



# Orientación a Objetos II

Sesión Experimental 01





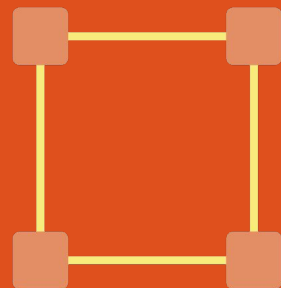
## Activación de conceptos



# ¿Qué es el polimorfismo?

Polimorfismo

Muchas Formas



## Las clases Profesor y Estudiante

```
public class Profesor extends Persona {  
    public Profesor(String rut, String nombre, boolean presente) {  
        super(rut, nombre, presente);  
    }  
}
```

```
public class Estudiante extends Persona {  
    public Estudiante(String rut, String nombre, boolean presente) {  
        super(rut, nombre, presente);  
    }  
}
```

# Polimorfismo

```
ArrayList<Persona> lista = new ArrayList<>();

lista.add(new Estudiante("1", "Juan", true));
lista.add(new Estudiante("2", "Andres", true));
lista.add(new Estudiante("3", "Juan", false));
lista.add(new Profesor("10", "Jose", true));

for(Persona individuo : lista) {
    System.out.println(individuo.toString());
}
```

## Output

```
Persona [rut=10, nombre=Jose, presente=true]
Persona [rut=1, nombre=Juan, presente=true]
Persona [rut=2, nombre=Andres, presente=true]
Persona [rut=3, nombre=Juan, presente=false]
```

# Sintaxis para el casteo de clases

```
Estudiante estudiante = (Estudiante) instanciaPersona;
```

# Casteo de clases

```
for(Persona individuo : lista) {  
    Estudiante estudiante = (Estudiante) individuo;  
    System.out.println(individuo.getClass().getSimpleName());  
    System.out.println(estudiante.getDeuda());  
}
```

## Output

```
Estudiante  
1500.0  
Estudiante  
2000.0  
Estudiante  
3500.0  
Exception in thread "main" java.lang.ClassCastException: Modelo.Profesor  
cannot be cast to Modelo.Estudiante  
    at Main.Main.main(Main.java:18)
```



# Comprobando si es posible castear

```
for(Persona p : lista) {  
    System.out.println(p.getClass().getSimpleName());  
    if(p.getClass() == Estudiante.class) {  
        Estudiante est = (Estudiante) p;  
        System.out.println("Deuda: " + est.getDeuda());  
    }  
}
```

## Output

```
Estudiante  
Deuda: 1500.0  
Estudiante  
Deuda: 2000.0  
Estudiante  
Deuda: 3500.0  
Profesor
```

# Las interfaces

- Proveen de una lista de prototipos de métodos, lo que significa que sólo se declara tipo de retorno, nombre y parámetros de entrada.
- Permiten conocer la lista de métodos que tendrán las clases que las implementen sin conocer el comportamiento específico de cada una.
- Los métodos que se declaran en la interfaz deben existir en todas las clases que la implementen, por ende, ayudan a establecer la forma de las clases.



# Las interfaces

```
public interface nombreInterfaz{  
    void imprimirHola();  
}  
  
//Implementación:  
public class nombreClase implements nombreInterfaz[, nombreOtraInterfaz]{  
    @Override  
    public void imprimirHola(){  
        System.out.println("hola");  
    }  
}
```

# Polimorfismo con interfaces

```
1 package Personajes;  
2  
3 import Interfaces.Jugador;  
4 import Interfaces.Personaje;  
5  
6 public class Protagonista implements Personaje, Jugador{  
7  
8     private int  
9  
10    @Override  
11    public void  
12        xActua  
13    }  
14  
15 }  
16
```

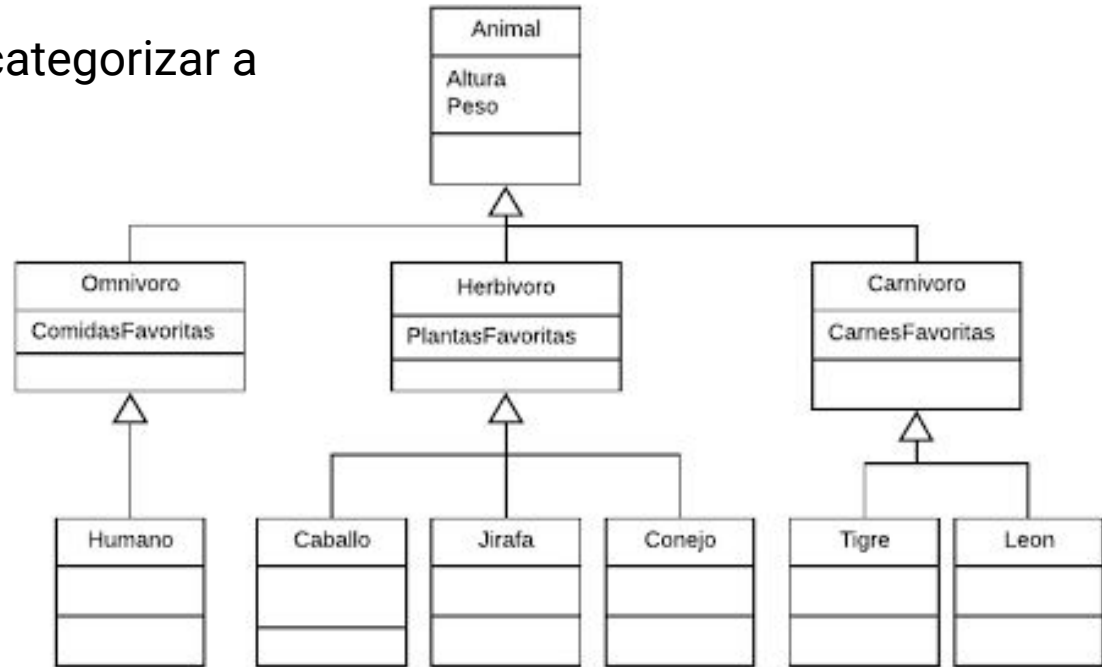
- Add unimplemented methods
- 📄 Create new JUnit test case for 'Protagonista.java'
- Make type 'Protagonista' abstract
- 🔄 Rename in file (Ctrl+2, R)
- 🔄 Rename in workspace (Alt+Shift+R)

## 2 methods to implement:

- Interfaces.Jugador.saltar()
- Interfaces.Jugador.ejecutarAccion()

# Las clases abstractas

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

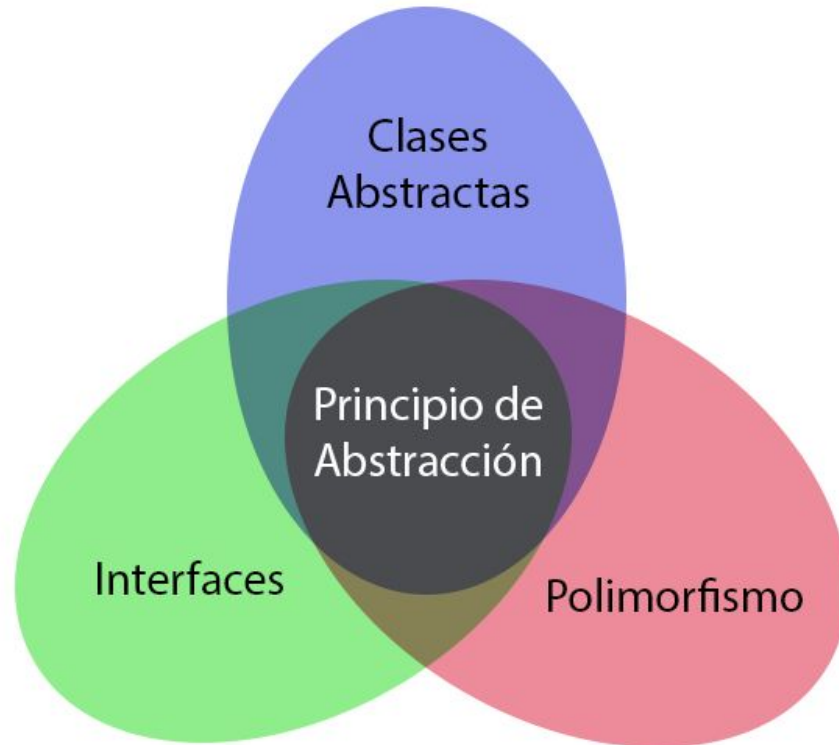


# Creando clases abstractas

```
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) ...  
}
```

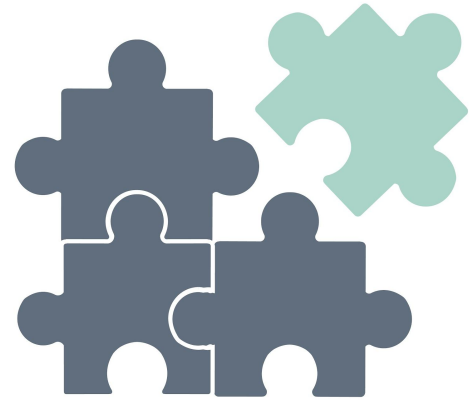
```
public abstract class Carnivoro extends Animal{  
    List<String> carnesFavoritas;  
  
    public List<String> getCarnesFavoritas() {...}  
  
    public void setCarnesFavoritas(List<String> carnesFavoritas) {...}  
}
```

# Principio de abstracción



# 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



## 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

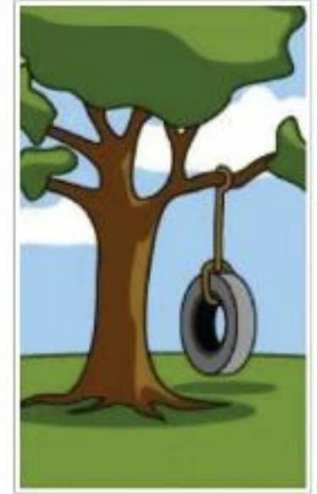
*You Are not Going to Need It*

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

Don't build this ...



if all you need is this.



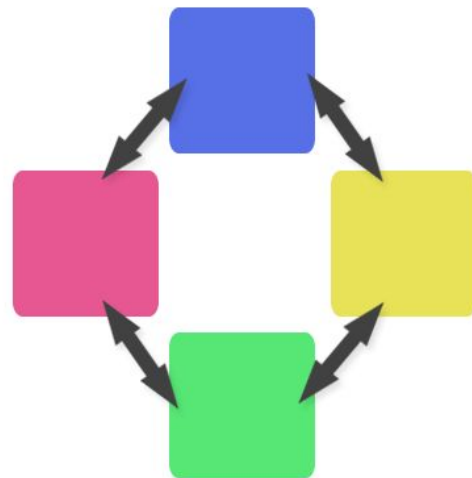
# Cohesión y Acoplamiento

- 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.

Alto acoplamiento



Alta cohesión





## Desafío



140 minutos


{desafío}  
latam\_



## Panel de discusión



# Panel de discusión



¿Qué fue lo que más me costó?

¿Qué fue lo que menos me costó?

¿Qué se necesita reforzar?

¿Cómo podría hacerlo mejor la próxima vez?



Cierre





**¿Existe algún concepto que no  
hayas comprendido?**

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

**Reflexionemos**



*Academia de  
talentos digitales*

[www.desafiolatam.com](http://www.desafiolatam.com)



/DesafioLatam



/DesafioLatam



/DesafioLatam



/DesafioLatam