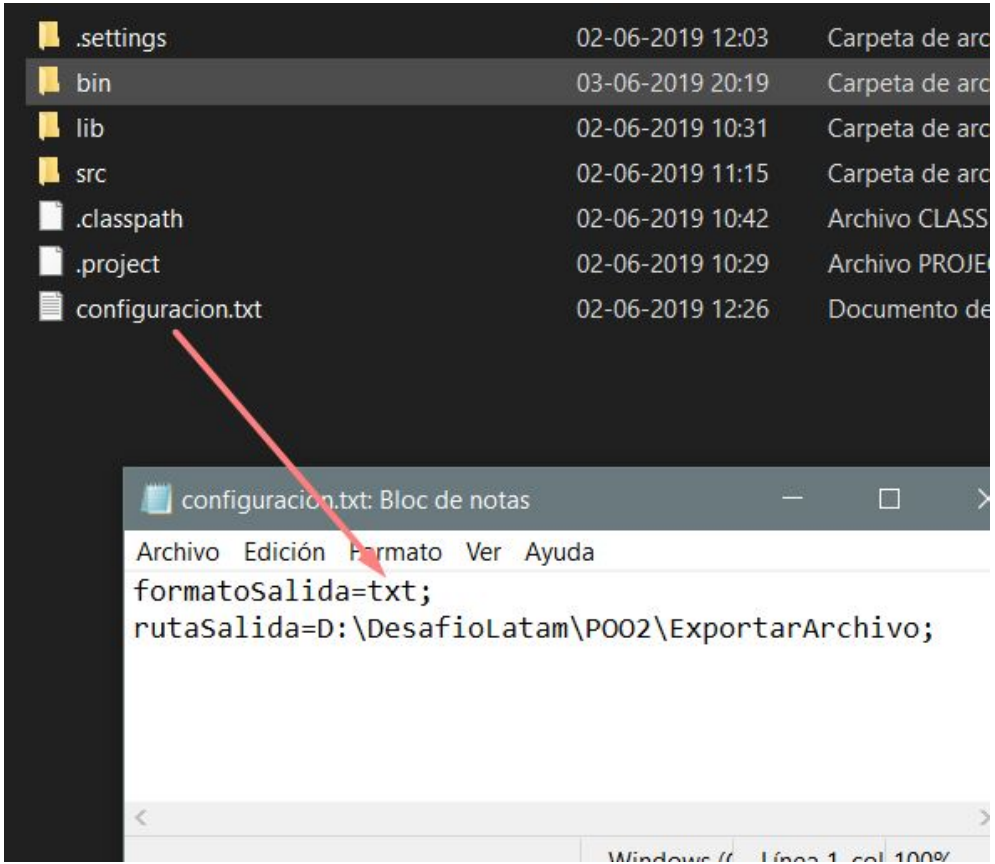
Probando la aplicación



{desafío} latam_

Creando archivo de configuración

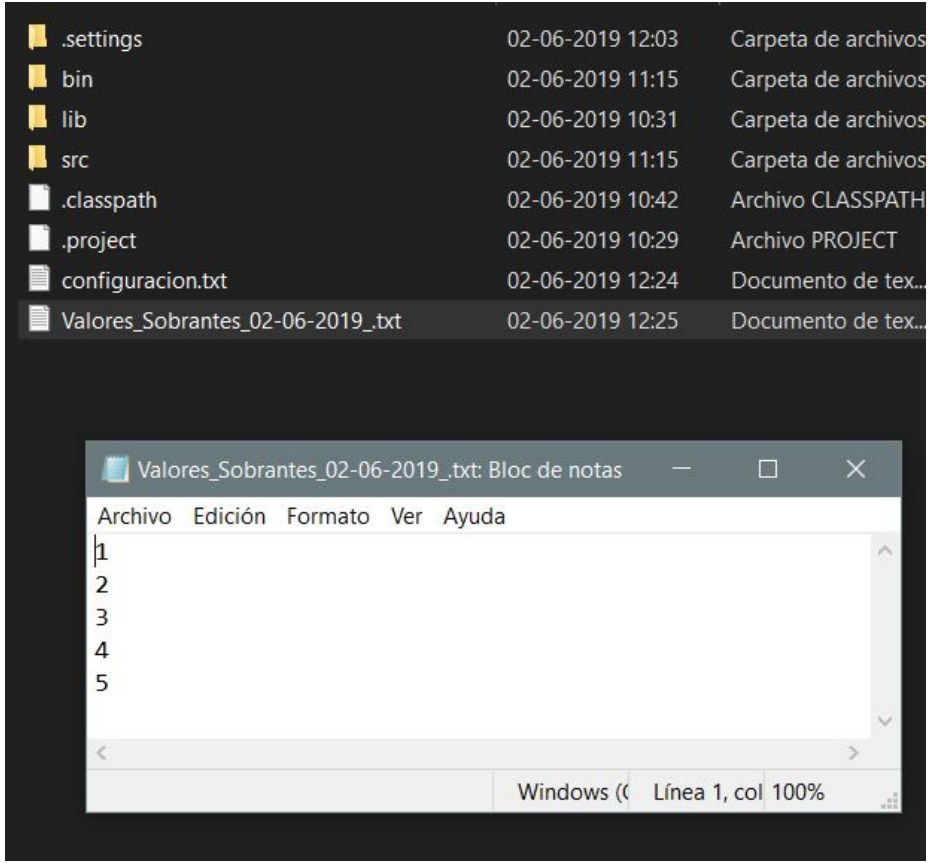
- La variable rutaSalida es la ruta de tu equipo donde se guardará el archivo creado, en este caso, lo crearemos en la carpeta raíz del proyecto.



{desafío} latam_

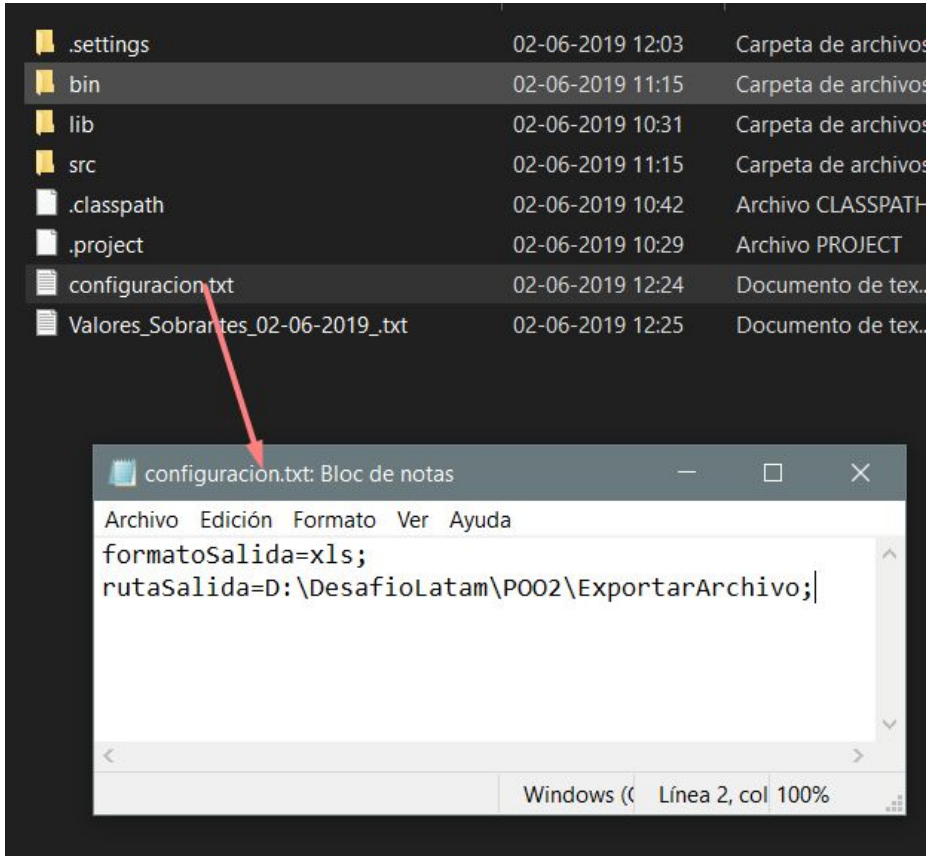
Probando la aplicación

- Al ejecutar se crea el archivo txt.



Probando la aplicación

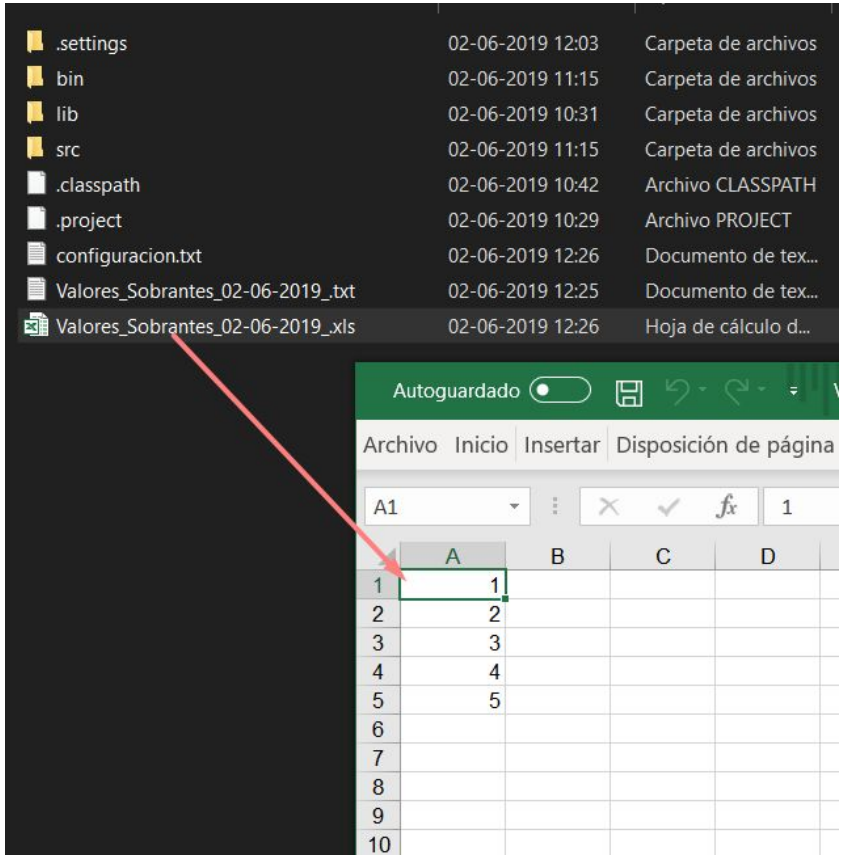
- Cambiamos el archivo de configuraciones, modificando el formato de salida a xls y ejecutamos nuevamente



{desafío} latam_

Probando la aplicación

- Se genera el archivo xls correctamente.

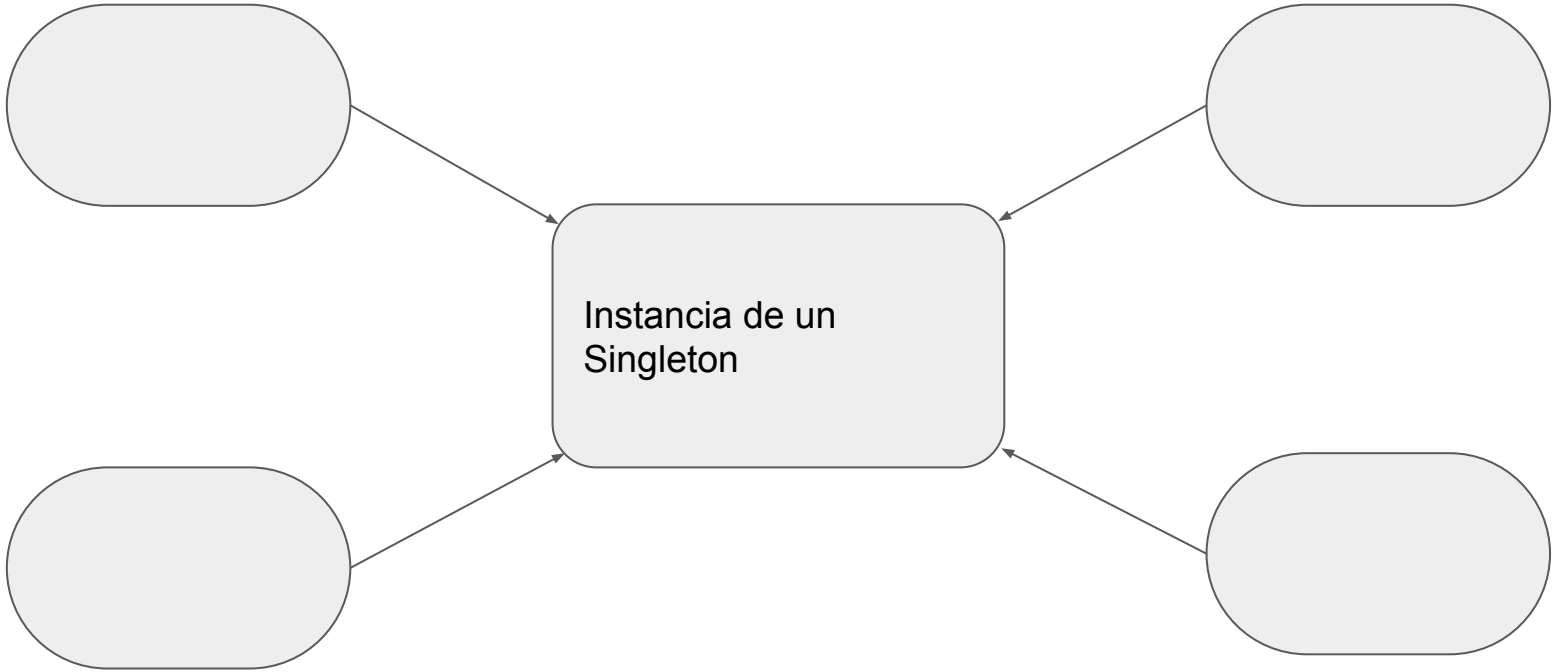


{desafío} latam_

Aplicación/módulo finalizado

Instancias únicas

Él patrón de diseño Singleton



Aplicando el patrón / Creando un Singleton

```
class Configurador{
	//Variable encapsulada y estática donde se almacenará la instancia única.
	private static Configurador config;

	//Constructor privado
	private Configurador() {}

	//Método estático encapsulador para acceder a la instancia única
	public static Configurador getConfig() {
		if (config== null) {
			config= new Configurador();
	    }
		return config;
	 }
}
```

Hilos / Threads

A pesar de que este código valida si existe una instancia antes de crearla, puede provocar un error si hay dos usuarios ejecutando el método al mismo tiempo, ya que podrían entrar a `if(config==null)` en el mismo instante y lograrían entrar al if, creando una instancia cada uno del Configurador.

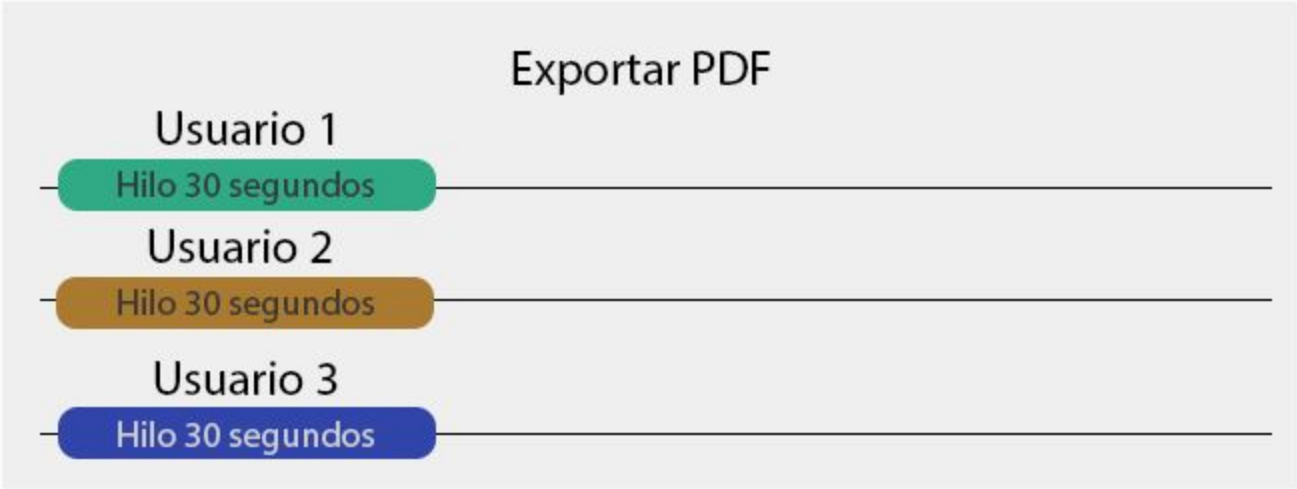
- En el software java, cada vez que se ejecuta un método o algoritmo, se crea un hilo, hasta que termina la ejecución de éste.

- Hay algunos algoritmos que tardan milésimas de segundo y otros que pueden llegar a durar meses ejecutándose, dependiendo de su complejidad.

- Por defecto, en aplicaciones online, cada usuario genera su propio hilo al ejecutar un método, esto permite que la aplicación no se bloquee con un cuello de botella, si dos usuarios quieren ejecutar el mismo método.

Hilos / Threads

Hilos / Threads



Synchronized

```
private static Configurador getConfig() {
    if (config == null) {
        synchronized(Configurador.class) {
            if (config == null) {
                config = new Configurador();
                System.out.println("Instancia creada");
            }
        }
    }
    System.out.println("Llamada al Configurador");
    return config
}
```

Llamada al Singleton

```
private static Configurador getConfig() {
    if (config == null) {
        synchronized(Configurador.class) {
            if (config == null) {
                config = new Configurador();
                System.out.println("Instancia creada");
            }
        }
    }
    System.out.println("Llamada al Configurador");
    return config
}
```

Llamada al Singleton

```
public static void main(String[] args){
        Configurador.getConfig();
        Configurador.getConfig();
        Configurador.getConfig();
}
```

Resultado:

```
Instancia creada
Llamada al Configurador
Llamada al Configurador
Llamada al Configurador
```

Cierre

{desafío}
latam_

¿Existe algún concepto que no hayas comprendido?

Volvamos a revisar los conceptos que más te hayan costado antes de seguir adelante

Reflexionemos

{desafío} latam_

{desafío} latam_

Academia de talentos digitales