



Universidad
Andrés Bello®
Conectar • Innovar • Liderar

FUNDAMENTOS DE PROGRAMACIÓN EN JAVA

HERENCIA Y POLIMORFISMO

Herencia de clases

Cuando se habla de herencia en la Programación Orientada a Objetos, se hace referencia a que una subclase deriva de una clase superior adoptando todos los atributos y métodos de la súper clase. Es una propiedad que permite que los objetos sean creados a partir de otros ya existentes, obteniendo características (métodos y atributos) similares a los ya existentes.

Por ejemplo: Si se declara una clase Auto derivada de una clase Vehículo todos los métodos y variables asociadas con la clase Auto son automáticamente heredados por la subclase Vehículo. En definitiva, la herencia permite crear nuevas clases que reutilizan, extienden y modifican el comportamiento que se define en otras clases. La clase cuyos miembros se heredan se denomina clase base (o padre) y la clase que hereda esos miembros se denomina clase derivada (o hija).

Existen dos tipos de herencia según la cantidad de clases bases que tenga una clase derivada:

- Herencia Simple: Indica que se pueden definir nuevas clases solamente a partir de una clase inicial
- Herencia Múltiple: Indica que se pueden definir nuevas clases a partir de dos o más clases iniciales.

Un ejemplo de herencia en código Java se vería de la siguiente forma:



```
public class Persona {
    private String nombre;
    private String apellido;
    private int edad;
    public Persona (String nombre, String apellido, int edad) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }

    public String getNombre () {
        return nombre;
    }

    public String getApellido () {
        return apellido;
    }

    public int getEdad () {
        return edad;
    }
}
```

La clase hija Profesor se escribiría de la siguiente manera en código Java:

```
public class Profesor extends Persona {
    private String IdProfesor;

    public Profesor (String nombre, String apellidos, int edad) {
        super(nombre, apellidos, edad);
    }

    public void setIdProfesor (String IdProfesor) {
        this.IdProfesor = IdProfesor;
    }

    public String getIdProfesor () {
        return IdProfesor;
    }

    public void mostrarNombreApellido() {
        System.out.println ("Nombre profesor: " + getNombre() + " " + getApellido() );
    }
}
```





Universidad
Andrés Bello®

Es muy importante destacar el uso de la palabra `super`, esta toma el rol de una llamada al constructor de la clase padre, por lo que en el ejemplo estaría Ejecutando el constructor con 3 parámetros de entrada como se aprecia en la clase `Persona`.

Interfaces

Las interfaces son una forma de especificar qué debe hacer una clase sin especificar el cómo. Las interfaces no son clases, sólo especifican requerimientos para la clase que las implementa.

Se pueden definir métodos que usen como parámetro objetos que implementen la interfaz, basta usar el nombre de la interfaz como el tipo del parámetro. Luego, las instancias de una clase que implemente la Interfaz, pueden tomar el lugar del argumento donde se espere alguien que implemente la interfaz. La declaración de una interfaz es similar a una clase, aunque emplea la palabra reservada `interface` en lugar de `class` y no incluye ni la declaración de variables de instancia ni la implementación del cuerpo de los métodos (sólo las cabeceras). La sintaxis de declaración de una interfaz es la siguiente:

```
public interface nombreInterfaz {
}
```

Una interfaz declarada como `public` debe ser definida en un archivo con el mismo nombre de la interfaz y con extensión `.java`. Las cabeceras de los métodos declarados en el cuerpo de la interfaz se separan entre sí por caracteres de punto y coma y todos son declarados implícitamente como `public` y `abstract` (se pueden omitir).

Por su parte, todas las constantes incluidas en una interfaz se declaran implícitamente como `public`, `static` y `final` (también se pueden omitir) y es necesario inicializarlas en la misma sentencia de declaración.

En código Java, la declaración de una interfaz con un único método se vería de la siguiente forma:

```
public interface Actualizacion {void incremento(int a);
}
```

El siguiente ejemplo corresponde a una interfaz con dos constantes reales:

```
public interface Constantes {
    double VALOR_MAX = 999999.0;
    double VALOR_MIN = -0.1;
}
```




Universidad
Andrés Bello

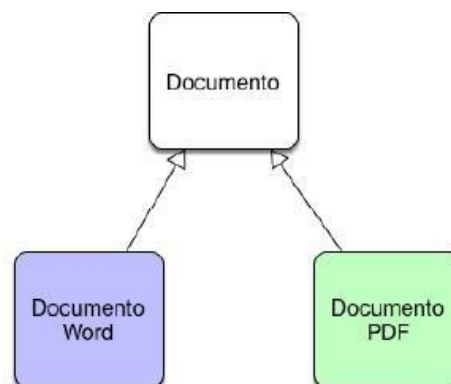
Para declarar una clase que implemente una interfaz es necesario utilizar la palabra reservada `implements` en la cabecera de declaración de la clase. Las cabeceras de los métodos (identificador y número y tipo de parámetros) deben aparecer en la clase tal y como aparecen en la interfaz implementada.

Por Ejemplo, la clase `Acumulador` implementa la interfaz `Modificacion` y por lo tanto debe declarar un método `incremento`:

```
public class Acumulador implements Modificacion {private int valor;

public Acumulador (int i) {this.valor = i;
}

public int daValor () {return this.valor;
}
public void incremento (int a) {this.valor += a;
}
}
```



Vamos a ver el código:

```
1. package com.arquitecturajava;
2.
3. public abstract class Documento {
4.
5.     private String titulo;
6.
7.
8.     public Documento(String titulo) {
9.
10.         this.titulo = titulo;
11.
12.     }
13.
14.     public String getTitulo() {
```



La cabecera con la palabra `implements` significa que la clase `Acumulador` define el método `incremento` declarado en la interfaz `Modificacion`.

El siguiente código Java muestra un ejemplo de uso de la clase `Acumulador`.

```
public class PruebaAcumulador {
    public static void main (String [] args) {
        Acumulador p = new Acumulador(25);
        p.incremento(12);
        System.out.println(p.daValor());
    }
}
```

La clase `Acumulador` tendría también la posibilidad de utilizar directamente las constantes declaradas en la interfaz si las hubiera. Para poder emplear una constante declarada en una interfaz, las clases que no implementen esa interfaz deben anteponer el identificador de la interfaz al de la constante.

Una clase puede implementar varias interfaces de los paquetes que se han importado dentro del programa, separando los nombres por comas. Java Herencia vs Interfaces.

Java Herencia vs Interfaces es una de las comparaciones más típicas cuando uno empieza a programar en Java. Siempre se generan dudas de cuando usar cada una de ellas ya que su comportamiento es similar. Vamos a construir un ejemplo sencillo que nos ayude a clarificar dudas. Vamos a suponer que tenemos una jerarquía de clases de tipos de documento que incluye documentos PDF y documentos Word.

```
6.
7. public String getVersion() {
8.     return version;
9. }
10.
11. public void setVersion(String version) {
12.     this.version = version;
13. }
14.
15. public DocumentoWord(String titulo, String version) {
16.     super(titulo);
17.     this.version = version;
18. }
19.
20. @Override
21. public void validar() {
22.
23.     System.out.println("el documento word con titulo" + getTitulo() + " ha sido
        validado");
24.
25. }
26.
27. }
```

Ya disponemos de la jerarquía de clases, todas ellas comparten un método validar que se usa para validar cada objeto. Nos queda diseñar una clase Servicio Validación que se encarga de delegar en el método validar de cada documento.

```

1. package com.arquitecturajava;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. public class ServicioValidacion {
7.
8.     private List<Documento> lista= new ArrayList<Documento>();
9.     public ServicioValidacion() {
10. // TODO Auto-generated constructor stub
11. }
12.
13. public void addDocumento(Documento d) {
14.
15.     lista.add(d);
16.
17. }
18.
19. public void validar() {

```

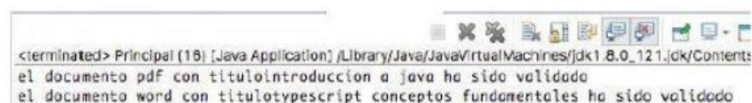
Creamos el programa principal

```

1. package com.arquitecturajava;
2.
3. public class Principal {
4.
5.     public Principal() {
6. // TODO Auto-generated constructor stub
7. }
8.
9.     public static void main(String[] args) {
10.
11.     DocumentoPDF doc1= new DocumentoPDF("introduccion a java",true);
12.     DocumentoWord doc2 = new DocumentoWord("typescript conceptos
    fundamentales", "word2010");
13.
14.     ServicioValidacion sc= new ServicioValidacion();
15.     sc.addDocumento(doc1);
16.     sc.addDocumento(doc2);
17.
18.     sc.validar();
19. }
20.
21. }

```

Ejecutamos y cada uno de los documentos será validado.



```

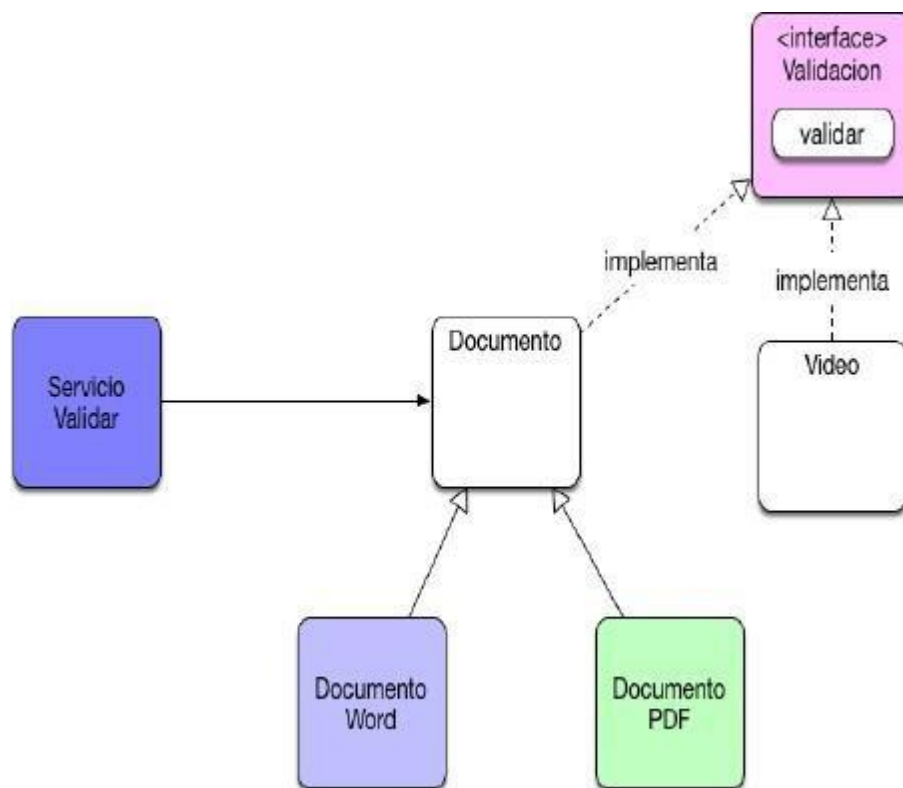
<terminated> Principal (15) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents
el documento pdf con titulo introduccion a java ha sido validado
el documento word con titulo typescript conceptos fundamentales ha sido validado

```

Java Herencia vs Interfaces

Todo es correcto, sin embargo, no es tan flexible como quisiéramos ya que la aplicación puede necesitar a futuro validar videos o audios. Lamentablemente ni los videos ni los audios son documentos y no los podemos encajar en la jerarquía.

¿Cómo podemos modificar el programa para conseguir que el servicio de validación valide otro tipo de clases? Podemos evolucionar el diseño y añadir una interface de validación de tal forma que otras clases puedan implementarlo (ajenas a la jerarquía).



Con este nuevo diseño podremos hacer que la clase de Servicio reciba un objeto que implemente el interface Validación.

Así podremos integrar la clase Video que no está en la jerarquía.

```

1. package com.arquitecturajava.ejemplo2;
2.
3. public class Video implements Validacion {
4.
5.     public Video() {
6.         // TODO Auto-generated constructor stub
7.     }
8.
9.     @Override
10.    public void validar() {
11.        System.out.println("validamos el video");
12.    }
13. }
14.
15. }

1. package com.arquitecturajava.ejemplo2;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.

```

```

6. public class ServicioValidacion {
7.
8. private List<Validacion> lista= new ArrayList<Validacion>();
9. public ServicioValidacion() {
10. // TODO Auto-generated constructor stub
11. }
12.
13. public void addDocumento(Validacion d) {
14.
15. lista.add(d);
16.
17. }
18.
19. public void validar() {
20.
21. for (Validacion d :lista) {
22.
23. d.validar();
24. }
25. }
26.
27. }

```

Creamos un nuevo programa principal:

```

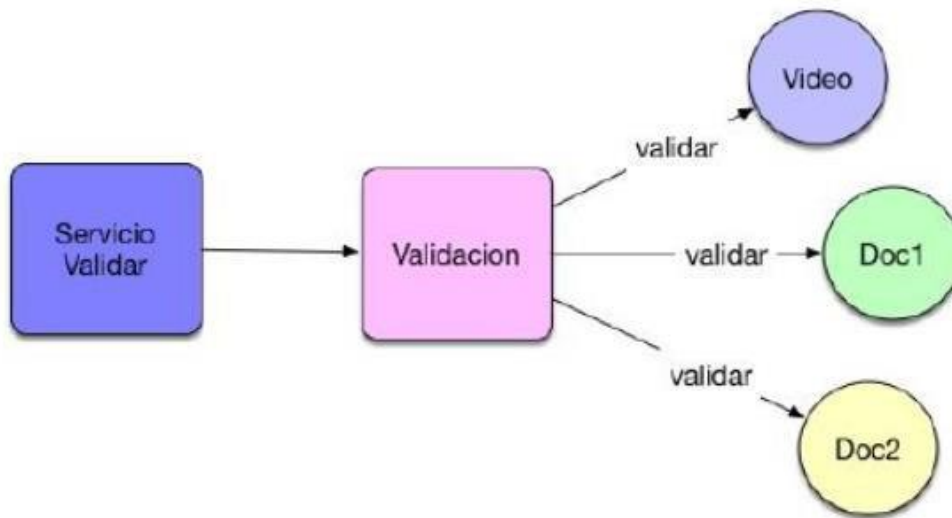
1. package com.arquitecturajava.ejemplo2;
2.
3. public class Principal {
4.
5. public Principal() {
6. // TODO Auto-generated constructor stub
7. }
8.
9. public static void main(String[] args) {
10.
11. DocumentoPDF doc1= new DocumentoPDF("introduccion a java",true);
12. DocumentoWord doc2 = new DocumentoWord("typescript conceptos
    fundamentales","word2010");
13. Video v1= new Video();
14. ServicioValidacion sc= new ServicioValidacion();
15. sc.addDocumento(doc1);
16. sc.addDocumento(doc2);
17. sc.addDocumento(v1);
18.
19. sc.validar();
20. }
21.
22. }

```

Ejecutamos:



Acabamos de integrar el concepto de Video en nuestro diseño utilizando interfaces:



Polimorfismo

El Polimorfismo es uno de los 4 pilares de la programación orientada a objetos (POO) junto con la Abstracción, Encapsulación y Herencia. Para entender qué 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.

Para empezar con esta entrada, se ha de decir que el término "Polimorfismo" es una palabra de origen griego que significa "muchas formas". Este término 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 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 basaren el siguiente diagrama de clases:

Implementando Polimorfismo mediante herencia

Para entender el Polimorfismo vamos a realizar un pequeño ejemplo que consistirá en una clase padre y dos clases hijas. Estas heredan del mismo padre, cada clase sobrescribirá un método del padre para ver cómo se comporta el polimorfismo.

Clase padre

Esta clase tiene un método llamado imprimir(), el cual solo va a imprimir un mensaje en la consola de salida.

```
public class Animal {
    private String especie;
    public Animal(String especie) {
        this.especie = especie;
    }

    public void imprimir() {
        System.out.println("Soy un animal de la especie: " + this.getEspecie());
    }

    public String getEspecie() {
        return especie;
    }

    public void setEspecie(String especie) {
        this.especie = especie;
    }
}
```

Las clases hijas

Cada una tiene sus métodos y atributos, las dos heredan la funcionalidad del padre, también las dos hijas están sobrescribiendo el método imprimir() del padre.

```
public class Perro extends Animal {
    private String nombre;
    public Perro(String especie, String nombre){
        super(especie);
        this.nombre = nombre;
    }

    public void imprimir(){
        super.imprimir();
        System.out.println("Soy un perro que ladra");
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```


La segunda clase derivada sería la siguiente:

```
public class Gato extends Animal{
    private String nombre;
    public Gato(String especie, String nombre){
        super(especie);
        this.nombre = nombre;
    }

    public void imprimir(){
        super.imprimir();
        System.out.println("Soy un gato que maulla");
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Ejecución del ejemplo de Polimorfismo en Java

Para probar el polimorfismo vamos a realizar un programa que implementa a las clases que acabamos de crear en las imágenes anteriores y vamos a imprimir con el método imprimir() paraver su comportamiento.

```
public class PolimorfismoMain {  
    public static void main(String []args){  
        Animal fido, snarf;  
        fido = new Perro("Perro", "Fido");  
        snarf = new Gato("Gato", "Snarf");  
  
        fido.printMensaje();  
  
        snarf.printMensaje();  
    }  
}
```

Como se puede apreciar en la imagen anterior, se están declarando dos variables de la clase Animal. Posteriormente esas variables son instanciadas con el operador “new” y pasan aser instancias de las clases “Perro” y “Gato” respectivamente.

Después cada objeto imprime su propio mensaje.

Salida: en el resultado podemos observar que se imprime un mensaje que esta declarado en el padre y otro mensaje cuando se sobrescribió el método printMensaje() en cada uno de los hijos.

Con esto podemos comprobar la herencia ya que los hijos están presentando comportamiento propio y también comportamiento de su clase padre.

Soy un animal de la especie: Perro Soy un perro que ladra

Soy un animal de la especie: Gato Soy un gato que maulla