Material descargable

Herencia

egg

¡Hola! 👋 Te damos la bienvenida, en esta parte teórica vamos a ver Herencia.

Hay ocasiones en las que varios conceptos de la vida real que queremos representar comparten características y comportamientos, pero también tienen sus propias particularidades.

Piensa que quieres diseñar un sistema para una escuela, y necesitas crear clases para 'Estudiante' y 'Profesor'. Ambas clases tienen atributos comunes como 'nombre', 'edad', 'dirección'. Además, cada una tiene atributos específicos, por ejemplo, 'grado' para Estudiantes y 'materia' para Profesores. ¿Hay alguna manera de evitar la redundancia de código y al mismo tiempo mantener cada clase con sus especificidades? ¡La respuesta está en la herencia de Java!

¡Que comience el viaje! 🚀

Herencia

La herencia en Java es un mecanismo en el cual un objeto adquiere todas las propiedades y comportamientos de la clase padre. Es una parte importante de POO (programación orientada a objetos).

Java soporta la herencia simple, lo que significa que una clase puede heredar sólo de una clase padre. Esto ayuda a evitar mucha confusión de múltiples herencias. Sin embargo, se puede implementar la herencia múltiple a través de interfaces que lo veremos en las siguientes clases.

Aplicando herencia

Para crear una clase padre o superclase en Java no necesitamos hacer nada, simplemente declaramos la clase como lo haríamos normalmente. La clase que debemos modificar es la clase hija o subclase.

El único detalle a tener en cuenta es que las propiedades en lugar de **private** deben ser declaradas como ***protected***. Esto permite que las propiedades solo puedan usarse en la clase padre y en las clases hijas.

```
public class Persona {
    protected String nombre;
    protected int edad;
}
```

Palabra clave extends

En las subclases tenemos que usar la palabra clave ***extends*** para que Java entienda que la clase debe heredar todos los atributos y métodos de otra clase.

```
public class Estudiante extends Persona {
    private String grado;
}
```

Ahora la clase Estudiante hereda las propiedades de nombre y edad.

Palabra clave super

La palabra clave ***super*** se utiliza para hacer referencia al constructor y métodos de la clase Padre de la misma manera que se usa ***this***.

```
// Superclase o clase padre
public class Persona {
    protected String nombre;
    protected int edad;

    public Persona() {
    }

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

```java
// Subclase o clase hija
public class Estudiante extends Persona {

    private String grado;

    public Estudiante() {
        super();
        this.edad = 14;
        this.nombre = "Elias";
        this.grado = "segundo";
    }

    public Estudiante(String nombre, int edad, String grado) {
        super(nombre, edad);
        this.grado = grado;
    }
}
```

En el ejemplo podemos usar las propiedades de la clase padre porque están declaradas con el modificador de acceso *protected*.

Clases selladas

Las clases selladas limitan la herencia al declarar, en la clase padre, qué subclases pueden "extenderla". Esto proporciona un control más granular que las clases finales (que no permiten ser heredadas), permitiendo la herencia, pero solo para una lista específica de clases.

Además, las clases que extienden una clase sellada deben ser declaradas como ***final***, ***sealed***, o ***non-sealed***.

- **final**: La clase no puede ser extendida.
- **sealed**: La clase puede ser extendida, pero debes especificar cuáles clases pueden hacerlo.
- **non-sealed**: La clase puede ser extendida por cualquier otra clase.

```java
public sealed class FiguraGeometrica
permits Triangulo, Circulo, Paralelogramo {
    // ...
}
```

```
public sealed class Triangulo extends FiguraGeometrica
permits Equilatero, Escaleno, Isosceles {
    // ...
}
```

```
public final class Circulo extends FiguraGeometrica {
    // ...
}
```

```
public non-sealed class Paralelogramo extends FiguraGeometrica {
    // ...
}
```

La clase Object

En Java, todos los objetos se heredan de una clase base común, la clase Object. Y lo hacen de manera implícita, es decir, no necesitas usar "*extends Object* " para que se produzca la herencia.

Esta clase tiene varios métodos que pueden ser sobrescritos en las clases derivadas para proporcionar un comportamiento específico. Estos métodos son equals(), hashCode(), toString(), clone(), finalize(), getClass(), notify(), notifyAll(), wait().

- **equals()**: Este método se utiliza para comparar dos objetos y determinar si son iguales. Por defecto, el método equals() compara la igualdad de referencia, pero es común que se sobreescriba en las clases personalizadas para realizar una comparación más significativa basada en los atributos de los objetos.
- **hashCode()**: Este método devuelve un código hash único para un objeto. Es utilizado en estructuras de datos como las tablas hash para indexar y buscar objetos de manera eficiente. Si se sobrescribe el método equals(), también es recomendable sobrescribir el método hashCode() para asegurar la coherencia entre ambos.
- **toString()**: Este método devuelve una representación en forma de cadena de texto del objeto. Es útil para imprimir información legible sobre el objeto, generalmente utilizada para fines de depuración o visualización.
- **clone()**: Este método se utiliza para crear y devolver una copia del objeto. Sin embargo, el método clone() realiza una copia superficial del objeto.

Esto significa que si el objeto contiene referencias a otros objetos, solo las referencias se copian, no los objetos a los que se refieren. Para realizar una copia completa de un objeto, incluyendo todos sus objetos internos, se debe implementar una un método de copia dentro de las clases de esos objetos internos. Este método ha caído en desuso y en muchos casos se considera una mala práctica debido a sus implicaciones con respecto a la correcta copia de objetos, en su lugar se recomienda usar métodos de copia propios.

- **finalize()**: Este método se llama antes de que el recolector de basura elimine un objeto, permitiendo una última oportunidad para limpiar recursos o realizar otras tareas de limpieza. Sin embargo, este método ha sido declarado obsoleto desde Java 9, ya que su ejecución no está garantizada por la JVM. El uso de finalize() puede llevar a comportamientos inesperados, retrasos en el rendimiento y problemas de memoria. En su lugar, se recomienda usar otras formas de gestión de recursos, como el bloque try-with-resources (que veremos en clases siguientes).
- **getClass()**: Este método se utiliza para obtener la clase de tiempo de ejecución del objeto. Este método no puede ser sobrescrito y es útil cuando necesitamos realizar operaciones de reflexión

  📍 *La reflexión es un tema avanzado y no será abordada durante el curso.*

- **notify(), notifyAll(), wait()**: Estos métodos se utilizan para la comunicación entre hilos en Java. Estos métodos eran fundamentales para la programación de sistemas multi-hilo, sin embargo, el manejo manual de la sincronización de hilos es complejo y propenso a errores. Actualmente, Java provee de clases más avanzadas para realizar este tipo de programación.

  📍 *La programación multihilo no será abordada durante el curso.*

Sobrescribir métodos

Los métodos de las superclases pueden ser sobrescritos por las subclases utilizando la anotación @Override, de esta manera se modifica el funcionamiento del método haciéndolo que se adapte a las particularidades de la clase hija

```java
public class Persona {
    public void hablar() {
        System.out.println("Soy una persona");
    }
}
```

```java
public class Estudiante extends Persona {
    @Override
    public void hablar() {
        System.out.println("Soy un estudiante");
    }
}
```

Si un objeto *Estudiante* invoca al método *hablar() se imprimirá en consola "Soy un estudiante." si no se sobreescribe el método hablar() se imprime "Soy una persona".*

La importancia de equals() hashCode() y toString()

Sobreescribir los métodos equals() y hashCode() es sumamente importante cuando creamos objetos personalizados, porque nos permiten hacer comparaciones por el valor de los atributos en lugar de hacerlo por la referencia en memoria del objeto.

Si tenemos dos variables que se refieren a dos instancias de una clase personalizada y estas variables contienen los mismos valores, al invocar el método equals() sin haberlo sobrescrito, obtendremos un resultado que indica que los objetos son diferentes. Esto se debe a que el método equals() predeterminado en Java compara las referencias de los objetos en la memoria, en lugar de comparar los valores de sus atributos internos.

Sobreescribir el método toString() nos permite devolver el valor de todos los atributos del objeto en formato String para poder luego imprimirlos en pantalla.

El método hashCode() en Java se utiliza para generar un valor entero que representa un objeto en memoria. Este valor se utiliza en estructuras de datos basadas en hash (las veremos en las siguientes clases), para determinar la ubicación en la estructura donde se debe almacenar y buscar el objeto. Cuando insertamos un objeto en una de estas estructuras, se llama al método hashCode() del objeto para determinar su ubicación de almacenamiento. Luego,

cuando quieres comprobar si existe el objeto en la estructura, se vuelve a llamar al método hashCode() para saber dónde encontrarlo rápidamente. Las estructuras basadas en hash aumentan el rendimiento para los procesos de almacenamiento y búsqueda, pero no para recuperación.

Mira el siguiente video para entender mejor cómo funcionan estos métodos y ver cómo implementarlos con vs code:

👉 **Herencia | Sobrescritura de métodos | JAVA | Egg**

Polimorfismo

El polimorfismo se refiere a la capacidad de un método u objeto de tomar múltiples formas. En términos prácticos significa que una interfaz o superclase puede tener múltiples implementaciones.

- **Polimorfismo de métodos:** Esto ocurre cuando hacemos sobrecarga y sobrescritura de los métodos de una clase, porque con un mismo nombre el método tiene múltiples implementaciones.
- **Polimorfismo de objetos:** Esto ocurre cuando una superclase se utiliza para referirse a un objeto de una subclase (Lo veremos con el uso de *instanceof*) o cuando una clase implementa interfaces.

Clases Abstractas

Son clases que usan la palabra clave *abstract*. Las mismas pueden contener los métodos comunes, que ya has aprendido a realizar, y métodos abstractos, que son métodos declarados pero no implementados.

No se puede crear una instancia de una clase abstracta. Te preguntarás ¿Para qué se usa entonces?, bueno las clases abstractas se pueden heredar, por lo tanto podemos declarar métodos abstractos en una clase y luego heredarlos en otra para ser implementados.

```
public abstract class Animal {
    public abstract void hacerSonido();
}
```

```
public class Perro extends Animal {
```

```
    @Override
    public void hacerSonido() {
        System.out.println("Guau Guau");
    }
}
```

Como puedes observar los métodos abstractos no tienen “cuerpo”, se declaran sin las llaves, y luego en las subclases es necesario usar la anotación @Override para implementar el método de la superclase.

Operador Instanceof

Este operador devuelve un booleano y se utiliza para probar si un objeto es una instancia de una clase, una subclase, o una clase que implementa una interfaz particular. Esto es útil para asegurarse de que tienes el tipo correcto antes de realizar una operación.

```
class Animal {}
class Perro extends Animal {}

public class Instanceof {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Animal perro = new Perro();

        System.out.println(perro instanceof Animal);  // Devuelve true
        System.out.println(perro instanceof Perro);  // Devuelve true
        System.out.println(animal instanceof Perro);  // Devuelve false
    }
}
```

Pattern matching

El Pattern Matching, o coincidencia de patrones, es un mecanismo común en la programación que permite verificar si un determinado valor (o valores) coincide con un patrón específico, y en función de eso, ejecutar cierto código.

Java ha estado incorporando el concepto de Pattern Matching en su lenguaje durante las últimas versiones, mejorando significativamente la simplicidad y la seguridad de ciertos patrones de programación comunes.

- **Pattern Matching para instanceof (Introducido en Java 16)**: Esto reduce la necesidad de una conversión explícita después de una operación exitosa de instanceof.

```java
// Antes de que exista el Pattern Matching con instanceof
public class Main {
    public static void main(String[] args) {
        Object obj = "Hola mundo";
        if (obj instanceof String) {
            // Tenías que hacer un casteo y declarar una nueva variable
            String s = (String) obj;
            System.out.println(s.toLowerCase());
        }
    }
}
```

```java
// Ahora
public class Main {
    public static void main(String[] args) {
        Object obj = "Hola mundo";
        if (obj instanceof String s) {
            // Puedes declarar la variable directamente al usar instanceof.
            System.out.println(s.toLowerCase());
        }
    }
}
```

- **Pattern Matching en switch expression (En revisión en Java 20)**: Proporciona mayor flexibilidad y poder expresivo en las declaraciones switch al permitir patrones complejos con case.

```java
// Antes cuando solo existía el Pattern Matching con instanceof
class Animal {}
class Perro extends Animal {}

public class Main {
    public static void main(String[] args) {
        Object[] objects = { "Hola", 10, 20.0, true, new Animal(), new
Perro() };
```

```
        for (Object obj : objects) {
            if (obj instanceof String s) {
                System.out.println("String: " + s);
            } else if (obj instanceof Integer i) {
                System.out.println("Integer: " + i);
            } else if (obj instanceof Double d) {
                System.out.println("Double: " + d);
            } else if (obj instanceof Boolean b) {
                System.out.println("Boolean: " + b);
            } else if (obj instanceof Perro p) {
                System.out.println("Es un perro");
            } else if (obj instanceof Animal a) {
                System.out.println("Es un animal");
            } else {
                System.out.println("Tipo desconocido");
            }
        }
    }
}
```

```
// Ahora con el Pattern Matching en switch expression
class Animal {}
class Perro extends Animal {}

public class Main {
    public static void main(String[] args) {
        Object[] objects = { "Hola", 10, 20.0, true, new Animal(), new
Perro() };

        for (Object obj : objects) {
            switch (obj) {
                case String s -> System.out.println("String: " + s);
                case Integer i -> System.out.println("Integer: " + i);
                case Double d -> System.out.println("Double: " + d);
                case Boolean b -> System.out.println("Boolean: " + b);
                case Perro p -> System.out.println("Es un perro");
                case Animal a -> System.out.println("Es un animal");
                default -> System.out.println("Tipo desconocido");
            }
        }
    }
```

```
}
```

> 💡 Actualmente en Java 20 salió la 4ta revisión y en Java 21 (que saldrá en septiembre), se introducirá en el lenguaje definitivamente.

Principios SOLID y herencia

Recordemos los principios SOLID:

- Single Responsibility Principle (SRP): Un módulo de código (clase, método, etc.) debería tener responsabilidad sobre una única parte de la funcionalidad proporcionada por el software.
- Open-Closed Principle (OCP): Los módulos de código deberían estar abiertos para extensión pero cerrados para modificación.
- Liskov Substitution Principle (LSP): Los objetos de un programa deberían poder ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.
- Interface Segregation Principle (ISP): Este principio sugiere que es mejor tener muchas interfaces pequeñas y específicas, en lugar de una grande y general.
- Dependency Inversion Principle (DIP): Este principio sugiere que los módulos de alto nivel no deberían depender de los módulos de bajo nivel. Ambos deberían depender de abstracciones.

Ya vimos cómo aplicar el primer principio con las clases personalizadas del modelo, las clases servicios y con la creación de métodos que realicen una única tarea. Ahora veamos qué principios se pueden aplicar con herencia.

Open-Closed Principle (OCP)

Este principio sugiere que deberíamos ser capaces de extender el comportamiento de un módulo, sin modificar el módulo en sí. La herencia es una de las formas en las que podemos lograr esto en un lenguaje orientado a objetos como Java. Podemos crear una nueva clase que hereda de la clase original y añadir o sobrescribir métodos para cambiar o extender su comportamiento.

¿Pero cómo me doy cuenta de que lo estoy usando bien?:

- **Bien aplicado:** Por ejemplo, estarías aplicando correctamente el principio OCP sí, para añadir un nuevo comportamiento a la clase FiguraGeometricaServicio, no necesitas modificar la clase en sí. En lugar de eso, creas subclases que extienden FiguraGeometricaServicio y se encargan de sobreescribir y agregar ese nuevo comportamiento. De este modo, la clase original permanece cerrada a las modificaciones, pero abierta a la extensión.

```
public abstract class FiguraGeometricaServicio {
    protected String nombreFiguraGeometrica;

    public abstract void dibujar();
}
```

```
class CirculoServicio extends FiguraGeometricaServicio {

    @Override
    public void dibujar() {
        System.out.println(this.nombreFiguraGeometrica+":");
        // dibujar circulo
    }
}
```

```
class RectanguloServicio extends FiguraGeometricaServicio {

    @Override
    public void dibujar() {
        System.out.println(this.nombreFiguraGeometrica+":");
        // dibujar rectángulo
    }
}
```

- **No aplicado:** No estarías aplicando correctamente el principio OCP sí, para añadir una nueva figura, tienes que modificar la clase FiguraGeometricaServicio cada vez. Este escenario puede suceder si la clase FiguraGeometricaServicio es la encargada de implementar directamente el método dibujar(). En lugar de estar abierta a la extensión, la clase estaría abierta a la modificación, lo cual va en contra del principio OCP.

```
public class FiguraGeometricaServicio {
    protected String nombreFiguraGeometrica;

    public void dibujar() {
        if ("círculo".equals(nombreFiguraGeometrica)) {
            // dibujar circulo
        } else if ("rectángulo".equals(nombreFiguraGeometrica)) {
            // dibujar rectangulo;
        }
        // Si quieres agregar una nueva forma, tienes que modificar
este método.
    }
}
```

Liskov Substitution Principle (LSP)

Este principio se aplica directamente a la herencia y sugiere que si una clase B es una subclase de una clase A, entonces deberíamos ser capaces de usar B en cualquier lugar donde se espera A, sin que el programa haga algo “extraño” o “incorrecto”. Esto significa que las subclases deben ser completamente intercambiables con sus clases base sin alterar el funcionamiento del programa.

- **Bien aplicado:** Por ejemplo, estaríamos aplicando correctamente LSP si tenemos una estructura de clases en la que cada subclase Hornero y Gallina heredan de la clase abstracta Ave. Cada una de estas subclases proporciona su propia implementación del método duermeEn(), lo que está perfectamente bien. Además, notarás que solo la clase Hornero tiene el método volar(). Luego en el “Main” se cumple el principio LSP de que se pueda usar cualquier instancia de una subclase donde se espera un objeto de tipo Ave.

```
public abstract class Ave {
    public String duermeEn() {
        return "Duerme en ";
    };
}
```

```
class Hornero extends Ave {
    @Override
    public String duermeEn() {
```

```
        return super.duermeEn()+"un nido de barro";
    }

    public void volar() {
        // ...
    }
}
```

```
class Gallina extends Ave {
     @Override
    public String duermeEn() {
       return super.duermeEn()+"un corral";
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Ave hornero = new Hornero();
        Bird gallina = new Gallina();
        metodo(hornero);
        metodo(gallina);
    }

    public static void metodo(Ave ave) {
        System.out.println(ave.duermeEn());
        if (ave instanceof Hornero h) {
            h.volar();
        }
    }
}
```

- **No aplicado**: No lo estaríamos aplicando si, siguiendo con el ejemplo anterior, el método volar() lo añadimos a la clase Ave, lo que sugiere que todas las aves pueden volar. Sin embargo, sabemos que esto no es cierto en el mundo real. Esto significaría que cuando pases una Gallina al método del "Main", se intentará invocar el método volar(), que no es correcto ya que una Gallina no puede volar. Por lo tanto, esto viola el

principio LSP, ya que estamos utilizando una subclase Gallina donde se espera una Ave, y el comportamiento del programa no es el esperado.

```
public abstract class Ave {
    public String duermeEn() {
        return "Duerme en ";
    };

    public void volar() {
        // ...
    }
}
```

```
class Hornero extends Ave {
    @Override
    public String duermeEn() {
        return super.duermeEn()+"un nido de barro";
    }
}
```

```
class Gallina extends Ave {
     @Override
    public String duermeEn() {
       return super.duermeEn()+"un corral";
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Ave hornero = new Hornero();
        Bird gallina = new Gallina();
        metodo(hornero);
        metodo(gallina);
    }

    public static void metodo(Ave ave) {
        System.out.println(ave.duermeEn());
        ave.volar();
    }
}
```

Patrones de Diseño y Herencia

Uno de los patrones de diseño que se beneficia de la herencia es el "Factory Method Pattern" o "Patrón de Método de Fábrica". Este patrón se considera una extensión del "Simple Factory Pattern" o "Patrón de Fábrica Simple" (donde no se cumple el principio de OCP).

Patrón Simple Factory

Este es un patrón de diseño que permite la creación de objetos sin exponer la lógica de creación al cliente. En su lugar, se utiliza un método común para crear los objetos, lo que permite un acoplamiento más suelto y una mayor flexibilidad.

Vamos a considerar un ejemplo simple en Java, donde tenemos diferentes tipos de pan y una fábrica de pan para crearlos.

```
public class PatronFactorySimple {}

public abstract class Pan {
    // atributos y métodos comunes a todos los panes
}

public class Baguette extends Pan {
    public Baguette() {// inicializar la Baguette}
}

public class Ciabatta extends Pan {
    public Ciabatta() {// inicializar la Ciabatta}
}

public class FabricaDePan {
    public static Pan crearPan(String tipo) {
        if (tipo.equals("Baguette")) {
            return new Baguette();
        } else if (tipo.equals("Ciabatta")) {
            return new Ciabatta();
        }
        return null;
    }
}
```

Patrón de Método de Fábrica

Al igual que Simple Factory, el Factory Method encapsula la creación de objetos. Sin embargo, en lugar de pedir a una fábrica que haga el trabajo, el objeto es delegado a las subclases del Factory Method para crearlo (aplicando el principio de OCP).

Siguiendo con nuestro ejemplo de pan, vamos a extenderlo usando el patrón de Método de Fábrica.

```
public class PatronFactorySimple {}

public abstract class Pan {
    // atributos y métodos comunes a todos los panes
}

public class Baguette extends Pan {
    public Baguette() {// inicializar la Baguette}
}

public class Ciabatta extends Pan {
    public Ciabatta() {// inicializar la Ciabatta}
}

public abstract class FabricaDePan {
    public abstract Pan crearPan();
}

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

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

```
    }
}
```

Aquí, FabricaDePan es la fábrica abstracta que define un método para crear un objeto. FabricaDeBaguette y FabricaDeCiabatta son las subclases concretas que implementan el método de fábrica para producir objetos Baguette y Ciabatta, respectivamente.