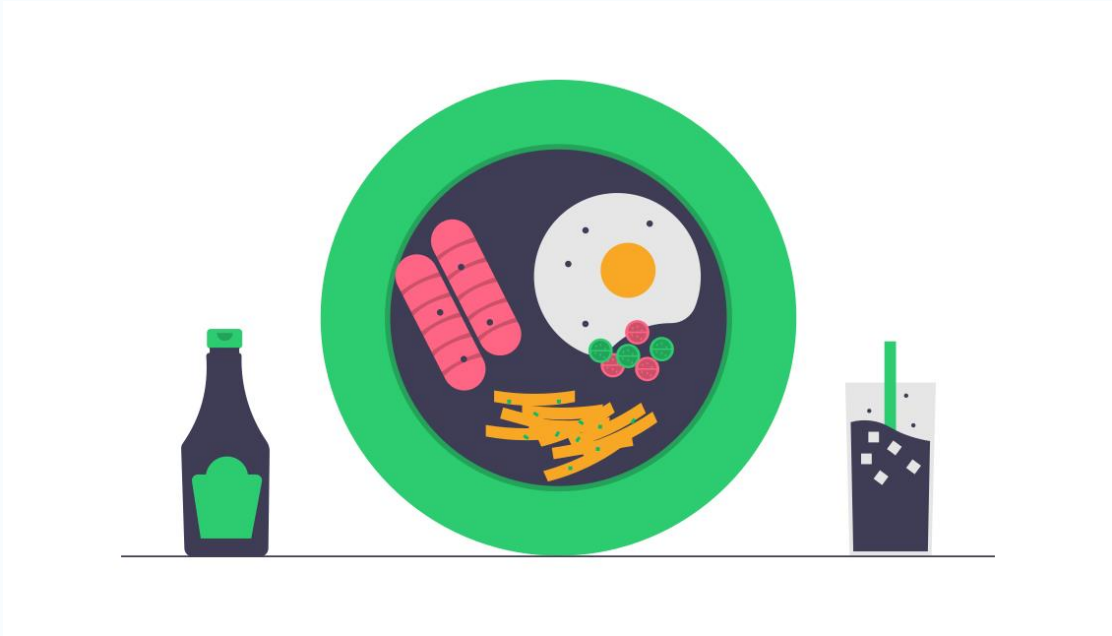
Programación Orientada a Objetos con TypeScript



Programación Orientada a Objetos?

La programación orientada a objetos es un paradigma basado en el concepto de envolver piezas de datos y el comportamiento relacionado con esos datos, en paquetes especiales llamados objetos, que se construyen a partir de un conjunto de "planos", definidos por un programador, llamados clases

Visibility

`+ = public`
`– = private`

Cat
+ name + gender + age + weight + color ...
+ breathe() + eat(food) + run(destination) + sleep(hours) + meow()

The ellipsis means that there's more stuff in the class, but it's not relevant at the moment.

Name

Fields
(state)

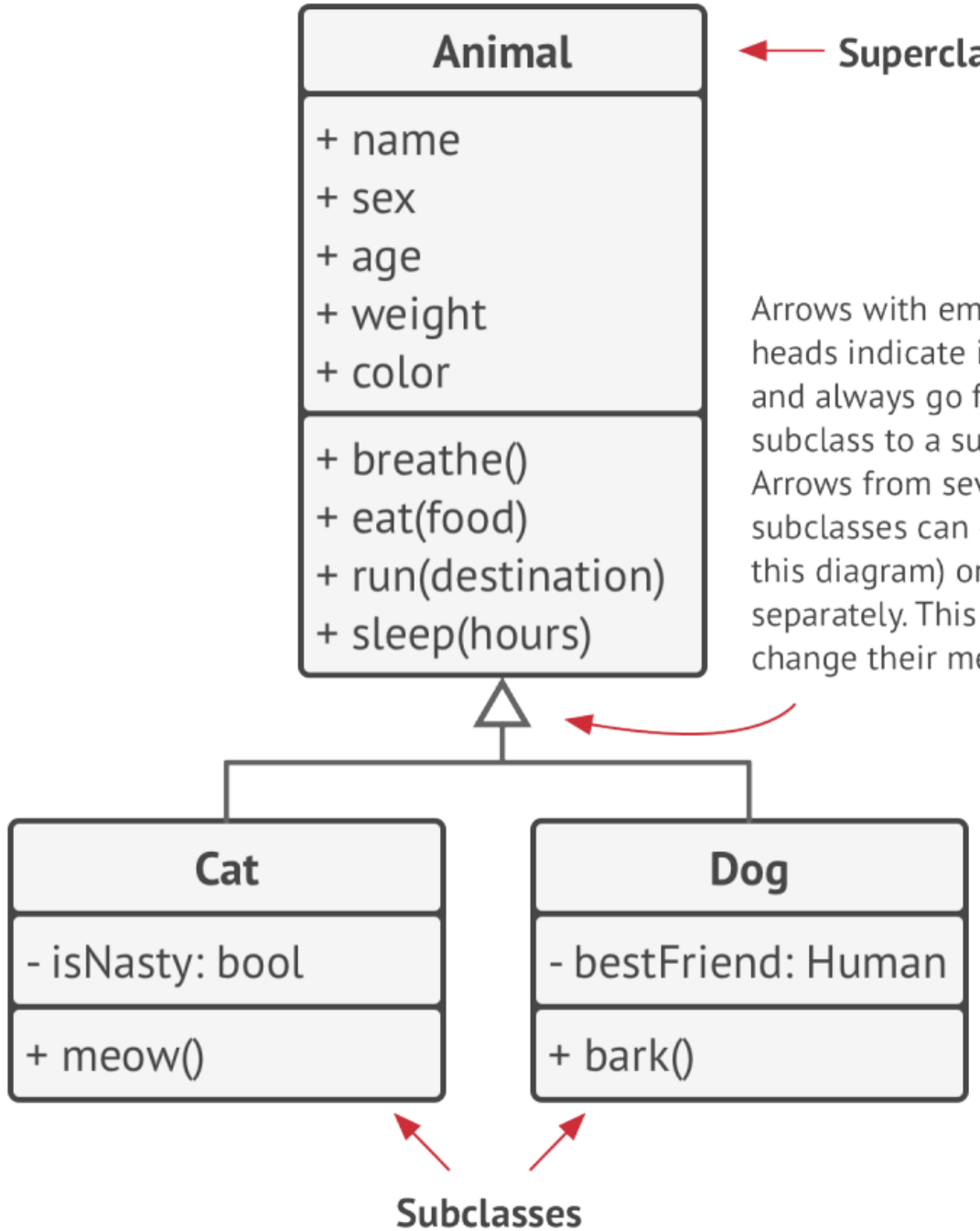
Methods
(behavior)

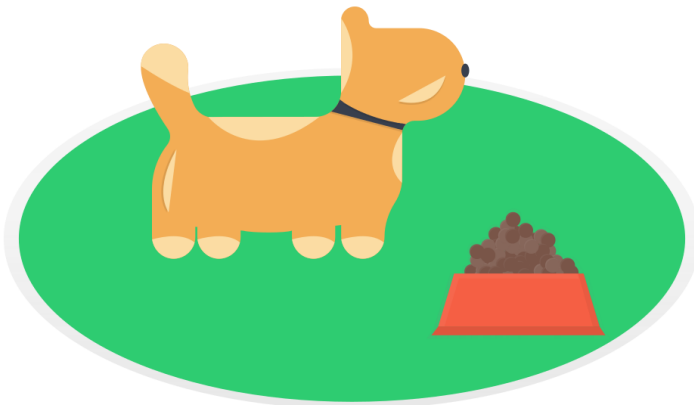
Los datos almacenados dentro de los campos del objeto a menudo se refieren como estado, y todos los métodos del objeto definen su comportamiento.

Entonces, una clase es como un plano que define la estructura de los objetos, que son instancias concretas de esa clase.

Jerarquía de clases

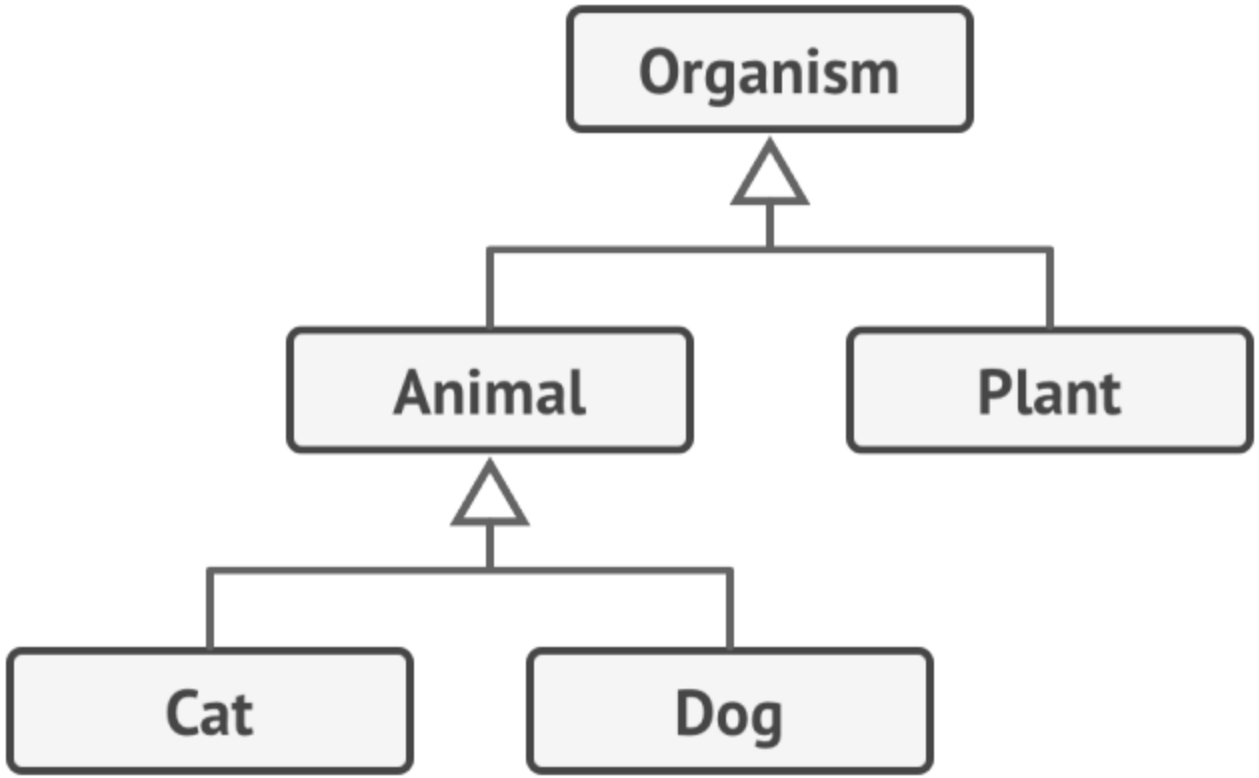
Por lo general nuestros programas contendrán mas de una clase o clases sueltas ya que muchas veces tendremos que generalizar reglas de negocio o conceptos y crear una JERARQUIA de clases.

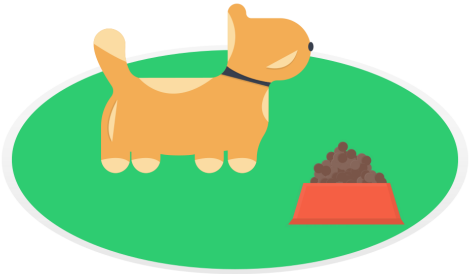

Animal
+ name
+ sex
+ age
+ weight
+ color
+ breathe()
+ eat(food)
+ run(destination)
+ sleep(hours)
Superclass
Arrows with empty triangle heads indicate inheritance and always go from a subclass to a superclass. Arrows from several subclasses can overlap (as in this diagram) or be drawn separately. This doesn't change their meaning.
Cat
- isNasty: bool
+ meow()
Dog
- bestFriend: Human
+ bark()
Subclasses



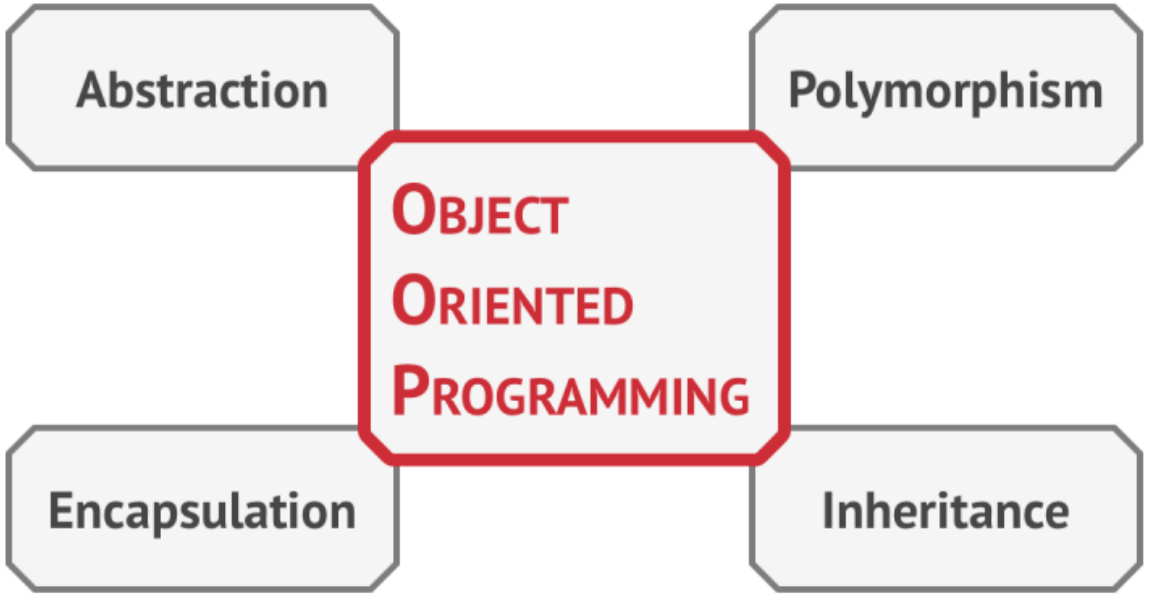
- Cada subclase hereda los atributos y métodos de la superclase.
- Cada subclase puede sobre escribir los métodos de la superclase.
- Cada subclase puede aumentar atributos según lo requiera la clase.

Según los requerimientos del sistema podremos incluso generalizar aun mas nuestras clases.


Organism
Animal
Plant
Cat
Dog

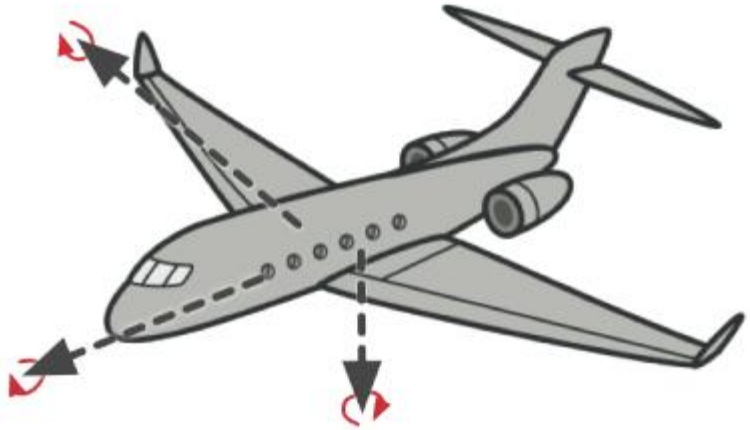


Pilares del OOP



Abstracción

La abstracción es un modelo de un objeto o fenómeno del mundo real, limitado a un contexto específico, que representa todos los detalles relevantes para este contexto con gran precisión y omite todo lo demás.



Airplane
- speed - altitude - rollAngle - pitchAngle - yawAngle
+ fly()

Video juego

Airplane
- seats
+ reserveSeat(n)

App de reservas

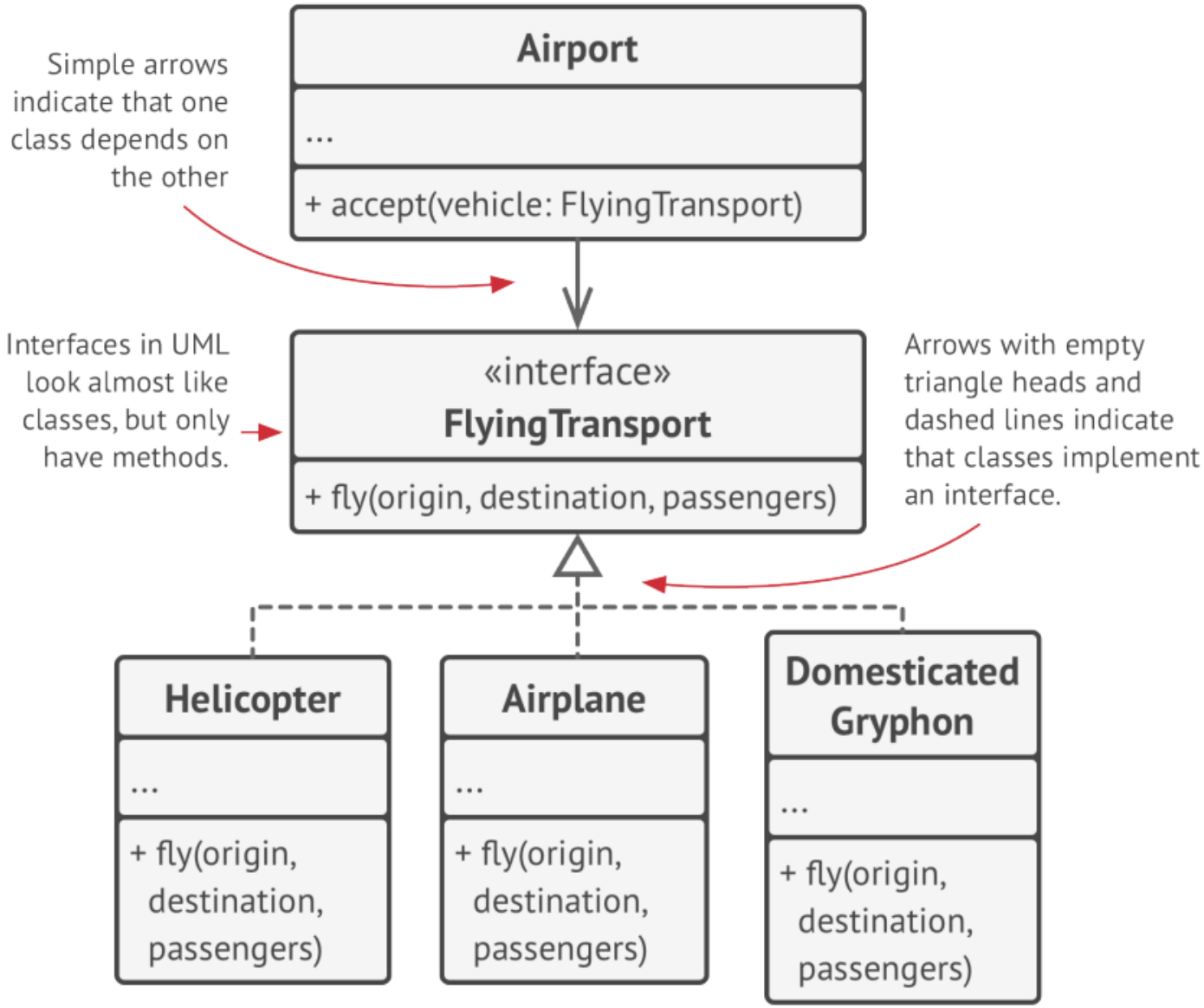
Encapsulación

Es la habilidad que tiene un objeto de ocultar partes de su estado y comportamiento a otros objetos, mostrando solo una *interface limitada* al resto del programa.

- **Public:** los atributos y métodos son visibles para cualquier clase.
- **Private:** los atributos y métodos solo son visibles desde la propia clase.
- **Protected:** los atributos y métodos son visibles desde la misma clase y sus subclases.

Interfaces o clases abstractas

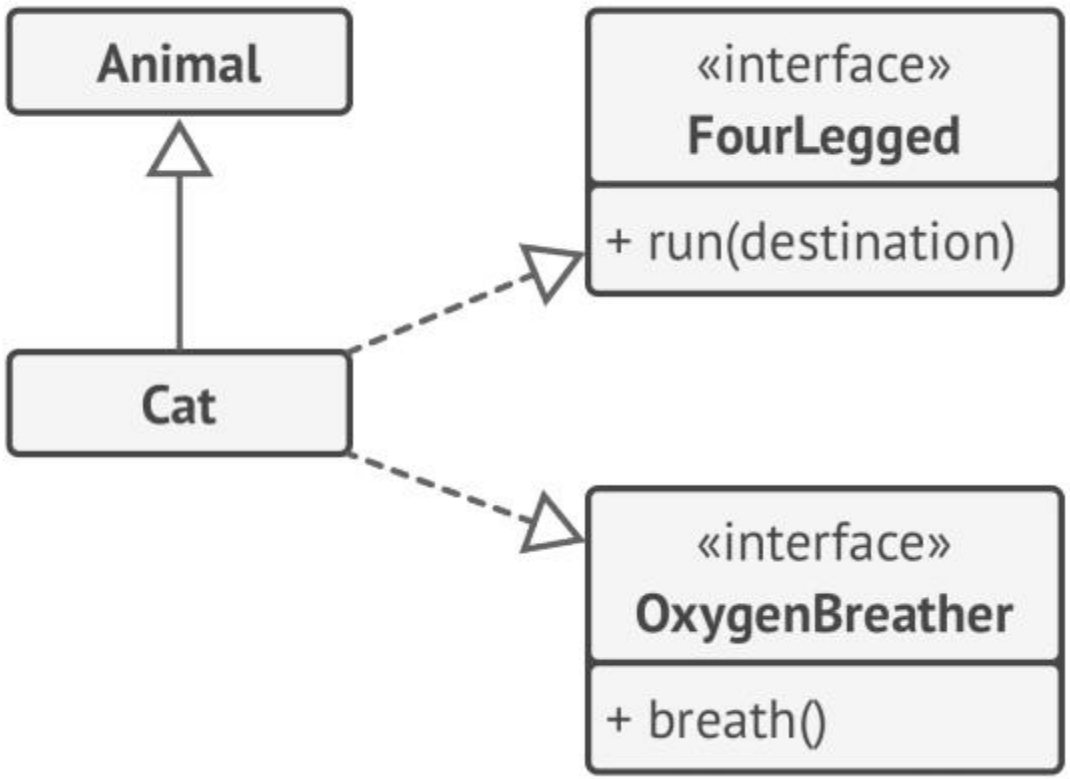
Te permiten definir las reglas de interacción entre clases, es una clase que solo tiene métodos que definen el comportamiento de sus subclases.


Simple arrows indicate that one class depends on the other
Airport
...
+ accept(vehicle: FlyingTransport)
Interfaces in UML look almost like classes, but only have methods.
«interface»
FlyingTransport
+ fly(origin, destination, passengers)
Arrows with empty triangle heads and dashed lines indicate that classes implement an interface.
Helicopter
...
+ fly(origin, destination, passengers)
Airplane
...
+ fly(origin, destination, passengers)
Domesticated Gryphon
...
+ fly(origin, destination, passengers)

Herencia

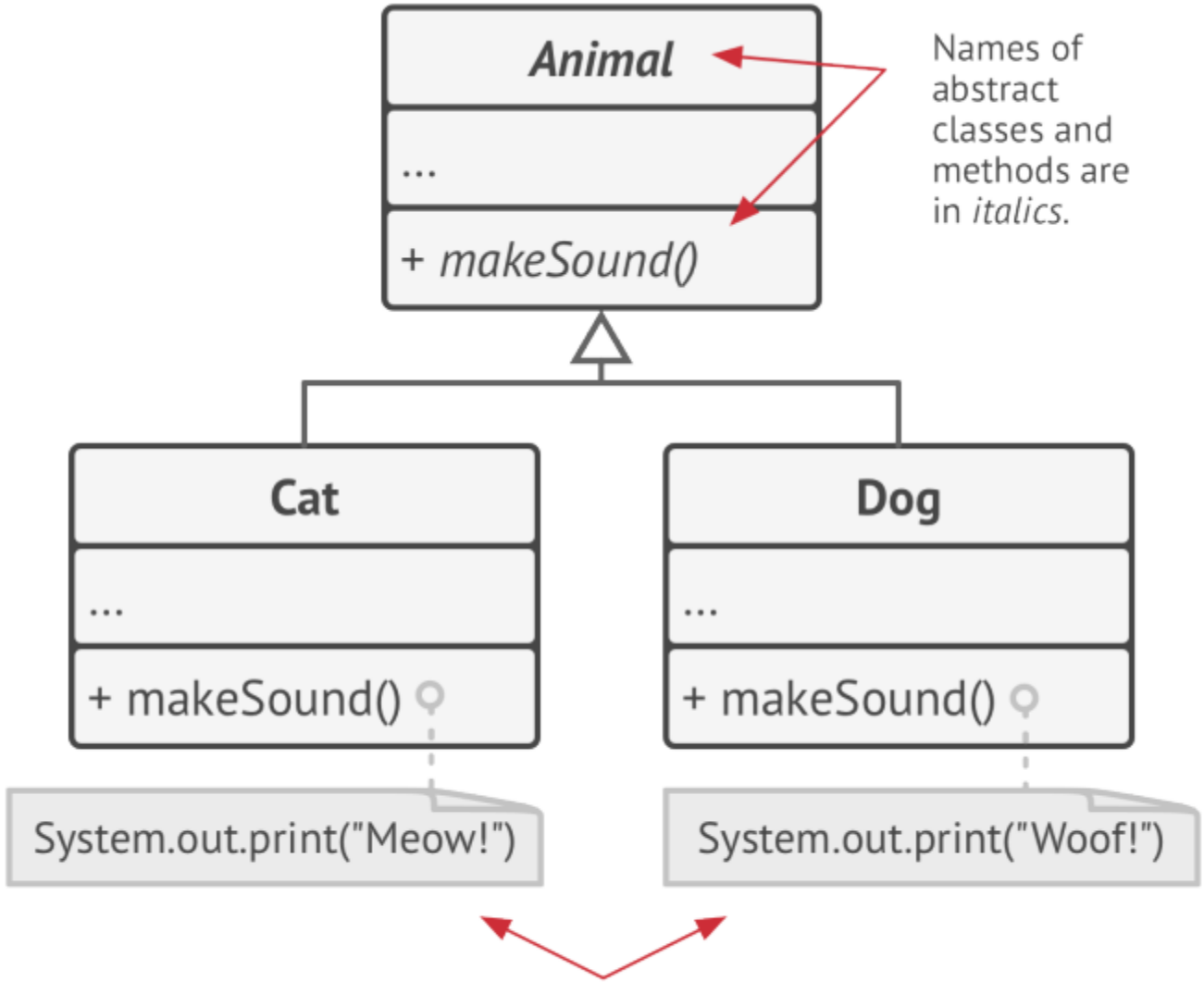
La herencia es la capacidad de construir nuevas clases sobre las existentes. El principal beneficio es la reutilización del código. Si desea crear una clase que sea ligeramente diferente de una existente, no es necesario duplicar el código. En su lugar, extiende la clase existente y coloca la funcionalidad adicional en una subclase resultante, que hereda los campos y métodos de la superclase.

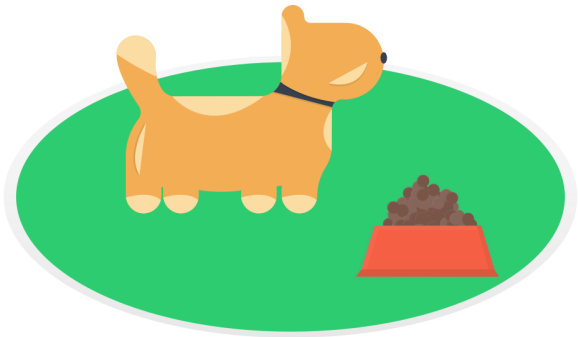
La consecuencia del uso de la herencia es que las subclases tienen la misma interfaz que su clase principal. No puede ocultar un método en una subclase si se declaró en la superclase. También debe implementar todos los métodos abstractos, incluso si no tienen sentido para su subclase.


Animal
«interface»
FourLegged
+ run(destination)
Cat
«interface»
OxygenBreather
+ breath()

Polimorfismo

El polimorfismo es la capacidad de un programa para detectar la clase real de un objeto y llamar a su implementación incluso cuando su tipo real es desconocido en el contexto actual.

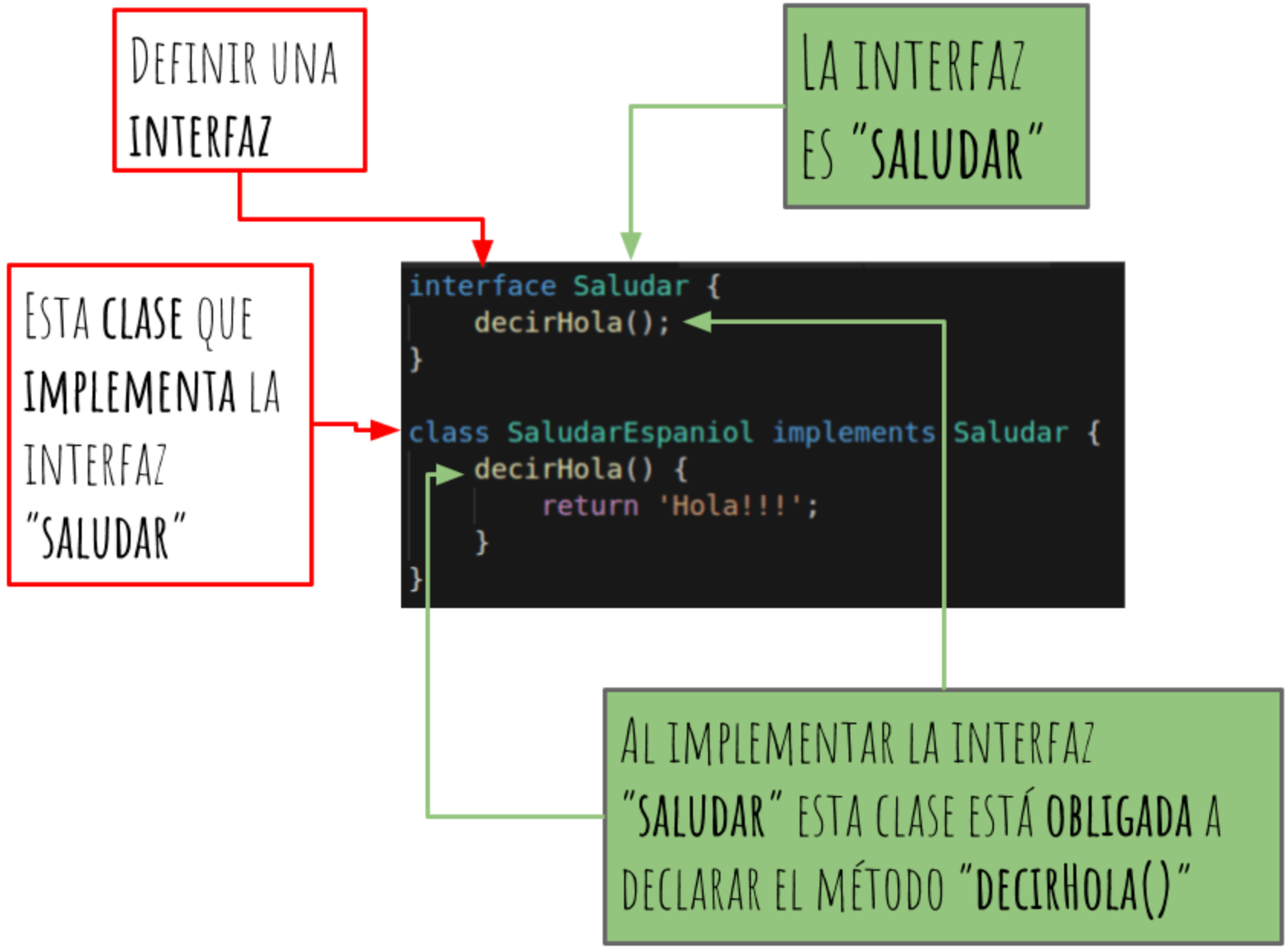

Animal
...
+ makeSound()
Names of abstract classes and methods are in *italics*.
Cat
...
+ makeSound()
Dog
...
+ makeSound()
System.out.print("Meow!")
System.out.print("Woof!")
These are UML comments. Usually they are explain implementation details of the given classes or methods.



Interfaces y Clases Abstractas

Interfaces

Interfaces, y a veces llamadas firmas, es el mecanismo que usa Typescript para definir tipos en las clases.
Nos ayudan a definir la estructura de una clase tanto en su estado como en su comportamiento sin tener que instanciar ningún método o atributo, esto nos ayuda a ocultar estados o comportamientos que no queremos mostrar afuera de la clase.


Definir una interfaz
La interfaz es "saludar"
Esta clase que implementa la interfaz "saludar"
interface Saludar {
decirHola();
}
class SaludarEspaniol implements Saludar {
decirHola() {
return 'Hola!!!';
}
}
Al implementar la interfaz "saludar" esta clase está obligada a declarar el método "decirHola()"

Clases Abstractas

Las clases abstractas son clases base a partir de las cuales se pueden extender otras clases y que no se pueden instanciar.

- Pueden implementar métodos propios.
- Pueden definir métodos que las clases heredadas deben implementar.

pueden considerarse conceptualmente como una combinación de una interfaz y una clase .

Implementación

Implementación de clase Abstracta.

```
abstract class Machine {
    constructor(public manufacturer: string) {
    }

    // Puede instanciarse métodos o no
    summary(): string {
        return `${this.manufacturer} makes this machine.`;
    }

    // Las clases heredadas deben implementar los métodos
    // declarados en la clase abstracta
    abstract moreInfo(): string;
}
```

Implementación de Subclase.

```
class Car extends Machine {

    constructor(manufacturer: string, public position: number, protected speed: number) {
        super(manufacturer);
    }

    move() {
        this.position += this.speed;
    }

    moreInfo() {
        return `This is a car located at ${this.position} and going ${this.speed}mph!`;
    }
}

let myCar = new Car("Konda", 10, 70);
myCar.move(); // position is now 80
console.log(myCar.summary()); // prints "Konda makes this machine."
console.log(myCar.moreInfo()); // prints "This is a car located at 80 and going 70mph!"
```

En síntesis una Clase abstracta es equivalente a una Super clase de las cuales ya vimos su forma de construcción en clase.

Relaciones

UML Association. Professor communicates with students.

La asociación es un tipo de relación en la que un objeto usa o interactúa con otro. En general, utiliza una asociación para representar algo como un campo en una clase, aunque en una interfaz, solo puede indicar la presencia de un método que devolverá el cliente del pedido.

UML Dependency. Professor depends on salary.

La dependencia es una variante de asociación más débil que generalmente implica que no existe un vínculo permanente entre los objetos. La dependencia típicamente (pero no siempre) implica que un objeto acepta a otro objeto como parámetro de método, crea instancias o usa otro objeto.

UML Composition. University consists of departments.

La composición es una relación de "parte completa" entre dos objetos, uno de los cuales está compuesto por una o más instancias del otro. La distinción entre esta relación y otras es que el componente solo puede existir como parte del contenedor.

UML Aggregation. Department contains professors.

La agregación es una variante menos estricta de la composición, donde un objeto simplemente contiene una referencia a otro. El contenedor no controla el ciclo de vida del componente. El componente puede existir sin el contenedor y puede vincularse a varios contenedores al mismo tiempo.