UNIDAD DIDÁCTICA 7

UTILIZACIÓN AVANZADA DE CLASES

MÓDULO PROFESIONAL: PROGRAMACIÓN

CESUR
Tu Centro Oficial de FP

Índice

RESUMEN INTRODUCTORIO

En esta unidad se va a tratar, en primer lugar, el concepto de composición de clases. Posteriormente, se tratará el importante concepto de Herencia, en la que se verán los conceptos de superclases, clases padres, subclases y clases hijas. Más adelante se introduce el concepto de clases y métodos abstractos y clases, métodos y variables finales y estáticas. Se verá, a continuación, la sobreescritura de métodos, revisando el tratamiento de los constructores en la herencia.

Se finalizará la unidad tratando el uso de interfaces y como los implementan las clases para adquirir funcionalidad de varias fuentes diferentes lo que se conoce por herencia múltiple por interfaz.

Para cada uno de los apartados veremos ejemplos en el lenguaje de programación Java.

INTRODUCCIÓN

Un conjunto de objetos aislados tiene escasa capacidad para resolver un problema. En una aplicación real los objetos colaboran e intercambian información, existiendo distintos tipos de relaciones entre ellos. La relación de herencia es un mecanismo que permite sobrescribir o extender las funcionalidades de una clase ya existente. De este modo se favorece la reutilización del código, se evitan redundancias, ya que no se repite de forma innecesaria la escritura de código ya implementado y se modela de forma más fidedigna la realidad.

Conocer y manejar los diferentes aspectos y conceptos relacionados con la composición de clases, herencia, polimorfismo, superclases, subclases, clases abstractas, métodos finales, métodos estáticos, sobreescritura de métodos, constructores y herencia en la programación orientada a objetos es fundamental por varias razones. Estos conceptos son fundamentales en la programación orientada a objetos y forman la base de cómo se modelan y organizan los sistemas de software. Comprender estos conceptos es esencial para crear aplicaciones efectivas y eficientes en POO.

La herencia y la composición de clases permiten la reutilización de código, evitando la duplicación y promoviendo la eficiencia en el desarrollo. El polimorfismo facilita la creación de código genérico y flexible, permitiendo tratar objetos de diferentes clases de manera uniforme. Esto es útil en situaciones donde se necesita manejar una variedad de objetos de diferentes subclases.

El conocimiento de estos conceptos permite la creación de sistemas de software extensibles y fáciles de mantener. También facilita la creación de arquitecturas sólidas y eficientes, evita errores de diseño y ayuda a optimizar el rendimiento de una aplicación. En resumen, estos conceptos son esenciales para desarrolladores de software y forman la base de la programación orientada a objetos.

CASO INTRODUCTORIO

El equipo de trabajo que diriges en la empresa en la que trabajas, está desarrollando un proyecto en el que se han escrito todas las superclases necesarias. Se ha llegado al punto en el que hay que implementar subclases para no tener que reescribir código que ya se ha escrito gracias a la herencia se podrá diseñar un proyecto modular, eficiente y fácil de mantener que facilitará la división del trabajo en pequeños equipos de desarrollo que podrán trabajar en clases específicas de una jerarquía sin afectar a otras partes del sistema, siempre que se cumplan los contratos que se establecen con la herencia.

Al finalizar el estudio de la unidad, conocerás los conceptos de herencia y composición de clases, serás capaz de implementar dichos conceptos en un lenguaje de programación orientado a objetos, identificarás cuándo utilizar clases y métodos abstractos, así como clases, métodos y variables finales y estáticas, conocerás el concepto de método constructor y su aplicación en las relaciones entre clases, definirás y utilizarás variables y métodos estáticos que pertenecen a la clase en lugar de una instancia específicas, identificarás y utilizarás interfaces para definir contratos comunes que las clases deben implementar y utilizarás clases y métodos abstractos y conocerá como se declaran e implementan ambos.

1. COMPOSICIÓN DE CLASES

En lugar de reescribir el código que ya ha sido cuidadosamente desarrollado en las superclases, deseas utilizar un enfoque de composición de clases para lograr tus objetivos. La razón detrás de esta decisión es que la composición de clases te permite agregar nuevas características y funcionalidades al sistema sin modificar directamente las superclases existentes. En lugar de heredar de las superclases, crearás nuevas clases que se compondrán de objetos de las superclases y agregarás la lógica específica de tu negocio a estas nuevas subclases.

La composición es un tipo de relación **dependiente** en la que un objeto más complejo es conformado por objetos más pequeños. En esta situación, la frase "***Tiene un***", debe tener sentido:

El coche *tiene* **llantas**

El portátil *tiene* **un teclado**.

EJEMPLO PRÁCTICO

Se representa la relación: El portátil tiene un teclado. Donde teclado es un objeto instanciado de la clase Teclado (la clase Teclado debe estar declarada para que podamos usarla).

```
public class Portatil {

    private String fabricante;
    private String modelo;
    private String servicioTecnico;
    private Teclado teclado = new Teclado();

      public Portatil() {
            // Lo que haga el constructor...
      }

}
```

La composición generalmente se usa cuando se quieren tener las características de una **clase existente** dentro de otra **clase**. En Java la composición se consigue creando clases que contienen instancias de otras clases como miembros o atributos, que van a representar las partes que componen al objeto más grande.

Para hacer esto, se sigue el típico patrón de alojar objetos privados de clases existentes en su nueva clase.

En ocasiones, tiene sentido permitir que el usuario de la clase acceda a la composición de su clase, esto es, hacer públicos los miembros objeto. De esta forma el objeto compuesto podrá acceder a funcionalidades y propiedades de los objetos contenidos, lo cual va a facilitar la creación de estructuras más complejas y modulares. Los miembros objeto usan su control de accesos, entonces es seguro y cuando el usuario conoce que está formando un conjunto de piezas, hace que la interfaz sea más fácil de entender.

RECUERDA

Los objetos que componen a la clase contenedora deben existir desde el principio. (También pueden ser creados en el constructor, no sólo al momento de declarar las variables como se muestra en el ejemplo).

No hay momento en que la clase contenedora pueda existir sin alguno de sus objetos componentes. Por lo que la existencia de estos objetos no debe ser abiertamente manipulada desde el exterior de la clase.

ARTÍCULO DE INTERÉS

Para ampliar información sobre la composición y ver más ejemplos se recomienda realizar la lectura:

En la unidad 4 vimos un ejemplo sobre un Parking que estaba compuesto por plazas de aparcamiento como ejemplo de composición, antes de ver el siguiente ejemplo es interesante volver a recordarlo.

Este es el código de la clase Plaza Aparcamiento y Parking.

```
public class PlazaAparcamiento {

    private String numero;
    private double tamanho;

    public PlazaAparcamiento(String numero, double tamanho) {
        this.numero = numero;
        this.tamanho = tamanho;
    }

    public String getNumero() {
        return numero;
    }

    public void setNumero(String numero) {
        this.numero = numero;
    }

    public double getTamanho() {
        return tamanho;
    }

    public void setTamanho(double tamanho) {
        this.tamanho = tamanho;
    }

}


public class Parking {

    private int numPlazas;
    private PlazaAparcamiento[] plazas;

    public Parking(int numPlazas) {
        this.numPlazas = numPlazas;
        plazas = new PlazaAparcamiento[numPlazas];
    }

    public int getNumPlazas() {
        return plazas.length;
    }

    public void setNumPlazas(int numPlazas) {
        this.numPlazas=numPlazas;
    }

    public PlazaAparcamiento[] getPlazas() {
        return plazas;
    }

    public void setPlazas(PlazaAparcamiento[] plazas) {
        this.plazas = plazas;
    }

}
```

En la unidad anterior ya vimos las colecciones que proporciona Java y que nos facilitan en gran parte el proceso de utilización y manejo de ciertas estructuras de almacenamiento, ahora vamos a poner en práctica el concepto de composición de clases, pero utilizando una de estas colecciones, en concreto una lista de tipo ArrayList.

EJEMPLO PRÁCTICO

Desarrolla el código necesario, utilizando clases en Java para que diferentes usuarios puedan tener en préstamo uno o varios libros de una biblioteca.

- Si el libro ya está prestado entonces no se puede asignar o prestar a otro usuario, solo se pueden prestar libros que no están prestados.
- Hay que tener en cuenta que si un usuario quiere devolver un libro que no tiene prestado no podrá hacerlo, solo puede devolver libros que le han sido prestados.

Utiliza la **composición** de clases de forma que tengas:

- Una clase **Usuario** con dos atributos; un nombre y una lista de libros que se prestan al usuario.
- Una clase **Libro** con dos atributos; un título y un indicador de si está prestado o no.

Ambas clases deben tener constructor, getters y setters implementados, así como otros métodos necesarios que ayuden a resolver el problema.

La clase controladora será la clase **Biblioteca** que contendrá un método **main()** donde se realizarán las instanciaciones de objetos tanto de tipo Libro como de tipo Usuario y se llamarán a los métodos necesarios para asignar y devolver libros a cada usuario.

En todo momento será necesaria la emisión de mensajes por consola para informar al usuario de lo que está ocurriendo.

SOLUCIÓN

Clase Libros.java

```
package ud7.libros;

public class Libro {
	private String título;
	private boolean prestado;

	public Libro(String título) {
			this.título = título;
			prestado = false;
		}

	public String getTítulo() {
		return título;
	}

	public void setTítulo(String título) {
		this.título = título;
```

```
	}

	public boolean isPrestado() {
		return prestado;
	}

	public void setPrestado(boolean prestado) {
		this.prestado = prestado;
	}

	public void prestarLibro() {
		if (!prestado)prestado = true;
	}

	public void devolverLibro() {
		if (prestado)prestado = false;
	}
}
```

Clase Usuario.java

```
package ud7.libros;
import java.util.ArrayList;
public class Usuario {
	private String nombre;
	private ArrayList<Libro> librosPrestadosAUsuario = new
	  ArrayList<>();

	public Usuario(String nombre) {
		this.nombre = nombre;
	}

	public String getNombre() {
		return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public ArrayList<Libro> getLibrosPrestadosAUsuario() {
		return librosPrestadosAUsuario;
	}

public void setLibrosPrestadosAUsuario(ArrayList<Libro>
librosPrestadosAUsuario) {
		this.librosPrestadosAUsuario = librosPrestadosAUsuario;
	}

	public void asignarLibro(Libro libroAPrestar) {
		if (!libroAPrestar.isPrestado()) {
			/* si no está prestado se puede asignar y
			 cambiamos prestado a true*/
			 libroAPrestar.prestarLibro();
			 // añadimos el libro a la lista de libros
			 librosPrestadosAUsuario.add(libroAPrestar);
			 /* añadimos la lista de libros actualizada al
			  Usuario*/
			 setLibrosPrestadosAUsuario(librosPrestadosAUsuario);
```

```
                System.out.println("\nEl libro '"  +
            libroAPrestar.getTítulo() + "' se ha prestado a " +
                    this.getNombre());
            } else {
                /* el libro ya está prestado no se puede asignar a
                usuario*/
                System.out.println("\nEl libro '" +
                libroAPrestar.getTítulo() +
                                    "' no está disponible");
            }
      }

      public void devolverLibro(Libro libroAPrestar) {
            // si no está prestado no se puede devolver
            if (!libroAPrestar.isPrestado()) {
                System.out.println("\nEl libro '" +
                                    libroAPrestar.getTítulo()
                                    + "' no ha sido prestado ");
            } else {
                /*si prestado es true, se podría devolver pero
                  antes se comprueba si el libro ha sido prestado al
                  usuario que quiere devolverlo*/

      if(this.librosPrestadosAUsuario.contains(libroAPrestar)) {
                /*Si el libro se le ha prestado, cambiamos el
                valor de prestado de nuevo a false para que se pueda
                volver a prestar a otro usuario*/
                libroAPrestar.devolverLibro();
                /* eliminamos el libro de la lista de libros de
                dicho usuario*/
                librosPrestadosAUsuario.remove(libroAPrestar);

                System.out.println("\nEl libro '" +
                                    libroAPrestar.getTítulo() +
                  "' ha sido devuelto por " + this.getNombre());
                }else {
                      /*el usuario no puede devolver un libro que
                      no se le ha prestado*/
                      System.out.println("\n" + this.getNombre() +
                                  " no puede devolver el libro '" +
                              libroAPrestar.getTítulo() + "'" );
                }
            }
      }

      public void listarLibrosUsuario() {
            System.out.println("\n" + this.nombre);
            // se pinta un subrayado
            for (int i = 0; i < this.nombre.length(); i++) {
                System.out.print("-");
            }
            if (this.librosPrestadosAUsuario.isEmpty()) {
                System.out.println("\nNo tiene libros prestados");
            } else {// tiene libros prestados, se muestran
                  /*Se puede utilizar otro tipo de estructura
                    'for' */
                    for (Libro libro : librosPrestadosAUsuario) {
                     System.out.println(" \nTítulo: " +
                                                libro.getTítulo());
```

```
                }
            }
        }
}
```

Clase Biblioteca.java

```
package ud7.libros;
public class Biblioteca {
	public static void main(String[] args) {
		Libro libro1 = new Libro("El Principito");
		Libro libro2 = new Libro("Harry Potter");

		Usuario usuario1 = new Usuario("Ana");
		Usuario usuario2 = new Usuario("Carlos");

		usuario1.asignarLibro(libro1);

		usuario1.asignarLibro(libro2);

		System.out.println("\n\n--Libros prestados actualmente--" );
		usuario1.listarLibrosUsuario();
		usuario2.listarLibrosUsuario();
		System.out.println("\n" );

		usuario2.asignarLibro(libro1);
		usuario1.devolverLibro(libro2);
		usuario2.asignarLibro(libro2);
		usuario1.devolverLibro(libro2);
		usuario1.devolverLibro(libro1);
		usuario2.asignarLibro(libro1);

		System.out.println("\n\n--Libros prestados actualmente--" );
		usuario1.listarLibrosUsuario();
		usuario2.listarLibrosUsuario();
	}
}
```

La salida por consola será:

```
El libro 'El Principito' se ha prestado a Ana

El libro 'Harry Potter' se ha prestado a Ana


--Libros prestados actualmente--

Ana
---
Título: El Principito

Título: Harry Potter

Carlos
------
```

```
No tiene libros prestados



El libro 'El Principito' no está disponible

El libro 'Harry Potter' ha sido devuelto por Ana

El libro 'Harry Potter' se ha prestado a Carlos

Ana no puede devolver el libro 'Harry Potter'

El libro 'El Principito' ha sido devuelto por Ana

El libro 'El Principito' se ha prestado a Carlos


--Libros prestados actualmente--

Ana
---
No tiene libros prestados

Carlos
------
Título: Harry Potter

Título: El Principito
```

2. HERENCIA Y POLIMORFISMO

Siguiendo con el desarrollo del sistema de gestión empresarial en tu empresa, tu equipo ha alcanzado el punto en el que debe implementar subclases para agregar funcionalidades específicas al sistema. Para ello utilizarás la herencia y el polimorfismo como estrategia de diseño para aprovechar la funcionalidad común de las superclases y crear subclases especializadas que se adapten a las necesidades específicas de tu proyecto. Esta decisión está orientada a mantener la consistencia, eficiencia y facilidad de mantenimiento en el desarrollo del software.

2.1 Herencia

La herencia es uno de los 4 pilares de la programación orientada a objetos (POO) junto con la **Abstracción**, **Encapsulación** y **Polimorfismo**. *La herencia es un mecanismo que permite la **definición de una clase** a partir de la definición de **otra ya existente**. La herencia permite **compartir** automáticamente **métodos** y datos entre clases, subclases y objetos.*

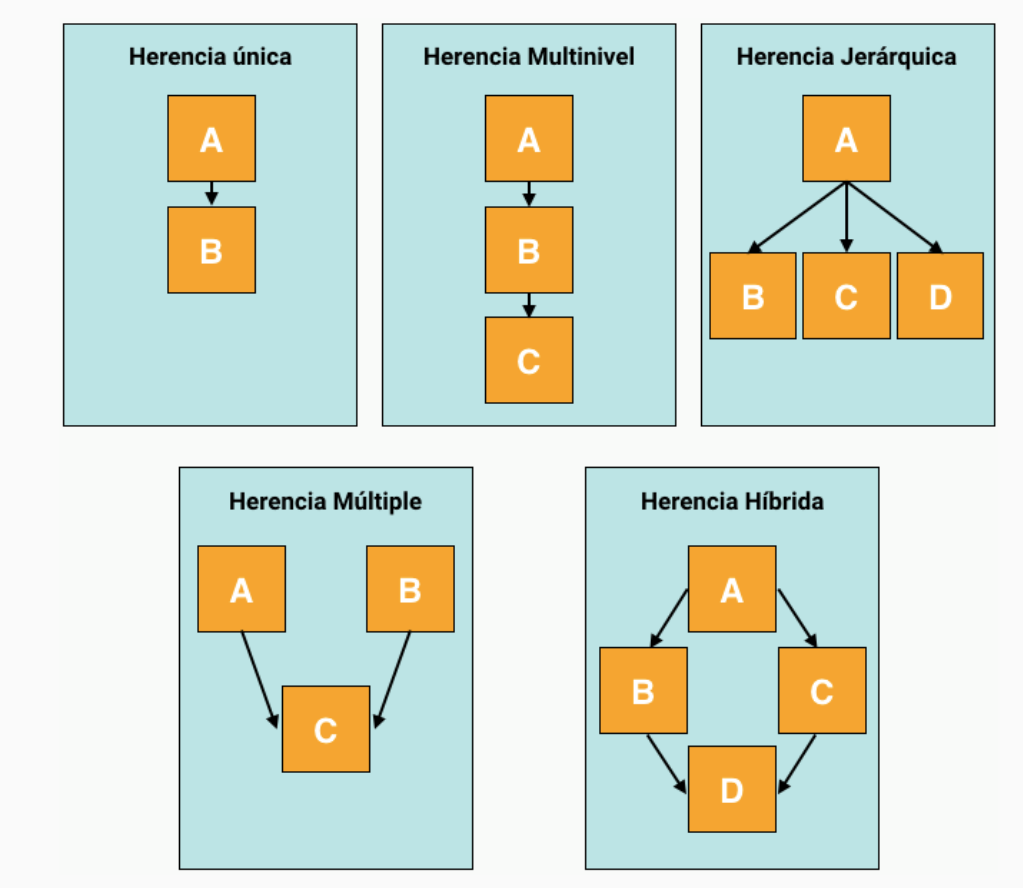
Java permite **heredar** a las clases **características** y **conductas** de una o varias clases denominadas base. Las clases que heredan de clases base se denominan **derivadas o clases hijas**, estas a su vez pueden ser clases bases para otras clases derivadas. Se establece así una **clasificación jerárquica**, similar a la existente en Biología con los animales y las plantas.

La herencia ofrece una **ventaja** importante, permite la **reutilización del código**. Una vez que una clase ha sido depurada y probada, el código fuente de dicha clase no necesita modificarse. Su funcionalidad se puede cambiar derivando una nueva clase que herede la funcionalidad de la clase base y le añada otros comportamientos. Reutilizando el código existente, el programador ahorra tiempo y dinero, ya que solamente tiene que verificar la nueva conducta que proporciona la clase derivada.

¿Qué ocurre en las subclases (Clases derivadas o hijas)?

- Heredan todas las características de la superclase (Clase padre).
- Podrán definir y añadir otras características adicionales.
- Pueden volver a redefinir las características que hereda.

En esta imagen se muestra los tipos de herencia que existen en programación, pero recuerda que en Java no se permite la herencia múltiple.



Tipos de herencia

Fuente: https://ifgeekthen.nttdata.com/es/herencia-en-programacion-orientada-objetos

ARTÍCULO DE INTERÉS

Conoce más información sobre la herencia y cómo implementarla en Java en este enlace:

VÍDEO DE INTERÉS

Un tutorial muy interesante de visionar sobre herencia en Java se puede encontrar en YouTube en la siguiente URL:

Se va a ver un ejemplo simple de la herencia. Para este caso se creará una clase que se heredará en la cual se encuentre el apellido, y otra clase donde se asignará el nombre de una persona.

La clase que se heredará (clase padre):

```
package ud7.herencia;
public class ClaseHeredada {
	String Apellido;

	public ClaseHeredada(String Dato) {
		this.Apellido = Dato;
	}
	public String getApellido() {
		return Apellido;
	}
}
```

Una vez que se tiene la clase que se va a heredar, se crea la clase en la que se encontrará el nombre de la persona y se le asigna el siguiente código:

```
package ud7.herencia;
public class Herencia extends ClaseHeredada {
	String Nombre;

	public Herencia(String Apellido, String nombre) {
		super(Apellido);
		Nombre = nombre;
	}
	public void setPersona(String NombrePer) {
		this.Nombre = NombrePer  ;
```

```
	}
	public String getPersona() {
		return Nombre;
	}
}
```

Se usa extends para indicar que se está heredando la clase “ClaseHeredada” en la que se encuentra el apellido. La palabra super se usa para indicar que estamos instanciando al constructor de la clase que se está heredando.

Una vez realizado este proceso, se instanciará un objeto de la clase “Herencia” en el método main de la siguiente forma:

```
package ud7.herencia;
public class Main {
	public static void main(String[] args) {
	   Herencia X=new Herencia(" Arias Figueroa" ,"Kevin
Arnold");
	   /*Se puede comprobar que el objeto X(clase hija) puede
	    acceder a métodos de la clase “ClaseHeredada” (clase
	    padre), por tanto, ‘apellido’ forma parte del objeto X
	    aunque internamente 'apellido' se encuentre declarado
	    en la 'ClaseHeredada' (clase padre) */
	    System.out.println(X.getPersona() + X.getApellido());
	}
}
```

VÍDEO DE INTERÉS

Visualiza este vídeo para ver un ejemplo de herencia entre una clase Persona y otros estudiantes:

2.2 Polimorfismo

El polimorfismo es uno de los conceptos fundamentales de la programación orientada a objetos (POO) y está presente en el lenguaje de programación Java. La palabra polimorfismo está compuesta por la palabra “Poli” que significa muchos y por “morfismo” que significa formas.

El polimorfismo se refiere a la capacidad de un objeto de una clase de comportarse de diferentes maneras o asumir diferentes formas en función del contexto en el que se utiliza. En Java, el polimorfismo se logra a través de dos mecanismos principales: el polimorfismo de subtipos y la sobrecarga de métodos.

Polimorfismo de subtipos:
Proporciona la capacidad de redefinir por completo el método de una superclase en una subclase. Es decir, en una subclase se definirá un método que tendrá el mismo nombre y misma lista de argumentos que un método existente en la superclase (hay relación de herencia entre estas clases) pero el método definido tendrá un comportamiento diferente.

Tenemos las clases Animal.java, Gato.java y Perro.java, donde Animal.java es la superclase o clase padre y las otras dos son las clases hijas. La clase padre contiene un método hacerSonido() que muestra en pantalla el texto “Este animal emite algún ruido”. En la subclase Gato se vuelve a definir este método hacerSonido() pero se cambia el comportamiento para que muestre en pantalla el mensaje “El gato maulla”, e igualmente se realiza la misma operación en la clase Perro pero esta vez para que muestre “El perro ladra”.

En la clase Main, que contendrá un método main(String[] args), se instancia un objeto de tipo Perro y otro de tipo Gato, pero al llamar al método hacerSonido(), Java sabrá qué método tiene que ejecutar dependiendo del tipo de objeto que realiza la llamada, bien el método declarado en la clase Perro o bien en la clase Gato.

Clase Animal.java

```
package polimorfismo;
class Animal {
     void hacerSonido() {
               System.out.println("Este animal emite algún
ruido");
          }
}
```

Clase Gato.java

```
package polimorfismo;
class Gato extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("El gato maulla");
    }
}
```

Clase Perro.java

```
package polimorfismo;
class Perro extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("El perro ladra");
    }
}
```

Clase Main.java

```
package polimorfismo;
public class Main {
     public static void main(String[] args) {
          Animal miMascota;

          // Polimorfismo de subtipos
          miMascota = new Perro();

          // Llama al método de la subclase Perro
          miMascota.hacerSonido();

          // Polimorfismo de subtipos
          miMascota = new Gato();

          // Llama al método de la subclase Gato
          miMascota.hacerSonido();
     }
}
```

Sobrecarga de métodos:

La sobrecarga de métodos, en programación, se refiere a la capacidad de definir múltiples métodos en una misma clase con el mismo nombre, pero con diferentes parámetros. Los métodos sobrecargados tienen el mismo nombre, pero se diferencian por la cantidad, el tipo o el orden de los parámetros que reciben. Esto permite a los programadores crear métodos con funcionalidades similares pero adaptadas para manejar diferentes situaciones o tipos de datos.

Algunas consideraciones importantes sobre la sobrecarga de métodos en Java:

- **Nombre del método:** Todos los métodos sobrecargados deben tener el mismo nombre.
- **Diferenciación por parámetros:** Los métodos sobrecargados deben tener una diferencia en la lista de parámetros, ya sea en términos de tipo, cantidad o ambos.
- **Valor de retorno:** La sobrecarga de métodos no se basa en el valor de retorno; dos métodos con el mismo nombre y la misma lista de parámetros, pero con valores de retorno diferentes, no se consideran sobrecargados.
- **Llamadas a métodos:** Cuando se llama a un método sobrecargado, el compilador de Java determina cuál de los métodos coincidentes ejecutar en función de los argumentos proporcionados.

En el siguiente ejemplo, podemos observar que la clase Calculadora tiene dos métodos llamados *sumar*, uno para enteros y otro para números de punto flotante. Java selecciona automáticamente el método correcto según el tipo de datos proporcionado en las llamadas. Esto es un ejemplo simple de sobrecarga de métodos.

Clase Calculadora.java

```
package SobrecargaMetodos;
class Calculadora {
    int sumar(int a, int b) {
        return a + b;
    }

    double sumar(double a, double b) {
        return a + b;
    }
}
```

Clase Main.java

```
package SobrecargaMetodos;
public class Main {
    public static void main(String[] args) {
        Calculadora calculadora = new Calculadora();

        // Llama al primer método
        int sumaEnteros = calculadora.sumar(6, 2);

        // Llama al segundo método
        double sumaDobles = calculadora.sumar(1.5, 5.1);

        System.out.println("Suma de enteros: " +
sumaEnteros);
        System.out.println("Suma de dobles: " +
sumaDobles);
    }
```

```
}
```

Diferencia entre polimorfismo y sobrecarga:
El **polimorfismo** se relaciona con la herencia y permite que las subclases se comporten de manera específica cuando se llaman a métodos a través de una referencia de una superclase, mientras que la **sobrecarga** de métodos se relaciona con la definición de múltiples versiones de métodos en una sola clase para manejar diferentes tipos de datos o situaciones. Ambos conceptos son útiles en la programación orientada a objetos, pero se aplican en diferentes contextos y con propósitos diferentes.

Pero estos dos conceptos **no son mutuamente excluyentes** y a menudo se utilizan en conjunto para crear una interfaz más versátil. Un escenario común en el que se combinan la sobrecarga y el polimorfismo es cuando una superclase define un método con un nombre específico y las subclases sobrescriben ese método. Además, en las subclases, es posible tener métodos adicionales con el mismo nombre, pero con diferentes parámetros. Estos métodos sobrecargados en las subclases no reemplazan los métodos de la superclase, ya que tienen una firma diferente debido a los diferentes parámetros.

PARA SABER MÁS

Para saber más podrías combinar el polimorfismo junto con la sobrecarga de métodos en el ejemplo anterior de las clases Animal, Perro y Gato.

3. SUPERCLASES Y SUBCLASES

Reúnes a tu equipo, como de costumbre, en lo que conocéis como una 'Daily Scrum Meeting' y les explicas las ventajas de utilizar la herencia ya que, en lugar de crear nuevas clases desde cero, se crearán subclases que heredan atributos y métodos de las superclases existentes. Estas subclases añadirán o modificarán la funcionalidad heredada para adaptarse a los requisitos específicos del proyecto. Esta estrategia permitirá aprovechar la estructura y el código existente, lo que ahorrará tiempo y reducirá la redundancia de código. Todo el equipo se pone manos a la obra.

En Java, como en otros lenguajes de programación orientados a objetos, las clases pueden derivar desde otras clases. La clase derivada (la clase que proviene de otra clase) se llama **subclase**. La clase de la que está derivada se denomina **superclase**.

De hecho, en Java, todas las clases deben derivar de alguna clase. Lo que lleva a la cuestión ¿Dónde empieza todo esto? La clase más alta, la clase de la que todas las demás descienden, es la clase **Object,** definida en **java.lang**. Object es la raíz de la herencia de todas las clases, por lo que se puede decir que Object es la superclase de todas las clases en Java.

Las subclases heredan el **estado** y el **comportamiento** en forma de las **variables** y los **métodos** de su superclase. La subclase puede utilizar los ítems heredados de su superclase tal y como son, o puede **modificarlos** o **sobreescribirlos**. Por eso, según se va bajando por el árbol de la herencia, las clases se convierten en más y **más especializadas**. Una **subclase** es una clase que desciende de otra clase. Una subclase hereda el estado y el comportamiento de todos sus ancestros.

El término **superclase** se refiere a la clase que es el ancestro más directo, así como a todas las clases ascendentes.

Se declara que una clase es una subclase de otra clase dentro de La declaración de la Clase.

Se supone que se quiere crear una subclase llamada SubClase de otra clase llamada SuperClase. Se escribiría:

```
  public class SubClase extends SuperClase {

}
```

Esto declara que SubClase es una subclase de SuperClase. Y también declara implícitamente que SuperClase es la superclase de SubClase. Una subclase también hereda variables y miembros de las superclases de su superclase, y así a lo largo del árbol de la herencia.

ENLACE DE INTERÉS

Conoce mejor el concepto de subclase y superclase visitando este enlace:

4. ABSTRACT, FINAL Y STATIC

En esta etapa del proyecto, te das cuenta de que se necesita definir y organizar las clases y métodos de manera especial para lograr los objetivos propuestos. Estás considerando la posibilidad de utilizar clases abstractas, y métodos abstractos, así como clases, métodos y variables finales y estáticas en tu diseño.

La razón detrás de esta elección es que deseas establecer una estructura sólida para el código, asegurando que ciertas clases no puedan ser heredadas o modificadas, mientras que otras deben proporcionar implementaciones específicas. También quieres utilizar variables estáticas para almacenar información compartida en toda la aplicación.

En este apartado veremos las palabras reservadas, y en Java, su significado y utilidad cuando se aplican a clases, métodos o variables.

4.1 Clases y métodos abstractos

A veces, una clase que se ha definido representa un concepto abstracto y como tal, no debe ser ejemplarizado. Por ejemplo, la comida en la vida real. ¿Ha visto algún ejemplar de comida? No. Lo que ha visto son ejemplares de manzanas, pan, y chocolate. Comida representa un **concepto abstracto** de cosas que son comestibles. **No tiene sentido** que exista un ejemplar de comida.

En la programación orientada a objetos se podrían modelar conceptos abstractos, y, a su vez, no querer que se creen ejemplares de ellos.

Por ejemplo, la clase Number del paquete java.lang representa el concepto abstracto de número. Tiene sentido modelar números en un programa, pero no tiene sentido crear un objeto genérico de números. En su lugar, la clase Number sólo tiene sentido como superclase de otras clases como Integer y Float que implementan números de tipos específicos. Las clases como Number, que implementan conceptos abstractos y no deben ser ejemplarizadas, son llamadas **clases abstractas**. Una clase abstracta es una clase que **sólo puede tener subclases** no puede ser ejemplarizada o instanciada, siendo las subclases o clases hijas las encargadas de definir los detalles.
Para declarar que una clase es una clase abstracta, se utiliza la palabra clave **abstract** en la declaración de la clase.

```
abstract class Number {
    . . .
}
```

Si se intenta ejemplarizar una clase abstracta, el compilador mostrará un **error** similar a este y no compilará el programa.

```
AbstractTest.java:6: class AbstractTest is an abstract class. It can't be
instantiated.
        new AbstractTest();
        ^
1 error
```

Una clase abstracta puede contener atributos, métodos, constructores, etc. al igual que una clase convencional y además contener **métodos abstractos**, esto es, métodos que **no tienen implementación** (sin cuerpo). De esta forma, una clase abstracta puede definir una estructura de programación completa, incluso proporciona a sus subclases la declaración de todos los métodos necesarios para implementar esta estructura de programación. Sin embargo, las clases abstractas pueden dejar algunos detalles o toda la implementación de aquellos métodos a sus subclases.

ENLACE DE INTERÉS

Lo mejor para aprender a utilizar las clases y métodos abstractos es mediante ejemplos. En este enlace tienes acceso a ello:

Se va a detallar a continuación un ejemplo de cuándo sería necesario crear una clase abstracta con métodos abstractos. En una aplicación de dibujo orientada a objetos, se pueden dibujar círculos, rectángulos, líneas, etc… Cada uno de esos objetos gráficos comparten ciertos estados (posición, caja de dibujo) y comportamiento (movimiento, redimensionado, dibujo). Se pueden aprovechar esas similitudes y declararlos todos a partir de un mismo objeto padre - `ObjetoGrafico`.
Sin embargo, los objetos gráficos también tienen diferencias substanciales: dibujar un círculo es bastante diferente a dibujar un rectángulo. Los objetos gráficos no pueden compartir estos tipos de estados o comportamientos.

Por otro lado, todos los objetos `ObjetoGrafico` deben saber cómo dibujarse a sí mismos; se diferencian en cómo se dibujan unos y otros. Esta es la situación perfecta para una clase abstracta.

Primero se debe declarar una clase abstracta, `ObjetoGrafico`, para proporcionar las variables "miembro" y los métodos que van a ser compartidos por todas las subclases, como la posición actual y el método **`moverA()`**.

También se deberían declarar métodos abstractos como **`dibujar()`**, que necesita ser implementado por todas las subclases, pero de manera completamente diferente (no tiene sentido crear una implementación por defecto en la superclase).

Una clase abstracta **no necesita** contener un método abstracto. Pero **todas las clases que contengan un método abstracto** o no proporcionen implementación para cualquier método abstracto declarado en sus superclases, deben ser declaradas como una **clase abstracta**.

EJEMPLO PRÁCTICO

La clase ObjetoGrafico se parecería a:

```
public abstract class ObjetoGrafico {
int x, y;...

      void moverA(int nuevaX, int nuevaY) {
           ...
          }

      abstract void dibujar();

}
```

Todas las subclases no abstractas de ObjetoGrafico como son Circulo o Rectángulo deberán proporcionar una implementación para el método dibujar().

Clase Circulo:

```
public class Circulo extends ObjetoGrafico {
    void dibujar() {
        . . .
    }
}
```

Clase Rectángulo:

```
public class Rectangulo extends ObjetoGrafico {
    void dibujar() {
        . . .
    }
}
```

RECUERDA

Una clase abstracta tiene las siguientes características:

- Una clase abstracta puede tener, o no, métodos abstractos.
- Una clase con uno o más métodos abstractos debe ser declarada abstracta.
- Una clase abstracta puede combinar métodos abstractos con métodos ya implementados.
- Una clase abstracta no puede ser instanciada directamente.

RECUERDA

Un método abstracto tiene las siguientes características:

- No tiene cuerpo, solo consta de la signatura y termina en punto y coma.

Solo existirá dentro de una clase abstracta, es decir, si una clase incluye un método abstracto, forzosamente la clase será una clase abstracta.

4.2 Clases, métodos y variables finales

Se puede declarar que una **clase** sea **final**; esto es, que la clase **no pueda tener subclases**. Existen (al menos) dos razones por las que se querría hacer esto:

- **Seguridad**: un mecanismo que los hackers utilizan para atacar sistemas es crear subclases de una clase y luego sustituirla por el original. Las subclases parecen y sienten como la clase original, pero hacen cosas bastante diferentes, probablemente causando daños u obteniendo información privada. Para prevenir esta clase de subversión, se puede declarar que la clase sea final y así prevenir que se cree cualquier subclase.

  La clase String del paquete java.lang es una clase final sólo por esta razón. La clase String es tan vital para la operación del compilador y del intérprete que el sistema Java debe garantizar que siempre que un método o un objeto utilicen un String, obtenga un objeto java.lang.String y no algún otro string. Esto asegura que ningún string tendrá propiedades extrañas, inconsistentes o indeseables.
  Si se intenta compilar una subclase de una clase final, el compilador mostrará un **mensaje de error** y **no compilará** el programa.

  Además, los bytecodes verifican que no está teniendo lugar una subversión, al nivel de byte comprobando que una clase no es una subclase de una clase final.

- **Diseño**: otra razón por la que se podría querer declarar una clase final son razones de diseño orientado a objetos. Se podría pensar que una clase es "perfecta" o que, conceptualmente hablando, la clase no debería tener subclases.

Para especificar que una clase es una clase final, se utiliza la palabra clave **final** antes de la palabra clave **class** en la declaración de la clase. Por ejemplo, si quisiéramos declarar AlgoritmodeAjedrez como una clase final (perfecta), la declaración se parecería a esto.

```
final class AlgoritmodeAjedrez {
    . . .
}
```

Cualquier intento posterior de crear una subclase de AlgoritmodeAjedrez resultará en el siguiente error del compilador.

```
Chess.java:6:    Can't    subclass    final    classes:    class
AlgoritmodeAjedrez
class MejorAlgoritmodeAjedrez extends AlgoritmodeAjedrez {
      ^
1 error
```

Para los **métodos finales**, si realmente lo que se quiere es proteger algunos métodos de una clase para que no sean sobre escritos, se puede utilizar la palabra clave **final** en la declaración del método para indicar al compilador que este método **no puede ser sobre escrito** por las subclases.
Se podría desear hacer que un método fuera final si el método tiene una implementación que no debe ser cambiada y que es crítica para el estado consistente del objeto. Por ejemplo, en lugar de hacer AlgoritmodeAjedrez como una clase final, podríamos hacer **siguienteMovimiento()** como un método final.

```
class AlgoritmodeAjedrez {
    . . .
    final void siguienteMovimiento(Pieza piezaMovida,
                  PosicionenTablero nuevaPosicion) {
    }
    . . .
}
```

Si una **variable** es declarada como **final**, quiere decir que no puede ser modificada después de su inicialización. Esto significa que el valor de la variable final no puede ser cambiado después de que se le haya asignado un valor durante la ejecución del programa. En general, se recomienda utilizar variables finales siempre que sea posible, para representar valores constantes que no queremos que cambien accidentalmente dentro de un método o bloque de código durante la ejecución del programa. El identificador de la constante se suele escribir en mayúsculas.

Ventajas de usar variables finales

- **Mejorar la legibilidad del código**: al utilizar variables finales, se pueden nombrar valores constantes de manera clara y concisa. Esto puede hacer que el código sea más fácil de entender y mantener.
- **Prevenir errores:** al declarar una variable como final, se garantiza que su valor no será modificado accidentalmente por el código del programa. Esto puede prevenir errores que podrían surgir si se intenta modificar accidentalmente el valor de una variable constante.

Optimizar el rendimiento: en algunos casos, el compilador puede optimizar el código cuando se utiliza una variable final en lugar de una variable normal. Esto se debe a que el compilador sabe que el valor de la variable no cambiará y puede hacer ciertas optimizaciones en el código.

EJEMPLO PRÁCTICO

Utiliza dos constantes de tipo de entero, una para almacenar los días de vacaciones y el número de asuntos propios que tienen los trabajadores de una determinada empresa.

```
package ud7.constantes;

public class EjemploConstantesEnJava {
	public static void main(String[] args) {
		final int DIASVACACIONES = 20;
		final int DIASASUNTOSPROPIOS = 5;

		System.out.println("Los días de vacaciones son: " +
                                                    DIASVACACIONES);
		System.out.println("El número de días de asuntos propios
                                        son: " + DIASASUNTOSPROPIOS);
	}
}
```

VÍDEO DE INTERÉS

Repase el uso de los modificadores static y final con este vídeo:

4.3 Clases, métodos y variables estáticas

El modificador static en Java se puede utilizar con clases, métodos o variables.

Una **clase estática** es aquella que no puede ser instanciada y pertenece a la definición de la clase que la contiene, es decir, no es posible crear objetos de una clase estática utilizando el operador 'new'. Las clases estáticas se utilizan a menudo como clases auxiliares que proporcionan utilidades o funcionalidades compartidas entre diferentes instancias de otra clase principal.

Una clase estática muy utilizada en la API de Java es la clase **Math**. La clase Math proporciona métodos estáticos para realizar operaciones matemáticas comunes, como cálculos trigonométricos, redondeos, funciones exponenciales, generación de números aleatorios, y más. Estos métodos pueden ser accedidos directamente a través del nombre de la clase, sin necesidad de crear instancias de esta.

Aquí tienes un ejemplo de cómo se pueden utilizar algunos métodos estáticos de la clase Math:

En este ejemplo puedes observar que la clase *Math* al ser estática no hace falta instanciarla, para llamar a sus métodos solo es necesario escribir el nombre de la clase seguido de un punto y a continuación el método.

```
package ud7.ClaseEstatica;

public class EjemploClaseEstaticaMath {
        public static void main(String[] args) {
            double numero = 3.7;

            // Redondear hacia arriba
```

```
                double redondeoArriba = Math.ceil(numero);
                System.out.println("Redondeo hacia arriba: " +
                                                redondeoArriba);

                // Redondear hacia abajo
                double redondeoAbajo = Math.floor(numero);
                System.out.println("Redondeo hacia abajo: " +
                                                redondeoAbajo);

                // Valor absoluto
                double valorAbsoluto = Math.abs(numero);
                System.out.println("Valor absoluto: " +
                                                valorAbsoluto);

                // Generar un número aleatorio entre 0 y 1
                double numeroAleatorio = Math.random() ;
                System.out.println("Número aleatorio: " +
                                                 numeroAleatorio);
// Generar un número aleatorio entre 0 y 100 incluido
                 numeroAleatorio = Math.random()*(100 + 1) ;
                System.out.println("Número aleatorio: " +
                                            Math.round(numeroAleatorio)
                                            );
            }
        }
```

Al ejecutar esta clase se ha obtenido la siguiente información por consola, pero hay que tener en cuenta que los números generados aleatoriamente, normalmente serán diferentes en distintas ejecuciones:

```
Redondeo hacia arriba: 4.0
Redondeo hacia abajo: 3.0
Valor absoluto: 3.7
Número aleatorio: 0.019457625884677587
Número aleatorio: 67
```

Si añadimos el modificador static a un **método de una clase**, entonces el método pertenecerá a la clase, por tanto, podremos usarlo a través del nombre de la clase, sin tener que realizar instanciación de objetos. Sin embargo, hay que tener en cuenta que los métodos estáticos declarados en una clase solo podrán acceder a atributos estáticos u otros métodos estáticos de la clase, pero nunca a atributos o métodos que no lo sean.

Las **variables estáticas** son útiles cuando se necesita almacenar información que es compartida por todas las instancias de una clase, o cuando se necesita llevar un registro de alguna información en la clase.

Algunos de los usos más comunes de las variables estáticas son:

- **Conteo de objetos**: las variables estáticas se pueden utilizar para llevar un registro del número de objetos creados de una clase en particular (se incluyen sus hijas).
- **Constantes**: las variables estáticas también se pueden utilizar para definir constantes que se utilizan en toda la clase. Por ejemplo, una constante PI en una clase de geometría.
- **Compartir información**: las variables estáticas se pueden utilizar para compartir información entre diferentes objetos de una clase.
- **Almacenamiento en caché**: las variables estáticas se pueden utilizar para almacenar en caché valores que son costosos de calcular y que se necesitan frecuentemente.

Para contabilizar el número de objetos Persona que se están creando en un momento dado, se puede usar una variable estática que incrementaremos dentro del constructor, de forma que cada vez que se instancie un nuevo objeto de tipo Persona, este será contado. Como es una variable de clase, el valor de esta variable será el mismo para todas las instancias que se creen de esta clase.

Clase Persona.java

```
package ud7.MetodosEstaticos;
public class Persona {
	/*contador es una variable estática, por tanto, tendrá el
mismo
	valor para todas las instancias de la clase Persona
	almacena el número de instancias creadas de la clase
Persona.
	Inicialmente se pone a 0 y se incrementa en 1 cada vez que
se
	crea una nueva instancia de Persona(cuando se llama a su
	constructor)*/
	private static int contador = 0;
	private String nombre;
	public Persona(String nombre) {
		this.nombre = nombre;
		contador++;
	}
	//Devuelve el valor de la variable estática contador
	public static int getContador() {
		return contador;
	}
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
```

```
            this.nombre = nombre;
      }
}
```

Clase MainPersona.java

```
package ud7.MetodosEstaticos;
public class MainPersona {
      public static void main(String[] args) {
            Persona persona1 = new Persona("Juan");
            // Imprime "Total de Personas creadas: 1"
            System.out.println("Total de Personas creadas: " +
                                    Persona.getContador());
            // Imprime "Total de Personas creadas: 2"
            Persona persona2 = new Persona("Maria");
            System.out.println("Total de Personas creadas: " +
                                     Persona.getContador());
            // Imprime "Total de Personas creadas: 3"
            Persona persona3 = new Persona("Pedro");
            System.out.println("Total de Personas creadas: " +
                                     Persona.getContador());
            System.out.println("\n\nEstos son sus nombres..\n");
            System.out.println(persona1.getNombre());
            System.out.println(persona2.getNombre());
            System.out.println(persona3.getNombre());
      }
}
```

Se puede observar que cada vez que se instancia un nuevo objeto la variable contador se incrementa en 1, además cuando se llama al método getContador() como es un método también estático y pertenece a la clase Persona para invocarlo se hace usando la propia clase Persona(Persona.getContador()).

Por otro lado, podemos observar que el atributo nombre (no estático) será propiedad de cada uno de los objetos instanciados de la clase Persona a diferencia del atributo contador que es propiedad de la propia clase, por lo que cada objeto será dueño de su atributo nombre y este podrá cambiar en cada uno de los objetos sin afectar a los demás.

¿Una variable final tiene que ser también estática(static)?

No necesariamente, las variables finales pueden ser estáticas o no estáticas.

Si una variable final es a la vez estática, significa que su valor es compartido entre todas las instancias de la clase y solo puede ser inicializado una vez, generalmente en el bloque de inicialización estático o en la definición de la variable.

Por otro lado, si una variable final no es estática, significa que su valor es específico para cada instancia de la clase y solo puede ser inicializado una vez, generalmente en el constructor o en la definición de la variable.

5. SOBREESCRITURA DE MÉTODOS

Con la sobreescritura de métodos se podrá personalizar en tu aplicación la forma en que se tratan ciertos datos según sus características específicas. Esto implica crear subclases y reemplazar el método o métodos implementados en la superclase por métodos con igual nombre, pero con una implementación adaptada a la funcionalidad requerida en cada una de las subclases.

Una subclase puede:

- Sobrescribir completamente la implementación de un método de una superclase.
- Añadir implementación a un método de la superclase.
- Métodos que una subclase no puede sobrescribir.

5.1 Reemplazar la implementación de un método de una superclase

Algunas veces, una subclase querría reemplazar completamente la implementación de un método de su superclase. De hecho, muchas superclases proporcionan implementaciones de métodos vacías con la esperanza de que la mayoría, si no todas, de sus subclases reemplacen completamente la implementación de ese método.

Un ejemplo de esto es el método **run()** de la clase Thread. La clase Thread proporciona una implementación vacía (el método no hace nada) para el método **run()**, porque, por definición, este método depende de la subclase. La clase Thread posiblemente no puede proporcionar una implementación medianamente razonable del método **run()**.

Para reemplazar completamente la implementación de un método de la superclase, simplemente se llama a un método con el mismo nombre que el del método de la superclase y se sobrescribe el método con la misma firma que la del método sobrescrito.

```
class ThreadSegundoPlano extends Thread {
    void run() {
        . . .
    }
}
```

La clase ThreadSegundoPlano sobreescribe completamente el método **run()** de su superclase y reemplaza completamente su implementación.

ENLACE DE INTERÉS

Para entender en su totalidad el código fuente escrito en Java, es imprescindible tener muy claro el concepto de sobreescritura de métodos. Amplía información en este enlace:

5.2 Añadir implementación a un método de la superclase

Otras veces una subclase querrá mantener la implementación del método de su superclase y posteriormente **ampliar** algún comportamiento específico de la subclase. Por ejemplo, los métodos constructores de una subclase lo hacen normalmente: la subclase quiere preservar la inicialización realizada por la superclase, pero proporciona inicialización adicional específica de la subclase.

Se supone que se quiere crear una subclase de la clase Windows del paquete java.awt. La clase Window tiene un constructor que requiere un argumento del tipo Frame que es el padre de la ventana.

```
public Window(Frame parent)
```

Este constructor realiza alguna inicialización en la ventana para que trabaje dentro del sistema de ventanas. Para asegurarse de que una subclase de Window también trabaja dentro del sistema de ventanas, se deberá proporcionar un constructor que realice la misma inicialización.

Mucho mejor que intentar recrear el proceso de inicialización que ocurre dentro del constructor de Window se podría utilizar lo que la clase Window ya hace. Se puede utilizar el código del constructor de Window llamándolo desde dentro del constructor de la subclase Window.

```
class Ventana extends Window {
    public Ventana(Frame parent) {
        super(parent);
        . . .
            // Ventana especifica su inicialización aquí
        . . .
    }
}
```

El constructor de **Ventana** llama primero al constructor de su superclase, y no hace nada más. Típicamente, este es el comportamiento deseado de los constructores. Las superclases deben tener la oportunidad de realizar sus tareas de inicialización antes que las de su subclase. Otros tipos de métodos podrían llamar al constructor de la superclase al final del método o en el medio.

VÍDEO DE INTERÉS

Visualiza este interesante curso sobre los métodos constructores y, la sobreescritura de métodos utilizando el método toString muy utilizado en Java:

5.3 Métodos que una subclase no puede sobrescribir

Los métodos que una subclase no puede sobrescribir son los que se explican a continuación:

- Una subclase no puede sobrescribir métodos que hayan sido declarados como **final** en la superclase (por definición, los métodos finales no pueden ser sobrescritos). Si intentamos sobrescribir un método final, el compilador mostrará un mensaje y no compilará el programa.
- Una subclase tampoco puede sobrescribir métodos que se hayan declarado como **static** en la superclase. En otras palabras, una subclase no puede sobrescribir un método de clase.

6. CONSTRUCTORES Y HERENCIA

Durante el desarrollo del proyecto empiezan a surgir dudas en el equipo sobre el orden en el que se ejecutan los constructores en las relaciones de herencia. Esta cuestión debe quedar muy clara antes de continuar, por lo que en la siguiente 'Daily' harás una breve exposición para indicar que cuando se crea un objeto de una subclase, primero se ejecuta el constructor de la clase base y luego el constructor de la subclase. Esto garantiza que se inicialicen adecuadamente tanto los atributos de la clase base como los atributos específicos de la subclase.

A diferencia de lo que ocurre con los métodos y atributos no privados, **los constructores no se heredan**. Además de esta característica, deben tenerse en cuenta algunos aspectos sobre el comportamiento de los constructores dentro del contexto de la herencia, ya que dicho comportamiento es sensiblemente distinto al del resto de métodos.

Cuando existe una relación de herencia entre diversas clases y se crea un objeto de una subclase S, se ejecuta no sólo el constructor de S sino también el de todas las superclases de S. Para ello se ejecuta en primer lugar el constructor de la clase que ocupa el nivel más alto en la jerarquía de herencia y se continúa de forma ordenada con el resto de las subclases.

Se definen tres clases: A, B y C. Cada una de estas clases tiene un constructor que muestra un mensaje en la consola cuando se crea una instancia de la clase.

La clase B extiende la clase A, lo que significa que hereda sus características, incluido el constructor.

La clase C extiende la clase B, lo que también implica la herencia de características y constructores.

En el método main de la clase Constructores_y_Herencia, se crea una instancia de la clase C llamada obj.

Cuando se crea una instancia de C (que es la subclase más profunda), se ejecutan los constructores de todas las clases en la jerarquía de herencia, en orden de arriba a abajo. En este caso, se mostrarán los mensajes "En A", "En B" y "En C" en ese orden en la consola.

Entonces, la salida en la consola será:

En A

En B

En C

EJEMPLO PRÁCTICO

El siguiente ejemplo ilustra este comportamiento:

```
class A {

    A() {
        System.out.println("En A");
    }
}

class B extends A {

    B() {
        System.out.println("En B");
    }
}

class C extends B {

    C() {
        System.out.println("En C");
    }
}

class Constructores_y_Herencia {

    public static void main(String[] args) {
        C obj = new C();
    }
}
```

Es posible que una misma clase tenga más de un constructor (sobrecarga del constructor), tal y como se muestra en este ejemplo:

```
class A {
    A() {
        System.out.println("En A");
    }
    A(int i) {
        System.out.println("En A(i)");
    }
```

```
}
  class B extends A {
      B() {
          System.out.println("En B");
      }
      B(int j) {
          System.out.println("En B(j)");
      }
}
```

ENLACE DE INTERÉS

Este enlace nos lleva a un tutorial muy interesante sobre herencia y constructores en Java:

La cuestión que se plantea es: ¿qué constructores se invocarán cuando se ejecuta la sentencia **`B obj = new B(5);`**? Puesto que hemos creado un objeto de tipo B al que le pasamos un entero como parámetro, parece claro que en la clase B se ejecutará el constructor **`B(int j)`**. Sin embargo, puede haber confusión acerca de qué constructor se ejecutará en A. La regla en este sentido es clara: mientras no se diga explícitamente lo contrario, en la superclase se ejecutará siempre el constructor sin parámetros. Por tanto, ante la sentencia **`B obj = new B(5);`** se mostraría:

```
En A
En B(j)
```

EJEMPLO PRÁCTICO

En el siguiente ejemplo se especifica de forma explícita el constructor que deseamos ejecutar en A:

```
class A {

    A() {
        System.out.println("En A");
    }

    A(int i) {
        System.out.println("En A(i)");
    }
}

class B extends A {

    B() {
        System.out.println("En B");
    }

    B(int j) {
        super(j);     // Ejecutar en la superclase un
constructor que acepte un entero
        System.out.println("En B(j)");
    }
}
```

Ha de tenerse en cuenta que en el caso de usar la sentencia **`super ()`**, ésta deberá ser obligatoriamente **la primera sentencia** del constructor. Esto es así porque se debe respetar el orden de ejecución de los constructores comentado anteriormente.

En resumen, cuando se crea una instancia de una clase, para determinar el constructor que debe ejecutarse en cada una de las superclases, en primer lugar, se **exploran** los constructores en **orden jerárquico ascendente** (desde la subclase hacia las superclases). Con este proceso se decide el constructor que debe ejecutarse en cada una de las clases que componen la jerarquía. Si en algún constructor no aparece explícitamente una llamada a **`super(lista_de_argumentos)`** se entiende que de forma implícita se está invocando a **`super()`** (constructor sin parámetros de la superclase). Finalmente, una vez que se conoce el constructor que debe ejecutarse en cada una de las clases que componen la jerarquía, éstos **se ejecutan** en **orden jerárquico descendente** (desde la superclase hacia las subclases).

Podemos considerar que todas las clases en Java tienen de forma implícita un **constructor por defecto** sin parámetros y sin código. Ello permite crear objetos de dicha clase sin necesidad de incluir explícitamente un constructor. Por ejemplo, dada la clase:

```
class A {
    int i;
}
```

No hay ningún problema en crear un objeto:

```
A obj = new A();
```

Ya que, aunque no lo escribamos, la clase A lleva implícito un constructor por defecto:

```
class A {

    int i;

    A() {
        // Constructor por defecto
    }
}
```

Sin embargo, es importante saber que dicho constructor por defecto **se pierde** si escribimos cualquier otro constructor. Por ejemplo, dada la clase:

```
class A {

    int i;

    A(int valor) {
        i = valor;
    }
}
```

La sentencia **`A obj = new A();`** generaría un **error de compilación**, ya que en este caso no existe ningún constructor en A sin parámetros. Hemos perdido el constructor por defecto. Lo correcto sería, por ejemplo, **`A obj = new A(5);`** Esta situación debe tenerse en cuenta igualmente cuando exista un esquema de herencia.
Se tiene la siguiente jerarquía de clases:

```
class A {

    int i;

    A(int valor) {
        i = valor;
```

```
    }
}

class B extends A {

    int j;

    B(int valor) {
        j = valor;
    }
}
```

En este caso la sentencia **`B obj = new B(5)`**; generará igualmente un error ya que, puesto que no hemos especificado un comportamiento distinto mediante **`super()`**, en A debería ejecutarse el constructor sin parámetros, sin embargo, tal constructor no existe puesto que se ha perdido el constructor por defecto. La solución pasaría por sobrecargar el constructor de A añadiendo un constructor sin parámetros.

```
class A {

    int i;

    A() {
        i = 0;
    }

    A(int valor) {
        i = valor;
    }
}

class B extends A {

    int j;

    B(int valor) {
        j = valor;
    }
}
```

O bien indicar explícitamente en el constructor de B que se desea ejecutar en A un constructor que recibe un entero como parámetro, tal y como se muestra a continuación:

```
class A {

   int i;

   A(int valor) {
       i = valor;
```

```
    }
}
class B extends A {

    int j;

    B(int valor) {
        super(0);
        j = valor;
    }
}
```

7. INTERFACES

Para garantizar la interoperabilidad y la comunicación efectiva entre los diferentes módulos de tu aplicación, decides utilizar interfaces en tu diseño. Las interfaces actúan como contratos que especifican un conjunto de métodos o comportamientos que deben ser implementados por cualquier clase que desee interactuar con dicha interface. Por tanto, varias clases diferentes pueden implementar esa interfaz de manera que todas cumplan con los mismos métodos especificados en la misma, pero a su vez cada clase puede proporcionar su propia implementación única para esos métodos. Esto facilitará el poder cambiar las implementaciones de las clases que siguen una interface sin afectar la funcionalidad global del sistema, favoreciendo la adaptación y la evolución del mismo con el tiempo.

En Java, cuando estamos hablando de clases, una interface es una colección de métodos abstractos y constantes que se pueden implementar en cualquier clase (no confundir con el término interfaz gráfica). Se suele decir que una interface es como un contrato, una plantilla, etc. para la construcción de la clase que lo implementa. Como se ha comentado anteriormente, Java no permite la herencia múltiple, pero el uso de interfaces nos va a dar la posibilidad de poder implementar herencia múltiple.

Una interface es similar a una clase abstracta y no se puede instanciar (no se pueden crear objetos de ella), pero a diferencia de una clase abstracta, una interface no puede tener implementaciones de métodos en su interior, es decir, métodos con cuerpo, salvo a partir de Java 8 y en adelante que se incorporan los métodos default y estáticos que pueden tener cuerpo. En este apartado de la unidad nos centraremos en la explicación de interfaces según las características de versiones anteriores a Java 8 y solo tendremos en cuenta que los métodos dentro de la interface no tendrán código solo la declaración del método.

Los métodos en una interface son por defecto "abstractos", lo que significa que no tienen una implementación concreta, solo se define su firma o signatura, es decir, nombre del método, parámetros y tipo de retorno. Además, todas las variables definidas

en una interface son “public”, "final" y "static" por defecto, por tanto, se pueden omitir estos modificadores y es necesario inicializarlas en la misma sentencia de declaración, lo que significa que no se pueden modificar, que pertenecen a la interface en sí y se podría acceder a ellas sin necesidad de crear una instancia de la interfaz.

Características

- Una interfaz solo podrá extender o heredar de otras interfaces (no hay un máximo).
- Una clase solo puede heredar(extends) de una clase padre, pero, sin embargo, puede implementar(implements) cualquier número de interfaces.
- Normalmente, las interfaces o no tienen atributos o tienen atributos que son constantes.
- Solo pueden tener métodos públicos.
- Si una clase implementa una o varias interfaces entonces debe definir e implementar obligatoriamente estos métodos que declara la o las interfaces.

El siguiente ejemplo nos servirá para ilustrar el uso de interfaces en Java.

Supongamos que tenemos distintos dispositivos y todos tienen la funcionalidad de reproducir y pausar. Además, el reproductor de MP3 podrá grabar. Pero según sea el dispositivo la funcionalidad reproducir, pausar o grabar estará adaptada al dispositivo. Cuando una clase implementa una interface estará obligada a implementar todos los métodos declarados en la interface que implementa, pero tiene libertad para añadirle el código que sea necesario para desarrollar la funcionalidad requerida.

`Reproducible.java` y `Grabable.java` son interfaces. `ReproductoCDs.java`, `ReproductoDVD.java` y `ReproductorMP3.java` son clases. Todas estas clases implementan la interface `Reproducible`, pero `ReproductorMP3`, además, implementa la interface `Grabable`.

Como se ha comentado anteriormente, una clase puede implementar más de una interface.

Interface Reproducible.java

```
public interface Reproducible {
	void reproducir();
	void pausar();
}
```

Interface Grabable.java

```
package ud7EjemplosInterfaces;

public interface Grabable {
```

```
	void grabar();
}
```

Clase ReproductorCDs.java

```
package ud7EjemplosInterfaces;
public class ReproductorCDs implements Reproducible {
	public void reproducir() {
		System.out.println("Reproduciendo canción en CD");
	}
	public void pausar() {
		System.out.println("Pausando canción en CD");
	}
}
```

Clase ReproductorDVD.java

```
package ud7EjemplosInterfaces;
public class ReproductorDVD implements Reproducible {
	public void reproducir() {
		System.out.println("Reproduciendo canción en DVD");
	}
	public void pausar() {
		System.out.println("Pausando canción en DVD");
	}
}
```

Clase ReproductorMP3.java

```
package ud7EjemplosInterfaces;
	public class ReproductorMP3 implements Reproducible,
Grabable{
		public void reproducir() {
			System.out.println("Reproduciendo canción en
					MP3");
		}
		public void pausar() {
			System.out.println("Pausando canción en MP3");
		}
		public void grabar() {
			System.out.println("Grabando canción en MP3");

		}
	}
```

Clase Main.java

```
package ud7EjemplosInterfaces;
public class Main {
	public static void main(String[] args) {
		//creo una instancia de la clase ReproductorCDs
		ReproductorCDs  cd1 = new ReproductorCDs();
		ReproductorMP3 mp3_1 = new ReproductorMP3();
```

```
            ReproductorDVD dvd_1 = new ReproductorDVD();
            cd1.reproducir();
            mp3_1.reproducir();
            cd1.pausar();
            mp3_1.pausar();
            dvd_1.reproducir();
        }
}
```

La salida en consola será la siguiente:

Reproduciendo canción en CD
Reproduciendo canción en MP3
Pausando canción en CD
Pausando canción en MP3
Grabando canción en MP3
Reproduciendo canción en DVD

VÍDEO DE INTERÉS

Si quieres conocer más sobre qué es esto de las versiones de Java y cuando aparecen, visualiza este vídeo:

RECUERDA

Las interfaces no tienen implementaciones concretas de los métodos que definen, sino que simplemente declaran los métodos que deben ser implementados por cualquier clase que implemente la interface.
Salvo a partir de Java 8, con la incorporación de los métodos default y estáticos que pueden tener cuerpo, pero estos no se tratarán en la unidad.

RESUMEN FINAL

Al desarrollar una aplicación en Java, podemos reutilizar código utilizando la herencia de clases. Es muy importante no volver a programar lo que ya existe y poder reutilizarlo.

En esta unidad se ha comenzado analizando los conceptos de composición y herencia de clases. Hemos visto que la composición es un tipo de relación **dependiente** en la que un objeto más complejo es conformado por objetos más pequeños y se utiliza la frase "***Tiene un***" en la relación. El objeto compuesto tiene la posibilidad de acceder a funcionalidades y propiedades de los objetos contenidos, facilitando la creación de estructuras más complejas y modulares. Por otro lado, la herencia, se utiliza para definir una clase a partir de otra ya existente permitiendo heredar características y funcionalidades de la clase base. Siendo la herencia uno de los pilares de la programación orientada a objetos.

Posteriormente, se ha visto la jerarquía de clases definiendo superclases y subclases, resaltando que la clase Object, definida en el paquete **java.lang** es la raíz de la herencia de todas las clases y la superclase de todas las clases en Java.

A continuación, se han visto los significados de las palabras abstract, final y static que se pueden aplicar a clases, métodos o variables en Java. Una clase abstracta se utiliza como una clase base que proporciona una estructura común y ciertas implementaciones a las clases derivadas (subclases) que la heredan. Las clases abstractas al igual que las clases convencionales pueden contener atributos, métodos, constructores, etc. además de métodos abstractos. Los **métodos abstractos** no tienen implementación dentro de la clase abstracta, pero deben ser obligatoriamente implementados en las clases que extienden(heredan) de la clase abstracta. Con el modificador **final** se indica al compilador que una **clase no puede tener clases hijas**, que un **método no puede sobreescribirse** y que una **variable será una constante cuyo valor no va a poder cambiar** durante la ejecución de la aplicación. Una **clase** declarada con modificador **static no puede ser instanciada**, un **método estático** se podrá acceder solo con invocar a la clase que lo contiene seguido de un punto y a continuación el nombre del método y por último una **variable estática** será útil cuando se necesita almacenar **información** que es **compartida** por todas las instancias de una clase, o cuando se necesita llevar un registro de alguna información en la clase (conteo de objetos, constantes, etc).

Para finalizar esta unidad nos hemos adentrado en la sobreescritura de métodos de una superclase, haciendo énfasis en que una subclase no puede sobreescribir métodos que hayan sido declarados como **final** o **static** en la superclase **y también se ha visto el funcionamiento de los constructores en la herencia.**

Para terminar, se han tratado los interfaces, que se utilizan comúnmente en Java para definir contratos que deben ser cumplidos por las clases que implementan la interfaz, lo que favorece que sean intercambiables en diferentes contextos permitiendo la creación de código modular y flexible.