Auxiliar 1
OOP:
Objetivamente OP

Ignacio Slater Muñoz

¿Qué es OOP?

- La programación orientada a objetos* es un **paradigma** de computación que se organiza en base a **objetos** en vez de *acciones* y **datos** en vez de *lógica*.
- Aquí lo que realmente nos importa son los objetos que queremos manipular más que la lógica para manipularlos.

*OOP, porque POO suena muy feo

Ya, pero ¿Qué es un objeto?

- Un objeto es una **abstracción** que contiene información y maneras de manejar esta información.
- La información contenida dentro de un objeto **no es visible desde afuera** (esto se conoce como *transparencia**).
- Un objeto se compone de su **estado** (campos) y su **comportamiento** (métodos). *¿Y si tengo dos objetos con el mismo estado y comportamiento?*

*Sí, ya se que no hace sentido

Clases

- Una clase es una *plantilla* para *crear objetos*.
- A partir de ahora, cuando hablemos de un objeto, nos estaremos refiriendo a una *instancia de una clase*.

¿Cómo interactúan los objetos?

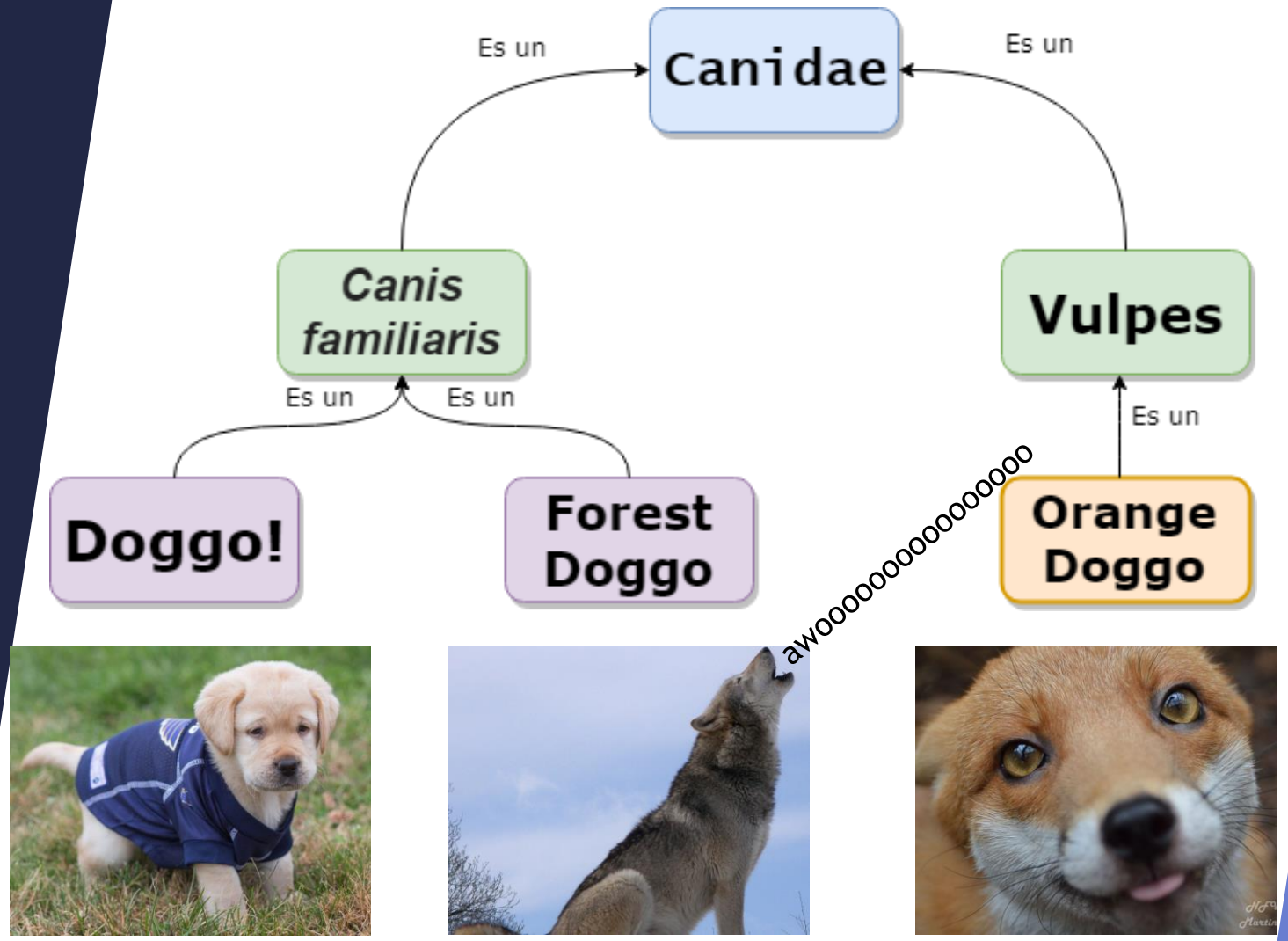
- Los objetos se comunican a través de mensajes (*message passing*), **un objeto no debe ser capaz de acceder a las variables internas de otro**
- Al recibir un mensaje, el objeto “decide” cómo responder
- El proceso de buscar el método adecuado para responder se conoce como *method lookup* (vamos a volver a esto en un ratito)

¿Qué hace OP al OOP?

- Encapsulación
  - Cada objeto se encarga de manejar su estado y comportamiento
- Composición
  - Objetos pueden contener a otros
- Separación de responsabilidades
  - Cada objeto maneja una parte de la lógica de un programa
- Delegación
  - “Si no es mi pega, se la encargo a otro”
- Herencia (merece su propia diapo)

Herencia

- Especialización de una clase
- Organización jerárquica
- Los hijos heredan funcionalidades del padre (desde ahora, superclase)
- La herencia **debe** tener coherencia lógica



¿Son los animales objetos?

¡NO!

Bueno sí, pero sólo en este contexto

¿Cómo afecta esto a Java?

- Java es un **lenguaje orientado a objetos-*ish***
  - Tipos primitivos no son objetos
- Clases se definen con la *keyword* **class** y se crean *instancias* con **new**
- Se define herencia con la *keyword* **extends**, todos los objetos heredan de la clase *Object*

Un ejemplo

```
public class Unit {
  public int hitPoints;
  public int strength;
}

public class Hero extends Unit {
  public String name;
}
```

```
public class Main {

  public static void main(
      String[] args) {
    Hero hero = new Hero();
    hero.hitPoints = 16;
    hero.strength = 4;
    hero.name = "Lyn";
    ...
  }
}
```

¿Lo podemos hacer mejor?

Constructores

```
public class Main {
  public static void main(
      String[] args) {
    Hero hero = new Hero("Lyn");
    ...
  }
}
public class Unit {
  public int hitPoints;
  public int strength;
  public Unit(int hitPoints,
      int strength) {
    this.hitPoints = hitPoints;
    this.strength = strength;
  }
}
```

```
public class Hero extends Unit {
  public String name;
  public Hero(String name) {
    super(16, 4);
    this.name = name;
  }
}
```

¿Compila el siguiente código?

```
public class Unit {

  public int hitPoints;
  public int strength;

  public Unit(int hitPoints,
      int strength) {
    this.hitPoints = hitPoints;
    this.strength = strength;
  }
}
```

```
public class Hero extends Unit {
  public String name;

  public Hero(String name) {
    this.hitPoints = 16;
    this.strength = 4;
    this.name = name;
  }
}
```

¿Y ahora?

```
public class Unit {

  public int hitPoints;
  public int strength;

  public Unit(int hitPoints,
      int strength) {
    this.hitPoints = hitPoints;
    this.strength = strength;
  }

  public Unit() { }
}
```

```
public class Hero extends Unit {
  public String name;

  public Hero(String name) {
    this.hitPoints = 16;
    this.strength = 4;
    this.name = name;
  }
}
```

Dudas existenciales

¿Métodos o funciones?

- Los métodos son procedimientos que están asociados a un mensaje y objeto
- Las funciones son procedimientos que están desligados de las estructuras de datos
- Java no tiene funciones, lo más parecido son métodos estáticos (van asociados a la clase, no al objeto)

¿Punteros? ¿Referencias?

- Los punteros son variables que guardan la dirección de memoria de algún valor
- Una referencia es un valor que le permite a un programa acceder de manera indirecta a algún objeto en la memoria
- En Java no existen punteros, sólo referencias

Cómo ser autorreferente

- Existen dos formas en que un objeto puede referirse a si mismo: **this** y **super**

- ¿Cuál es la diferencia?
  - **this** hace referencia al objeto que recibió el mensaje
  - **super** hace referencia al objeto que recibió el mensaje

Khé?!

Volvamos al *method lookup*

- Tanto **this** como **super** referencian al mismo objeto, pero:
  - El method lookup de **this** parte en la clase del objeto que recibió el mensaje
  - El method lookup de **super** parte en la *superclase de la clase que contiene la llamada a super*
- El method lookup busca un método que corresponda al mensaje que recibió
  - Si lo encuentra, retorna
  - Si no lo encuentra, busca en la superclase
  - Caso especial: métodos estáticos

De vuelta al código

```
public class Main {
  public static void main(
      String[] args) {
    Hero hero = new Hero("Lyn");
    ...
  }
}
public class Unit {
  public int hitPoints;
  public int strength;
  public Unit(int hitPoints,
      int strength) {
    this.hitPoints = hitPoints;
    this.strength = strength;
  }
}
```

```
public class Hero extends Unit {
  public String name;
  public Hero(String name) {
    super(16, 4);
    this.name = name;
  }
}
```

¿Es legal esto?

```
public class Main {
  public static void main(
      String[] args) {
    Unit hero = new Hero("Lyn");
    ...
  }
}
public class Unit {
  public int hitPoints;
  public int strength;
  public Unit(int hitPoints,
      int strength) {
    this.hitPoints = hitPoints;
    this.strength = strength;
  }
}
```

```
public class Hero extends Unit {
  public String name;
  public Hero(String name) {
    super(16, 4);
    this.name = name;
  }
}
```

Ideas para tatuajes

Polimorfismo

- Es la capacidad de un tipo A de verse y poder usarse como un tipo B

Principio de Liskov

- Los subtipos siempre deben ser reemplazables por su clase padre

Sobreescritura y sobrecarga

- Todos los métodos tienen una *firma*
  - *La firma se compone del nombre del método y la cantidad y tipo de argumentos que recibe*
- Dentro de la jerarquía de clases:
  - Hay **overriding** si se tienen métodos con igual firma
  - Hay **overloading** si se tienen métodos con igual nombre y distinta firma

Te la peleo

```
public class Unit {
  ...
  public void attack(Unit enemy,
      int damage) {
    enemy.hitPoints -= damage;
  }

  public void attack(Unit enemy) { }
}
```

```
public class Hero extends Unit {
  ...
  @Override
  public void attack(Unit enemy) {
    super.attack(enemy, strength);
  }
}
```

¿Tiene sentido crear instancias de la clase Unit?

Más jerarquía

Clases abstractas

- Son clases que no pueden ser instanciadas
- Se crean con el keyword **abstract**
- Deben tener al menos un método abstracto
- Los métodos abstractos **tienen que ser implementados** por las clases hijas
- **NO DEBEN USARSE COMO TIPO**

Interfaces

- Son contratos entre un usuario y un implementador
- Las clases las pueden implementar con el keyword **implements**
- Una clase puede implementar múltiples interfaces
- Hay *polimorfismo* con interfaces

Antes de volver al código: Visibilidad

- *Keywords* que modifican la visibilidad: **private**, **protected**, **public**
- Siempre debe restringirse la privacidad lo más posible
- Campos de las clases **siempre** deben ser privadas
  - ¿Y si quiero acceder a ellas desde afuera?

Ordenémonos un poco

```
public interface IUnit {
  int getHitPoints();
  void setHitPoints(int hitPoints);
  int getStrength();
  void setStrength(int strength);
  void attack(IUnit enemy);
}
```

```
public abstract class AbstractUnit
    implements IUnit {
  private int hitPoints;
  private int strength;
  ...
  protected void attack(IUnit enemy,
      int damage) {
    enemy.setHitPoints(
      enemy.getHitPoints()
          - damage);
  }
  public abstract void attack(
    IUnit enemy);
  ...
}
```

Estructuras de datos de vidas pasadas

Listas

- Java posee implementaciones de listas que heredan de una interfaz **List<E>**
- Algunas funcionalidades útiles: **add**, **remove**, **contains**, **get**, y podría seguir
- Una implementación en particular que nos va a interesar:
  - **ArrayList<E>**

Diccionarios

- Los diccionarios en Java implementan la interfaz **Map<K,V>**
- Algunas funcionalidades útiles: **put**, **get**, **remove**, **replace**, y más y más
- ¿Alguna implementación a la que ponerle atención?
  - **HashMap<K, V>**

El que se la sabe cante:

- ¿Por qué usar clases abstractas e interfaces en vez de clases concretas?
- ¿Pueden tenerse interfaces en Python?
- ¿Se puede definir el cuerpo de un método en una interfaz? Si se pudiera: ¿Por qué utilizar clases abstractas?
- ¿Qué otros ejemplos de Listas y Diccionarios de Java pueden dar?
- ¿Existen las constantes en Java?