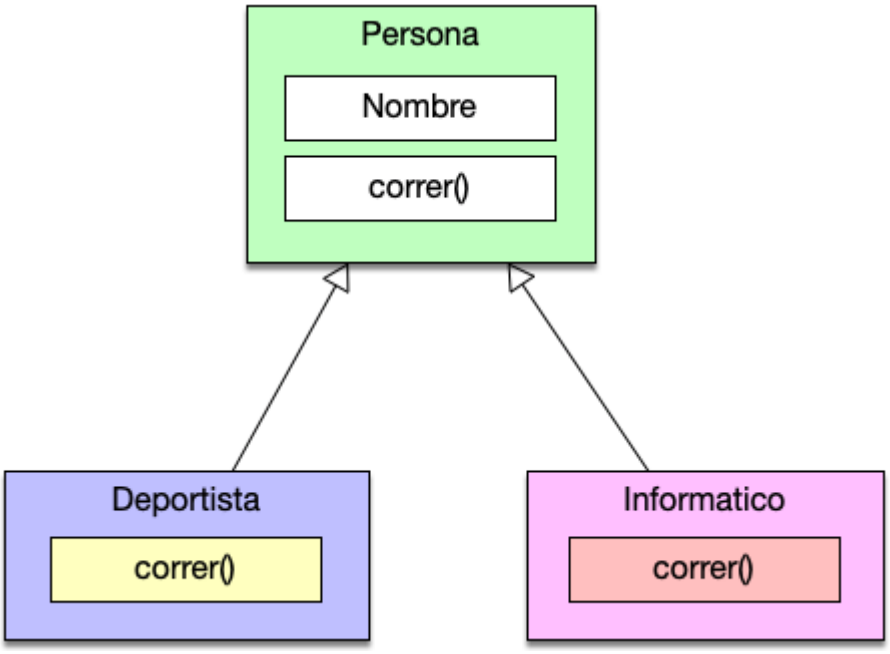
Los conceptos de Java Polimorfismo y herencia están intimamente relacionados . Siempre el Polimorfismo ha sido muy difícil de entender para la mayor parte de los programadores.

¿Para que sirve el Polimorfismo?

Vamos a explicarlo para ello partiremos de un ejemplo en el cual tenemos las siguientes clases

- Persona
- Deportista
- Informatico.

Estas clases están organizadas en una jerarquía de herencia en la cual la clase Persona es la clase padre y el resto heredan de ella.



El método correr de la clase Persona es un método abstracto y no tiene implementación . Por el contrario los métodos de la clases hijas tienen sobrecargado el método correr. El deportista correrá a 7 hm/hora y el informatico a 2km/h . Vamos a ver el código:

```
package com.arquitecturajava;
```

```
public abstract class Persona {

	private String nombre;

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public Persona(String nombre) {
		super();
		this.nombre = nombre;
	}
	public abstract int  correr() ;
}
```

```
package com.arquitecturajava;

public class Deportista extends Persona {

	public Deportista(String nombre) {
		super(nombre);
		// TODO Auto-generated constructor stub
	}

	@Override
	public int correr() {
		// TODO Auto-generated method stub
		return 7;
```

```
	}

}
```

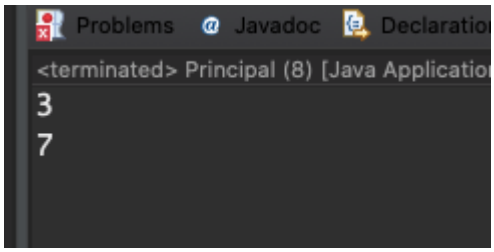
```
package com.arquitecturajava;

public class Ingeniero extends Persona{

	public Ingeniero(String nombre) {
		super(nombre);
	}

	@Override
	public int correr() {
		return 3;
	}

}
```

Si construimos un programa principal veremos cómo el método correr se ejecuta de forma diferente para cada una de las clases hijas . Es decir se ejecuta de forma polimorfica (diferente forma) en cada una de las clases . La clase persona lo tiene como método abstracto y cada clase hija lo sobre escribe:

```
package com.arquitecturajava;

public class Principal {

	public static void main(String[] args) {
		Ingeniero i= new Ingeniero("pedro");
		Deportista d= new Deportista("gema");
		System.out.println(i.correr());
```

```
                System.out.println(d.correr());
        }

}
```

El resultado se puede ver en la consola:

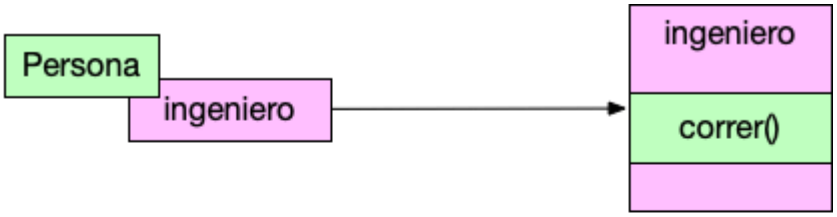


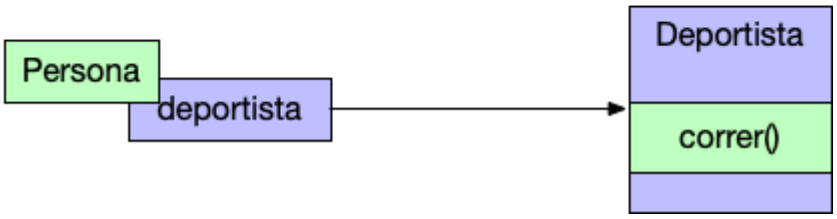
Ahora bien este mismo código se puede escribir de la siguiente manera:

```
package com.arquitecturajava;

public class Principal2 {

        public static void main(String[] args) {
                Persona i= new Ingeniero("pedro");
                Persona d= new Deportista("gema");
                System.out.println(i.correr());
                System.out.println(d.correr());
        }

}
```

Java Polimorfismo y uso de clases

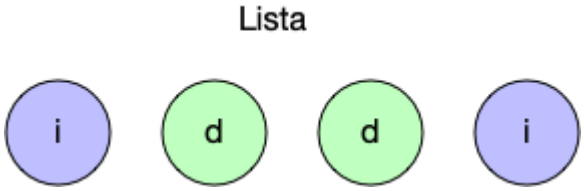
Cuando la gente empieza a programar en Java , este código puede sonarle raro ya que estamos apuntando con una variable de tipo Persona a un Ingeniero y a un Deportista . Esto no es problemático ya que ambas son personas y ambas sobreescriben el método correr() que persona tiene declarado como abstracto.





Java Polimorfismo y simplificación

Muchas veces a las programadores les resulta difícil entender porque esta solución es más flexible que la anterior. El uso del Polimorfismo en Java genera código más sencillo . La respuesta a esto viene si por ejemplo tenemos una lista de Deportistas e Ingenieros



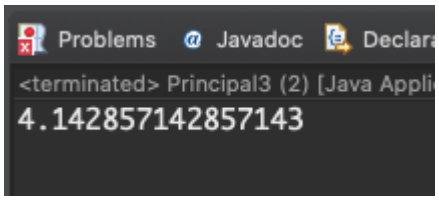
Queremos obtener la media de velocidad de todas las personas no nos importa que sean Deportistas o Ingenieros. No podemos tratar a cada tipo de forma independiente sino que tenemos que usar la clase padre para tratarlos de forma idéntica y generar una lista de Personas aunque estas almacenen Deportistas e Ingenieros.

```
import java.util.Arrays;
import java.util.List;

public class Principal3 {

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

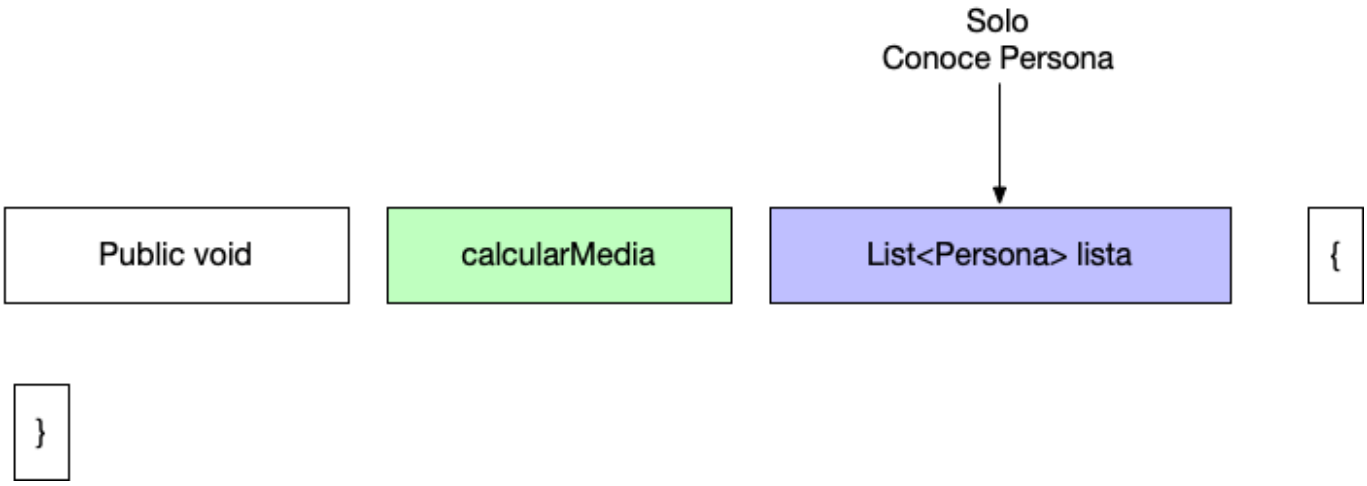
```
        Persona i1 = new Ingeniero("pedro");
        Persona d1 = new Deportista("gema");
        Persona i2 = new Ingeniero("angel");
        Persona i3 = new Ingeniero("antonio");
        Persona i4 = new Ingeniero("maria");
        Persona d5 = new Deportista("david");

        List<Persona> lista = Arrays.asList(i1, d1, i2, i3, i4, i4,
d5);
        double resultado = calcularMediaVelocidad(lista);
        System.out.println(resultado);
    }

    public static double calcularMediaVelocidad(List<Persona> lista) {
        double total = 0;
        int contador = 0;

        for (Persona p : lista) {
            total = total + p.correr();
            contador++;
        }

        if (contador > 0) {
            return total / contador;
        } else {
            return 0; // Manejar el caso en que la lista esté vacía
para evitar la división por cero.
        }
    }
}
```

Al recorrer la lista y usar la variable p de tipo Persona para invocar el método correr cada

objeto se comportará como el tipo que es . Devolviendo el resultado que corresponda y obteniendo la media



La clave se encuentra en darnos cuenta que el polimorfismo nos ayuda a simplificar el código y reducir el numero de clases que los programadores deben conocer ya que en este caso el desarrollador encargado del método que calcula la media no le hace falta conocer todos los tipos de clases existentes es suficiente con conocer la clase padre Persona.



Java polimorfismo y programación funcional

De forma muy similar podemos calcular la media utilizando Java Polimorfismo y combinandolo con programación funcional.

```
import java.util.Arrays;
import java.util.List;
import java.util.OptionalDouble;

public class Principal3 {
```

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

                Persona i1 = new Ingeniero("pedro");
                Persona d1 = new Deportista("gema");
                Persona i2 = new Ingeniero("angel");
                Persona i3 = new Ingeniero("antonio");
                Persona i4 = new Ingeniero("maria");
                Persona d5 = new Deportista("david");

                List<Persona> lista = Arrays.asList(i1, d1, i2, i3,
i4, i4, d5);
                OptionalDouble resultado =
calcularMediaVelocidad(lista);
                if (resultado.isPresent()) {

                        System.out.println(resultado.getAsDouble());
                }
        }

        public static OptionalDouble
calcularMediaVelocidad(List<Persona> lista) {

                return
lista.stream().mapToDouble(Persona::correr).average();
        }
}
```

Conclusiones

El concepto de Java Polimorfismo nos ayuda a la hora de generar flexibilidad en el código pero sobre todo también a la hora de simplificar el número de conceptos que un programador debe manejar.

El polimorfismo y la herencia son pilares fundamentales de la Programación Orientada a Objetos (POO). Aunque estos conceptos se presentan tempranamente en el aprendizaje de Java, su correcta aplicación en proyectos reales puede marcar la diferencia entre un código legible y fácil de mantener, o un sistema propenso a errores y difícil de extender.”

Herencia y limitaciones

Aunque la herencia puede parecer la solución ideal en muchos casos, su uso inapropiado puede generar sistemas rígidos y difíciles de mantener. Aplicar el principio de ‘composición sobre herencia’ es crucial para evitar diseños monolíticos y poco flexibles recordemoslo.

Otros artículos relacionados

- Java Herencia
- Java Herencia y limitaciones
- Interface y el concepto de simplicidad
- Curso Programación orientada a objeto
- Java Herencia ejemplos