

# Programación Orientada a Objetos. Utilización avanzada de clases

Encarnación Sánchez Gallego

# ÍNDICE

1. **Introducción**
2. Clases y métodos abstractos
3. Interfaces.
4. Polimorfismo

# Introducción

- La POO es una herramienta para reducir la complejidad de los sistemas de software. Usando la POO un programa que ocupa miles y miles de líneas de código puede utilizarse como una colección de pequeñas unidades (objetos), cada una con cierta independencia y ciertas responsabilidades.
- Un programa OO consisten en un conjunto de objetos que intercambian **mensajes**, cada objeto decide si debe o no aceptar los mensajes que recibe, así como la interpretación de cada mensaje. En un programa OO mediante compilado, los objetos no son totalmente independientes: unos heredan propiedades y métodos de otros, algunos necesitan consultar a otros para desempeñar su tarea, otros llevan dentro de sí más objetos.
- **La principal ventaja** de la POO es que un programa de objetos se extiende de manera más sencilla que uno sin ellos, el programador puede construir sus propios objetos de uno ya existente y personalizarlo acorde a sus necesidades.

## 2. Clases abstractas "Molde parcialmente relleno"

En Java se dice que son clases abstractas aquellas clases base (superclases) de las que no se permite la creación de objetos. Para ello, se utiliza la palabra clave **abstract**.

### Métodos abstractos

En una clase abstracta es posible definir métodos abstractos, los cuales se caracterizan por el hecho de que no pueden ser implementados en la clase base. De ellos, solo se escribe su signatura en la superclase, y su funcionalidad –polimórfica– tiene que indicarse en las clases derivadas (subclases).

**EJEMPLO** Dadas la siguientes clases

(**Figura**, **Cuadrado** y **Triangulo**):

En la clase **Figura** se ha definido un atributo (**color**), un constructor y dos métodos (**calcularArea** y **getColor**).

```
public abstract class Figura
{
    private String color;

    public Figura(String color)
    {
        this.color = color;
    }

    public abstract double calcularArea();

    public String getColor()
    {
        return color;
    }
}
```

## 2. Clases abstractas

```
public class Cuadrado extends Figura
{
    private double lado;

    public Cuadrado(String color, double lado)
    {
        super(color);
        this.lado = lado;
    }

    public double calcularArea()
    {
        return lado * lado;
    }
}
```

En la clase **Cuadrado** se ha definido un atributo (**lado**), un constructor y un método (**calcularArea**).

```
public class Triangulo extends Figura
{
    private double base;
    private double altura;

    public Triangulo(String color, double base, double altura)
    {
        super(color);
        this.base = base;
        this.altura = altura;
    }

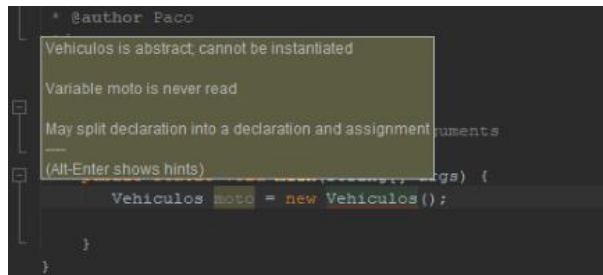
    public double calcularArea()
    {
        return (base * altura) / 2;
    }
}
```

En la clase **Triangulo** se han definido dos atributos (**base** y **altura**), un constructor y un método (**calcularArea**).

Como se puede observar, el método **calcularArea** ha sido definido abstracto (**abstract**) en la superclase abstracta **Figura**, indicándose solamente su signatura.

## 2. Clases abstractas "Molde parcialmente relleno"

- Son clases que no permiten la instanciación de objetos a partir de ellas.
- La idea es que otras clases hereden de ella, proporcionando la superclase abstracta un modelo genérico y algunos métodos de utilidad general.

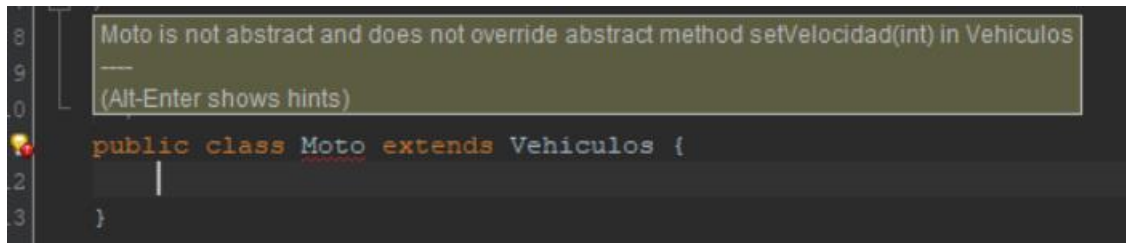


```
11 public class Moto extends Vehiculos {
12     private int cc;
13     private String marca;
14     private double velMax;
15
16     @Override
17     public String acelerar() {
18         if (cc > 125) {
19             return "Burruumbubbum";
20         } else {
21             return "birriipinin";
22         }
23     }
24 }
```

```
1 public abstract class Vehiculos {
2
3     private String medio;
4     private int velocidad;
5     private boolean encendido;
6
7     public abstract String acelerar(); //se implementa en las hijas
8
9     public void setEncendido(boolean encendido) { //se hereda así
10         this.encendido = encendido;
11     }
12 }
```

## 2. Clases abstractas

- A su vez podrán tener métodos abstractos, que tendrán que ser obligatoriamente programados en las clases hijas. Forzamos así a que se implemente el código, siendo distinto en cada clase hija (salvo que también sea abstracta) .



- Palabra reservada: abstract class.
- Los métodos no abstractos de una clase abstracta se heredarán en las hijas, los abstractos será obligatorio especificarlos y codificarlos en las hijas.
- Si una clase tiene un método abstracto, tiene que ser abstracta.
- Los métodos abstractos no pueden ser static.

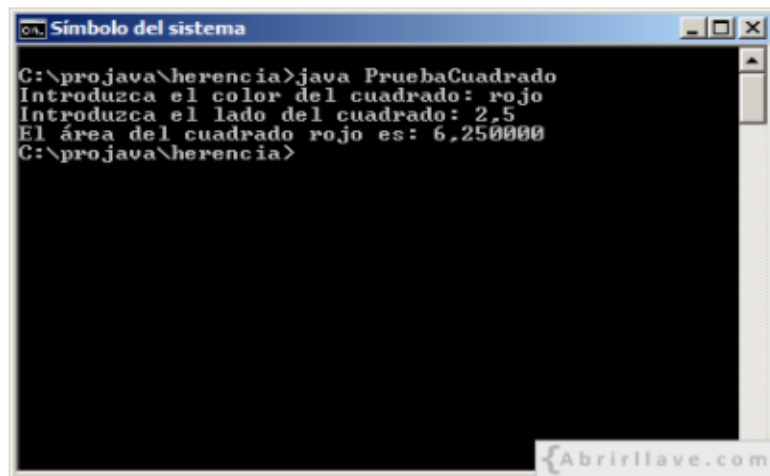
## 2. Clases abstractas

```
public abstract double calcularArea();
```

Por otro lado, véase que, en cada una de las subclases (**Cuadrado** y **Triangulo**) se ha implementado dicho método.

**EJEMPLO { PruebaCuadrado }** Al compilar y ejecutar el siguiente código:

Resultado:



```
C:\projava\herencia>java PruebaCuadrado
Introduzca el color del cuadrado: rojo
Introduzca el lado del cuadrado: 2.5
El área del cuadrado rojo es: 6.250000
C:\projava\herencia>
```

```
import java.util.Scanner;
```

```
public class PruebaCuadrado
{
```

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

```
        String colorDelCuadrado;
        double ladoDelCuadrado;
```

```
        Scanner teclado = new Scanner(System.in);
```

```
        System.out.print("Introduzca el color del cuadrado: ")
        colorDelCuadrado = teclado.nextLine();
```

```
        System.out.print("Introduzca el lado del cuadrado: ")
        ladoDelCuadrado = teclado.nextDouble();
```

```
        Triangulo triangulo1 = new Triangulo(colorDelTriangulo,
        baseDelTriangulo, alturaDelTriangulo);
```

```
        System.out.printf("El área del triángulo %s es: %f",
        triangulo1.getColor(), triangulo1.calcularArea());
```

```
    }
```

```
}
```



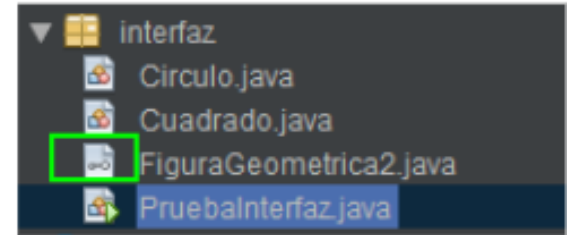
# 5. Interfaces

"Molde de moldes"

Una interfaz es una clase sin código, es decir tendrá atributos y métodos, pero no tendrá código dentro, va a servir para forzar a otras clases con su estructura, que si implementan su propio código en base a ese molde.

**Clase: molde para instanciar objetos.**

**Interfaz: molde para forzar clases.**



```
1 public interface FiguraGeometrica2{
12     //atributo: static final, será constante
13     public static final String COLOR = "blanco";
14     //métodos sin rellenar:
15     public double area();
16     public double perimetro();
17 }
```

```
11 public class Circulo implements FiguraGeometrica2{
12     private final double radio;
13     public Circulo (double r){
14         this.radio = r;
15     }
16     @Override
17     public double area(){
18         return (Math.PI * radio * radio);
19     }
20     @Override
21     public double perimetro(){
22         return (Math.PI * 2 * radio);
23     }
24 }
```

# 5. Interfaces

- Modificar de la interfaz: **interface**.
- En la clase implementada: **implements**.
- Como mínimo se sigue la estructura de la interfaz, pero no se pueden programar más elementos
- Especialmente importante el concepto de forzar: si se implementa una clase a partir de una interfaz, y no cumple la estructura, se obtiene un error.

```
public class Cuadrado implements FiguraGeometrica2{  
    private double lado;  
    public Cuadrado (double l){  
        this.lado = l;  
    }  
    public double area(){  
        return (lado * lado);  
    }  
    public double perimetro(){  
        return (4*lado);  
    }  
    public String getColor(){  
        return COLOR;  
    }  
}
```

Circulo is not abstract and does not override abstract method area() in FiguraGeometrica2

(Alt-Enter shows hints)

```
public class Circulo implements FiguraGeometrica2{  
    private double radio;  
    public Circulo (double r){  
        this.radio = r;  
    }  
    public double area(int num){  
        return (Math.PI * radio * radio);  
    }  
    public double perimetro(){  
        return (Math.PI * 2 * radio);  
    }  
}
```

Circulo is not abstract and does not override abstract method area() in FiguraGeometrica2

(Alt-Enter shows hints)

```
public class Circulo implements FiguraGeometrica2{  
    private double radio;  
    public Circulo (double r){  
        this.radio = r;  
    }  
    public double area(int num){  
        return (Math.PI * radio * radio);  
    }  
    public double perimetro(){  
        return (Math.PI * 2 * radio);  
    }  
}
```

# 5. Interfaces

- Si la interface va a tener atributos, éstos deben llevar las palabras reservadas static final y tener un valor inicial ya que funcionan como constantes. Métodos override.

```
11 public interface FiguraGeometrica2{
12     //atributo: static final, será constante
13     public static final String COLOR = "blanco";
14     //métodos sin rellenar:
15     public double area();
16     public double perimetro();
17 }
```

```
11 public class Circulo implements FiguraGeometrica2{
12     private final double radio;
13     public Circulo (double r){
14         this.radio = r;
15     }
16     @Override
17     public double area(){
18         return (Math.PI * radio * radio);
19     }
20     @Override
21     public double perimetro(){
22         return (Math.PI * 2 * radio);
23     }
24 }
```

¿Cómo se usan las interfaces en Java?

- Se declara la interfaz, que forzará a las clases implementadas:

```
public interface FiguraGeometrica2{  
    //atributo: static final, será constante  
    public static final String COLOR = "blanco";  
    //métodos sin rellenar:  
    public double area();  
    public double perimetro();  
}
```

## ¿Cómo se usan las interfaces en Java?

- Se programan las clases implementadas:

```
11 public class Circulo implements FiguraGeometrica2{
12     private final double radio;
13     public Circulo (double r){
14         this.radio = r;
15     }
16     @Override
17     public double area(){
18         return (Math.PI * radio * radio);
19     }
20     @Override
21     public double perimetro(){
22         return (Math.PI * 2 * radio);
23     }
24 }
```

```
11 public class Cuadrado implements FiguraGeometrica2{
12     private final double lado;
13     public Cuadrado (double l){
14         this.lado = l;
15     }
16     @Override
17     public double area(){
18         return (lado * lado);
19     }
20     @Override
21     public double perimetro(){
22         return (4*lado);
23     }
24     public String getColor(){
25         return COLOR;
26     }
27 }
```

¿Cómo se usan las interfaces en Java?

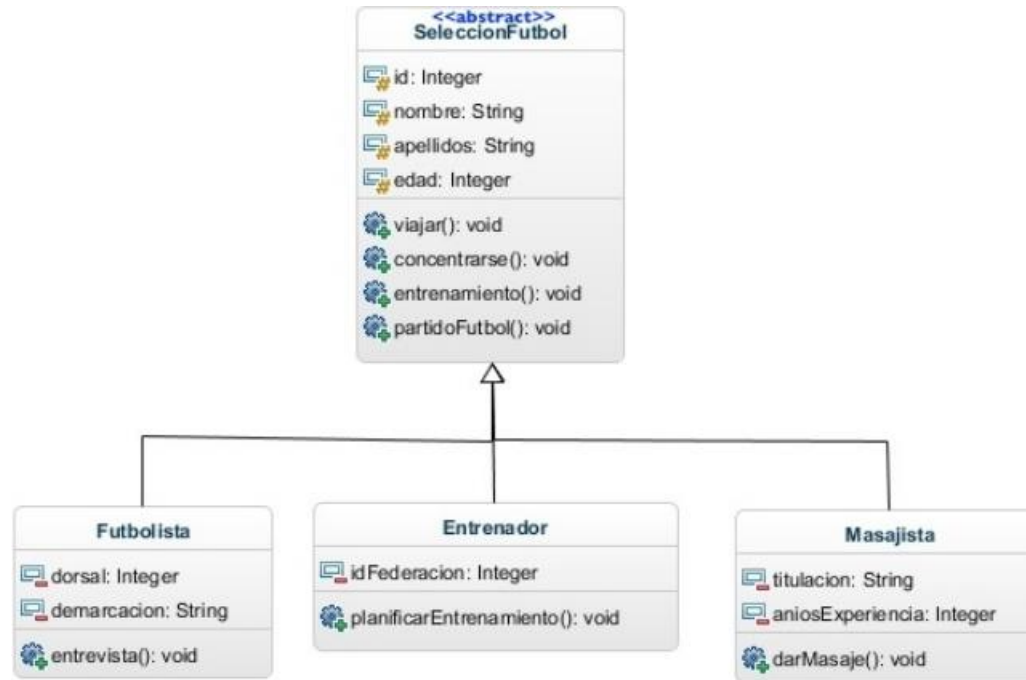
- Se instancian los objetos de las implementadas y se usan:

```
public class PruebaInterfaz {  
  
    public static void main(String[] args) {  
        Circulo circl = new Circulo(4);  
        System.out.println("El area es " + circl.area());  
    }  
}
```

<div data-bbox="104 102 156 157">≡</div> <div data-bbox="384 113 662 150">INTERFAZ</div>	<div data-bbox="1110 113 1630 150">CLASE ABSTRACTA</div> <div data-bbox="1765 102 1818 157">≡</div>
Se usa implementando, no herenciando	Se usa herenciando
Por lo anterior, permiten una especie de herencia múltiple: una clase puede implementar varias interfaces a la vez <div data-bbox="884 456 937 511">+</div>	Una clase sólo puede heredar de una única clase madre (no hay herencia múltiple)
No contienen código en sus métodos.	Sí contienen código en sus métodos, que puede heredarse o requerir la codificación explícita
Sólo define un conjunto de métodos comunes que se pueden implementar en muchas clases sin relación directa con la interfaz, no serán hijas	Servirá para generar clases hijas, que tendrán relación directa con la clase abstracta

## 4. Polimorfismo

- Vamos a partir de un ejemplo de un diagrama de clases:





## 4. Polimorfismo

- Vamos a tener una clase padre (SelecciónFutbol) en la que tendremos los atributos y métodos comunes a todos los integrantes que forman la selección española de fútbol (Futbolistas, Entrenadores, Masajistas, etc.) y en ella se van a implementar los métodos del comportamiento "genérico" que deben de tener todos los integrantes de la selección

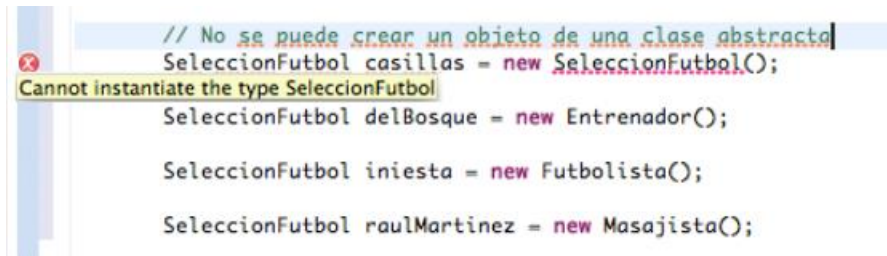
```
public abstract class SeleccionFutbol {  
  
    protected int id;  
    protected String nombre;  
    protected String apellidos;  
    protected int edad;  
  
    // constructores, getter y setter  
  
    public void viajar() {  
        System.out.println("Viajar (Clase Padre)");  
    }  
  
    public void concentrarse() {  
        System.out.println("Concentrarse (Clase Padre)");  
    }  
  
    // IMPORTANTE -> METODO ABSTRACTO => no se implementa en la clase abstracta pero si en la clases hijas  
    public abstract void entrenamiento();  
  
    public void partidoFutbol() {  
        System.out.println("Asiste al Partido de Fútbol (Clase Padre)");  
    }  
}
```

## 4. Polimorfismo

- La clase "SeleccionFutbol" es una clase abstracta y las clases abstractas no se pueden instanciar
- La palabra reservada abstract en un método (en el método entrenamiento). Esto quiere decir que todas las clases hijas de la clase "SeleccionFutbol" tienen que tener implementado ese método obligatoriamente.

```
public abstract class SeleccionFutbol {  
  
    protected int id;  
    protected String nombre;  
    protected String apellidos;  
    protected int edad;  
  
    // constructores, getter y setter  
  
    public void viajar() {  
        System.out.println("Viajar (Clase Padre)");  
    }  
  
    public void concentrarse() {  
        System.out.println("Concentrarse (Clase Padre)");  
    }  
  
    // IMPORTANTE -> METODO ABSTRACTO => no se implementa en la clase abstracta pero si en la clases hijas  
    public abstract void entrenamiento();  
  
    public void partidoFutbol() {  
        System.out.println("Asiste al Partido de Fútbol (Clase Padre)");  
    }  
}
```

- Por tanto con esto que se acaba de contar y diciendo que **la palabra "Polimorfismo" significa "muchas formas"**, podéis deducir que la clase "SeleccionFutbol" es una clase que puede adoptar diferentes formas y en este ejemplo puede adoptar las formas de "Futbolista", "Entrenador" y "Masajista".

A screenshot of a code editor showing a Java class named 'SeleccionFutbol'. The code contains four lines of instantiation: 'SeleccionFutbol casillas = new SeleccionFutbol();', 'SeleccionFutbol delBosque = new Entrenador();', 'SeleccionFutbol iniesta = new Futbolista();', and 'SeleccionFutbol raulMartinez = new Masajista();'. The first line is highlighted in blue and has a red 'X' icon to its left. A yellow tooltip points to this line with the text 'Cannot instantiate the type SeleccionFutbol'. Above the first line, a comment in Spanish reads '// No se puede crear un objeto de una clase abstracta'.

- Como vemos un "Entrenador", un "Futbolista" y un "Masajista" pertenecen a la misma clase padre y por eso se instancian diciendo que es una SeleccionFutbol y son nuevos objetos de las clases hijas. Por otro lado vemos que no se pueden crear objetos de una clase abstracta, por tanto el crearnos el objeto "casillas" nos da un error.

- Y ahora si hemos dicho que hemos definido en la clase padre un método abstracto que es obligatorio implementar en las clases hijas ¿Como lo hacemos?
- Las clases hijas se pueden especializar. Esto significa que una clase hija puede "redefinir" los métodos de su clase padre

```
public class Futbolista extends SeleccionFutbol {  
  
    private int dorsal;  
    private String demarcacion;  
  
    // constructor, getter y setter  
  
    @Override  
    public void entrenamiento() {  
        System.out.println("Realiza un entrenamiento (Clase Futbolista)");  
    }  
  
    @Override  
    public void partidoFutbol() {  
        System.out.println("Juega un Partido (Clase Futbolista)");  
    }  
  
    public void entrevista() {  
        System.out.println("Da una Entrevista");  
    }  
}
```

```
public class Entrenador extends SeleccionFutbol {  
  
    private int idFederacion;  
  
    // constructor, getter y setter  
  
    @Override  
    public void entrenamiento() {  
        System.out.println("Dirige un entrenamiento (Clase Entrenador)");  
    }  
  
    @Override  
    public void partidoFutbol() {  
        System.out.println("Dirige un Partido (Clase Entrenador)");  
    }  
  
    public void planificarEntrenamiento() {  
        System.out.println("Planificar un Entrenamiento");  
    }  
}
```

```
public class Masajista extends SeleccionFutbol {  
  
    private String titulacion;  
    private int aniosExperiencia;  
  
    // constructor, getter y setter  
  
    @Override  
    public void entrenamiento() {  
        System.out.println("Da asistencia en el entrenamiento (Clase Masajista)");  
    }  
  
    public void darMasaje() {  
        System.out.println("Da un Masaje");  
    }  
}
```

- Por último vamos a aplicar el concepto del polimorfismo a nuestra clase principal:

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

public static void main(String[] args) {

    SeleccionFutbol delBosque = new Entrenador(1, "Vicente", "Del Bosque", 60, 28489);
    SeleccionFutbol iniesta = new Futbolista(2, "Andres", "Iniesta", 29, 6, "Interior Derecho");
    SeleccionFutbol 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();
    }

    .....
}
```

- Ahora al ejecutar este fragmento de código vamos a ver que todos tienen el mismo comportamiento a la hora de "concentrarse()" y "viajar()"

```
Todos los integrantes comienzan una concentracion. (Todos ejecutan el mismo método)
Vicente Del Bosque -> Concentrarse (Clase Padre)
Andres Iniesta -> Concentrarse (Clase Padre)
Raúl Martínez -> Concentrarse (Clase Padre)

Todos los integrantes viajan para jugar un partido. (Todos ejecutan el mismo método)
Vicente Del Bosque -> Viajar (Clase Padre)
Andres Iniesta -> Viajar (Clase Padre)
Raúl Martínez -> Viajar (Clase Padre)
```



- ahora vamos a ver como cada uno de los integrante al lanzarse los mismos métodos ("entrenamiento()" y "partidoFutbol()") tienen un comportamiento diferente:

```
.....  
// ENTRENAMIENTO  
System.out.println("nEntrenamiento: Todos los integrantes tienen su función en un entrenamiento (Especialización)");  
for (SeleccionFutbol integrante : integrantes) {  
    System.out.print(integrante.getNombre() + " " + integrante.getApellidos() + " -> ");  
    integrante.entrenamiento();  
}  
  
// PARTIDO DE FUTBOL  
System.out.println("nPartido de Fútbol: Todos los integrantes tienen su función en un partido (Especialización)");  
for (SeleccionFutbol integrante : integrantes) {  
    System.out.print(integrante.getNombre() + " " + integrante.getApellidos() + " -> ");  
    integrante.partidoFutbol();  
}  
  
.....
```

```
Entrenamiento: Todos los integrantes tienen su función en un entrenamiento (Especialización)  
Vicente Del Bosque -> Dirige un entrenamiento (Clase Entrenador)  
Andres Iniesta -> Realiza un entrenamiento (Clase Futbolista)  
Raúl Martinez -> Da asistencia en el entrenamiento (Clase Masajista)
```

```
Partido de Fútbol: Todos los integrantes tienen su función en un partido (Especialización)  
Vicente Del Bosque -> Dirige un Partido (Clase Entrenador)  
Andres Iniesta -> Juega un Partido (Clase Futbolista)  
Raúl Martinez -> Asiste al Partido de Fútbol (Clase Padre)
```

- Por último vamos a ver que cada uno de los objetos puede ejecutar métodos propios que solamente ellos los tienen como son el caso de "planificarEntrenamiento(), entrevista() y darMasaje()" que solo los pueden ejecutar objetos de la clase Entrenador, Futbolista y Masajista respectivamente:

```
.....  
// PLANIFICAR ENTRENAMIENTO  
System.out.println("nPlanificar Entrenamiento: Solo el entrenador tiene el método para planificar un entrenamiento:");  
System.out.print(delBosque.getNombre() + " " + delBosque.getApellidos() + " -> ");  
((Entrenador) delBosque).planificarEntrenamiento();  
  
// ENTREVISTA  
System.out.println("nEntrevista: Solo el futbolista tiene el método para dar una entrevista:");  
System.out.print(iniesta.getNombre() + " " + iniesta.getApellidos() + " -> ");  
((Futbolista) iniesta).entrevista();  
  
// MASAJE  
System.out.println("nMasaje: Solo el masajista tiene el método para dar un masaje:");  
System.out.print(raulMartinez.getNombre() + " " + raulMartinez.getApellidos() + " -> ");  
((Masajista) raulMartinez).darMasaje();  
.....
```

```
Planificar Entrenamiento: Solo el entrenador tiene el método para planificar un entrenamiento:  
Vicente Del Bosque -> Planificar un Entrenamiento
```

```
Entrevista: Solo el futbolista tiene el método para dar una entrevista:  
Andres Iniesta -> Da una Entrevista
```

```
Masaje: Solo el masajista tiene el método para dar un masaje:  
Raúl Martínez -> Da un Masaje
```