Ejercicio:

Construir un programa para verificar que corredor hace la mayor cantidad de Km.
El programa debe gestionar una serie de corredores caracterizados por su número de atleta, nombre y km recorridos. Al final del programa debe mostrarse los datos del corredor con mayor km recorridos.

Herencia en POO

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.".
En resumen: la herencia no es más que un "***Copy-Paste Dinámico***" o una forma de "***sacar factor común***" al código que escribimos.

La idea de la herencia es permitir la creación de nuevas clases basadas en clases existentes.
Cuando heredamos de una clase existente, reusamos (o heredamos) métodos y campos, y agregamos nuevos campos y métodos para cumplir con la situación nueva.
Cada vez que encontremos la relación ***"es-un"*** *entre dos clases, estamos ante la presencia de herencia.*

Veamos un ejemplo:

Como se puede observar, vemos que en las tres clases tenemos atributos y métodos que son 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:

```java
public class Futbolista
{

        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private int dorsal;
        private String demarcacion;

        // constructor, getter y se
tter

        public void Concentrarse()
{
                ...
        }

        public void Viajar() {
                ...
        }

        public void jugarPartido()
{
                ...
        }

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

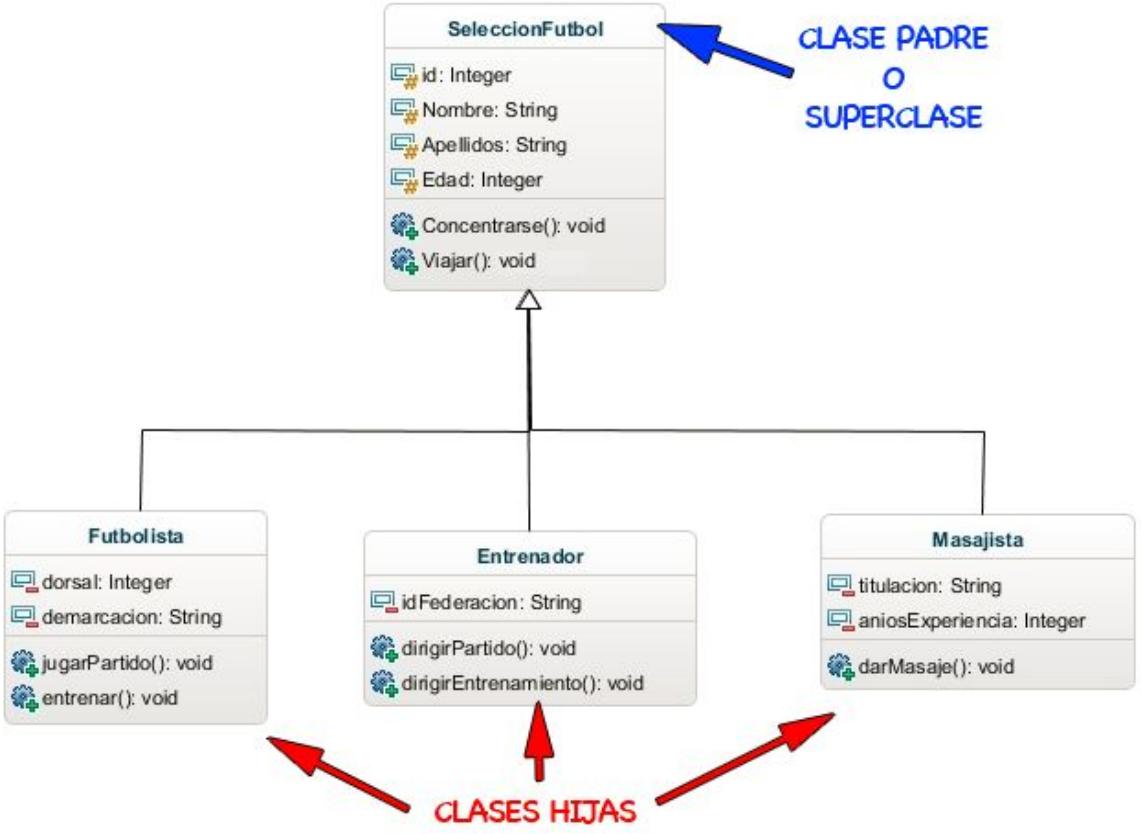
```java
public class Entrenador
{

        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private String idFederacion
;

        // constructor, getter y se
tter

        public void Concentrarse()
{
                ...
        }

        public void Viajar() {
                ...
        }

        public void dirigirPartido
() {
                ...
        }

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

```java
public class Masajista
{

        private int id;
        private String Nombre;
        private String Apellidos;
        private int Edad;
        private String Titulacion;
        private int aniosExperiencia;

        // constructor, getter y sett
er

        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 ahí y como veremos enseguida, decimos que la herencia consiste en "**sacar factor común**" para no escribir código de más, por tanto lo que haremos será **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 específico de cada clase**, lo dejaremos en ella, siendo **denominadas estas clases como "Clases Hijas",**.*

***IMPORTANTE**: **"Las Clases Hijas" heredan de la clase padre todos los atributos y métodos públicos o protegidos**. Pero **no van a heredar nunca los atributos y métodos privados de la clase padre**, así que mucho cuidado con esto.*

El Diagrama de Clase nos queda:



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

```java
public class SeleccionFutbol
{

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

        // constructor, getter y se
tter

        public void Concentrarse()
{
                ...
        }

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

```java
public class Futbolista extends Sel
eccionFutbol
{
        private int dorsal;
        private String demarcacion;

        public Futbolista() {
                super();
        }

        // getter y setter

        public void jugarPartido()
{
                ...
        }

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

```java
public class Entrenador extends Sel
eccionFutbol
{

        private String idFederacion
;

        public Entrenador() {
                super();
        }

        // getter y setter

        public void dirigirPartido
() {
                ...
        }

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

```java
public class Masajista extends Selecc
ionFutbol
{

        private String Titulacion;
        private int aniosExperiencia;

        public Masajista() {
                super();
        }

        // getter y setter

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

*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.*

*Ahora en el código que se ha escrito hemos usado las 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 cuál 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:

```java
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, 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
```

Al ejecutar todos el mismo método de la clase padre el código puesto funciona correctamente.

Posteriormente vamos a ejecutar código específico de las clases hijas, de ahí 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:

```java
// 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
```

ArrayList

Primero para poder usarlo debemos importar ***java.util.ArrayList .***
La clase ArrayList en Java, es una clase que permite almacenar datos en memoria de forma similar a los Arrays, con la ventaja de que el número de elementos que almacena, lo hace de forma dinámica, es decir, que no es necesario declarar su tamaño como pasa con los Arrays.

Los principales métodos para trabajar con los ArrayList son los siguientes:

```java
// Declaración de un ArrayList de "String". Puede ser de cualquier otro Elemento u Objeto (float, Boolean, Object, ...)
ArrayList<String> nombreArrayList = new ArrayList<String>();
// Añade el elemento al ArrayList
nombreArrayList.add("Elemento");
// Añade el elemento al ArrayList en la posición 'n'
nombreArrayList.add(n, "Elemento 2");
// Devuelve el numero de elementos del ArrayList
nombreArrayList.size();
// Devuelve el elemento que esta en la posición '2' del ArrayList
nombreArrayList.get(2);
// Comprueba se existe del elemento ('Elemento') que se le pasa como parametro
nombreArrayList.contains("Elemento");
// Devuelve la posición de la primera ocurrencia ('Elemento') en el ArrayList
nombreArrayList.indexOf("Elemento");
// Devuelve la posición de la última ocurrencia ('Elemento') en el ArrayList
nombreArrayList.lastIndexOf("Elemento");
// Borra el elemento de la posición '5' del ArrayList
nombreArrayList.remove(5);
// Borra la primera ocurrencia del 'Elemento' que se le pasa como parametro.
nombreArrayList.remove("Elemento");
//Borra todos los elementos de ArrayList
nombreArrayList.clear();
// Devuelve True si el ArrayList esta vacio. Sino Devuelve False
nombreArrayList.isEmpty();
// Copiar un ArrayList
ArrayList arrayListCopia = (ArrayList) nombreArrayList.clone();
// Pasa el ArrayList a un Array
Object[] array = nombreArrayList.toArray();
```

Otra cosa muy importante a la hora de trabajar con los ArrayList son los "Iteradores" (Iterator). Los Iteradores sirven para recorrer los ArrayList y poder trabajar con ellos. Los Iteradores solo tienen tres métodos que son el "hasNext()" para comprobar que siguen quedando elementos en el iterador, el "next()" para que nos de el siguiente elemento del iterador; y el "remove()" que sirve para eliminar el elemento del Iterador.

En el siguiente fragmento de código, declaramos un ArrayList de Strings y lo rellenamos con 10 Strings (Elemento i). Esto lo hacemos con el método "add()". Después añadimos un nuevo elemento al ArrayList en la posición '2' (con el metodo "add(posición,elemento)") que le llamaremos "Elemento 3" y posteriormente imprimiremos el contenido del ArrayList, recorriendolo con un Iterador. El fragmento de este código es el siguiente:

```java
// Declaración el ArrayList
ArrayList<String> nombreArrayList = new ArrayList<String>();

// Añadimos 10 Elementos en el ArrayList
for (int i=1; i<=10; i++){
        nombreArrayList.add("Elemento "+i);
}

// Añadimos un nuevo elemento al ArrayList en la posición 2
nombreArrayList.add(2, "Elemento 3");

// Declaramos el Iterador e imprimimos los Elementos del ArrayList
Iterator<String> nombreIterator = nombreArrayList.iterator();
while(nombreIterator.hasNext()){
        String elemento = nombreIterator.next();
        System.out.print(elemento+" / ");
}
```

Ejecutando esta código obtenemos por pantalla lo siguiente:

```
Elemento 1 / Elemento 2 / Elemento 3 / Elemento 3 / Elemento 4 / Elemento 5 / Elemento 6 / Elemento 7 / Elemento 8 / Elemento 9 / El
```

Como se observa en el resultado tenemos repetido el elemento "Elemento 3" dos veces y esto lo hemos puesto a proposito para mostrar el siguiente ejemplo. Ahora para seguir trabajando con los ArrayList, lo que vamos ha hacer es mostrar el numero de elementos que tiene el ArrayList y después eliminaremos el primer elemento del ArrayList y los elementos del ArrayList que sean iguales a "Elemento 3", que por eso lo hemos puesto repetido. El "Elemento 3" lo eliminaremos con el metodo "remove()" del iterador. A continuación mostramos el código que realiza lo descrito:

```java
// Recordar que previamente ya hemos declarado el ArrayList y el Iterator de la siguiente forma:
// ArrayList<String> nombreArrayList = new ArrayList<String>();
// Iterator<String> nombreIterator = nombreArrayList.iterator();

// Obtenemos el numero de elementos del ArrayList
int numElementos = nombreArrayList.size();
System.out.println("nnEl ArrayList tiene "+numElementos+" elementos");

// Eliminamos el primer elemento del ArrayList, es decir el que ocupa la posición '0'
System.out.println("n... Eliminamos el primer elemento del ArrayList ("+nombreArrayList.get(0)+")...");
nombreArrayList.remove(0);

// Eliminamos los elementos de ArrayList que sean iguales a "Elemento 3"
System.out.println("n... Eliminamos los elementos de ArrayList que sean iguales a "Elemento 3" ...");
nombreIterator = nombreArrayList.iterator();
while(nombreIterator.hasNext()){
        String elemento = nombreIterator.next();
        if(elemento.equals("Elemento 3"))
                nombreIterator.remove();        // Eliminamos el Elemento que hemos obtenido del Iterator
}

// Imprimimos el ArrayList despues de eliminar los elementos iguales a "Elemento 3"
System.out.println("nImprimimos los elementos del ArrayList tras realizar las eliminaciones: ");
nombreIterator = nombreArrayList.iterator();
while(nombreIterator.hasNext()){
        String elemento = nombreIterator.next();
        System.out.print(elemento+" / ");
}

// Mostramos el numero de elementos que tiene el ArrayList tras las eliminaciones:
numElementos = nombreArrayList.size();
System.out.println("nNumero de elementos del ArrayList tras las eliminaciones = "+numElementos);
```

Como salida a este código tenemos lo siguiente:

```
El ArrayList tiene 11 elementos

... Eliminamos el primer elemento del ArrayList (Elemento 1)...

... Eliminamos los elementos de ArrayList que sean iguales a "Elemento 3" ...

Imprimimos los elementos del ArrayList tras realizar las eliminaciones:
Elemento 2 / Elemento 4 / Elemento 5 / Elemento 6 / Elemento 7 / Elemento 8 / Elemento 9 / Elemento 10 /

Numero de elementos del ArrayList tras las eliminaciones = 8
```