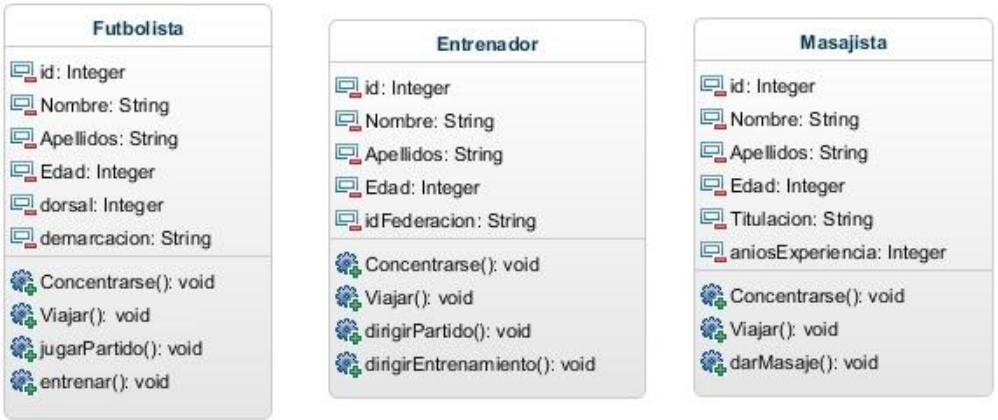
La Herencia[1]

La Herencia es uno de los 4 pilares de la programación orientada a objetos (POO) junto con la ***Abstracción***, ***Encapsulación*** y ***Polimorfismo***. Al principio cuesta un poco entender estos conceptos característicos del paradigma de la POO porque solemos venir de otro paradigma de programación como el paradigma de la programación estructurada (ver la entrada"Paradigmas de Programación), pero se ha de decir que la complejidad está en entender este nuevo paradigma y no en otra cosa. En esta entrada vamos a explicar de la mejor manera posible que es la herencia y lo vamos a explicar con un ejemplo.

Respecto a la herencia se han dado muchas definiciones como por ejemplo la siguiente: "*La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos.*". Así de primeras esta definición es un poco difícil de digerir para aquellos que estéis empezando con la POO, así que vamos a intentar digerir esta definición con un ejemplo en el que veremos que la herencia no es más que un "***Copy-Paste Dinámico***" o una forma de "***sacar factor común***" al código que escribimos.

El ejemplo que proponemos es un caso en el que vamos a simular el comportamiento que tendrían los diferentes integrantes de la selección española de futbol; tanto los Futbolistas como el cuerpo técnico (Entrenadores, Masajistas, etc…). Para simular este comportamiento vamos a definir tres clases que van a representaran a objetos Futbolista, Entrenador y Masajista. De cada unos de ellos vamos a necesitar algunos datos que reflejaremos en los ***atributos*** y una serie de acciones que reflejaremos en sus***métodos***. Estos atributos y métodos los mostramos en el siguiente diagrama de clases:



[1] Extret de: http://jarroba.com/herencia-en-la-programacion-orientada-a-objetos-ejemplo-en-java/

NOTA: en este diagrama y en adelante no vamos a poner los constructores y métodos getter y setter con el fin de que el diagrama nos quede grande e "intendible" aunque en un buen diagrama de clases deberían aparecer para respetar el principio de encapsulación de la POO

Como se puede observar, vemos que en las tres clases tenemos atributos y métodos que con iguales ya que los tres tienen los atributos *id, Nombre, Apellidos y Edad*; y los tres tienen los métodos de *Viajar* y *Concentrarse:*

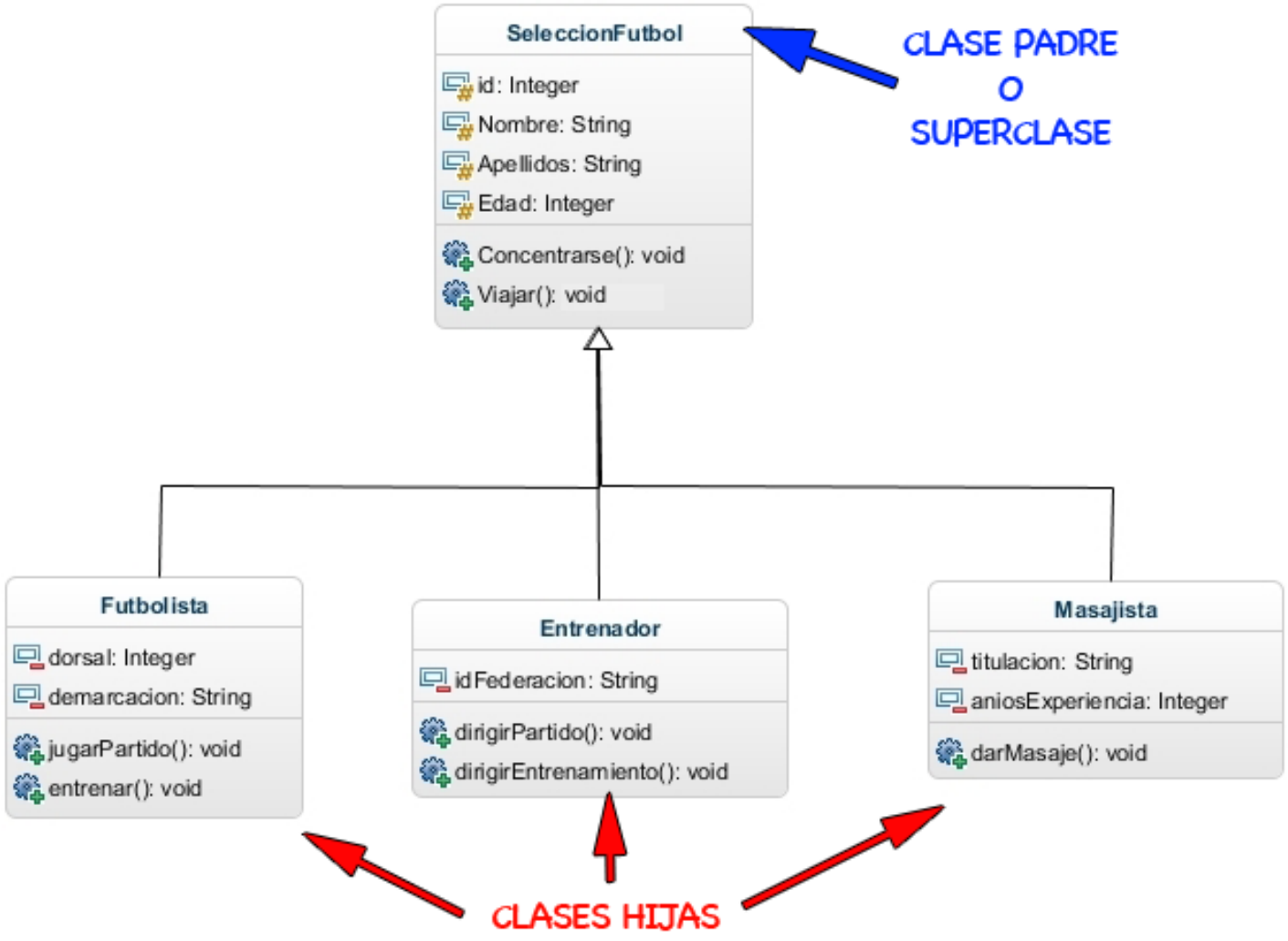
A nivel de código tenemos lo siguiente tras ver el diagrama de clases:

```
public class Futbolista
{
        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private int dorsal;
        private String demarcacion;

    // constructor, getter y setter

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }

        public void jugarPartido() {
                ...
        }

        public void entrenar() {
                ...
        } }
```

```
public class Entrenador
{
        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private String idFederacion;

                // constructor, getter y setter

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }

        public void dirigirPartido() {
                ...
        }

        public void dirigirEntreno() {
                ...
        }
}
```

```
public class Masajista
    {
        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private String Titulacion;
        private int aniosExperiencia;

        // constructor, getter y setter

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }

        public void darMasaje() {
                ...
        }
    }
```

Lo que podemos ver en este punto es que estamos escribiendo mucho código repetido ya que las tres clases tienen métodos y atributos comunes, de ahi y como veremos enseguida, decimos que la herencia consiste en "***sacar factor común***" para no escribir código de más, por tanto lo que haremos sera ***crearnos una clase con el "código que es común a las tres clases" (a esta clase se le denomina en la herencia como "Clase Padre o SuperClase")*** y ***el código que es especifico de cada clase***, lo dejaremos en ella, siendo ***denominadas estas clases como "Clases Hijas",*** las cuales ***heredan de la clase padre todos los atributos y métodos públicos o protegidos***. Es muy importante decir que las clases hijas ***no van a heredar nunca los atributos y métodos privados de la clase padre***, así que mucho cuidado con esto. En resumen para que veáis la ventaja de la herencia, tenemos ahora una clase padre con 'n' lineas de código y tres clases hijas con 'a', 'b' y 'c' lineas de códigos respectivamente, por tanto si hecháis cuentas, ***hemos reducido nuestro código en '2n' líneas menos ya que antes teníamos '(n+a)+(n+b)+(n+c)' líneas de código y ahora tras aplicar herencia tenemos 'n+a+b+c' líneas***, aunque también es cierto que tenemos una clase más, pero veremos un poco más adelante la ventaja de tener esa clase padre. En resumen, al "sacar factor común" y aplicar herencia, tenemos las siguientes clases:



A nivel de código, las clases quedarían implementadas de la siguiente forma:

```
public class SeleccionFutbol
{

        protected int id;
        protected String Nombre;
        protected String Apellidos;
        protected int Edad;

        // constructor, getter y setter

        public void Concentrarse() {
                ...
        }

        public void Viajar() {
                ...
        }
}
```

```
public class Futbolista extends SeleccionFutbol
{
  private int dorsal;
  private String demarcacion;

   public Futbolista() {
          super();
  }

   // getter y setter

  public void jugarPartido() {
                ...
   }

   public void entrenar() {
                ...
  }
}
```

```
public class Entrenador extends SeleccionFutbol
{

   private String idFederacion;

   public Entrenador() {
          super();
   }

   // getter y setter

   public void dirigirPartido() {
                ...
   }

   public void dirigirEntreno() {
                ...
   }
}
```

```
public class Masajista extends SeleccionFutbol
{

   private String Titulacion;
   private int aniosExperiencia;

   public Masajista() {
          super();
   }

   // getter y setter

   public void darMasaje() {
                ...
   }
}
```

Como podéis observar ahora queda un ***código mucho más limpio, estructurado y con menos líneas de código, lo que lo hace más legible***, cosa que es muy importante y lo que todavía lo hace más importante es que ***es un código reutilizable***, lo que significa que ahora si queremos añadir más clases a nuestra aplicación como por ejemplo una clase Médico, Utiller@, Jefe/a de prensa etc. que pertenezcan también al equipo técnico de la selección Española, lo podemos hacer de forma muy sencilla ya que en la clase padre (SeleccionFutbol) tenemos implementado parte de sus datos y de

su comportamiento y solo habrá que implementar los atributos y métodos propios de esa clase. ¿Empezáis a ver la utilidad de la herencia?.

Ahora si os habéis fijado bien en el código que se ha escrito y sino habéis tenido experiencia con la herencia en Java, habréis podido observar dos palabras reservadas "nuevas" como son "***extends***", "***protected***" y "***super***". Pues bien, ahora vamos a explicar el significado de ellas:

- ***extends***: Esta palabra reservada, indica a la clase hija cual va a ser su clase padre, es decir que por ejemplo en la clase Futbolista al poner "*public class Futbolista extends SeleccionFutbol*" le estamos indicando a la clase 'Futbolista' que su clase padre es la clase 'SeleccionFutbol' o dicho de otra manera para que se entienda mejor, al poner esto estamos haciendo un "***copy-paste dinámico***" diciendo a la clase 'Futbolista' que se 'copie' todos los atributos y métodos públicos o protegidos de la clase 'SeleccionFutbol'. De aquí viene esa 'definición' que dimos de que la herencia en un 'copy-paste dinámico'.
- ***protected***: sirve para indicar un tipo de visibilidad de los atributos y métodos de la clase padre y significa que cuando un atributo es 'protected' o protegido, solo es visible ese atributo o método desde una de las clases hijas y no desde otra clase.
- ***super***: sirve para llamar al constructor de la clase padre. Quizás en el código que hemos puesto no se ha visto muy bien, pero a continuación lo mostramos de formas más clara, viendo el constructor de los objetos pasándole los atributos:

```
public class SeleccionFutbol {

	.....
	public SeleccionFutbol() {
	}
	public SeleccionFutbol(int id, String nombre, String apellidos, int edad) {
		this.id = id;
		this.Nombre = nombre;
		this.Apellidos = apellidos;
		this.Edad = edad;
	}
	.....
public class Futbolista extends SeleccionFutbol {
	.....
	public Futbolista() {
		super();
	}
	public Futbolista(int id, String nombre, String apellidos, int edad, int dorsal, String
demarcacion) {
		super(id, nombre, apellidos, edad);
		this.dorsal = dorsal;
		this.demarcacion = demarcacion;
	}
	.....
```

Hasta aquí todo correcto, pero ahora vamos a ver como trabajamos con estas clases. Para ver este funcionamiento de forma clara y sencilla vamos a trabajar con un objeto de cada clase y vamos a ver como se crean y de que forma ejecutan sus método. Para ello empecemos mostrando el siguiente fragmento de código:

```
public class Main {

	// ArrayList de objetos SeleccionFutbol. Idenpendientemente de la clase hija a la que pertenezca el objeto
	public static ArrayList<SeleccionFutbol> integrantes = new ArrayList<SeleccionFutbol>();

	public static void main(String[] args) {

	  Entrenador delBosque = new Entrenador(1, "Vicente", "Del Bosque", 60, "284EZ89");
	  Futbolista iniesta = new Futbolista(2, "Andres", "Iniesta", 29, 6, "Interior Derecho");
	  Masajista raulMartinez = new Masajista(3, "Raúl", "Martinez", 41, "Licenciado en Fisioterapia", 18);

	  integrantes.add(delBosque);
	  integrantes.add(iniesta);
	  integrantes.add(raulMartinez);

		// CONCENTRACION
	  System.out.println("Todos los integrantes comienzan una concentracion. (Todos ejecutan el mismo método)");
	for (SeleccionFutbol integrante : integrantes) {
		System.out.print(integrante.getNombre()+" "+integrante.getApellidos()+" -> ");
		integrante.Concentrarse();
	}

	// VIAJE
	System.out.println("\nTodos los integrantes viajan para jugar un partido. (Todos ejecutan el mismo método)");
	for (SeleccionFutbol integrante : integrantes) {
		System.out.print(integrante.getNombre()+" "+integrante.getApellidos()+" -> ");
		integrante.Viajar();
	}
	.....
```

Lo primero que vemos es que nos creamos un objeto de cada clase, pasándole los atributos al constructor como parámetro y después "sorprendentemente" ***los metemos en un "ArrayList" de objetos de la clase "SeleccionFutbol"*** que es la clase padre. Esto evidentemente te lo permite hacer ya que todos los objetos son hijos de la misma clase padre. Luego como veis, recorremos el ArrayList y ejecutamos sus métodos "comunes" como son el 'Concentrarse' y el 'Viajar'. Este código da como salida lo siguiente:

Todos los integrantes comienzan una concentracion. (Todos ejecutan el mismo método)

Vicente Del Bosque -> Concentrarse

Andres Iniesta -> Concentrarse

Raúl Martinez -> Concentrarse

Todos los integrantes viajan para jugar un partido. (Todos ejecutan el mismo método)

Vicente Del Bosque -> Viajar

Andres Iniesta -> Viajar

Raúl Martinez -> Viajar

Como veis al ejecutar todos el mismo método de la clase padre el código puesto funciona correctamente.

Posteriormente vamos a ejecutar código especifico de las clases hijas, de ahi que ahora no podamos recorrer el ArrayList y ejecutar el mismo método para todos los objetos ya que ahora esos objetos son únicos de la clases hijas. El código es el siguiente:

```
// ENTRENAMIENTO
System.out.println("\nEntrenamiento: Solamente el entrenador y el futbolista tiene metodos para entrenar:");
System.out.print(delBosque.getNombre()+" "+delBosque.getApellidos()+" -> ");
delBosque.dirigirEntrenamiento();
System.out.print(iniesta.getNombre()+" "+iniesta.getApellidos()+" -> ");
iniesta.entrenar();

// MASAJE
System.out.println("\nMasaje: Solo el masajista tiene el método para dar un masaje:");
System.out.print(raulMartinez.getNombre()+" "+raulMartinez.getApellidos()+" -> ");
raulMartinez.darMasaje();

// PARTIDO DE FUTBOL
System.out.println("\nPartido de Fútbol: Solamente el entrenador y el futbolista tiene metodos para el partido de fútbol:");
System.out.print(delBosque.getNombre()+" "+delBosque.getApellidos()+" -> ");
delBosque.dirigirPartido();
System.out.print(iniesta.getNombre()+" "+iniesta.getApellidos()+" -> ");
iniesta.jugarPartido();
```

Como vemos aunque el entrenador y los futbolistas asistan a un entrenamiento, los dos hacen una función diferente en el mismo, por tanto hay que hacer métodos diferente para cada una de las

clases. Ya veremos cuando hablemos del polimorfismo que podremos ejecutar el mismo método para clases diferentes y que esos métodos hagan cosas distintas. Como resultado al código mostrado tenemos lo siguiente:

```
Entrenamiento: Solamente el entrenador y el futbolista tiene metodos para entrenar:
Vicente Del Bosque -> Dirige un entrenamiento
Andres Iniesta -> Entrena
Masaje: Solo el masajista tiene el método para dar un masaje:
Raúl Martinez -> Da un masaje
Partido de Fútbol: Solamente el entrenador y el futbolista tiene metodos para el partido de fútbol:
Vicente Del Bosque -> Dirige un partido
Andres Iniesta -> Juega un partido
```

CONCLUSIONES Y ACLARACIONES:

Esto ha sido todo lo que hemos contado sobre la herencia en esta entrada. El tema de la herencia es un tema que puede ser un poco más complejo de lo que lo hemos contado aquí, ya que solo hemos contado lo que es la herencia simple (ya que Java por el momento es el único tipo de herencia que soporta) y no la herencia múltiple, que es un tipo de herencia en la que una clase hija puede tener varios padres, aunque por el momento si estáis empezando a aprender el concepto de la herencia, con la herencia simple tenéis más que suficiente. Para los que os estéis iniciando en el mundo de la ingeniería informática, habréis podido ver que hemos puesto unos ejemplo mostrando unos diagramas "un poco raros"; pues bien, estos diagramas se llaman diagramas de clases (que los hemos realizado con la herramienta web de www.genmymodel.com) y sirven para representar de forma gráfica los atributos y métodos de las clases y las relaciones entre ellos, utilizando el lenguaje UML del cual intentaremos hablar más adelante en otros tutoriales. Por último decir y aclarar que en esta entrada quizás no hemos utilizado una terminología correcta para explicar la herencia, pero lo hemos explicado de una forma algo distinta a como esta explicada por ahi para que los que empeceis podais entender la herencia desde otro punto de vista.