Material descargable

egg

Interfaces

¡Hola! 👋 Te damos la bienvenida a interfaces

En nuestra búsqueda continua por mejorar nuestras habilidades de programación en Java, nos encontramos con el reto de que los sistemas que estamos creando se están volviendo cada vez más complejos y nos damos cuenta de que necesitamos herramientas más sofisticadas para manejar esa complejidad. Afortunadamente, Java tiene una solución para esto: las interfaces.

¡Que comience el viaje! 🚀

Interfaces

Las interfaces en Java son un tipo de referencia similar a las clases y son un conjunto de constantes y métodos abstractos, por lo tanto las interfaces son abstractas y no se pueden instanciar. Cuando una clase **implementa** una interfaz, la clase está obligada a **implementar** todos los métodos abstractos definidos en ella. En esencia, una interfaz es una forma de garantizar un contrato particular de comportamiento.

¿Cómo se usan?

Las interfaces se declaran usando la palabra clave “interface”, similar a cómo se declaran las clases. Aquí tienes un ejemplo de una interface sencilla:

```
public interface Dibujable {
    void dibujar();
}
```

Características a tener en cuenta:

- Al ser abstractas no pueden instanciarse

- Sus métodos abstractos son públicos por defecto, por lo tanto, no necesitas usar el modificador de acceso.
- No pueden usar variables, solo pueden declarar atributos finales (constantes).
- Pueden ser heredadas por otra interfaz usando la palabra clave “extends”.
- Pueden ser implementadas por una clase usando la palabra clave “implements”.

```
public class Circulo extends FiguraGeometrica implements Dibujable {
    @Override
    public void dibujar() {
       // implementación del método para dibujar la figura
    }
}
```

```
public class Rectangulo extends Paralelogramo implements Dibujable {
    @Override
    public void dibujar() {
       // implementación del método para dibujar la figura
    }
}
```

```
public class Escaleno extends Triangulo implements Dibujable {
    @Override
    public void dibujar() {
       // implementación del método para dibujar la figura
    }
}
```

Luego puedes declarar una variable del tipo de la interfaz y asignarle una instancia de un objeto que implemente esa interfaz:

```
public class Main {
    public static void main(String[] args) {
        Dibujable[] dibujables = {new Circulo(),
                                  new Rectangulo(), new Escaleno()};
        realizarDibujos(dibujables);
    }

    public static void realizarDibujos(Dibujable[] dibujable) {
```

```
        for (Dibujable d : dibujable) {
            d.dibujar();
        }
    }
}
```

El problema de la herencia de clases

La herencia de clases permite la reutilización de código y la creación de relaciones jerárquicas entre clases. Sin embargo, también presenta varios problemas y desafíos a medida que se crean sistemas más complejos. Aquí se explican algunas razones por las cuales se considera que la agregación y las interfaces son generalmente preferibles a la herencia de clases.

Acoplamiento fuerte: La herencia de clases crea un acoplamiento fuerte entre la clase base y la clase derivada, porque la clase derivada depende directamente de la implementación de la clase base. Esto puede hacer que los cambios en la clase base se propaguen a las clases derivadas y potencialmente causen problemas cuando se quiere modificar sistemas más complejos.

Rigidez: Una vez que se define una jerarquía de herencia, es difícil cambiarla. Si en un momento quieres extender tu programa y te percatas de que necesitabas una clase intermedia entre dos clases, para representar otro tipo de comportamiento común entre algunas subclases, eso te llevaría tener que modificar bastante parte del código. En cambio usando interfaces puedes extender el comportamiento de una clase sin necesidad de modificar el código existente.

Por ejemplo imagina que tienes una clase Empleado y otras subclases como Programador, Gerente y Diseñador.

```
public class Empleado {
   public void trabajar() {
       //...
   }
}
```

```java
public class Programador extends Empleado {
    @Override
    public void trabajar() {
        super.trabajar();
        programar();
    }

    public void programar() {
        //...
    }
}

public class Gerente extends Empleado {
    @Override
    public void trabajar() {
        super.trabajar();
        administrar();
    }
    public void administrar() {
        //...
    }
}

public class Diseñador extends Empleado {
    @Override
    public void trabajar() {
        super.trabajar();
        diseñar();
    }
    public void diseñar() {
        //...
    }
}
```

Pero, ¿qué pasa si ahora tienes una situación donde un empleado puede ser tanto un gerente como un programador, o un diseñador que también sabe programar? La herencia de clases no te permite tener un Gerente que también es un Programador. Podrías crear más subclases como GerenteProgramador, pero esto se volvería rápidamente complicado y difícil de mantener.

Aquí es donde las interfaces y la agregación se vuelven muy útiles. En lugar de extender una clase Empleado, podrías tener una agregación de trabajos y que cada uno implemente una interfaz genérica para cada habilidad:

```
public class Empleado {

    private Posicion[] posicion;

    public Empleado(Posicion[] posicion) {
        this.posicion = posicion;
    }

    public void trabajar(Posicion[] posiciones) {
        for (Posicion posicion : posiciones) {
            posicion.ejecutarTareaEspecifica();
        }
    }
}
```

```
public interface Posicion {
    void ejecutarTareaEspecifica();
}
```

```
public class Gerente implements Posicion {

    @Override
    public void ejecutarTareaEspecifica() {
        gerenciar();
    }

    public void gerenciar() {
        //implementación
    }

}
```

```
public class Diseñador implements Posicion {

    @Override
    public void ejecutarTareaEspecifica() {
```

```
        diseñar();
    }

    public void diseñar() {
        //implementación
    }

}
```

```
public class ProgramadorJunior implements Posicion {

    @Override
    public void ejecutarTareaEspecifica() {
        programarComojunior();
    }

    public void programarComojunior() {
        //implementación
    }

}
```

```
public class ProgramadorSenior implements Posicion {

    @Override
    public void ejecutarTareaEspecifica() {
        programarComoSenior();
    }

    public void programarComoSenior() {
        //implementación
    }

}
```

Método *default*

El método default en las interfaces es una característica que permite agregar métodos con implementación en una interfaz. Antes de Java 8, todas las declaraciones de métodos en una interfaz eran implícitamente abstractas, lo que

significaba que las clases que implementaban la interfaz debían proporcionar una implementación para cada método.

💡*Los métodos por default también pueden ser estáticos y privados.*

```
public interface MenuServicio {
    default void iniciar() {
        Integer opcion = 0;
        do {
            imprimirTitulo();
            imprimirOpciones();
            opcion = obtenerOpcion();
            seleccionarOpcion();
        } while (validarSalida());
        saludo();
    }

    void imprimirTitulo();

    void imprimirOpciones();

    Integer obtenerOpcion();

    void seleccionarOpcion(Integer opcion);

    Boolean validarSalida(Integer opcion);

    void saludo();
}
```

Algunos puntos clave sobre los métodos default en las interfaces son:

- Los métodos default se definen utilizando la palabra clave default seguida del método y su implementación en la interfaz.
- Los métodos default pueden ser utilizados por todas las clases que implementan la interfaz sin requerir una implementación explícita en cada una de ellas.
- Si una clase implementa múltiples interfaces que contienen un método default con la misma firma, debe sobreescribirlo proporcionando su propia implementación para resolver la ambigüedad.

- Las clases que implementan una interfaz pueden optar por no sobrescribir un método default si desean utilizar la implementación predeterminada proporcionada por la interfaz.
- Los métodos default en las interfaces son útiles para agregar nuevas funcionalidades a las interfaces existentes sin afectar el código existente.

Herencia múltiple

La herencia múltiple es un concepto en el que una clase puede heredar comportamientos y características de más de una superclase. Java no permite la herencia múltiple directamente a través de clases, pero sí a través de interfaces. Una clase puede implementar múltiples interfaces, permitiendo de este modo la herencia múltiple.

```
public class Pato implements Volador, Nadador {

    @Override
    public void volar() {
        System.out.println("El pato está volando");
    }

    @Override
    public void nadar() {
        System.out.println("El pato está nadando");
    }
}
```

Clase anónimas para implementar interfaces

Las clases anónimas son una forma de declarar e instanciar una subclase de una clase abstracta, es decir, de alguna manera son una subclase del tipo de la variable a la que se le está asignando la instancia, pero esta clase anónima no tiene nombre y, por lo tanto, no puedes referenciar su tipo. Son útiles cuando necesitas crear una implementación de una interfaz en el momento.

```
interface Presentador {
    void presentarse(String nombre);
}
```

```
public class Main {
    public static void main(String[] args) {
        Presentador presentacion1 = new Presentador() {
            public void presentarse(String nombre) {
                System.out.println("Mi nombre es "+nombre);
            }
        };
        Presentador presentacion2 = new Presentador() {
            public void presentarse(String nombre) {
                System.out.println("Hola! me llamo "+nombre+" y tu?");
            }
        };
        Presentador presentacion3 = new Presentador() {
            public void presentarse(String nombre) {
                System.out.println("Hola me dicen "+nombre+" pero no me
apetece conocerte, adiós!");
            }
        };

        presentacion1.presentarse("Nahuel");
        presentacion2.presentarse("Nahuel");
        presentacion3.presentarse("Nahuel");
    }
}
```

Mira el siguiente video para entender mejor cómo funciona:
👉 **Interfaces | Clases anónimas | JAVA | Egg**

Interfaz funcional

En Java, una interfaz funcional es aquella que contiene solo un método abstracto. Estas interfaces son fundamentales en la programación funcional de Java y se utilizan como base para las expresiones lambda, las cuales explicaremos en la siguiente sección. Algunos ejemplos comunes de interfaces funcionales que pueden haber visto son Runnable, Callable y Comparator. Si queremos crear nuestra propia interfaz funcional solo necesitamos declarar una interfaz con un solo método abstracto. También se utiliza la anotación @FunctionalInterface, para evitar que los desarrolladores agreguen inadvertidamente más métodos a la interfaz, ya que el compilador asegura que las interfaces con esa anotación cumplan la regla de tener solo un método.

📍*La programación funcional no se profundizará en este curso.*

Las expresiones Lambda

Una expresión lambda es una función anónima (sin nombre) que podemos usar para crear delegados o tipos de referencia. Proporcionan una forma clara y concisa de representar un método de interfaz funcional y son un reemplazo de la implementación a través de clases anónimas que vimos anteriormente. Esta es su sintaxis básica:

```
(parámetros...) -> cuerpo;
```

Si el método de la interfaz funcional no recibe parámetros:

```
() -> cuerpo;
```

No necesitas usar el "*return*", automáticamente devolverá lo que pongas en el cuerpo. Si en la implementación necesitas usar más de una línea de código debes usar llaves y si el método devuelve algún valor también tienes que usar "*return*":

```
() -> {
        // línea de código
        // línea de código
        return // valor
};
```

La función lambda marcará error si la implementación que haces no concuerda con el método esperado.Mira el siguiente video para ver cómo usar las expresiones lambdas:

👉 **Interfaces | Interfaces funcionales lambdas | JAVA | Egg**

Funciones como parámetro

La introducción de interfaces funcionales y expresiones lambda en Java 8 representó un cambio fundamental en el lenguaje. Permitió una mayor

capacidad para desarrollar código más limpio y conciso, facilitando la programación funcional en un lenguaje que anteriormente era puramente orientado a objetos. Entre las innovaciones más importantes se encuentra la posibilidad de pasar funciones como parámetros a otros métodos.

Esta es una característica que antes no existía en Java, y su introducción ha permitido a los desarrolladores escribir código más flexible y reutilizable. Es particularmente útil para operaciones que se benefician de la personalización, como el ordenamiento y la filtración de colecciones.

Aquí hay un ejemplo de cómo se puede utilizar la interfaz funcional Comparator junto con la clase Arrays para realizar distintos ordenamientos a un objeto personalizado Persona:

```
public class Persona {
    private String nombre;
    private Integer edad;
    //Constructor por defecto
    //Constructor por parámetros
    //Getters and Setters
    //toString()
    //hashCode()
    //equals()
}
```

```
import java.util.Arrays;
import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        Persona[] personas = {new Persona("Agustin",40),
                              new Persona("Juan",30),
                              new Persona("Pedro", 20)};

        // Usamos una lambda para definir un Comparator personalizado
        Comparator<Persona> ordenarPorNombre = (persona1, persona2) ->
                        persona1.getNombre().compareTo(persona2.getNombre());
        Comparator<Persona> ordenarPorEdad = (persona1, persona2) ->
                            persona1.getEdad().compareTo(persona2.getEdad());
```

```
        // Pasamos la función que ordena por nombre como parámetro al método
sort de Arrays
        Arrays.sort(personas, ordenarPorNombre);
        // Imprimimos el array ordenado
        System.out.println("\nOrdenado por nombre");
        imprimirPersonas(personas);

        // Pasamos la función que ordena por edad como parámetro al método
sort de Arrays
        Arrays.sort(personas, ordenarPorEdad);
        // Imprimimos el array ordenado
        System.out.println("\nOrdenado por edad");
        imprimirPersonas(personas);


    }

    public static void imprimirPersonas(Persona[] personas) {
        for(Persona persona : personas) {
            System.out.println(persona.toString());
        }
    }
}
```

```
Consola:
Ordenado por nombre
Persona [nombre=Agustin, edad=40]
Persona [nombre=Juan, edad=30]
Persona [nombre=Pedro, edad=20]

Ordenado por edad
Persona [nombre=Pedro, edad=20]
Persona [nombre=Juan, edad=30]
Persona [nombre=Agustin, edad=40]
```

Mira el siguiente video para entender cómo podemos usar la interfaz comparator:

👉 **Interfaces | Comparator | JAVA | Egg**

Interfaces y Principios SOLID

En lo que se refiere a las interfaces, los dos últimos principios son los que nos interesan: El principio de Segregación de Interfaces y el principio de Inversión de Dependencias.

Principio de Segregación de Interfaces (ISP):

Este principio establece que los clientes no deben ser forzados a depender de interfaces que no utilizan. En otras palabras, es mejor tener muchas interfaces específicas que una sola interfaz genérica.

- **No aplicado:** Supongamos que tenemos la interfaz AccionesPerro

```
public interface AccionesPerro {
    void comer();
    void dormir();
    void defecar();
    void morderseLaCola();
}
```

  Como verás esta interfaz genera el problema de que hay perros que no se pueden morder la cola, por lo tanto hay clases de perros que tendrían este método implementado pero que no lo utilizan, no se estaría cumpliendo el principio de ISP.

- **Bien aplicado:** Para aplicarlo correctamente podemos tomar el ejemplo anterior y construir una interfaz para cada método, y ahora se pueden usar incluso para otros animales que no sean perros pero tengan ese comportamiento:

```
public interface Comedor {
    void comer();
}
public interface Dormidor {
    void dormir();
}
public interface Defecador {
    void defecar();
}
```

```
public interface MordedorDeCola {
    void morderseLaCola();
}
```

Principio de Inversión de Dependencias (DIP):

Este principio establece que los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones. Además, las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones. En términos de interfaces, esto significa que las clases deben depender de interfaces, no de implementaciones concretas.

- **No aplicado:** Supongamos que tenemos una clase Computadora que depende de una clase ProcesadorIntel:

```
public class ProcesadorIntel {
    void ejecutar() {
        // implementacion
     }
}

public class Computadora {
    private ProcesadorIntel procesadorIntel;

    public Computadora(ProcesadorIntel procesadorIntel) {
        this.procesadorIntel = procesadorIntel;
    }

    void run() {
        procesadorIntel.ejecutar();
    }
}
```

> En esta situación, si tenemos una computadora con un procesador AMD no podríamos representarlo porque estamos dependiendo de una clase concreta en lugar de depender de una abstracción, no se estaría aplicando el principio DIP.

- **Bien aplicado:** Siguiendo con el ejemplo anterior para aplicar el principio DIP tendríamos que crear una interfaz Procesador y que nuestra clase Computadora dependa de ella, entonces luego podemos pasarle por parámetro al constructor cualquier clase concreta que implemente esa interfaz.

```java
public interface Procesador {
    void ejecutar();
}

public class ProcesadorIntel implements Procesador {
    @Override
    public void ejecutar() {
        // implementacion
    }
}
public class ProcesadorAMD implements Procesador {
    @Override
    public void ejecutar() {
        // implementacion
    }
}
```

```java
class Computadora {
    private Procesador procesador;

    public Computadora(Procesador procesador) {
        this.procesador = procesador;
    }

    public void run() {
        procesador.ejecutar();
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Procesador procesadorIntel = new ProcesadorIntel();
        Procesador procesadorAMD = new ProcesadorAMD();
        Computadora computadoraConIntel = new Computadora(procesadorIntel);
        Computadora computadoraConAMD = new Computadora(procesadorAMD);
```

```
    }
}
```

Interfaces y Patrones de Diseño

Hay varios patrones de diseño que utilizan interfaces, vamos a ver algunos de ellos:

Patrón Strategy

Este patrón se utiliza para crear una familia de algoritmos bajo un mismo tipo de interfaz, de esta manera se vuelven intercambiables entre sí. Por ejemplo, podemos tener diferentes algoritmos de ordenamiento:

```
public interface SortStrategy {
    void sort(int[] numbers);
}
```

```
public class QuickSortStrategy implements SortStrategy {
    public void sort(int[] numbers) {
        // Implementación del QuickSort
    }
}
```

```
public class BubbleSortStrategy implements SortStrategy {
    public void sort(int[] numbers) {
        // Implementación del BubbleSort
    }
}
```

Patrón Decorator

El patrón Decorator extiende el comportamiento de una clase. En este patrón, se crea una clase decoradora que envuelve la clase original y proporciona funcionalidades adicionales manteniendo intacta la firma de los métodos de la clase objetivo. Esto se hace a través de la herencia de interfaces y la agregación de la clase.

El patrón Decorator es especialmente útil en situaciones donde hay una necesidad de extender la funcionalidad de una clase, pero no es la mejor opción crear una subclase.

Imagina que tiene una interfaz y una clase que lo que hacen es enviar un mensaje

```java
public interface Notificador {
    void enviar(String mensaje);
}
```

```java
public class NotificadorSimple implements Notificador {
    public void enviar(String mensaje) {
        //implementación de enviar un mensaje
    }
}
```

Ahora quieres extender esta clase para que encripte el mensaje antes de enviarlo, pero te das cuenta de que es mejor usar interfaces y agregación en lugar de crear una subclase aumentando el acoplamiento del código, entonces decide aplicar el patrón decorador.

```java
public interface NotificadorEncriptado extends Notificador {
    String encriptar(String mensaje);
}
```

```java
public class NotificadorEncriptadorImpl implements NotificadorEncriptado {

    private Notificador notificador;

    @Override
    public void enviar(String mensaje) {
        notificador.enviar(encriptar(mensaje));
    }

    @Override
    public String encriptar(String mensaje) {
        // implementación de encriptacion del mensaje
        return mensaje;
    }
 }
```

💡*Este patrón también es conocido como Wrapper, y es el que utilizan los wrappers de Java, envuelven a un tipo de dato primitivo y le agregan métodos útiles para trabajar con ellos.*

Patrón Factory

El patrón factory ya lo hemos utilizado con la herencia de clases, en este caso debemos reemplazar la clase padre abstracta por una interfaz para aplicarlo:

```
public interface FabricaDePan {
    public Pan crearPan();
}

public class FabricaDeBaguette implements FabricaDePan {
    @Override
    public Pan crearPan() {
        return new Baguette();
    }
}

public class FabricaDeCiabatta implements FabricaDePan {
    @Override
    public Pan crearPan() {
        return new Ciabatta();
    }
}
```