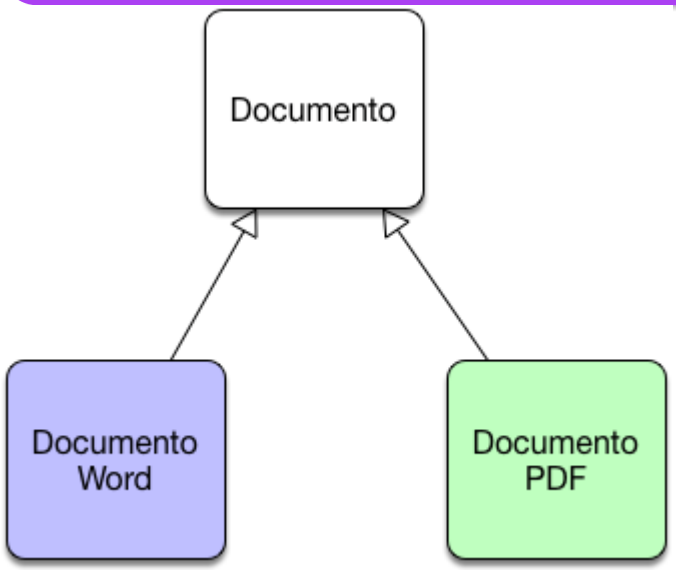
Java Herencia vs Interfaces es una de las comparaciones más típicas cuando uno empieza a programar en Java. Siempre se generan dudas de cuando usar cada una de ellas ya que su comportamiento es similar. Vamos a construir un ejemplo sencillo que nos ayude a clarificar dudas. Vamos a suponer que tenemos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



Vamos a ver el código:

```
package com.arquitecturajava;

public abstract class Documento {
```

```
	private String titulo;

	public Documento(String titulo) {

		this.titulo = titulo;

	}

	public String getTitulo() {
		return titulo;
	}

	public void setTitulo(String titulo) {
		this.titulo = titulo;
	}

	public abstract void validar();

}
```

```
package com.arquitecturajava;

public class DocumentoPDF extends Documento{

	private boolean protegido;

	public boolean isProtegido() {
		return protegido;
```

```
	}

	public void setProtegido(boolean protegido) {
		this.protegido = protegido;
	}

	public DocumentoPDF(String titulo,boolean protegido) {
		super(titulo);
		this.protegido=protegido;
		// TODO Auto-generated constructor stub
	}

	@Override
	public void validar() {

		System.out.println("el documento pdf con titulo" +
getTitulo()+" ha sido validado");

	}

}


package com.arquitecturajava;

public class DocumentoWord extends Documento{

	private String version;

	public String getVersion() {
		return version;
```

```
	}

	public void setVersion(String version) {
		this.version = version;
	}

	public DocumentoWord(String titulo,String version) {
		super(titulo);
		this.version=version;
	}

	@Override
	public void validar() {

		System.out.println("el documento word con titulo" +
getTitulo()+" ha sido validado");

	}

}
```

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 ServicioValidación que se encarga de delegar en el método validar de cada documento.

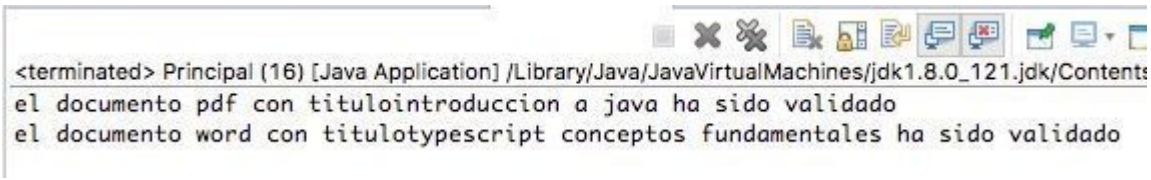
```
package com.arquitecturajava;

import java.util.ArrayList;
```

```
import java.util.List;

public class ServicioValidacion {

	private List&amp;lt;Documento&amp;gt; lista= new
ArrayList&amp;lt;Documento&amp;gt;();
	public ServicioValidacion() {
		// TODO Auto-generated constructor stub
	}

	public void addDocumento(Documento d) {

		lista.add(d);

	}

	public void validar() {

		for (Documento d :lista) {

			d.validar();
		}
	}

}
```

Creamos el programa principal:

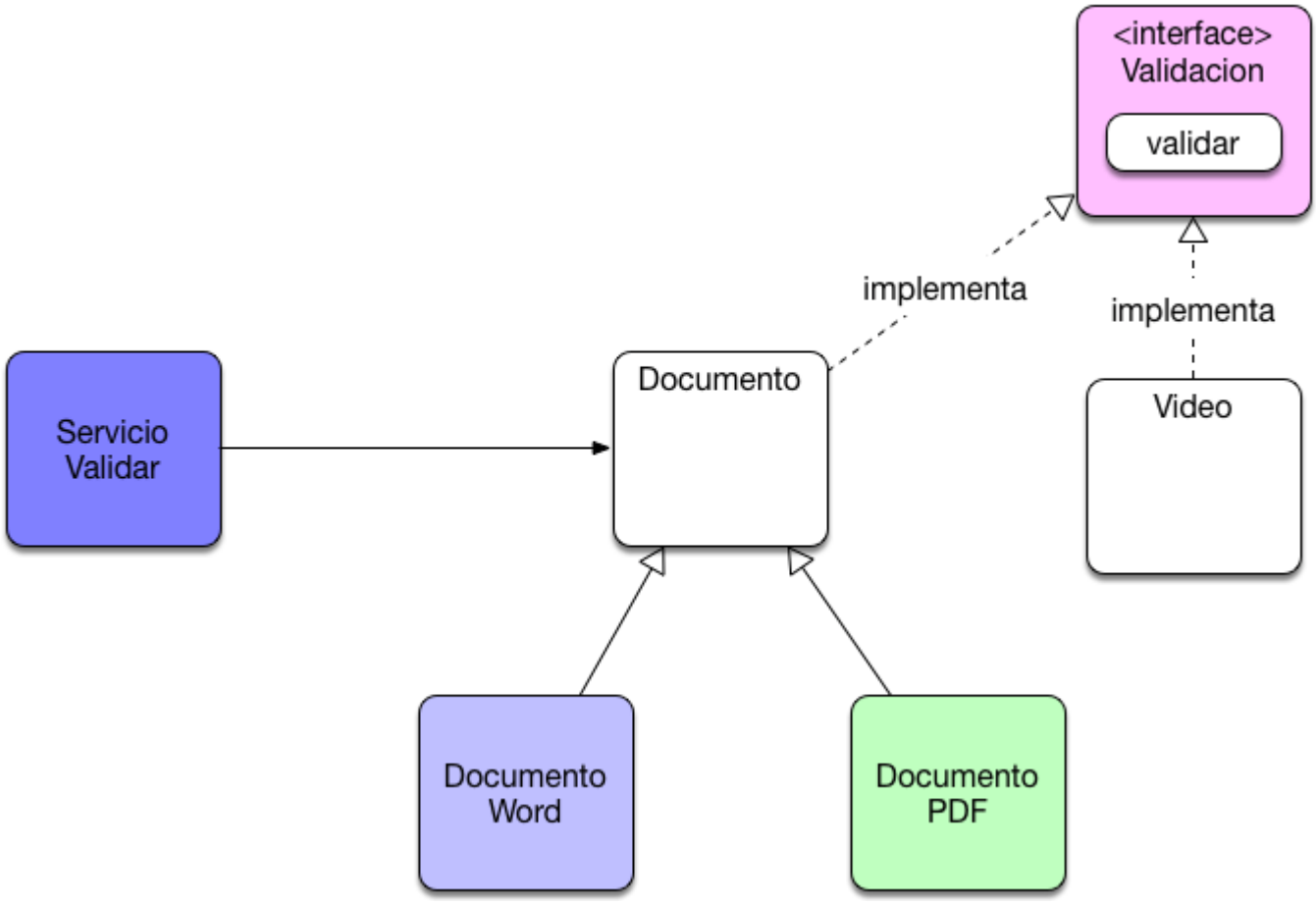
```
package com.arquitecturajava;

public class Principal {

	public Principal() {
		// TODO Auto-generated constructor stub
	}

	public static void main(String[] args) {

		DocumentoPDF doc1= new DocumentoPDF("introduccion a
java",true);
		DocumentoWord doc2 = new DocumentoWord("typescript
conceptos fundamentales","word2010");

		ServicioValidacion sc= new ServicioValidacion();
		sc.addDocumento(doc1);
		sc.addDocumento(doc2);

		sc.validar();
	}

}
```

Ejecutamos y cada uno de los documentos será validado.

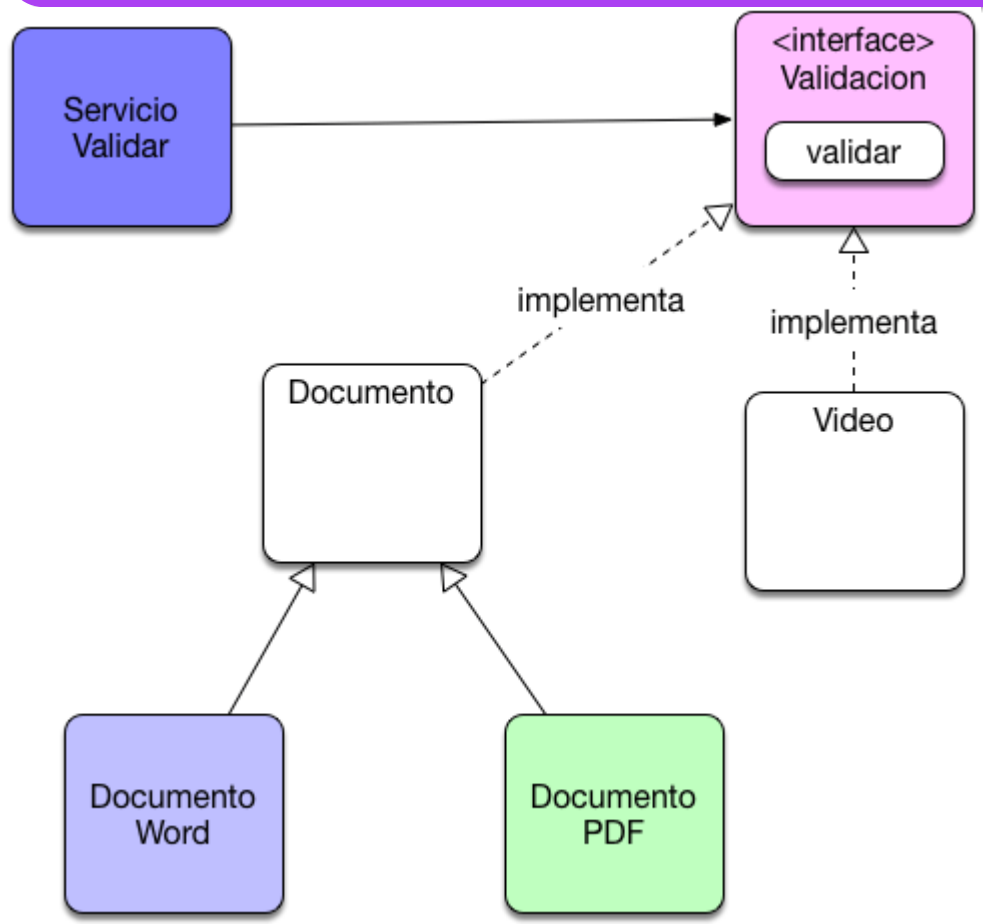


Java Herencia vs Interfaces

Todo es correcto , sin embargo no es tan flexible como quisieramos 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 un 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 Validacion.



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

```
package com.arquitecturajava.ejemplo2;

public class Video  implements Validacion {
```

```
	public Video() {
		// TODO Auto-generated constructor stub
	}

	@Override
	public void validar() {
		System.out.println("validamos el video");

	}

}
```

```
package com.arquitecturajava.ejemplo2;

import java.util.ArrayList;
import java.util.List;

public class ServicioValidacion {

	private List&amp;lt;Validacion&amp;gt; lista= new
ArrayList&amp;lt;Validacion&amp;gt;();
	public ServicioValidacion() {
		// TODO Auto-generated constructor stub
	}

	public void addDocumento(Validacion d) {

		lista.add(d);
```

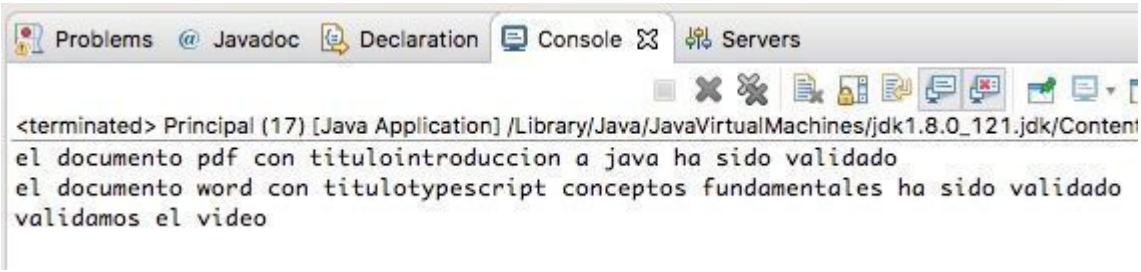
```
	}

	public void validar() {

		for (Validacion d :lista) {

			d.validar();
		}
	}

}
```

Creamos un nuevo programa principal:

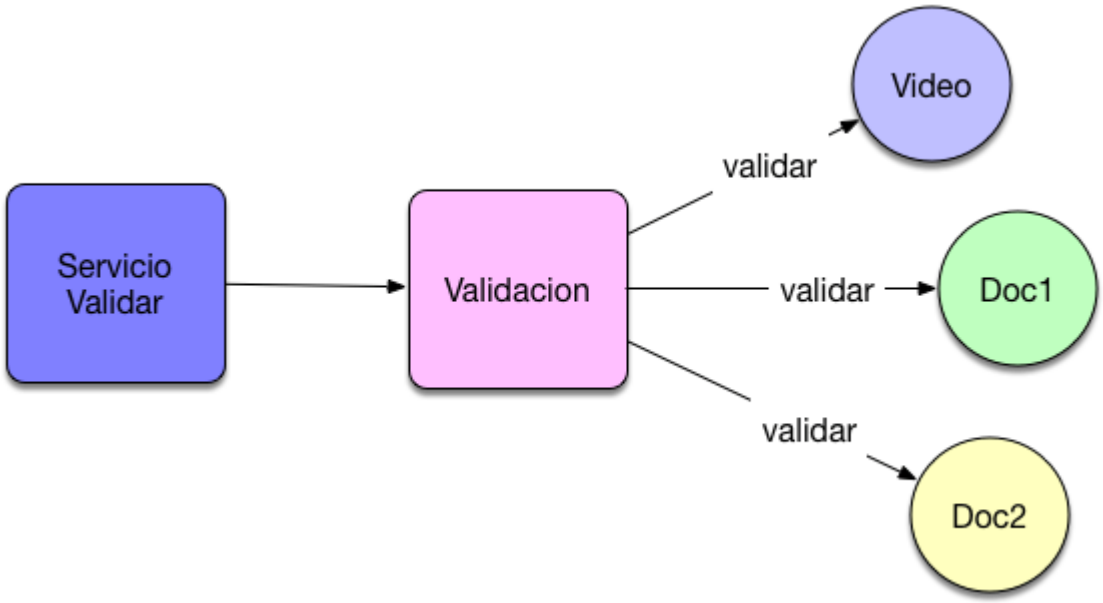
```
package com.arquitecturajava.ejemplo2;

public class Principal {

	public Principal() {
		// TODO Auto-generated constructor stub
	}

	public static void main(String[] args) {

		DocumentoPDF doc1= new DocumentoPDF("introduccion a
java",true);
		DocumentoWord doc2 = new DocumentoWord("typescript
conceptos fundamentales","word2010");
```

```
                Video v1= new Video();
                ServicioValidacion sc= new ServicioValidacion();
                sc.addDocumento(doc1);
                sc.addDocumento(doc2);
                sc.addDocumento(v1);

                sc.validar();
        }

}
```

Ejecutamos:



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



Cuando hablamos de Java Herencia vs Interfaces los interfaces ganan a nivel de flexibilidad.

Otros artículos relacionados:

1. Java 8 interface static methods y reutilizacion
2. Java Predicate Interface y sus métodos
3. Java Mixins, un ejemplo sencillo
4. Java Herencia