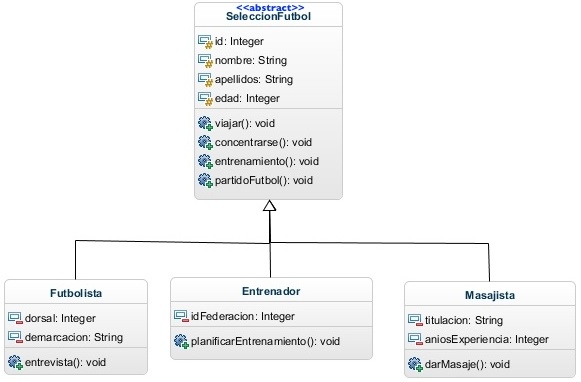
Polimorfismo en Java (Parte I), con ejemplos

El proyecto de este post lo puedes descargar pulsando [AQUI](https://github.com/jarroba/PolimorfismoJarroba).

Post original <https://jarroba.com/polimorfismo-en-java-parte-i-con-ejemplos/>

***El Polimorfismo*** es uno de los 4 pilares de la programación orientada a objetos ([POO](http://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos)) junto con la ***Abstracción***, ***Encapsulación*** y [***Herencia***](https://jarroba.com/herencia-en-la-programacion-orientada-a-objetos-ejemplo-en-java/)***.***Para entender que es el polimorfismo es muy importante que tengáis bastante claro el concepto de la Herencia, por tanto recomendamos que veáis la entrada en la que hablamos de la Herencia: [Herencia en Java, con ejemplos](https://jarroba.com/herencia-en-la-programacion-orientada-a-objetos-ejemplo-en-java/).

Para empezar con esta entrada, se ha de decir que el término ***"Polimorfismo"*** es una palabra de origen griego que significa***"muchas* formas"**. Este termino se utiliza en la POO para "***referirse a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos***". Como esta definición quizás sea algo difícil de entender, vamos a explicarla con el ejemplo que pusimos en la entrada de la [herencia](https://jarroba.com/herencia-en-la-programacion-orientada-a-objetos-ejemplo-en-java/) en la que queríamos simular el comportamiento que tendrían los diferentes integrantes de la selección española de fútbol; tanto los Futbolistas como el cuerpo técnico (Entrenadores, Masajistas, etc…). Para este ejemplo nos vamos a basar en el siguiente diagrama de clases:

[](https://jarroba.com/wp-content/uploads/2014/04/PolimorfismoFutbol-diag.jpg)

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

En este ejemplo 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.*** Como ya dijimos en la entrada de la herencia, la herencia no es más que sacar "factor común" del código que escribimos, así que los atributos y métodos de la clase SeleccionFutbol los tendrán también los objetos de las clases Futbolista, Entrenador y Masajista. Antes de seguir vamos a mostrar el código de la clase "SeleccionFutbol" para ver algunas peculiaridades:

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();

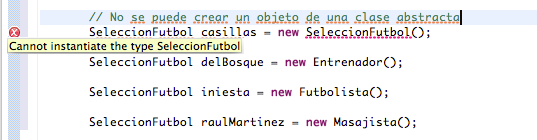
public void partidoFutbol() {

System.out.println("Asiste al Partido de Fútbol (Clase Padre)");

}

}

Lo primero que nos debe de llamar la atención al ver este código es que utilizamos dos veces la palabra reservada "***abstract***". Esta palabra nos indica que ***la clase "SeleccionFutbol" es una clase abstracta y las clases abstractas no se pueden instanciar***, por tanto nunca podremos hacer un "new SeleccionFutbol()". Otra cosa que vemos es que también utilizamos la***palabra reservada abstract en un método*** (en el método entrenamiento). Esto quiere decir que ***todas las clases hijas de la clase "SeleccionFubol" tienen que tener implementado ese método obligatoriamente***. 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"***.

[](https://jarroba.com/wp-content/uploads/2014/04/Ejm_polimorfismo_jarroba.png)

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?. Bueno vamos por partes. Una cosa muy buena que tiene la herencia y el polimorfismo, es que las clases hijas no solo heredan los métodos (o la implementación de los métodos) de las clases padre, sino que ***las clases hijas se pueden especializar***.  Esto significa que una clase hija puede "***redefinir***" los métodos de su clase padre; es decir, que se puede volver a escribir ese método y de ahi la especialización. Para ello vamos a ver la implementación de las clases hijas:

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)");

}

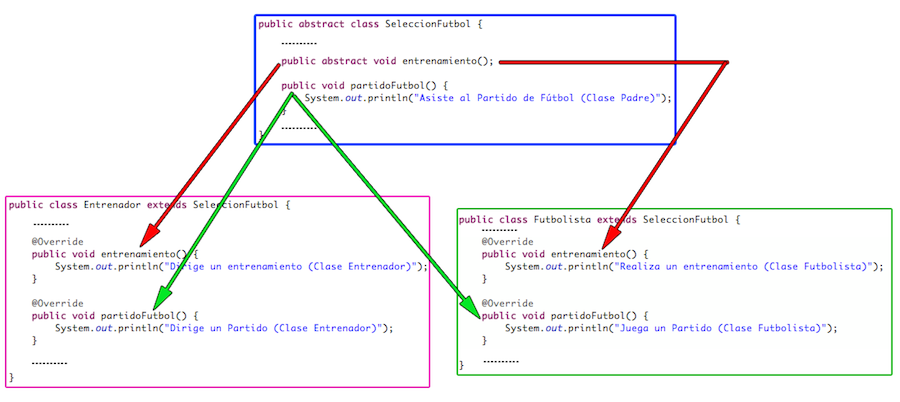
public void darMasaje() {

System.out.println("Da un Masaje");

}

}

Como vemos en el código todas las clases hijas tienen implementada el método "entrenamiento()" ya que como dijimos al tenerlo en la clase padre como método abstracto, es obligatorio que todas las clases hijas tengan ese método. Por otro lado observamos en el código que encima del método "entrenamiento()" y otros métodos, tenemos la etiqueta "***@Override***". Esta etiqueta sirve para indicar en el código que estamos ***"re-escribiendo o especializando" un método que se encuentra en la clase padre y que queremos redefinir en la clase hija***. Si os fijáis esta etiqueta solo y exclusivamente esta en los métodos de las clases hijas que tenemos definida en la clase padre, por tanto cuando se llame a esos métodos, las clases hijas ejecutaran el método redefinido en la clase hija y las que no lo hayan redefinido se ejecutará es método de la clase padre. En la siguiente imagen vemos como hacemos estas especializaciones:

[](https://jarroba.com/wp-content/uploads/2014/04/Polimorfismo_especializacion_jarroba.png)

Con todo esto ya podemos empezar a ejecutar el programa que simulará el comportamiento de los integrantes de la selección española y ver las diferentes formas que adoptan cada uno de los integrantes de la selección. 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) {

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();

}

.........

}

Como vemos nos hemos creado tres objetos de la clase SeleccionFutbol que adoptan una de las tres formas que pueden adaptar (Entrenador, Futbolista y Masajista)  y los metemos en un “[ArrayList](https://jarroba.com/arraylist-en-java-ejemplos/)” de objetos de la clase “SeleccionFutbol”. Ahora al ejecutar este fragmento de código vamos a ver que todos tienen el mismo comportamiento a la hora de "concentrarse()" y "viajar()", por tanto ejecutarán el método de la clase padre:

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

Vicente Del Bosque -> Concentrarse (Clase Padre)

Andres Iniesta -> Concentrarse (Clase Padre)

Raúl Martinez -> 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 Martinez -> Viajar (Clase Padre)

Hasta el momento nada nuevo y sorprendente, pero 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();

}

........

Vemos el resultado al ejecutar este fragmento de código:

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)

En este caso vemos que todos los integrantes ejecutan el método "entrenamiento()" de forma diferente ya que al ser este método abstracto en la clase padre, les forzamos a las clases hijas a que implementen ese método. Por el contrario al ejecutar el método "partidoFutbol()" vemos que el objeto de la clase Masajista utiliza el método implementado en la clase padre y en cambio los objetos de la clase Futbolista y Entrenador ejecutan sus método "re-implementados o especializados" que se volvieron a escribir en sus clases.

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();

........

Como resultado de la ejecución de este fragmento de código tenemos lo siguiente:

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 Martinez -> Da un Masaje

CONCLUSIONES Y ACLARACIONES:

Como hemos visto el polimorfismo es un concepto un poco más avanzado que la herencia y puede ser muy util a la hora de jerarquizar y querer dar un patrón de comportamiento común a una serie de objetos que heredan de la misma clase. En esta entrada no hemos visto todo lo referente al polimorfismo ya que nos quedaría ver un concepto un poco más avanzado en Java (y en otros lenguajes también) como son las "[Interface](https://jarroba.com/polimorfismo-en-java-interface-parte-ii-con-ejemplos/)" (clases abstractas puras) de las cuales hablaremos en otra entrada para terminar de ver lo que es el polimorfismo.

Por último es muy probable para los que estéis empezando con la POO que no veáis mucho sentido a esto del polimorfismo y al principio es normal. Solo os debo de decir que a base de experiencia se le encuentra sentido al polimorfismo, por tanto si teneis que hacer alguna práctica en la universidad o lo que sea en la que tengais que usar el polimorfismo intentar entenderlo y hacer lo que os pidan porque entenderlo 100% es complicado y se requiere de experiencia para ello.