Diferencia entre herencia, interface y composición en Java

Interface: las interfaces son los planos de las clases. Especifican qué debe hacer una clase pero no cómo. Como una clase, una interface puede tener métodos y variables, pero los métodos declarados son abstractos por default; es decir, solo contienen la firma del método. Las interfaces se utilizan para implementar una abstracción completa.

Herencia: es un mecanismo en Java mediante el cual una clase puede heredar las características de otra clase. Hay varias herencias posibles en java:

Herencia única (single inheritance): las subclases heredan las características de una superclase. En la imagen siguiente, la clase A sirve como clase base para la clase derivada B.

Herencia multinivel (multilevel inheritance): una clase derivada heredará de una clase base y también actuará como clase base para otra clase derivada. Por ejemplo, la clase A sirve como clase base para la clase derivada B, que a su vez sirve como clase base para la clase derivada C. En Java, una clase no puede acceder directamente a los miembros del abuelo.

Herencia jerárquica (hierarchical inheritance): una clase sirve como superclase (clase base) para más de una subclase. Por ejemplo, la clase A sirve como clase base para las clases derivadas B, C y D.

La siguiente tabla describe la diferencia entre herencia e interface:

Categoría	Herencia	Interface
Descripción	La herencia es el mecanismo en Java por el cual una clase puede heredar las características de otra clase.	La interface es el plano de la clase. Especifica qué debe hacer una clase pero no cómo. Como una clase, una interface puede tener métodos y variables, pero los métodos declarados son abstractos por default (solo firma, sin cuerpo).
Uso	Se usa para obtener las características de otra clase.	Se utiliza para proporcionar una abstracción total.
Sintaxis	class Nombre_subclase extends Nombre_superclase { }	interface Nombre { }
Tipos de herencia	Se utiliza para proporcionar 4 tipos de herencia. (herencia multinivel, simple, híbrida y jerárquica)	Se utiliza para proporcionar 1 tipo de herencia (múltiple).
Palabras clave	extends	implements
Herencia	Podemos heredar menos clases que con Interface.	Podemos heredar muchísimas más clases que con la Herencia.
Definición de métodos	Los métodos se pueden definir (implementar) dentro de la clase en caso de herencia.	Los métodos no se pueden definir dentro de la Interface (excepto mediante el uso de palabras clave como static y otras predeterminadas).
Sobrecarga	Sobrecarga el sistema si se intenta extender muchas clases.	El sistema no se sobrecarga, no importa cuántas clases implementemos.
Funcionalidad	No proporciona la funcionalidad de acoplamiento débil (loose coupling*).	Si proporciona la funcionalidad de acoplamiento débil.
Herencia múltiple	No podemos hacer herencia múltiple (causa un error en tiempo de compilación).	Podemos hacer herencia múltiple usando interfaces.

Diferencia entre herencia y composición

#	Herencia	Composición
1	Definimos la clase que estamos heredando (superclase) y, lo más importante, no se puede cambiar en tiempo de ejecución.	Definimos el tipo que queremos usar cuya implementación puede cambiar en tiempo de ejecución. Por lo tanto, la composición es mucho más flexible que la herencia.
2	Aquí solo podemos extender una clase, en otras palabras, más de una clase no se puede extender ya que Java no admite herencia múltiple.	Permite utilizar la funcionalidad de otras clases diferentes.
3	Necesitamos la clase padre para probar la clase hijo.	Permite probar la implementación de las clases que estamos usando independientemente de las clases padre o hijo.
4	No puede extender una clase final.	Permite la reutilización de código incluso de las clases final.
5	Es una relación es-un (**is-a**).	Es una relación tiene-una (**has-a**).

Ejemplo de herencia:

```
class A {
	int a, b;
	public void add(int x, int y)
	{
		a = x;
		b = y;
		System.out.println(
			"suma de a + b es:"
			+ (a + b));
	}
}

class B extends A {
	public void sum(int x, int y)
	{
		add(x, y);
	}

	// Driver Code
	public static void main(String[] args)
	{
		B b1 = new B();
		b1.sum(5, 6);
	}
}

Salida:

suma de a+b es:11
```

Aquí, la clase B es la clase derivada que hereda la propiedad (método add()) de la clase base A.

Ejemplo de composición:

La composición también proporciona reutilización de código, pero la diferencia aquí es que no heredamos (extend) la clase para esto.

```
// programa Java para ilustrar el concepto de Composición

import java.io.*;
import java.util.*;

// class book
class Book {

	public String title;
	public String author;

	Book(String title, String author)
	{

		this.title = title;
		this.author = author;
	}
}

// la clase Library contiene una lista de libros
class Library {

	// referencia a la lista de libros.
	private final List<Book> books;

	Library(List<Book> books)
	{
		this.books = books;
	}

	public List<Book> getTotalBooksInLibrary()
	{

		return books;
	}
}

// metodo main()
class Test {
	public static void main(String[] args)
	{

		// Creando los Objetos de la clase Book
		Book b1 = new Book(
			"EffectiveJ Java",
			"Joshua Bloch");
		Book b2 = new Book(
			"Thinking in Java",
			"Bruce Eckel");
		Book b3 = new Book(
			"Java: The Complete Reference",
			"Herbert Schildt");
```

```
            // Creando la lista que contiene los libros
            List<Book> books = new ArrayList<Book>();
            books.add(b1);
            books.add(b2);
            books.add(b3);

            Library library = new Library(books);

            List<Book> bks = library.getTotalBooksInLibrary();

            for (Book bk : bks) {

                  System.out.println("Titulo : "
                                          + bk.title + " y "
                                          + " Autor : "
                                          + bk.author);
            }
      }
}
```

```
Salida:
Titulo : EffectiveJ Java y  Autor : Joshua Bloch
Titulo : Thinking in Java y  Autor : Bruce Eckel
Titulo : Java: The Complete Reference y  Autor : Herbert Schildt
```

** **Loose coupling** (acoplamiento flojo, débil o flexible): El acoplamiento débil en Java significa que las clases son independientes entre sí. El único conocimiento que una clase tiene sobre otra, es lo que esta última ha expuesto a través de sus interfaces en acoplamiento débil. Una situación de acoplamiento débil requiere que se utilicen objetos del exterior.*