UTEC
Universidad Tecnológica

Relaciones

1. Introducción

En el presente punto, exploraremos detalladamente la metodología para llevar a cabo la implementación de relaciones entre dos clases en el contexto de programación orientada a objetos. Examinaremos estrategias y técnicas disponibles para establecer conexiones significativas y eficientes entre instancias de clases en un sistema. Además, se presentarán ejemplos concretos y casos de uso, lo que permitirá una comprensión más profunda y contextualizada de cómo estas relaciones pueden ser aplicadas de manera efectiva en proyectos de desarrollo de software.

2. Relación de Asociación

Existe una relación de asociación entre dos clases cuando estaban conectadas de alguna forma. Si decimos que una Persona tiene al menos un Auto. Existe una relación entre la clase Persona y la clase Auto. También podríamos decir que una Persona puede tener varios autos, en este caso también hay una relación entre las clases.

Por otro lado, por ejemplo, si imaginamos un sistema de gestión de una concesionaria de automóviles, donde se desea modelar la relación entre las clases Cliente y Vehículo. En este contexto, la asociación entre estas dos clases se manifiesta de manera clara y significativa.

La clase Cliente representa a los individuos que visitan la concesionaria en busca de adquirir uno o varios vehículos. Estableceremos una relación de asociación entre Cliente y Vehículo para reflejar la conexión entre un cliente específico y los automóviles que ha adquirido o está interesado en comprar.

UTEC
Universidad Tecnológica

En primer lugar, consideremos el escenario donde un cliente puede tener al menos un automóvil. Esta relación se materializa cuando un cliente realiza una compra y adquiere un vehículo de la concesionaria. Aquí, la clase Cliente se asocia con la clase Vehículo, indicando que un cliente posee al menos un automóvil.

Sin embargo, también es importante contemplar la posibilidad de que un cliente pueda tener varios automóviles. En este caso, la relación de asociación entre las clases se expande para permitir que un cliente, en diferentes momentos o situaciones, adquiera múltiples vehículos. Esto podría deberse a diversas razones, como la compra de autos para diferentes miembros de la familia o la adquisición de vehículos con propósitos específicos, como trabajo y ocio.

En consecuencia, de lo anterior, la relación de asociación entre las clases Cliente" y Vehículo se convierte en un componente esencial para modelar eficientemente la interacción dinámica entre los individuos y los vehículos en el contexto de la concesionaria.

Con este ejemplo se demuestra cómo la flexibilidad y la comprensión de los distintos tipos de asociaciones pueden enriquecer la representación de relaciones entre clases en un sistema de software orientado a objetos.

UTEC
Universidad Tecnológica

2.1 Implementación de una Relación: una Persona puede tener un Auto

En esta primera parte, veremos cómo implementar una relación de asociación. Como ejemplos base, tomaremos la clase “Persona” y la clase “Auto”:

```
public class Auto {
    public String marca;
    public String modelo;
    public String matricula;
    public int anio;
    public Auto(String marca, String modelo, String matricula, int anio){
        this.marca = marca;
        this.modelo = modelo;
        this.matricula = matricula;
        this.anio = anio;
    }
}

public class Persona {

    public String nombre;
    public String ci;
    public boolean mayorEdad;

    public Persona(String nombre, String ci, boolean mayorEdad){
        this.nombre = nombre;
        this.ci = ci;
        this.mayorEdad = mayorEdad;
    }
}
```

Supongamos, que queremos modelar, que la Persona puede tener hasta un Auto.

Pregunta: *¿Cómo podemos hacer esto?*

Respuesta: Agregando en la clase Persona un atributo de tipo de dato la clase Auto.

```java
public class Persona {

    public String nombre;
    public String ci;
    public boolean mayorEdad;
    public Auto auto;

    public Persona(String nombre, String ci, boolean mayorEdad, Auto auto){
        this.nombre = nombre;
        this.ci = ci;
        this.mayorEdad = mayorEdad;
        this.auto = auto;
    }
}
```

Esto significa que cuando cree una instancia de una persona, debo pasarle al constructor, una instancia previamente creada de la clase auto. De esta forma relacionamos una instancia de Auto con una instancia de Persona.

Pregunta: *¿Cómo podríamos con esta misma clase, representar que una persona, no tiene auto?*

Respuesta: Como vimos, los tipos de datos tienen un valor por defecto, por ejemplo, los tipos de datos numéricos este valor es 0, en String es vacío (“”). Y en los tipos de datos complejos (como son las clases) el valor por defecto es null. Esto quiere decir que la variable de tipo de dato complejo no está cargada, no tiene una instancia creada. Entonces si al crear el objeto persona, le decimos que el auto es null estaríamos representando que la persona no tiene auto, pues no existe una instancia creada del auto.

Universidad Tecnológica

Veamos algunos Ejemplos:

```java
package Relaciones;

public class Principal {
    public static void main(String[] args) {
        // --------- Ejemplo donde relacionamos un auto y una persona

        //Creamos una instancia de un auto
        Auto auto1 = new Auto( marca: "Chevrolet",  modelo: "Aveo G3",  matricula: "AAA1158",  anio: 2014);

        //Ahora creamos instancia de persona que se relaciona con el auto1
        Persona persona1 = new Persona( nombre: "Guillermo",  ci: "1.123.456-7",  mayorEdad: true, auto1);

        //Obtenemos el auto relacionado con la persona
        Auto autoDePersona1 = persona1.auto;

        // ---------- Ejemplo donde tenemos una persona que no tiene auto relacionado
        Persona persona2 = new Persona( nombre: "Martín",  ci: "4.232.879-9",  mayorEdad: false,  auto: null);
        Auto autoPersona2 = persona2.auto;

        //¿qué valor tiene autoPersona2?

        if (autoPersona2 == null){
            System.out.println("Persona2 no tiene auto!");
        } else {
            System.out.println("Persona2 tiene auto!!");
        }


    }
}
```

En el ejemplo, creamos una instancia de Auto llamada auto1. Luego creamos una instancia de Persona llamada persona1 y la vinculamos con el auto creado.

Luego, definimos otra instancia de Persona llamada persona2 a la cual no le relacionamos ninguna instancia de la clase Auto (utilizando null). En este caso modelamos que la persona2 no tiene auto y la persona1 tiene el auto1.

Cuando estamos relacionando instancias de esta manera, decimos que “persona1 tiene una referencia a auto1”. Esto implica que, si se modifica algún atributo de la instancia auto1, lo modificamos para todas las referencias que tenga.

Veamos el siguiente Ejemplo:

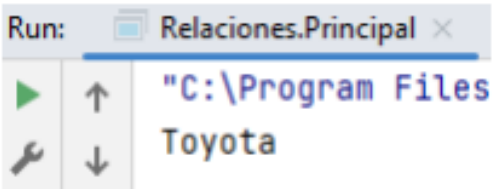
```
public class Principal {
    public static void main(String[] args) {
        // --------- Ejemplo donde relacionamos un auto y una persona

        //Creamos una instancia de un auto
        Auto auto1 = new Auto( marca: "Chevrolet",  modelo: "Aveo G3",  matricula: "AAA1158",  anio: 2014);

        //Ahora creamos instancia de persona que se relaciona con el auto1
        Persona persona1 = new Persona( nombre: "Guillermo",  ci: "1.123.456-7",  mayorEdad: true, auto1);

        //Obtenemos el auto relacionado con la persona
        Auto autoDePersona1 = persona1.auto;

        autoDePersona1.marca = "Toyota";

        System.out.println(persona1.auto.marca);
    }
}
```

En este caso, luego de crear las instancias y relacionarlas, obtuvimos la instancia de auto relacionada con la persona1 y la asignamos a la variable autoDePersona1.

Lo importante a destacar, es que, al tratarse de objetos, tanto la variable autoDePersona1 y el atributo auto de la instancia de la persona persona1 apuntan al mismo lugar de memoria donde está almacenada la instancia de auto.

Esto quiere decir que, si hago una modificación en la variable autoPersona1 en alguno de sus atributos, se modificará también para el objeto relacionado con persona1.

Por lo tanto, *¿Qué imprime el print del programa de ejemplo?*



En este ejemplo que hemos visto, tenemos que, dado un auto, podemos relacionarlo con muchas personas, puesto que es la persona quien tiene la referencia al auto y el auto no tiene la referencia a la persona.

Si queremos decir que un auto es de una persona, deberíamos agregarle a los atributos del auto, uno que sea de tipo Persona.

Hagamos el cambio en la clase Auto para contemplar esto:

```
public class Auto {

    public String marca;
    public String modelo;
    public String matricula;
    public int anio;
    public Persona propietario;

    public Auto(String marca, String modelo, String matricula, int anio, Persona propietario){
        this.marca = marca;
        this.modelo = modelo;
        this.matricula = matricula;
        this.anio = anio;
        this.propietario = propietario;
    }
}
```

En este caso hemos agregado un atributo de tipo Persona que se llama propietario. La clase Persona, queda intacta. Y de esta manera, ambas clases tienen una referencia a la otra. Veamos un ejemplo de generación de instancias con este nuevo cambio:

```
public class Principal {
    public static void main(String[] args) {
        // --------- Ejemplo donde relacionamos un auto y una persona

        //Creamos una instancia de un auto
        Auto auto1 = new Auto( marca: "Chevrolet",  modelo: "Aveo G3",  matricula: "AAA1158",  anio: 2014,  propietario: null);

        //Ahora creamos instancia de persona que se relaciona con el auto1
        Persona persona1 = new Persona( nombre: "Guillermo",  ci: "1.123.456-7",  mayorEdad: true, auto1);

        auto1.propietario = persona1;
    }
}
```

Como podemos ver, primero instanciamos un objeto de tipo Auto llamado auto1, al cual no le asignamos ningún propietario (porque le pasamos en el constructor, la instancia de persona en null). Esto lo hacemos, pues aún no tenemos la instancia de persona creada para relacionarla.

Luego creamos la instancia de persona que queremos. A este si, ya le relacionamos la instancia de auto que hemos creado previamente.

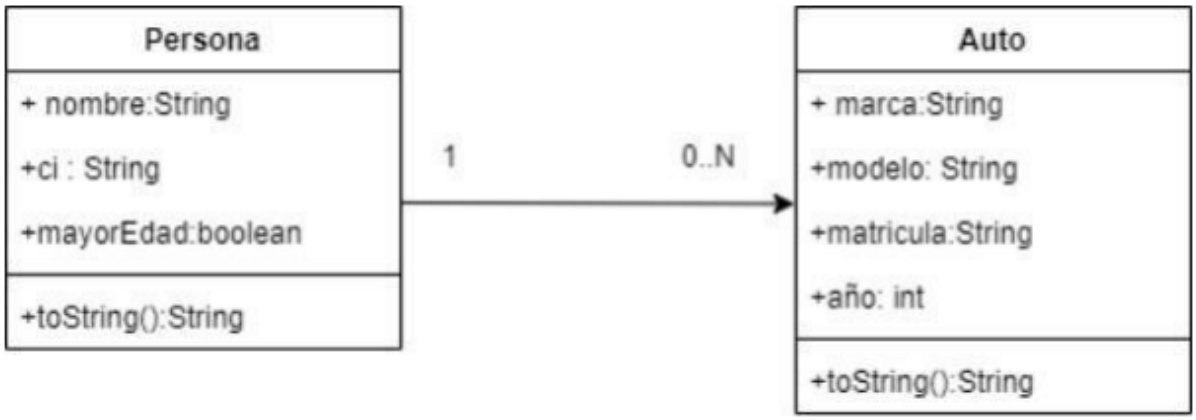
Luego, a la instancia de auto en el atributo propietario, le asignamos la persona que hemos creado. De esta manera, quedan referenciadas las instancias mutuamente.

2.2 Implementación de una relación: una Persona puede tener varios Autos

Para poder relacionar una Persona con varios Autos, tendremos que utilizar una estructura de datos que nos permita almacenar varios autos relacionados con la persona.

Por ejemplo:

La representación de la relación sería la siguiente:



UTEC
Universidad Tecnológica

La implementaremos de la siguiente manera:

```
import java.util.LinkedList;

public class Persona {

    public String nombre;
    public String ci;
    public boolean mayorEdad;
    LinkedList<Auto> autos = new LinkedList<>();

    public Persona(String nombre, String ci, boolean mayorEdad, LinkedList<Auto> autos){
        this.nombre = nombre;
        this.ci = ci;
        this.mayorEdad = mayorEdad;
        this.autos = autos;
    }
}
```

Definimos una lista de autos como atributo, lo recibimos en el constructor como parámetro y se lo asignamos al atributo.

En la clase Principal, en el método main, cuando se quiere instanciar a una Persona, por ejemplo, lo podríamos hacer de la siguiente manera:

```
import java.util.LinkedList;

public class Principal {
    public static void main(String[] args) {

        Auto auto1 = new Auto( marca: "Chevrolet",  modelo: "Aveo G3",  matricula: "AAA1158",  anio: 2014,  propietario: null);
        Auto auto2 = new Auto( marca: "Chevrolet",  modelo: "JOY plus",  matricula: "SSA2258",  anio: 2021,  propietario: null);
        Auto auto3 = new Auto( marca: "Chevrolet",  modelo: "JOY",  matricula: "SSA2299",  anio: 2021,  propietario: null);


        LinkedList<Auto> autosDeGuille = new LinkedList<>();
        autosDeGuille.add(auto1);
        autosDeGuille.add(auto2);
        autosDeGuille.add(auto3);


        // Ahora creamos la instancia de persona y le pasamos la lista de autos
        Persona persona1 = new Persona( nombre: "Guillermo",  ci: "1.123.456-7",  mayorEdad: true, autosDeGuille);
```

```
        //Si además, en cada auto quiero identificar a su propietario, lo indico
        auto1.propietario = persona1;
        auto2.propietario = persona1;
        auto3.propietario = persona1;

        System.out.println(persona1);
    }
}
```

Herencia

1. Introducción

En el presente punto, exploraremos los diversos enfoques y técnicas para llevar a cabo la implementación de herencia entre clases en el paradigma de programación orientada a objetos. Esto, nos permitirá no solo comprender los principios fundamentales inherentes a la herencia, sino también analizar casos prácticos y estrategias para su aplicación efectiva en el desarrollo de software. Examinaremos cómo esta técnica facilita la reutilización de código, la organización jerárquica de clases y la construcción de sistemas más flexibles y mantenibles.

Además, exploraremos los conceptos relacionados con la herencia, como las clases base y derivadas, los métodos y atributos heredados, y la capacidad de sobreescritura de métodos. Con ejemplos concretos, abordaremos las mejores prácticas para diseñar jerarquías de clases efectivas, garantizando una estructura coherente y adaptable en nuestros proyectos de desarrollo de software.

2. Concepto de Herencia

Es un mecanismo que permite crear una nueva clase a partir de una ya existente. La clase original se denomina *Clase Base* o *Superclase*, mientras que la nueva clase se denomina

Clase Derivada o *Subclase*. La clase derivada HEREDA todos los atributos y métodos que posee su clase base correspondiente. Además, incorpora nuevos, propios de ella.

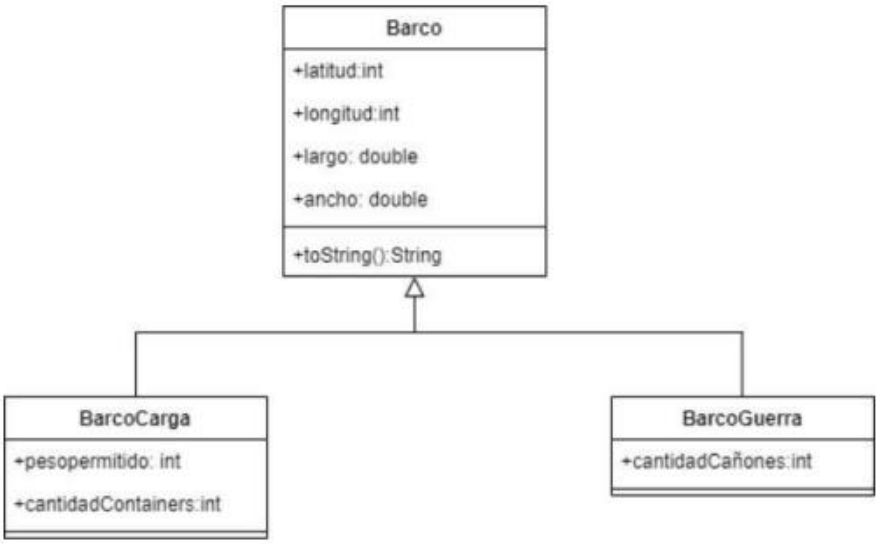
De esta manera en la clase derivada no se tiene que volver a programar los atributos y métodos heredados porque esta ya los posee mediante la herencia. Es por ello que la herencia es una forma de reutilización del código. Este aspecto es esencial en el contexto de la reutilización del código, ya que permite optimizar el desarrollo de software al aprovechar la estructura y funcionalidades previamente definidas en la clase base.

La herencia, por ende, se ve como un pilar clave para la construcción de jerarquías de clases coherentes y flexibles en sistemas orientados a objetos. Facilita la creación de modelos de datos y la organización modular del código, promoviendo la mantenibilidad y escalabilidad de los proyectos. No solo ofrece una manera eficaz de compartir y expandir funcionalidades, sino que también contribuye significativamente a la estructuración eficiente y sostenible de software en el desarrollo de aplicaciones complejas y robustas.

2.1 Implementación de Herencias

En esta sección veremos cómo implementar herencias en Java y como instanciamos las distintas clases, tanto padres como hijas. A la clase padre, o de la cual se hereda (en nuestro ejemplo la clase Barco), la denominaremos clase Base. Luego el resto, son herencias de esta.

La representación UML de la herencia, tiene el siguiente aspecto:



La implementaremos de la siguiente manera:

```
public class Barco {

    public int latitud;
    public int longitud;
    public double largo;
    public double ancho;

    public Barco(int latitud, int longitud, double largo, double ancho) {
        this.latitud = latitud;
        this.longitud = longitud;
        this.largo = largo;
        this.ancho = ancho;
    }
}
```

Esta sería la clase base o superclase, de la cual queremos que otras hereden atributos y métodos. Como vemos no tiene ninguna particularidad extra, es una clase java como las ya vistas.

Universidad Tecnológica

Veamos ahora, qué pasa si queremos tener la clase BarcoGuerra. Para que esta, sea hija de la clase Barco y heredar todos los atributos y métodos debemos crear un nuevo archivo, con el nombre de la nueva Clase (BarcoGuerra.java) que tenga el siguiente código:

```
public class BarcoGuerra extends Barco{

    public int cantidadCañones;

 public BarcoGuerra(int latitud, int longitud, double largo, double ancho, int cantidadCañones) {
        super(latitud, longitud, largo, ancho);
        this.cantidadCañones = cantidadCañones;
    }
}
```

Podemos ver que esta clase sí tiene una particularidad. Primero, agregamos extends Barco en la definición de la clase.

Esto, le indica al compilador que queremos que nuestra clase BarcoGuerra herede de la clase barco. Se utiliza “extends” porque a la herencia también puede llamársela extensión.

Luego en nuestro constructor, utilizamos super(…) para llamar al constructor de la clase padre, en este caso Barco.

Esto quiere decir que, cuando se llame al constructor de BarcoGuerra, este utilizará el constructor de la clase Barco para construir los atributos que corresponden “a la parte Barco” de mi clase: latitud, longitud, largo y ancho.

Luego debo asignar valor a los atributos particulares de BarcoGuerra por separado. Es decir, el método “super”, invoca al constructor de la clase padre o madre, o base o superclase, para cargar los atributos heredado y luego se deben asignar los valores a los atributos propios de la clase hija.

Como podemos ver, a las clases hijas no les agregamos los atributos de la clase padre, ya que como extiende una clase de la otra, se entiende que BarcoGuerra ya contiene los atributos y métodos de Barco implícitamente.

Veamos cómo sería para la clase BarcoCarga:

```
public class BarcoCarga extends Barco {

    public int pesoPermitido;
    public int cantidadContainers;

    public BarcoCarga(int latitud, int longitud, double largo, double ancho, int pesoPermitido, int cantidadContainers){
        super(latitud, longitud, largo, ancho);
        this.pesoPermitido = pesoPermitido;
        this.cantidadContainers = cantidadContainers;
    }
}
```

Como se puede ver, es muy similar en los dos casos.

Veamos ahora ejemplos de cómo poder instanciar cada uno de los objetos:

```
public class Principal {

    public static void main (String[] args) {

        // ----- Instanciamos un Barco

        Barco barco1 = new Barco( latitud: 10,  longitud: 15,  largo: 450,  ancho: 58);

        double largobarco1 = barco1.largo;

        // ----- Intanciamos un BarcoGuerra

        BarcoGuerra barcoGuerra1 = new BarcoGuerra( latitud: 10,  longitud: 15,  largo: 87.2,  ancho: 152.36,  cantidadCañones: 10);

        // obtenemos un atributo heredado
        double anchoBarcoGuerra = barcoGuerra1.ancho;

        // obtenemos un atributo propio
        int cañones = barcoGuerra1.cantidadCañones;
    }
}
```

Debemos tener en cuenta, que podemos instanciar un objeto de la clase Barco, que no tiene por qué ser ni BarcoGuerra ni BarcoCarga.

Si embargo, si instancio un BarcoGuerra o un BarcoCarga, siempre son, además “de ellos mismos” son implícitamente del tipo Barco.

2.2 InstanceOf

Java, nos brinda un método para obtener, dado un objeto instanciado, saber si es de determinada clase. Este se llama “instance of”: podemos preguntar si una variable es de cierto tipo de dato.

var1 instanceof Clase : devuelve true si var1 es de tipo de dato Clase, false en caso contrario.

Veamos un Ejemplo:

UTEC
Universidad Tecnológica

2.3 Beneficios de la Herencia

- Re-uso de clases y código.
- Código con menos defectos.
- Extensión de clases.
- Código genérico.

2.4 Desventajas de la Herencia

- Requiere conocimiento de la jerarquía (ancho y altura).
- Dificultad para mantener la jerarquía.
- Violación del encapsulamiento de los objetos.

Polimorfismo

1. Introducción

El polimorfismo, una característica esencial en la programación orientada a objetos, constituye un principio poderoso que eleva la flexibilidad y extensibilidad del código en Java.

Este concepto se manifiesta a través de la capacidad de las clases para adoptar múltiples formas y comportarse de manera variable según el contexto de ejecución. En este punto, exploraremos el polimorfismo en Java, desglosando sus dos manifestaciones principales: *el polimorfismo de compilación* y el *polimorfismo de ejecución*.

2. Concepto de Polimorfismo

El polimorfismo es un concepto fundamental en la programación orientada a objetos y es una de las características clave de Java. Se refiere a la capacidad de una clase para tomar varias formas o manifestarse de diferentes maneras. El polimorfismo en Java se puede lograr a través de dos mecanismos principales: *polimorfismo de compilación* (también conocido como sobrecarga de métodos) y *polimorfismo de ejecución* (también conocido como sobrescritura de métodos).

UTEC
Universidad Tecnológica

2.1 Polimorfismo de Compilación (Sobrecarga de Métodos)

La sobrecarga de métodos, una expresión sutil del polimorfismo en Java, amplía significativamente la versatilidad y la claridad del código al permitir que una clase ofrezca múltiples versiones de un método, todas compartiendo el mismo nombre pero diferenciándose por la cantidad o tipo de parámetros que aceptan.

Esta estrategia de diseño aporta una capa adicional de flexibilidad al lenguaje, ya que facilita la creación de interfaces más intuitivas y adaptables. Al invocar un método sobrecargado, el compilador desempeña un papel crucial al determinar cuál de las múltiples implementaciones utilizar, basándose en la información proporcionada por la llamada al método.

Esta resolución dinámica se lleva a cabo en tiempo de compilación, lo que significa que la decisión sobre qué versión del método ejecutar se toma antes de que el programa se ejecute. Este enfoque anticipado contribuye a una mayor eficiencia y rendimiento del programa, ya que se resuelven posibles ambigüedades y se optimiza la ejecución del código.

Así, la sobrecarga de métodos es una herramienta fundamental para estructurar clases de manera más expresiva y adaptable, facilitando una interacción más fluida y legible en el desarrollo de software en Java.

Veamos el siguiente Ejemplo:

```java
public class OperacionesMatematicas {

    public int sumar(int a, int b) { //Método llamado sumar.
        return a + b;
    }

    public double sumar(double a, double b) { //Método llamado sumar.
        return a + b;
    }

    //Ambos métodos se llaman igual pero tienen distintos parámetros.
}
```

El ejemplo anterior demuestra la sobrecarga de métodos al tener dos versiones del método sumar, cada una diseñada para manejar un tipo específico de datos (enteros y números de punto flotante). Al llamar al método sumar con diferentes tipos de argumentos, el compilador determinará automáticamente cuál versión del método debe invocarse, basándose en los tipos de datos proporcionados. Esto mejora la flexibilidad y la usabilidad de la clase, ya que puede adaptarse a diferentes tipos de datos sin necesidad de cambiar el nombre del método.

Ejemplo de uso:

```java
OperacionesMatematicas calculadora = new OperacionesMatematicas();

int resultadoEntero = calculadora.sumar(5, 10); // Llama al primer método
double resultadoDouble = calculadora.sumar(5.5, 10.5); // Llama al segundo método
```

Recordemos:

- La sobrecarga de métodos permite a una clase tener múltiples métodos con el mismo nombre, pero con diferentes parámetros.
- El compilador determina cuál método utilizar en función de la cantidad y el tipo de argumentos proporcionados al llamar al método.
- Aquí, el polimorfismo se resuelve en tiempo de compilación.

2.2 Polimorfismo de Ejecución (Sobrescritura de Métodos)

La sobrescritura de métodos, una faceta del polimorfismo en Java, desencadena una dinámica interacción entre superclases y subclases, permitiendo a estas últimas proporcionar implementaciones específicas de métodos que ya están definidos en sus superiores jerárquicos. Este proceso es esencial para la creación de una jerarquía de clases que, si bien comparte una interfaz común en términos de métodos, permite a las subclases adaptar y personalizar estas implementaciones según sus propias necesidades y contextos.

La anotación @Override emerge como un distintivo clave en este panorama, sirviendo para indicar de manera explícita que un método en la subclase está sobrescribiendo un método en la superclase. Este nivel de claridad no solo mejora la comprensión del código, sino que también proporciona un mecanismo de verificación en tiempo de compilación para garantizar la correcta implementación de la sobrescritura.

El polimorfismo de ejecución, intrínseco a este tipo de sobrescritura, revela su ser durante la ejecución del programa. A diferencia del polimorfismo de compilación, donde las decisiones se toman en tiempo de compilación, aquí, la determinación de qué método específico invocar se realiza en tiempo de ejecución. Esto significa que la selección dinámica de la implementación de un método particular se lleva a cabo cuando el programa está en marcha, dependiendo del tipo real del objeto en cuestión. Este enfoque dinámico no solo permite una flexibilidad considerable en el diseño de software, sino que también facilita la

creación de sistemas más adaptables y extensibles, donde las subclases pueden evolucionar y ajustarse a medida que los requisitos del programa cambian con el tiempo.

Veamos el siguiente Ejemplo:

```java
public class Animal {
    public void hacerSonido() {
        System.out.println("Sonido genérico de un animal");
    }
}

public class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("Guau guau");
    }
}
```

```java
public class Gato extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("Miau");
    }
}
```

En este ejemplo, tanto Perro como Gato son subclases de Animal, y cada una sobrescribe el método hacerSonido() de la clase base. Esto permite que se invoque el método específico de la instancia en tiempo de ejecución.

UTEC
Universidad Tecnológica

Ejemplo de uso:

```
Animal perro = new Perro();
Animal gato = new Gato();

perro.hacerSonido(); // Imprime "Guau guau"
gato.hacerSonido();  // Imprime "Miau"
```

En este caso, la variable perro es de tipo Animal, pero en tiempo de ejecución, el método hacerSonido() del objeto Perro es llamado debido al polimorfismo de ejecución. El mismo principio se aplica al objeto gato.

Recordemos:

- La sobrescritura de métodos permite a una subclase proporcionar una implementación específica de un método que ya está definido en su superclase.
- Se utiliza la anotación @Override para indicar que un método en la subclase está sobrescribiendo un método en la superclase.
- Este tipo de polimorfismo se resuelve en tiempo de ejecución.