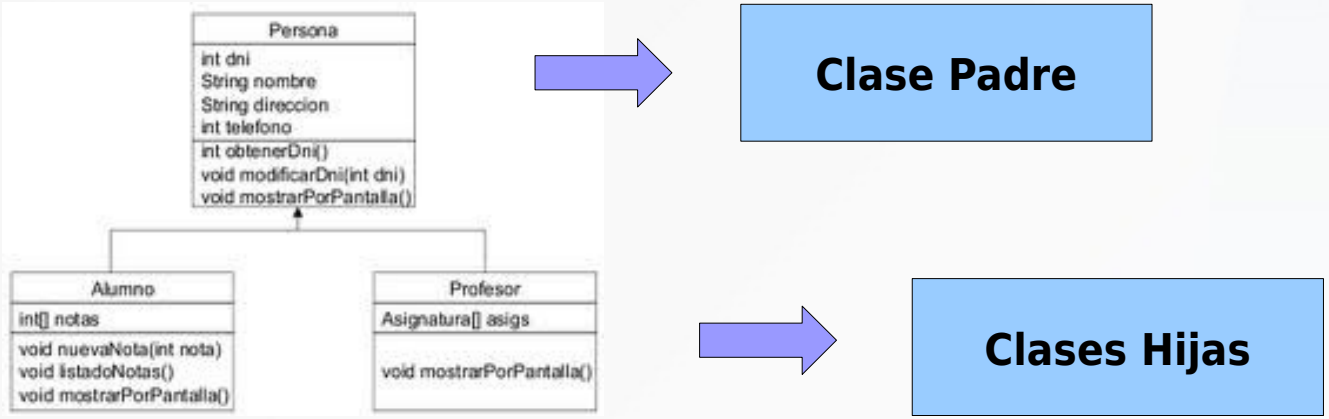
Programación IV

Programación Orientada a Objetos.

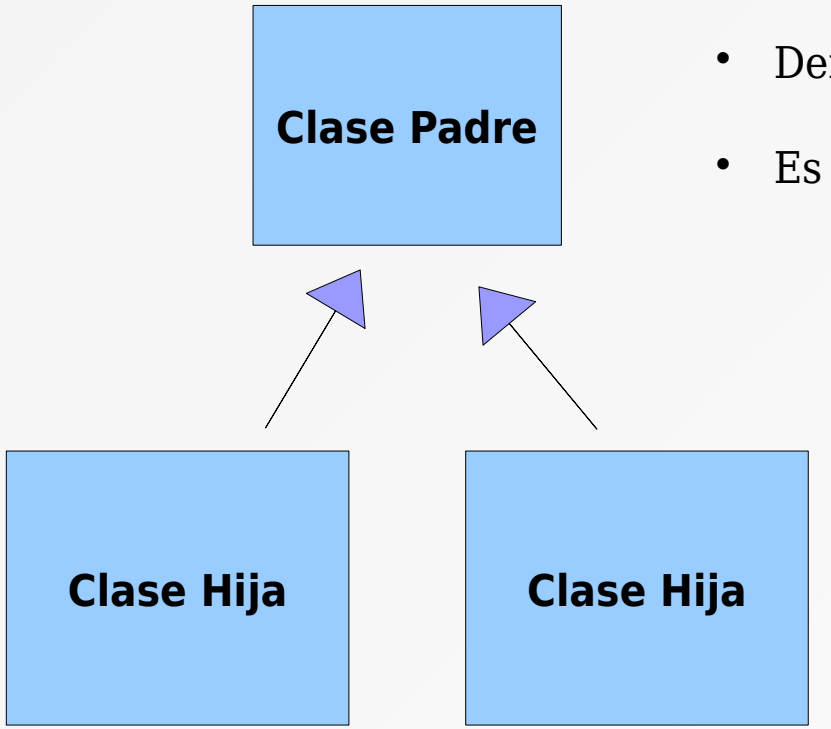
UNIVERSIDAD ADVENTISTA DEL PLATA

Herencia

- La reusabilidad puede lograrse mediante ***herencia***.
- Un comportamiento definido en una *superclase* es heredado por sus *subclases*.
- Las subclases extienden la funcionalidad heredada
- Esto nos permite definir la mayor cantidad de funcionalidades y atributos y luego reutilizarlas.



Herencia

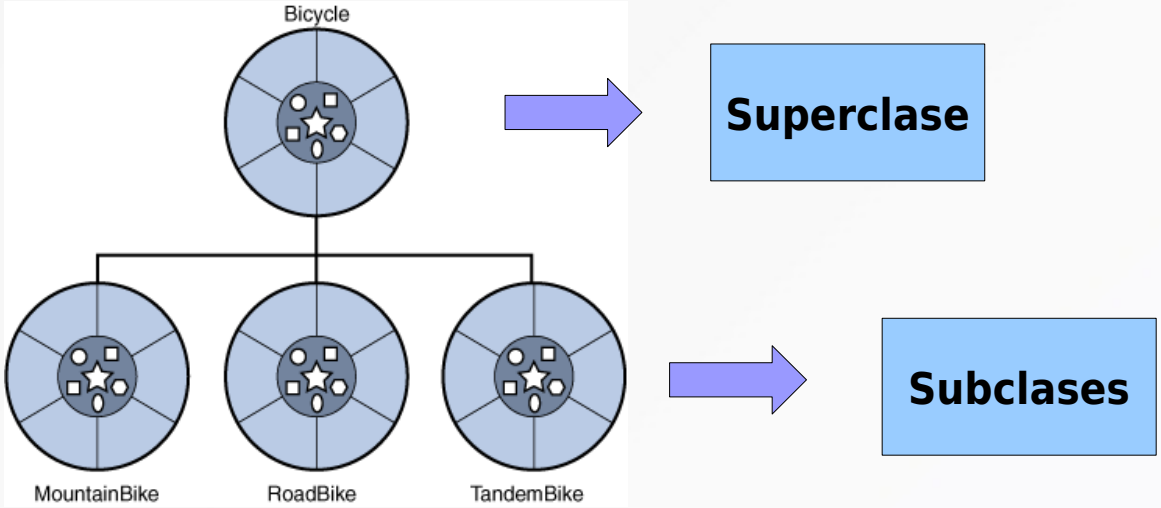


- Definimos atributos y comportamientos comunes.
- Es considerada como un tipo de generalización.

- Reutilizamos todo lo definido en la clase padre.
- Podemos ampliar atributos y comportamientos o *redifinirlos.*

Herencia

- Una clase hereda de su padre todos los atributos y métodos públicos y protegidos.
- Los constructores no son heredados pero pueden invocarse.
- Los métodos y atributos privados se heredan pero no son visibles.

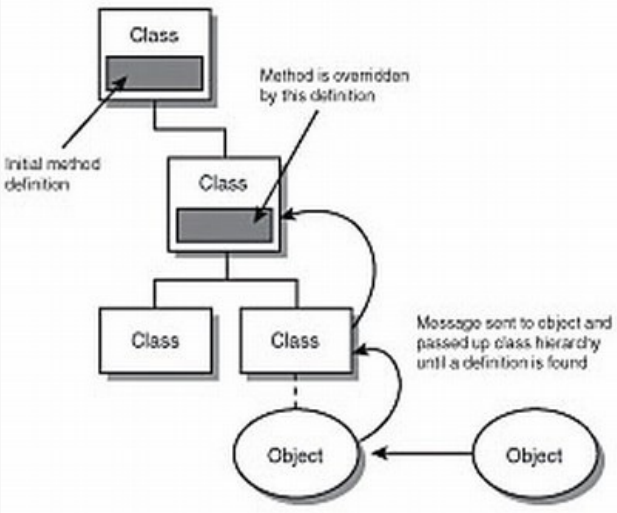


Herencia en Typescript

```
class Animal {
  constructor(public name: string) {}

  makeSound(): void {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  constructor(name: string, public breed: string) {
    super(name); // Llama al constructor de la clase base
  }

  makeSound(): void {
    console.log(`${this.name}, the ${this.breed}, barks.`);
  }
}

const myDog = new Dog("Buddy", "Golden Retriever");
myDog.makeSound(); // Output: Buddy, the Golden Retriever, barks.
```

Sobreescritura de métodos

- Se da en el contexto de relaciones de herencia.
- Consiste en reescribir la implementación de un método de *instancia*, con su mismo nombre y argumentos (*firma del método*).
- Si una clase hija sobreescribe un método de su clase padre, entonces ocultará al mismo.



Sobreescritura de métodos

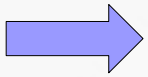
```
class Animal {
  hacerSonido(): void {
    console.log("Algún sonido genérico...");
  }
}

class Perro extends Animal {
  // Sobrescribimos el método de la clase padre
  hacerSonido(): void {
    console.log("¡Guau guau!");
  }
}

class Gato extends Animal {
  hacerSonido(): void {
    console.log("¡Miau!");
  }
}

const animales: Animal[] = [new Animal(), new Perro(), new Gato()];

animales.forEach(a => a.hacerSonido());
// Output:
// Algún sonido genérico...
// ¡Guau guau!
// ¡Miau!
```

```
class PerroGuardia extends Perro {
  hacerSonido(): void {
    super.hacerSonido();
    // Llama al "¡Guau guau!" de Perro
    console.log("¡Alerta! Intruso detectado.");
  }
}

new PerroGuardia().hacerSonido();
// ¡Guau guau!
// ¡Alerta! Intruso detectado.
```

Sobrecarga de métodos

- En TypeScript el compilador solo permite declarar varias firmas (overload signatures), pero la implementación es única.
- Esto es porque al final se transpila a JavaScript, y este no soporta sobrecarga real.

```
public class Artista {
    public void dibujar(int s) {
    }

    public void dibujar(String s) {
    }

    public void dibujar(int s, long y) {

    }
}
```

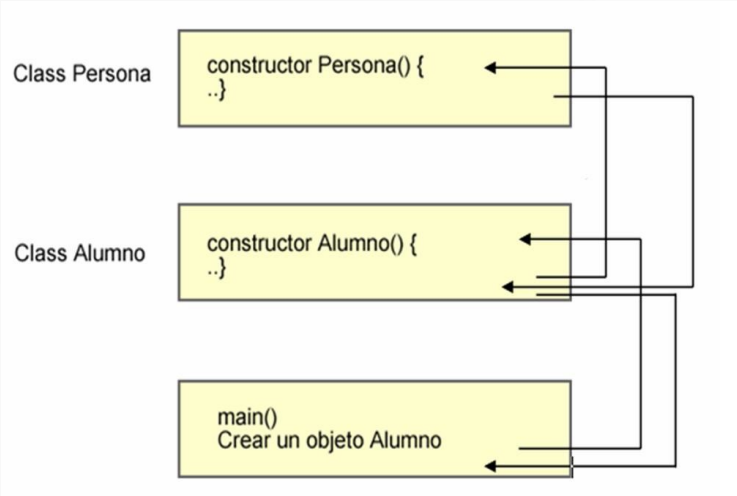


Métodos con el mismo nombre, pero con distinta firma.

Encadenamiento de constructores

- Debido a que los constructores son un tipo especial de métodos, también pueden sobrecargarse.

- Esto nos permite asegurar mínimamente como se inicialicen los objetos, no importando que constructor se utilice.



Encadenamiento de constructores

```
class Persona {
  nombre: string;
  edad: number;

  constructor(nombre: string, edad: number) {
    this.nombre = nombre;
    this.edad = edad;
    console.log(`Constructor Persona: ${this.nombre}, ${this.edad} años`);
  }
}

class Empleado extends Persona {
  puesto: string;

  constructor(nombre: string, edad: number, puesto: string) {
    super(nombre, edad); // Llamada obligatoria al constructor padre
    this.puesto = puesto;
    console.log(`Constructor Empleado: ${this.puesto}`);
  }
}

class Gerente extends Empleado {
  departamento: string;

  constructor(nombre: string, edad: number, puesto: string, departamento: string) {
    super(nombre, edad, puesto); // Encadenamiento hacia Empleado -> Persona
    this.departamento = departamento;
    console.log(`Constructor Gerente: ${this.departamento}`);
  }
}
```

Métodos y clases *abstract*

El calificador *abstract* condiciona el diseño de una jerarquía de herencia.

Clases Abstract:

- No pueden ser instanciadas.

```
abstract class Figura
```

Métodos Abstract:

- No pueden tener implementación.
- Deben ser implementados para las clases no abstractas que extiendan de su clase.

```
abstract calcularArea(): number;
```

Métodos y clases *abstract*

- Cuando un método está presente en todas las clases de una jerarquía pero se implementa en forma diferente conviene definirlo como abstracto.
- Una clase con al menos un método abstracto debe declararse como abstracta.
- Una clase que extiende de una clase abstracta debe:
  - ✓ Implementar todos sus métodos abstractos o …
  - ✓ Declararse como abstracta.

```
public class Alumno extends Persona {

    public void trabajar() {
        System.out.println("Ir a la escuela");

    }

}
```

```
public class Profesor extends Persona {

    @Override
    protected void trabajar() {
        System.out.println("Dar clases");

    }

}
```

Interfaces

- Bloques de códigos que poseen declaraciones de métodos y opcionalmente constantes.
- Todos sus métodos son abstractos, por ende no se definen.
- Una interface puede ser implementada por N clases.
- Una una clase implementa una interfaz se compromete a cumplir con su implementación, es decir se firma un contrato entre la clase y la interfaz.

Animal Interface

```
public interface Animal
 {
   public void locomotion();
   public void eat();
 }
```

Class Shark

```
public class Shark implements Animal
{
   public void locomotion()
   {
     System.out.println("I swim.");
   }
   public void eat()
   {
     System.out.println("I hunt for seals.");
   }
}
```

Class Dog

```
public class Dog implements Animal
{
  public void locomotion()
   {
     System.out.println("I run on four legs.");
   }
  public void eat()
   {
     System.out.println("I eat kibble.");
   }
}
```

```
public class  AnimalTest
{
  public static void main(String[] arg)
    {
      Shark shark = new Shark();
      shark.locomotion();
      shark.eat();
      Dog dog = new dog();
      dog.locomotion();
      dog.eat();
    }
}
```

Interfaces en Typescript

```
interface Person {
  name: string;
  age: number;
  greet(): void;
}

const person: Person = {
  name: "John",
  age: 30,
  greet() {
    console.log("Hello!");
  }
};

person.greet(); // Output: Hello!
```

```
interface Animal {
  name: string;
  age: number;
}

interface Dog extends Animal {
  breed: string;
}

const myDog: Dog = {
  name: "Buddy",
  age: 3,
  breed: "Golden Retriever"
};
```

Interfaces en Typescript

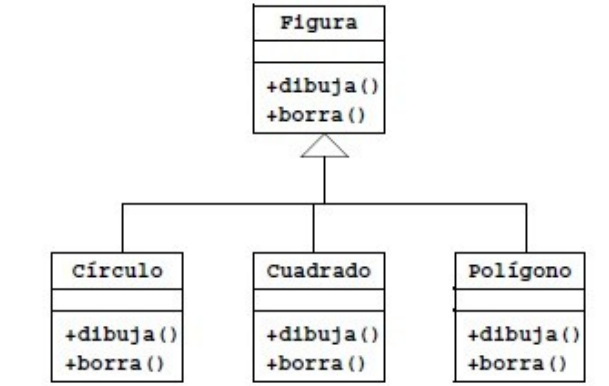
```
interface Drivable {
  drive(): void;
}

class Car implements Drivable {
  drive(): void {
    console.log("Driving the car.");
  }
}

const myCar = new Car();
myCar.drive(); // Output: Driving the car.
```

Polimorfismo

- Una variable de referencia cambia el comportamiento según el tipo de objeto al que apunta.

- Una variable traba a objetos de una clase como objetos de una superclase y se invoca dinámicamente el método correspondiente (*binding dinámico*)

- Si tenemos un método que espera como parámetro una variable de clase X, podemos invocarlo usando subclases pasando como parámetros referencias a objetos instancia de subclases de X.



Preguntas


UNIVERSIDAD
ADVENTISTA
DEL PLATA