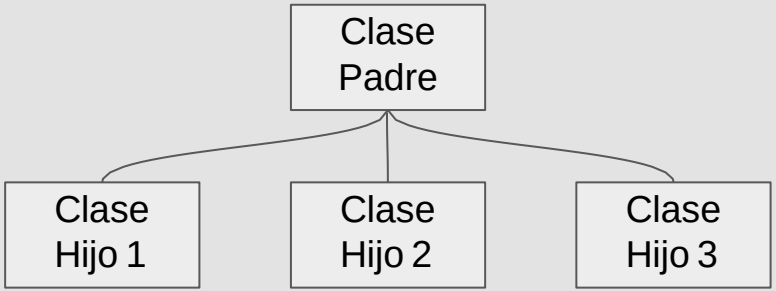
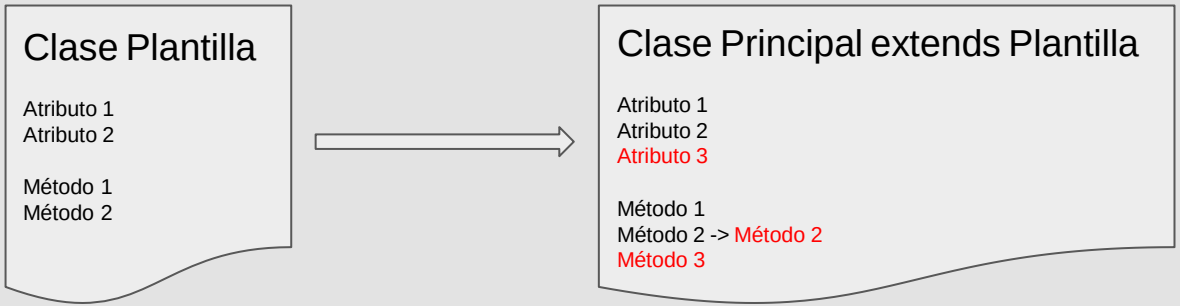
Utilización Avanzada de Clases

Índice

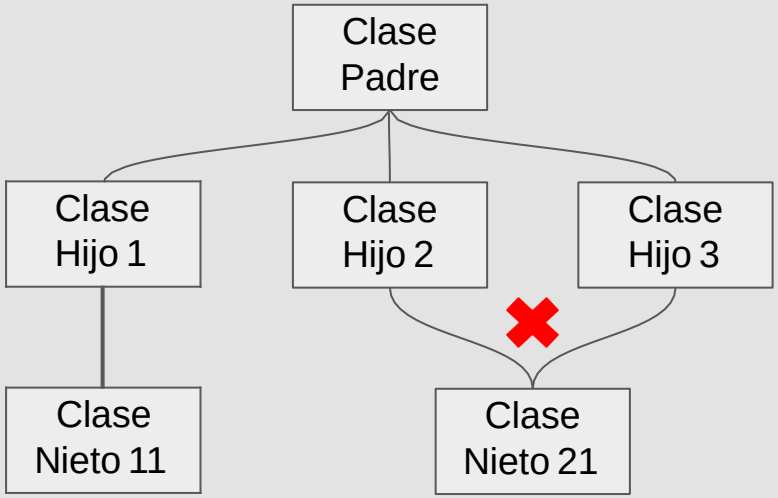
1. Herencia. ¿Qué es?.



La herencia es una característica de la Programación Orientada a Objetos que permite la reutilización del código.

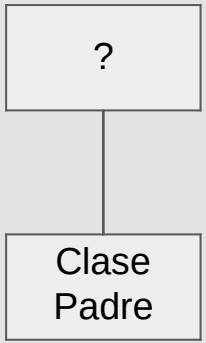


1. Herencia. Árboles.



Una clase puede heredar de varios niveles de clases, pero sin embargo en Java una clase no puede heredar de dos. En C++ y otros lenguajes, si está implementada esta funcionalidad.

1. Herencia. Por defecto.



Si definimos una clase sin herencia. ¿Cuál es la clase de la que hereda por defecto?

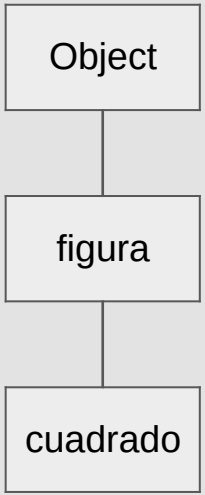
1. Herencia. Por defecto.

Si definimos una clase sin herencia. ¿Cuál es la clase de la que hereda por defecto?

La clase Object

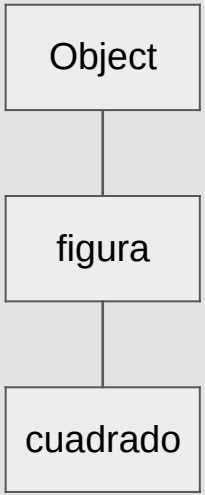
1. Herencia. Implementación.

```
public class figura{

	String color;

	public void setColor(String s){
		color = s;
	}

	public void getColor(){
		return color;
	}
}

public class cuadrado extends figura{

	private int lado;

	cuadrado(int l){
		this.lado = 1;
	}

	public int getArea(){
		return lado*lado;
	}

}
```



1. Herencia. Implementación.

```
public class figura{

	String color;

	public void setColor(String s){
		color = s;
	}

	public void getColor(){
		return color;
	}
}

public class cuadrado extends figura{

	private int lado;

	cuadrado(int l){
		this.lado = 1;
	}

	public int getArea(){
		return lado*lado;
	}

}
```



¿Podría la clase figura hacer uso de la función getArea()?

¿Qué ocurriría si declaramos el método getColor() de "figura" como privado?

1. Herencia. Ejercicio.

Se deben implementar 3 clases que estructuren nuestro ejemplo:

Clase **Personaje**, va a ser el padre de nuestras clases y tendrá como atributo:

int vida

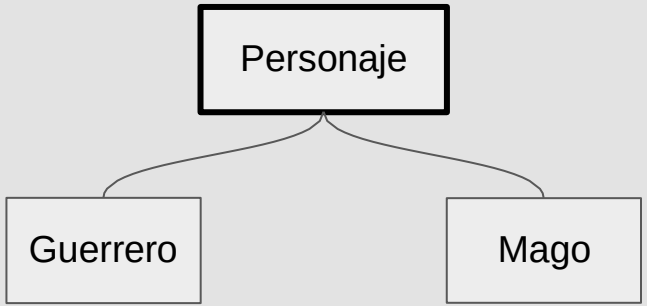
Y como métodos:

Constructor de Personaje que inicialice nuestro atributo:

- vida=100

Método getVida() que nos devuelva el valor de vida del personaje.

Método setVida() que establece el nuevo valor de vida del personaje.



1. Herencia. Ejercicio.

Clase **Guerrero**, que hereda de la clase Personaje y tendrá como atributos:

int **ataque**

Y como métodos:

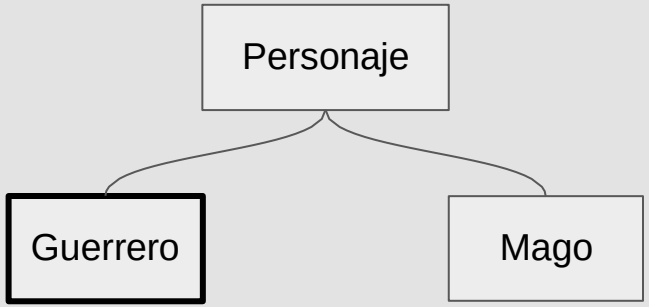
Constructor de Guerrero que inicialice nuestro atributo:

- ataque=10

Método **movimientoLucha()** que nos devuelva el valor de ataque y muestra por consola la sentencia "******Espadazo******".

Extra: añadir aleatoriedad a nuestra lucha con la librería math random

*int numero = (int) (Math.random() * ataque);*



1. Herencia. Ejercicio.

Clase **Mago**, que hereda de la clase Personaje y tendrá como atributos:

int **magia**

Y como métodos:

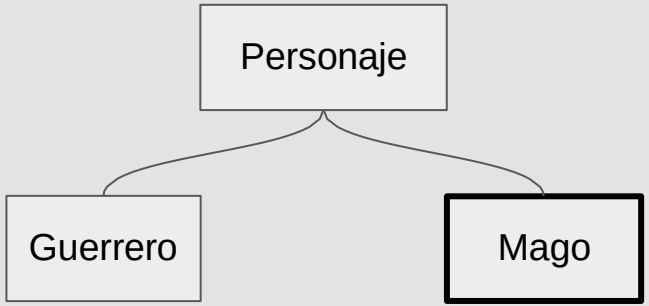
Constructor de Mago que inicialice nuestro atributo:

- magia=10

Método **movimientoLucha()** que nos devuelva el valor de la magia y muestra por consola la sentencia "******Hechizo******".

Extra: añadir aleatoriedad a nuestra lucha con la librería math random,

*int numero = (int) (Math.random() * magia);*

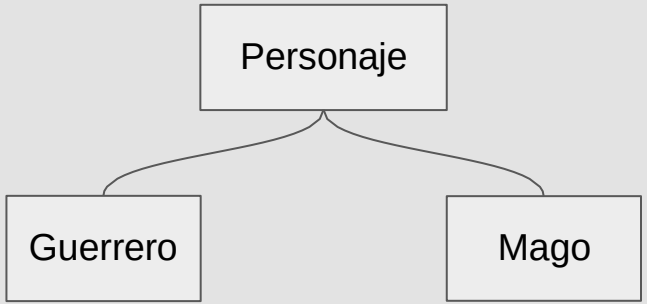


1. Herencia. Ejercicio.

Finalmente, construir una clase llamada “Arena” que contendrá el método Main, que va a ejecutar la siguiente secuencia:

- Crear un Guerrero .
- Mostrar su nivel de vida.
- Ejecutar un movimiento de lucha.

- Crear un Mago
- Mostrar su nivel de vida.
- Ejecutar un movimiento de lucha.



2. Paso por valor y por referencia.

Cuando pasamos un parámetro a un método, éste puede tener un comportamiento distinto según el tipo de dato:

- **Por valor:** el valor del parámetro se copia a la variable del método, por lo tanto por mucho que modifiquemos su valor, la variable original queda intacta. Para pasar por valor:
  - Pasar un tipo primitivo, o pasar otros objetos básicos de Java como un String.

```
public void modificarVariable( int miVariable ){
      miVariable = 5;
}
```

- **Por referencia:** lo que se pasa como argumento, no es el valor de la variable, sino la posición de la memoria (de forma transparente para nosotros), por lo que si modificamos su valor cambiamos el original. Para pasar por referencia, tenemos dos opciones:
  - Pasar un Array.
  - Pasar un Objeto.

```
public void modificarVariable( int[] miArray ){
      miArray[0] = 5
}
```

2. Paso por valor y por referencia. Ejercicio.

Añadir a nuestra clase "**Arena**" un método, llamado "**Lucha**" donde:

- Le pasaremos por parámetro el **objeto Mago** y el **objeto Guerrero**.

  ¿Cómo estamos pasando el parámetro, por referencia o por valor?

- Dentro del método el **Guerrero** ejecutará **movimientoLucha** y el valor que nos devuelva la función, se le **restará a la vida del Mago**.

- Dentro del método el **Mago** ejecutará **movimientoLucha** y el valor que nos devuelva la función, se le **restará a la vida del Guerrero**.

Desde nuestro Main, llamaremos a la función Lucha y mostraremos por consola la vida que le resta a nuestro Guerrero y nuestro Mago.

3. Wrappers. ¿Qué son?.

Un wrapper es un objeto con las mismas propiedades que un tipo primitivo, pero con las características propias de un objeto.

- Facilidad de conversión entre tipos primitivos y cadenas.
- Al ser objetos, si son pasados a un método como argumentos.

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

3. Wrappers. Constructores y métodos.

El wrapper Integer tiene dos constructores:

- Integer(int): le pasamos como parámetro inicializador el tipo primitivo que le corresponde.
- Integer(String): le pasamos como parámetro una cadena, que convertirá al tipo Integer.
    - Muy útil si la fuente de datos es externa, por ejemplo, si el información leída proviene de un fichero, input del usuario por consola, etc. ya que nos convierte el dato a entero para su posterior tratamiento.

Métodos del wrapper Integer:

- Funciones de conversión con datos de tipo primitivo:
    - byteValue()
    - shortValue()
    - intValue()
    - longValue()
    - doubleValue()
    - floatValue()
- Funciones de conversión a tipo Integer:
    - Integer decode(String)
    - Integer parseInt(String)
    - Integer parseInt(String, int)
    - Integer valueOf(String)
    - String toString()

3. Wrappers. Ejercicio.

En nuestra clase arena.

Se deben crear dos **propiedades que serán inicializadas a 0 al comienzo del Main**:

- **danoValor**: será un entero de tipo primitivo.
- **danoReferencia**: será un entero de tipo wrapper **(¿Pasará por referencia realmente?)**

Donde almacenarán ambas el daño acumulado en el combate sumando los puntos de vida restados al mago y al guerrero.

- Si al guerrero le restan 7 y al mago le restan 9 de vida, el daño acumulado será 16.

A nuestro método Lucha se le deben pasar por parámetro estas dos propiedades. Cuando termine el combate (método lucha), en el Main deben tener guardado el mismo valor y escribir por pantalla lo siguiente:

**** El daño acumulado por referencia es 0 ****
**** El daño acumulado por valor es 0 ****
El motivo es que ambos están pasando por valor, y en el main no podemos acceder a los valores que se calcularon en el método Lucha.

4. Tratamiento de Fechas y Horas.

La clase Date es una utilidad que permite representar un instante concreto en milisegundos. Internamente se corresponde a una variable de tipo **Long** que registra los milisegundos transcurridos desde el 1 de enero de 1970.

Para representar este dato en un formato legible, se puede utilizar la clase **GregorianCalendar** que convierte Date en una representación donde las horas son un entero de 0 a 23, los días entre 1 y 31 y los años son cuatro dígitos. Hay que tener cuidado ya que los meses son representados por un entero que va de 0 a 11, por lo que hay que sumar 1 al valor que nos devuelve el calendario.

```
Date d = new Date();
GregorianCalendar c = new GregorianCalendar();
c.setTime(d);
System.out.print(c.get(Calendar.DAY_OF_MONTH));
//Revisar API Calendar para ver todas las
opciones
```

4. Tratamiento de Fechas y Horas. Ejercicio.

Añadir todos constructores de nuestras clases de ejemplo:

- Que saque por pantalla el mensaje relativo a la clase que se está construyendo.
- Que escriba la fecha y hora actual de creación.

**** Construyendo un Guerrero a las 19:46 ****

5. Clases y métodos abstractos y finales.

Clase abstracta

- Se define una clase como las pensadas para ser genéricas.
- No tiene sentido definir objetos de clases abstractas.

Método abstracto

- Cuando defines un método abstracto, desconoces la implementación, pero indica a la subclase que hereda de la clase abstracta que ese método tiene que ser implementado.
- Un método abstracto no puede ser estático.
- Si una clase tiene al menos un método abstracto, ésta debe ser declarada también como abstracta

```
public abstract class personaje{

	int vida;

	public void setVida(int v){
		vida=v;
	}

	public int getVida(){
		return vida;
	}

	public abstract int movimientoLucha();

}
```

5. Clases y métodos abstractos y finales. Ejercicio.

Nuestra clase personaje pasa a ser abstracta y crearemos un método abstracto llamado:

- movimientoLucha()

Éste obliga a implementar el método momiviento Lucha en cualquier subclase que extienda al personaje.

```
public abstract class personaje{

	int vida;

	public void setVida(int v){
		vida=v;
	}

	public int getVida(){
		return vida;
	}

	public abstract int movimientoLucha();

}
```

5. Clases y métodos abstractos y finales.

Objetos finales

- Impedirá que haya otro objeto con la misma referencia, no puede ser reasignado.

```
final cuadrado c1 = new cuadrado(5);
cuadrado c2 = new cuadrado(15);
c1=c2
```

Métodos finales

- Declaramos ante el compilador que ese método no va a cambiar, es decir, que no va a ser sobreescrito en una subclase.
- Ganancia de eficiencia.

Clase final

- Cuando una clase se declara como final, indicamos que ésta no va a tener descendencia (no puede tener subclases).

5. Clases y métodos abstractos y finales. Ejercicio.

Convertir nuestros objetos Guerrero y Mago en final.

¿Cual es el objetivo de hacer esto?

5. Clases y métodos abstractos y finales. Ejercicio.

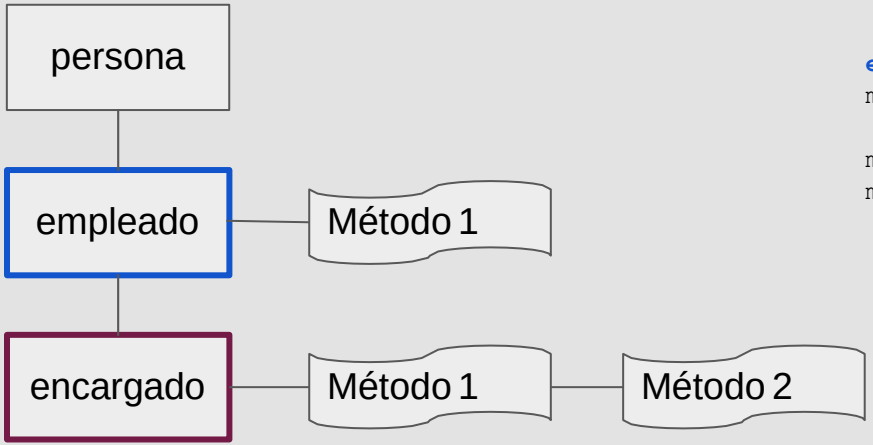
Convertir nuestros objetos Guerrero y Mago en final.

¿Cual es el objetivo de hacer esto?

- Evitar que los objetos sean sobreescritos y perdamos la información almacenada.

6. Polimorfismo.

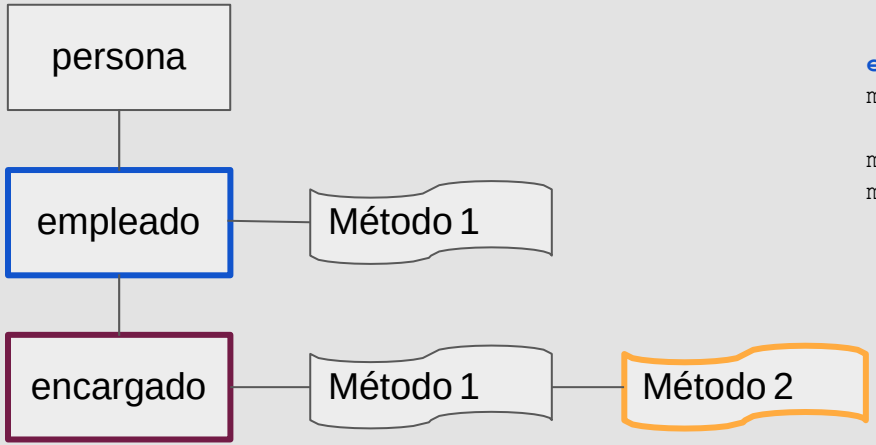
El polimorfismo se define como la característica que tienen los objetos de una clase base referenciada para apuntar a objetos de una clase derivada o subclase.




```
empleado miEmpleado;           //referencia a empleado
miEmpleado = new encargado();  //apunta a encargado

miEmpleado.metodo2();
miEmpleado.metodo1();
```

6. Polimorfismo.

El polimorfismo se define como la característica que tienen los objetos de una clase base referenciada para apuntar a objetos de una clase derivada o subclase.

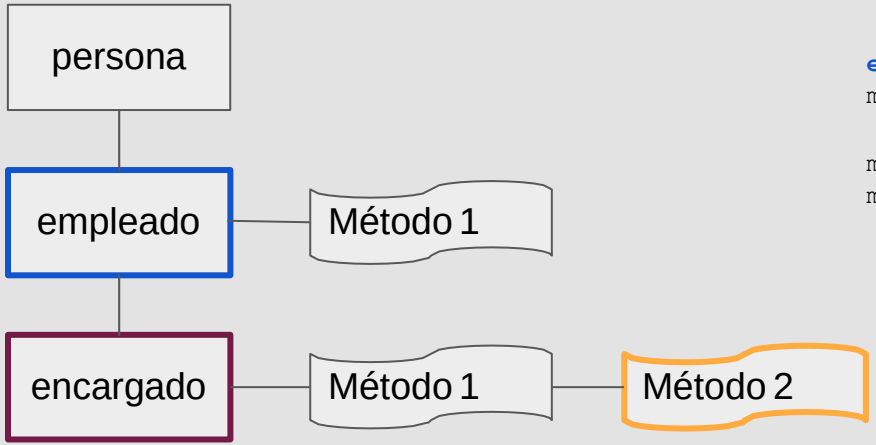



```
empleado miEmpleado;
miEmpleado = new encargado();

miEmpleado.metodo2();
miEmpleado.metodo1();
```

¿Que va a pasar con el método2?

6. Polimorfismo.

El polimorfismo se define como la característica que tienen los objetos de una clase base referenciada para apuntar a objetos de una clase derivada o subclase.



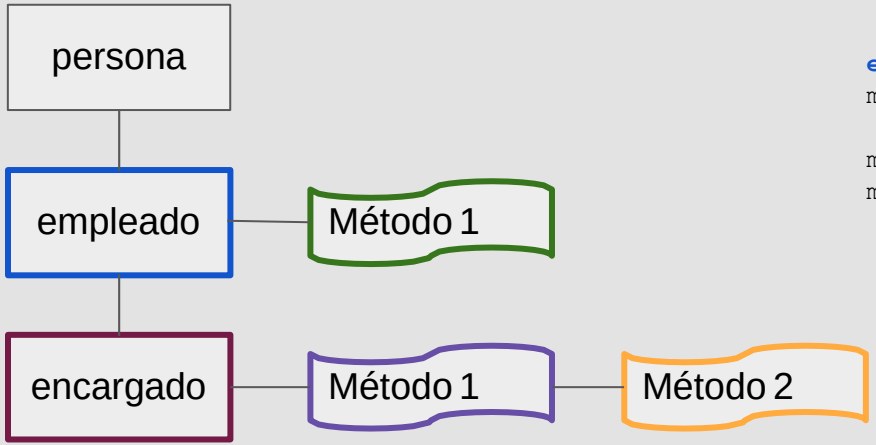

```
empleado miEmpleado;
miEmpleado = new encargado();

miEmpleado.metodo2(); ✖ //error de compilación
miEmpleado.metodo1();
```

¿Que va a pasar con el método2?

Nuestro empleado hace llamadas a los métodos de la clase que la referencia, por lo tanto, método2 no existe ya que se define sólo para encargado.

6. Polimorfismo.

El polimorfismo se define como la característica que tienen los objetos de una clase base referenciada para apuntar a objetos de una clase derivada o subclase.

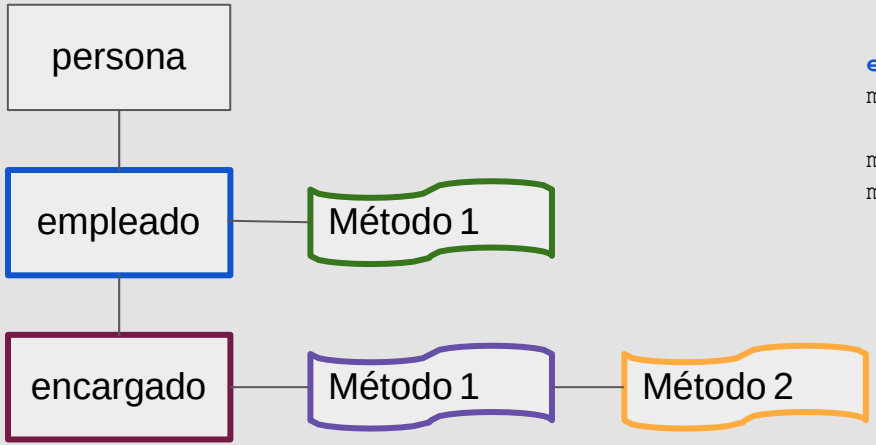



```
empleado miEmpleado;
miEmpleado = new encargado();

miEmpleado.metodo2();
miEmpleado.metodo1();
```

¿Que va a pasar con el método1?

6. Polimorfismo.

El polimorfismo se define como la característica que tienen los objetos de una clase base referenciada para apuntar a objetos de una clase derivada o subclase.




```
empleado miEmpleado;
miEmpleado = new encargado();

miEmpleado.metodo2();
miEmpleado.metodo1();
```

¿Que va a pasar con el método1?

Utiliza el método de la clase a la que apunta no a la que referencia, ya que al estar **sobrescrito** no existe error de compilación y en tiempo de ejecución prima la clase encargado a la que apunta.

7. Sobreescritura de métodos.

Para sobreescribir un método debe:

- Tener el mismo nombre.
- El retorno debe ser el mismo en el padre que en el hijo.
- La lista de argumentos debe ser la misma y con los mismos tipos.

```
public class clasePadre{

      public int miMetodo(int valor){

            valor = valor + 10;

            return valor;
      }
}
```

```
public class claseHijo extends clasePadre{

      public int miMetodo(int valor){

            valor = valor + 20;

            return valor;
      }
}
```

7. Sobreescritura de métodos. Ejercicio

Crear una clase que se llame GuerreroFuerte que herede que la clase Guerrero.

Ésta debe sobreescribir el método movimientoLucha sumando 5 al resultado de math.Random.

8. Sobrecarga de métodos.

La sobrecarga de métodos es igual que la sobrecarga de constructores, puedes tener más de un método con el mismo nombre, con la diferencia que la lista de argumentos cambia.

Puede ser un método heredado (ejemplo siguiente), o de la misma clase.

```
public class clasePadre{

      public int miMetodo(int valor){

            valor = valor + 10;

            return valor;
      }
}
```

```
public class claseHijo extends clasePadre{

      public int miMetodo(int valor, int suma){

            valor = valor + suma;

            return valor;
      }
}
```

8. Sobrecarga de métodos. Ejercicios.

Se debe sobrecargar el método **lucha** de nuestra clase **Arena**. En el main crearemos un GuerreroFuerte, y llamaremos al nuevo método Lucha.

Lucharán:

- **Mago vs Guerrero**.
- **Mago vs GuerreroFuerte**.

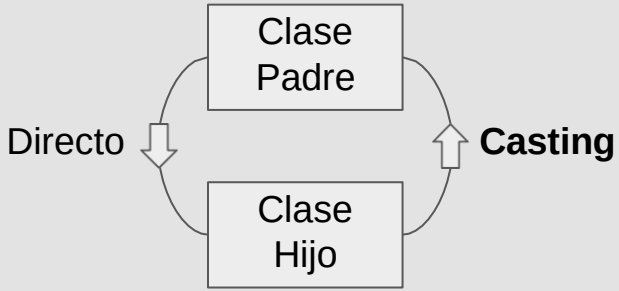
9. Asignaciones entre objetos distintos

1. Cuando asignamos un objeto a otro de distinta clase, ocurre lo mismo que al asignar objetos de la misma clase, que el objeto asignado pasa a ser el objeto a asignar, pero con distinto nombre.
2. Para que se asigne un objeto de una clase a otro de otra clase, debe haber una relación de herencia entre ellos.
3. Al asignar un objeto de una clase padre a un objeto de una clase hijo, se haría de manera automática, ya que el hijo contiene toda la información que posee el padre:

```
objetoHijo = objetoPadre;
```

al revés no sucede lo mismo porque el objeto Padre no contiene todo lo que el objeto hijo, y debemos hacer casting:

```
objetoPadre = (ClasePadre) objetoHijo;
```



9. Asignaciones entre objetos Ejercicio.

En nuestra clase Arena tenemos definida primero una Lucha (primera ronda) entre el Mago y el GuerreroFuerte. Cuando esta termine, debemos crear otro Guerrero normal a partir de nuestro GuerreroFuerte, y éste debe luchar contra el mago (segunda ronda).

- Observamos que al crear el nuevo Guerrero y asignarlo al GuerreroFuerte, nuestro nuevo Guerrero pasa a ser el GuerreroFuerte “disfrazado” de Guerrero, por lo que al luchar contra el Mago, no tendrá un ataque normal de Guerrero, sino que tendrá el ataque de GuerreroFuerte. ¿El motivo? En realidad está luchando con GuerreroFuerte.

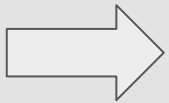
```
Guerrero gnormal = (Guerrero) gFuerte;
```

10. Acceso a métodos de la superclase.

El método super() nos sirve para acceder a los métodos de la clase que hereda la subclase.

Su utilidad principal reside en que si queremos sobreescribir un método, pero no queremos mantener de alguna forma la funcionalidad implementada en el padre.

```
public class clasePadre {

	int i = 0;

	public void suma(int j){
		i = i + j;
	}

}
```



```
public class claseHijo extends clasePadre{

	public void suma(int j){
		j = j + 10;
		super.suma(j);
	}

}
```

10. Acceso a métodos de la superclase. Ejercicio.

- Modificar el constructor del Mago para que su vida sea un 10% superior a la de cualquier personaje, utilizando la función super().
    - Pista: tendremos que hacer uso de las funciones de Personaje getVida y setVida.

11. Clases anidadas.

Una clase anidada, es una clase dentro de otra clase.

- **Implementación**:

```
public class claseExterna{
      public class claseAnidada{
      }
}
```

- **Objetivo**: este tipo de anidación se utiliza para que el mantenimiento del código y su legibilidad sea más sencilla.

****Importante****: Sólo se debe utilizar si el uso de la **clase anidada** es **exclusivo** para la **clase externa**. Antes de anidar una clase, se debe estar seguro que es la mejor opción.