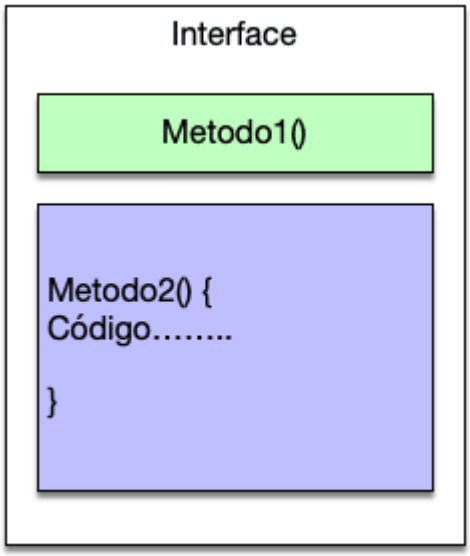
Los Default Methods son relativamente nuevos ya que muchos desarrolladores se encuentran empezando a trabajar con Java 8 y comienzan a tener un mayor conocimiento de esta versión de la plataforma . ¿ Para que sirven los métodos por defecto en Java? . La respuesta es sencilla , sirven para añadir métodos con implementación a un interface concreto de tal forma que el interface tenga ya una implementación y no tengamos la necesidad de construirla a posteriori .

[ihc-hide-content ihc_mb_type="show" ihc_mb_who="4" ihc_mb_template="1"]



En principio sus ventajas son más que evidentes ya que aportan una gran flexibilidad a la hora de diseñar una clase . Vamos a ver un ejemplo sencillo que nos ayude a clarificar estos conceptos. Supongamos que partimos de un interface Azulejo que dispone de tres implementaciones sencillas.

```
package com.arquitecturajava;

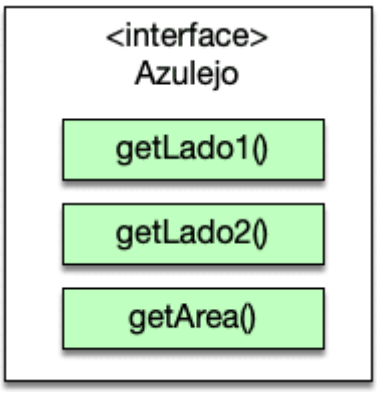
import java.util.List;
```

```
public interface Azulejo {

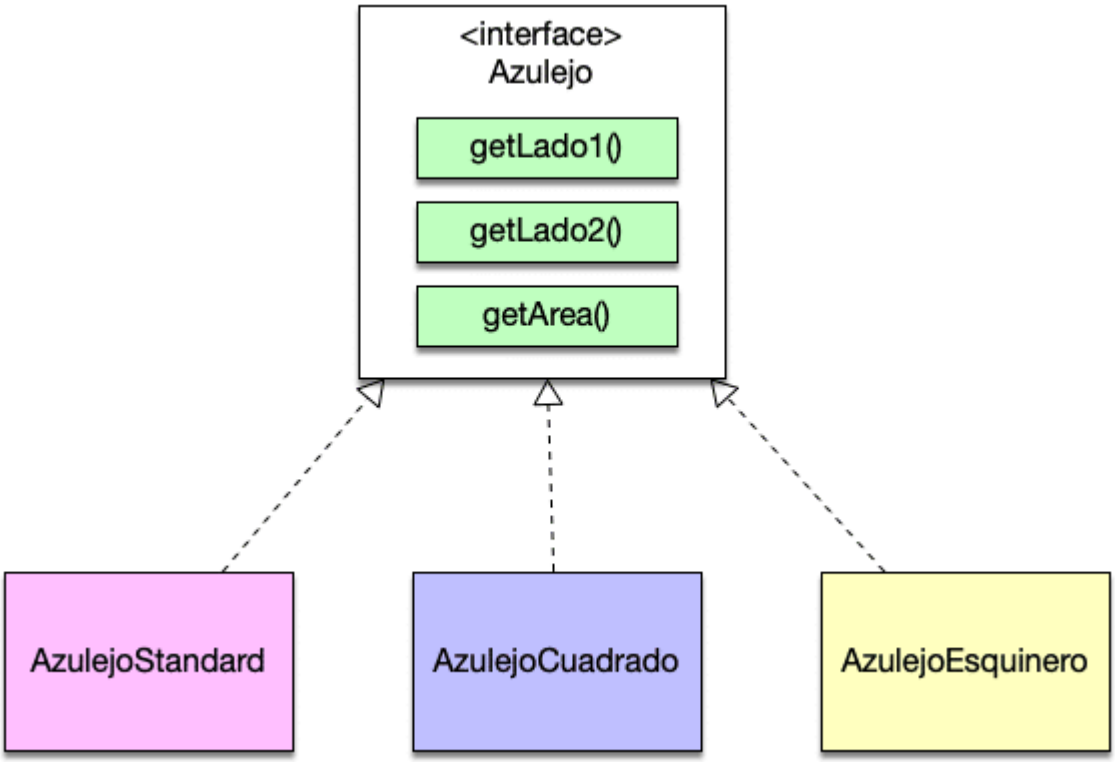
  double getLado1();
  double getLado2();
  double getArea();

}
```

Normalmente de un Azulejo deseamos conocer sus lados y el area que ocupa de una superficie determinada . Por ello hemos diseñado un interface bastante pequeño que incluya únicamente los 3 métodos fundamentales.



Es momento de ver un diagrama de clases con las clases que están asociadas a este interface.



Vamos a construirlas a nivel de código:

```
package com.arquitecturajava;

public class AzulejoStandard implements Azulejo {

  private double lado1;
  private double lado2;

  public double getLado1() {
    return lado1;
  }

  public void setLado1(double lado1) {
    this.lado1 = lado1;
```

```
  }

  public double getLado2() {
    return lado2;
  }

  public void setLado2(double lado2) {
    this.lado2 = lado2;
  }

  public double getArea() {

    return getLado1() * getLado2();
  }

}
public class AzulejoCuadrado  implements Azulejo{

  private double lado;
  @Override
  public double getLado1() {
    // TODO Auto-generated method stub
    return this.lado;
  }

  @Override
  public double getLado2() {
    // TODO Auto-generated method stub
    return this.lado;
  }
  public AzulejoCuadrado(double lado) {
```

```
    super();
    this.lado = lado;
  }

  public void setLado1(double lado) {
    this.lado=lado;
  }
  public void setLado2(double lado) {
    this.lado=lado;
  }

  @Override
  public double getArea() {
    // TODO Auto-generated method stub
    return lado*lado;
  }


}
```

Default Methods en Java

Estas son las dos clases sencillas , el AzulejoStandard y el AzulejoCuadrado. El primero hace referencia a un azulejo con dos lados de tamaños diferentes algo que es lo más habitual y el segundo hace referencia a un azulejo que tiene ambos lados iguales. En este caso nos encontramos ante una situación en la que podemos diseñar una clase abstracta que contenga el método getArea() o diseñar un método a nivel del interface que sea el encargado de implementar dicha funcionalidad . Vamos a ver esta opción:

```
package com.arquitecturajava;

import java.util.List;
```

```
public interface Azulejo {

  double getLado1();
  double getLado2();

  default double getArea() {
    return this.getLado1()* this.getLado2();
  }

}
```

De esta manera simplificamos nuestras clases ya que ahora ninguna de ellas necesitará sobreescribir el método Area , por lo tanto nuestra clase Azulejo Standard quedará :

```
package com.arquitecturajava;

public class AzulejoStandard implements Azulejo {

  private double lado1;
  private double lado2;

  public double getLado1() {
    return lado1;
  }

  public void setLado1(double lado1) {
    this.lado1 = lado1;
  }

  public double getLado2() {
    return lado2;
  }
```

```
  public void setLado2(double lado2) {
    this.lado2 = lado2;
  }

}
```

De la misma forma quedaría el AzulejoCuadrado , hemos hecho un avance en cuanto a diseño de nuestras clases . Sin embargo en muchas ocaciones nos olvidamos que él añadir funcionalidad a un interface es una responsabilidad importante ya que afectará a todas las clases que lo implementen. Vamos a ver un caso que nos ayude a clarificar . Supongamos que tenemos otro azulejo denominado AzulejoEsquinero. Este azulejo hace referencia a un azulejo que es cortado para encajar en una esquina.

```
package com.arquitecturajava;

public class AzulejoEsquinero  implements Azulejo{

  private int lado1;
  private int lado2;
  public AzulejoEsquinero(int lado1, int lado2) {
    super();
    this.lado1 = lado1;
    this.lado2 = lado2;
  }

  @Override
  public double getLado1() {
    // TODO Auto-generated method stub
    return lado1;
  }

  @Override
```

```
  public double getLado2() {
    // TODO Auto-generated method stub
    return lado2;
  }

  @Override
  public double getArea() {
    // TODO Auto-generated method stub
   throw  new UnsupportedOperationException();
  }

}
```

Como vemos se trata de una clase muy muy sencilla . Tiene una peculiaridad y es que no sabemos calcular su Area ya que al ser un Azulejo cortado a medida no es sencillo tener una formula a nuestra disposición . Es un sacrificio que tenemos para poder tener este tipo de azulejo y poderlo de alguna manera contabilizar . No parece algo mucho más problemático . Lamentablemente las cosas no son tan sencillas .

Default Methods , temporalidad e implementación

Los métodos por defecto se usan habitualmente para añadir una nueva funcionalidad al interface que todas las clases implementen de forma automática . Una opción sencilla que nos puede ser útil es el método esMayor() que nos dirá si el area de un azulejo es mayor que la otra. Vamos a construir su implementación que es bastante sencilla:

```
package com.arquitecturajava;

import java.util.List;
```

```
public interface Azulejo {

  double getLado1();
  double getLado2();

  default boolean esMayor(Azulejo azulejo) {
    return this.getArea()>azulejo.getArea();
  }
  default double getArea() {
    return this.getLado1()* this.getLado2();
  }

}
```

Esta implementación nos genera un problema claro . El azulejoEsquinero no encaja ya que no puede hacer un uso del método area. No solo eso sino que podemos tener a nivel del interface también un método estático que calcule el area total de un conjunto de azulejos:

```
package com.arquitecturajava;

import java.util.List;

public interface Azulejo {

  double getLado1();
  double getLado2();

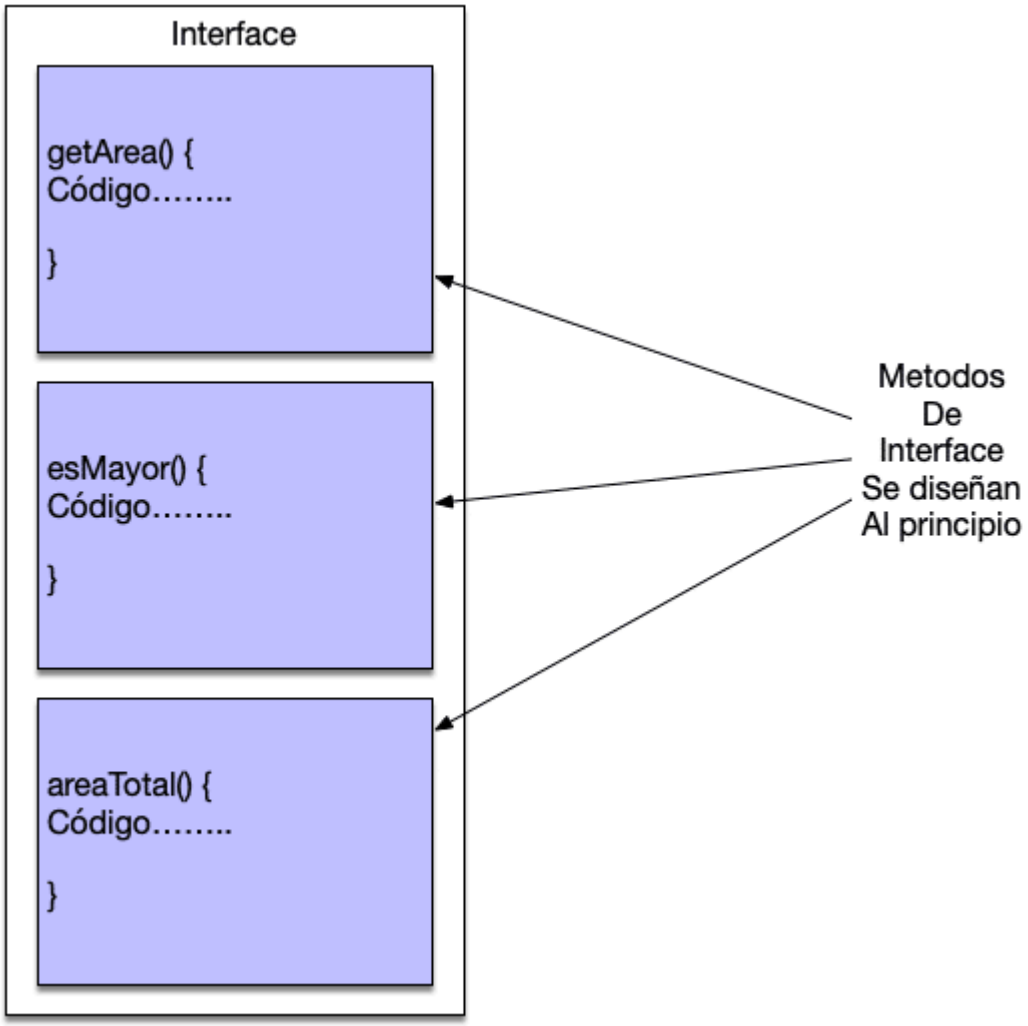
  default boolean esMayor(Azulejo azulejo) {
    return this.getArea()>azulejo.getArea();
  }
  default double getArea() {
    return this.getLado1()* this.getLado2();
```

```
  }
  static double areaTotal(List<Azulejo> azulejos) {
    return azulejos.stream().mapToDouble(Azulejo::getArea).sum();
  }

}
```

Métodos por defecto y sus problemas

Este método tampoco encaja con él azulejo esquinero . Sin embargo estamos obligando a todas las clases a disponer de él. ¿Qué es lo que esta pasando? . Lo que sucede es que hemos añadido estos métodos al diseño en un momento “posterior” a la construcción de la jerarquía de clases . No cuando la propia jerarquía de clases se construye de tal forma que las personas que diseñan la jerarquía de clases puedan darse cuenta de este problema.



Por lo tanto siempre que diseñemos métodos a nivel de interface debemos tener en cuando que ha de hacerse en un momento inicial del desarrollo . Añadir métodos default o estáticos a los interfaces en un desarrollo avanzado implica asumir grandes riesgos. En nuestro caso la solución es sencilla , el AzulejoEsquinero de alguna forma debe devolver un valor en su Area. Si esto se sabe a nivel de diseño inicial ya que vemos los métodos que el interface tiene . Es razonable asumir por ejemplo que su area es 1/3 del azulejo completo.

```
package com.arquitecturajava;

public class AzulejoEsquinero  implements Azulejo{
```

```
  private int lado1;
  private int lado2;
  public AzulejoEsquinero(int lado1, int lado2) {
    super();
    this.lado1 = lado1;
    this.lado2 = lado2;
  }

  @Override
  public double getLado1() {
    // TODO Auto-generated method stub
    return lado1;
  }

  @Override
  public double getLado2() {
    // TODO Auto-generated method stub
    return lado2;
  }

  @Override
  public double getArea() {
    return getArea()/3;
  }

}
```

Tengamos siempre esto en cuanta cuando diseñamos interfaces y añadamos los métodos en el inicio del desarrollo:

Otros artículos relacionados

1. Optional Map y el concepto de delegación
2. Java Stream Partition y el manejo de Listas
3. Java Strategy Pattern herencia vs composición
4. Java 8

[/ihc-hide-content]