Polimorfisme (I). Classes Abstractes.

El **Polimorfisme** és un dels 4 pilars bàsics de la programació orientada a objectes (POO) juntament amb **l'Abstracció**, **l'Encapsulació** i **l'Herència**. Per poder entendre què és el **Polimorfisme** es imprescindible tindre clar el concepte **d'herència**.

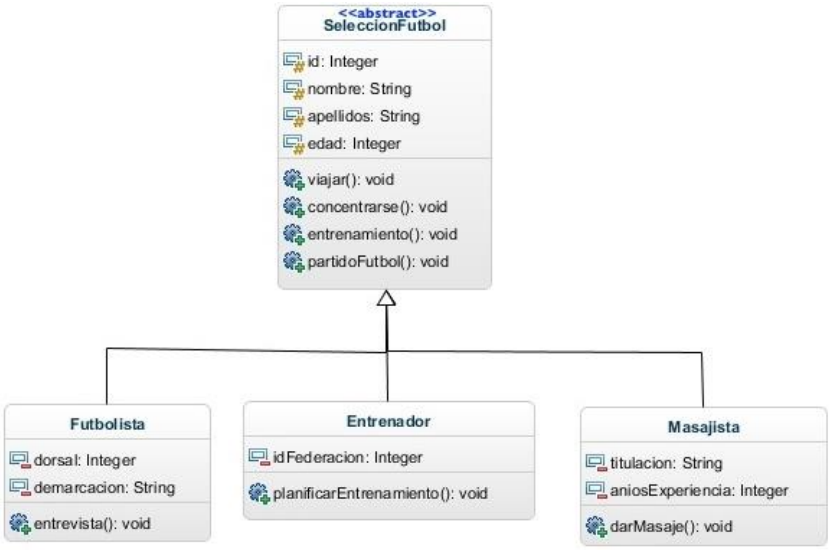
El terme polimorfisme és una paraula d'origen grec que significa "moltes formes".

En anteriors punts, ja hem vist alguna de les característiques del polimorfisme, com era la **sobrecàrrega de mètodes.**

En POO s'anomena **Polimorfisme** a la capacitat que tenen els objectes de distint tipus (de distintes classes) en respondre al mateix mètode.

Per poder entendre esta definició, veiem un exemple ja conegut del tema d'Herència: Integrants de la selecció de futbol.

Nota: En este exemple, no apareixen ni els constructors ni els mètodes **getter**'s i **sette**r's amb l'objectiu de fer simplificar l'exemple, encara que deurien aparèixer per tal de respectar el principi de **ENCAPSULAMENT** de la POO.



En aquest exemple tenim una **superclasse "Selección Futbol"** on s'implementa el comportament genèric de tots els integrants de la selecció **("Fubolista", "Entrenador" i "Masajista"),** es a dir els atributs i mètodes comuns a tots.

Com ja vam dir, l'herència no és més que traure "**factor comú**" del codi que escrivim, així que els atributs i mètodes de la classe **"SeleccióFutbol"** els tindran també els objectes de les classes (subclasses) **"Fubolista", "Entrenador" i "Masajista".**

Classe Abstracta:

Al definir una classe **abstract** estem indicant que és una classe que no podemm instanciar, es a dir, **NO** podrem fer:

=new SeleccionFutbol();

Mètode Abstracte:

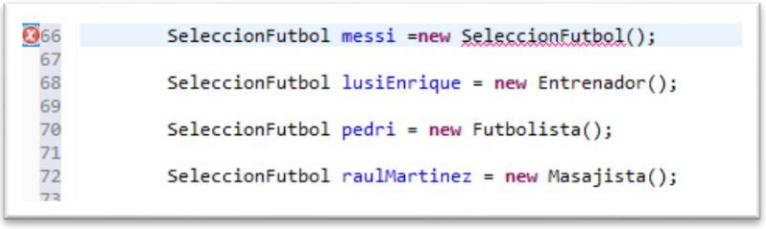
Al definir un mètode **abstract** estem indicant que **TOTES** les subclasses han de tindre implementades este mètode **OBLIGATÒRIAMENT.**
El mètode **abstract** **NO** s'implementa en la superclasse.

```
public abstract class SeleccionFutbol {

    protected int id;
    protected String nombre;
    protected String apellidos;
    protected int edad;

    // constructors, getter's y setter's

    public void viajar() {
        System.out.println("Viajar (Clase Padre)");
    }

    public void concentrarse() {
        System.out.println("Concentrarse (Clase Padre)");
    }

    // MÈTODE ABSTRACTE => no s'implementa en la classe abstracta però sí en les subclasses.

    public abstract void entrenamiento();

    public void partidoFutbol() {
        System.out.println("Asiste al Partido de Fútbol (Clase Padre)");
    }
}
```

Si instanciem objectes de les classes de l'exemple podem veure que:

Un **"Fubolista"** un **"Entrenador"** i un **"Masajista",** pertanyen a **la mateixa classe pare** i per això s'instancien dient que és una **"SeleccionFutbol"** i **són nous objectes de les classes filles.**

D'altra banda veiem que no es poden crear objectes d'una classe abstracta, per tant el crear-nos l'objecte **"messi"** ens dona un error.



Segons l'exemple anterior, i que polimorfisme vol dir moltes formes, podem afirmar que:

La classe **"SeleccionFutbol"** és una classe que **pot adoptar múltiples formes**, y en este exemple pot adoptar les formes de **"Fubolista", "Entrenador" i "Masajista".**

A l'exemple hem **definit** un mètode **abstracte** a la superclasse "**SelecionFutbol"** que **no hem implementat**, l'hem deixat totalment buit (ni tan sol hem posat les claus {}), este mètode l'hem **d'implementar obligatòriament a totes les subclasses**.

```
public abstract void entrenamiento();
```

Un dels avantatges que té l'herència i el polimorfisme, és que les classes filles no sols hereten els mètodes (o la implementació dels mètodes) de les classes pare, sinó que les classes filles es poden **ESPECIALTIZAR**. Això significa que una classe filla pot "**redefinir**" els mètodes de la seua classe pare; és a dir, que es pot tornar a escriure aqueix mètode i d'ahi l'especialització.

Subclasse Futbolista

```
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");
  }
}
```

Subclasse Entrenador

```
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");
  }
}
```

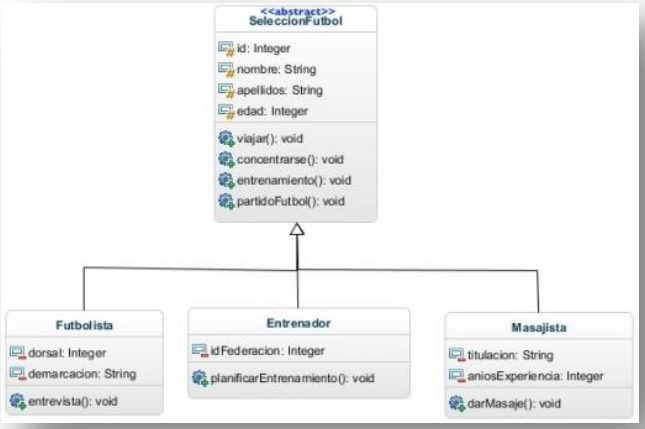
Subclasse Masajista

```
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");
  }
}
```

Observa que damunt del mètode "entrenamiento()" i d'altres mètodes, tenim l'etiqueta **"@Override"**.

Aquesta etiqueta serveix per a indicar en el codi que estem **"re-escrivint o especialitzant" un mètode que es troba en la classe pare i que volem redefinir en la classe filla**.

@Override: Esta etiqueta s'utilitza exclusivament en els mètodes de les classes filles que tenim definida en la classe pare, per tant quan es cride a estos mètodes, **les classes filles executaren el mètode redefinit** en la classe filla **i les que no l'hagen redefinit s'executarà és mètode de la classe pare**



En la següent imatge veiem com fem aquestes especialitzacions:

```
public abstract class SeleccionFutbol {
    ..........
    public abstract void entrenamiento();

    public void partidoFutbol() {
        System.out.println("Asiste al Partido de Fútbol (Clase Padre)");
    }
    ..........
}
```

```
public class Entrenador extends SeleccionFutbol {
    ..........
    @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 class Futbolista extends SeleccionFutbol {
    ..........
    @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 class Main {

// ArrayList d'objetos SeleccionFutbol. Independentment de la clase filla a la que pertany l'objecte

public static ArrayList<SeleccionFutbol> integrantes = new ArrayList<SeleccionFutbol>();

public static void main(String[] args) {

  SeleccionFutbol luisEnrique = new Entrenador(1, "Luis Enrique", "Martinez", 45, 284EZ89);
  SeleccionFutbol pedri = new Futbolista(2, "Pedro", "González", 18, 8, "Mediapunta");
  SeleccionFutbol raulMartinez = new Masajista(3, "Raúl", "Martinez", 41, "Licenciado en Fisioterapia", 18);

  integrantes.add(luisEnrique);
  integrantes.add(pedri);
  integrantes.add(raulMartinez);

 // CONCENTRACIÓ
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();
  }

// VIATJE
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();
}
    .........
}
```

Ens hem creat tres objectes de la classe "**SeleccionFutbol**" que adopten una de les 3 formes que poden adoptar **"Entrenador", "Futbolista" i "Masajista".**

Després els introduïm els tres objectes en un **"ArrayList"** d'objectes de la classe **"SeleccionFutbol"**

```
Todos los integrantes comienzan una concentracion. (Todos ejecutan el mismo método)
Luis Enrique Martinez -> Concentrarse (Clase Padre)
Pedro Gonzalez -> Concentrarse (Clase Padre)
Raúl Martinez -> Concentrarse (Clase Padre)

Todos los integrantes viajan para jugar un partido. (Todos ejecutan el mismo método)
Luis Enrique Martinez -> Viajar (Clase Padre)
Pedro Gonzalez -> Viajar (Clase Padre)
Raúl Martinez -> Viajar (Clase Padre)
```

Les tres subclasses hereten els mètodes **"concentrarse"** i **"viajar"** definits en la superclasse **"SeleccionFutbol".**

➔ **HERÈNCIA**

Fins ara, l'anterior execució no ens mostra res de nou, però en el següent fragment de codi, cadascun dels integrants de **l'ArrayList** de tipus "**SeleccionFutbol" tenen un comportament diferent al invocar als mateixos mètodes "entrenamiento()" i "partidoFutbol()"**

➔ **POLIMORFISME**

```
.....
// 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();
}
.....
```

Classe SeleccionFutbol (SUPERCLASSE)

```
public abstract void entrenamiento();

public void partidoFutbol() {
        System.out.println("Asiste al Partido de Fútbol (Clase Padre)");
```

Classe Entrenador

```
@Override
public void entrenamiento() {
   System.out.println("Dirige un entrenamiento (Clase Entrenador)");
   }
@Override
public void partidoFutbol() {
   System.out.println("Dirige un Partido (Clase Entrenador)");
   }
```

Classe Fubolista

```
@Override
public void entrenamiento() {
   System.out.println("Realiza un entrenamiento (Clase Futbolista)");
   }
@Override
public void partidoFutbol() {
   System.out.println("Juega un Partido (Clase Futbolista)");
```

Classe Masajista

```
@Override
public void entrenamiento() {
   System.out.println("Da asistencia en el entrenamiento (Clase Masajista)");
   }
```

Tots els integrants de la selecció executen el mètode **"entrenamiento()"** d'una manera diferent, ja que al ser un mètode **abstract** en la superclasse **obliguem a les subclasses a implementar este mètode.**

```
Entrenamiento: Todos los integrantes tienen su función en un entrenamiento (Especialización)
Luis Enrique Martinez -> Dirige un entrenamiento (Clase Entrenador)
Pedro Gonzalez -> 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)
Luis Enrique Martinez -> Dirige un Partido (Clase Entrenador)
Pedro Gonzalez -> Juega un Partido (Clase Futbolista)
Raúl Martinez -> Asiste al Partido de Fútbol (Clase Padre)
```

Quan executem el mètode **"partidoFutbol()"** l'objecte de la classe **"Masajista"** **utilitza el mètode implementat en la classe pare .**
En canvi els objectes de les classes **"Futbolista"** i **"Entrenador"** executen els **seus** mètodes **"re-implementats o especialitzats"** que es van tornar a escriure en les seues classes.

Finalment veurem que cadascun dels objectes pot executar mètodes propis que solament ells els tenen com són el cas de "**planificarEntrenamiento(), entrevista() i darMasaje()**" que **només els poden executar objectes** de la **classe Entrenador, Futbolista i Masajista** respectivament.