Trabajo práctico Nro. 2

|  |  |  |
| --- | --- | --- |
| Logo Departamento 2.jpg | **Asignatura: Programación II** | |
|  | |
| **Cursado:** Segundo semestre | **Horas semanales**: |
|  | **Horas semestrales:**  *Cantidad estimada de horas*  *semestrales/anuales.* |
| **Carrera**: *Tecnicatura Universitaria en*  *Programación* | **Nivel (Año):**  1°  2°  3° |
| **Ciclo Lectivo: 2024** |

**Integrantes de la Cátedra:**

* **DOCENTES:**

|  |  |  |
| --- | --- | --- |
| **Nombre del Profesor** | **Periodo** | **Cantidad horas materia** |
| **Cinthia Rigoni** |  | 6 horas |

* 1. **¿Qué es la herencia en java? ¿Cuáles son sus beneficios?**  
     En Java, la herencia es un concepto fundamental de la programación orientada a objetos (POO) que permite que una clase herede atributos y métodos de otra clase, conocida como la clase base o superclase. La clase que hereda se conoce como subclase o clase derivada. La herencia en Java se implementa utilizando la palabra clave extends.

Los beneficios principales de la herencia en Java son:

Reutilización de código: La herencia permite que las clases subclase aprovechen la funcionalidad y comportamiento definidos en una clase base, lo que evita la duplicación de código y promueve la reutilización de código.

Facilita la organización y mantenimiento del código: La herencia ayuda a organizar el código de manera jerárquica, ya que las clases pueden agruparse según su relación de herencia. Esto facilita la comprensión del código y su mantenimiento, ya que los cambios realizados en la clase base se propagan automáticamente a todas las clases derivadas.

Polimorfismo: La herencia permite el polimorfismo, que es la capacidad de tratar objetos de una clase derivada como objetos de su clase base. Esto significa que un objeto de una clase derivada puede utilizarse en cualquier lugar donde se requiera un objeto de su clase base, lo que brinda flexibilidad y extensibilidad al diseño del software.

Extensibilidad: La herencia permite agregar nuevos métodos y atributos a las clases derivadas sin modificar la clase base. Esto facilita la extensión y la adaptación del sistema a medida que los requisitos cambian con el tiempo.

La herencia en Java es una técnica poderosa que promueve la reutilización, la organización y la extensibilidad del código, lo que contribuye a la creación de software modular, flexible y fácil de mantener.

* 1. **¿Cómo se representa la herencia en un diagrama UML?**

La relación de herencia se muestra mediante una flecha sólida que apunta desde la clase hija hacia la clase padre, con una línea sólida que la conecta.

* 1. **¿Cuándo es recomendable usar la herencia?**

La herencia es recomendable en Java cuando se cumplen ciertas condiciones o cuando se busca alcanzar ciertos objetivos en el diseño de un sistema de software. Aquí hay algunas situaciones en las que es apropiado y recomendable utilizar la herencia:

Cuando existe una relación de "es un": La herencia es ideal cuando una clase subclase es una versión especializada o más específica de una clase base. Por ejemplo, si tienes una clase Animal y quieres crear subclases como Perro, Gato, Pájaro, etc., todas estas subclases son tipos de animales y se beneficiarían de heredar características comunes de la clase Animal.

Para reutilizar código común: Si hay comportamientos o atributos comunes entre varias clases, puedes definirlos en una clase base y hacer que las clases derivadas hereden esos atributos y métodos. Esto ayuda a evitar la duplicación de código y promueve la reutilización, lo que facilita el mantenimiento y la escalabilidad del sistema.

Para implementar polimorfismo: La herencia facilita el polimorfismo, que es la capacidad de tratar objetos de clases derivadas como objetos de su clase base. Esto permite escribir código que sea más flexible y extensible, ya que puedes escribir métodos que operan en objetos de la clase base y pueden manejar automáticamente objetos de sus subclases.

Para extender y especializar la funcionalidad: La herencia permite agregar nuevas funcionalidades a las clases derivadas sin modificar la clase base. Esto facilita la extensión y la especialización del sistema a medida que evolucionan los requisitos del software.

Sin embargo, es importante usar la herencia con cuidado y tener en cuenta que puede introducir acoplamiento entre las clases, lo que puede hacer que el código sea más difícil de mantener si no se utiliza correctamente. También es esencial tener en cuenta que la herencia debe ser una relación lógica y coherente en el diseño del software. En algunos casos, la composición u otros patrones de diseño pueden ser más apropiados que la herencia.

* 1. **¿Qué es una interfaz en java?**

En Java, una interfaz es una colección de métodos abstractos y constantes (variables estáticas finales) que define un conjunto de comportamientos que una clase debe implementar. En otras palabras, una interfaz define qué debe hacer una clase, pero no cómo lo debe hacer.

Características:

. Métodos abstractos: Todos los métodos en una interfaz son implícitamente abstractos y no tienen cuerpo. Esto significa que no proporcionan implementaciones y deben ser definidos por las clases que implementan la interfaz.

. Constantes: Las interfaces pueden contener constantes, que son variables declaradas con el modificador `final` y que deben ser inicializadas con un valor. Estas constantes son públicas, estáticas y finales de manera predeterminada.

. Implementación múltiple: En Java, una clase puede implementar múltiples interfaces. Esto permite que una clase adquiera comportamientos de varias fuentes diferentes, lo que facilita la creación de código modular y reutilizable.

. No pueden contener variables de instancia: Las interfaces no pueden tener variables de instancia, ya que todas las variables declaradas en una interfaz son implícitamente públicas, estáticas y finales.

. No pueden ser instanciadas: No se pueden crear instancias directas de una interfaz en Java. Sin embargo, las interfaces pueden ser referenciadas por variables de referencia de interfaz, lo que permite el polimorfismo y el trabajo con múltiples implementaciones de la misma interfaz.

En resumen, una interfaz en Java define un conjunto de métodos que las clases que la implementan deben proporcionar. Proporciona un mecanismo para la abstracción y la especificación de comportamientos comunes, lo que facilita la creación de aplicaciones más flexibles y extensibles.

* 1. **¿Cómo se representa una interfaz en un diagrama UML?**

En un diagrama UML, una interfaz se representa mediante una forma similar a una clase, pero con el nombre del interfaz, precedido por una barra inclinada o una flecha hacia abajo. La interfaz se nombra en la parte superior del rectángulo que representa la interfaz.

* 1. **¿Cuándo es recomendable usar una interfaz?**

Es recomendable usar una interfaz en los siguientes casos:

. Cuando se necesita proporcionar un contrato común para un conjunto de clases: Las interfaces definen un conjunto de métodos que deben implementar las clases que las utilicen. Esto proporciona un contrato común que las clases deben seguir, lo que permite una mayor cohesión y flexibilidad en el diseño del software.

. Cuando se quiere lograr una mayor modularidad y reutilización de código: Las interfaces permiten separar la especificación de una funcionalidad de su implementación. Esto facilita la reutilización del código, ya que las clases pueden implementar múltiples interfaces y heredar funcionalidad de interfaces sin la necesidad de heredar de una clase base específica.

. Cuando se desea lograr un bajo acoplamiento entre componentes: El uso de interfaces reduce el acoplamiento entre componentes del sistema, ya que las clases dependen de las interfaces en lugar de depender directamente de otras clases concretas. Esto hace que el código sea más flexible y fácil de mantener.

. Cuando se quiere permitir la creación de implementaciones personalizadas: Las interfaces permiten que múltiples clases proporcionen diferentes implementaciones de un conjunto común de métodos. Esto es útil cuando se necesita soportar diferentes comportamientos o estrategias en una aplicación.

En resumen, se recomienda usar interfaces cuando se busca definir contratos comunes, promover la reutilización del código, reducir el acoplamiento entre componentes y permitir la creación de implementaciones personalizadas de funcionalidad.

* 1. **¿Cuál es la diferencia entre una interfaz y una clase abstracta?**

La diferencia principal entre una interfaz y una clase abstracta radica en su naturaleza y en cómo se utilizan en el diseño de software:

Interfaz:

-Una interfaz en Java define un contrato que las clases pueden implementar.

-Solo puede contener métodos abstractos (sin cuerpo) y constantes (variables finales).

-No puede contener campos de datos ni implementaciones de métodos.

-Permite a una clase implementar múltiples interfaces.

-Proporciona una separación clara entre la especificación de una funcionalidad y su implementación.

-Se utiliza principalmente para definir comportamientos comunes que pueden ser compartidos por diferentes clases.

Clase abstracta:

-Una clase abstracta es una clase que no puede ser instanciada directamente y puede contener tanto métodos abstractos como concretos.

-Puede contener campos de datos, métodos abstractos, métodos concretos y constructores.

-Puede proporcionar cierta implementación por defecto para los métodos.

-Una clase puede extender solo una clase abstracta.

-Se utiliza cuando se desea proporcionar una implementación base común para un conjunto de clases relacionadas, pero no necesariamente idénticas.

-Puede tener métodos abstractos (sin cuerpo) y métodos concretos (con cuerpo).

En resumen, mientras que una interfaz define un contrato que las clases pueden implementar, una clase abstracta proporciona una implementación parcial y puede contener una mezcla de métodos abstractos y concretos, así como campos de datos. La elección entre usar una interfaz o una clase abstracta depende del diseño y los requisitos específicos de la aplicación.

* 1. **¿Qué tipos de relaciones pueden existir entre clases y objetos?**

Existen varios tipos de relaciones que pueden existir entre clases y objetos en el contexto de la programación orientada a objetos:

Asociación:

-Es una relación entre dos clases en la que un objeto de una clase está relacionado con uno o más objetos de otra clase.

-Puede ser unidireccional o bidireccional.

-Puede ser de cardinalidad uno a uno, uno a muchos o muchos a muchos.

Agregación:

-Es una relación de tipo "todo-parte" donde un objeto de una clase es parte de un objeto de otra clase.

-El objeto parte puede existir independientemente del objeto todo.

-Se representa por una línea de flecha con un rombo blanco en el extremo del objeto todo.

Composición:

-Es una relación de tipo "todo-compuesto" donde un objeto de una clase está compuesto por uno o más objetos de otra clase.

-El objeto compuesto no puede existir independientemente del objeto todo.

-Se representa por una línea de flecha con un rombo negro en el extremo del objeto todo.

Herencia:

-Es una relación en la que una clase (subclase) hereda atributos y métodos de otra clase (superclase).

-La subclase puede añadir nuevos atributos y métodos o redefinir los existentes.

-Se representa por una línea de flecha con una punta de flecha en la clase hija apuntando a la clase padre.

Dependencia:

-Es una relación en la que una clase (cliente) utiliza directa o indirectamente los servicios de otra clase (proveedor).

-Puede ser temporal o permanente.

-Se representa por una línea de flecha punteada desde la clase cliente hacia la clase proveedora.

Estas son las relaciones más comunes que pueden existir entre clases y objetos en un sistema orientado a objetos, y cada una de ellas tiene sus propias características y se utiliza en diferentes contextos de diseño de software.

* 1. **¿Cómo se representa cada relación en un diagrama UML?**

Asociación:

-Se representa con una línea sólida que conecta las clases involucradas, generalmente con una flecha en uno o ambos extremos para indicar la dirección de la asociación.

-Se pueden añadir etiquetas en las líneas para especificar el tipo y la multiplicidad de la asociación.

Agregación:

-Se representa con una línea de conexión sólida que también puede tener una flecha en uno de sus extremos.

-En el extremo del objeto todo (clase contenedora), se dibuja un rombo blanco.

-La multiplicidad puede ser especificada con etiquetas en la línea de conexión.

Composición:

-Similar a la agregación, se representa con una línea de conexión sólida, posiblemente con una flecha en uno de sus extremos.

-En el extremo del objeto todo (clase contenedora), se dibuja un rombo negro.

-Al igual que con la agregación, la multiplicidad puede ser especificada con etiquetas en la línea de conexión.

Herencia:

-Se representa con una línea de conexión sólida con una flecha dirigida desde la subclase hacia la superclase.

-Puede haber una etiqueta indicando la relación de herencia.

Dependencia:

-Se representa con una línea de conexión punteada desde la clase que depende hacia la clase de la que depende.

-No hay símbolos adicionales en los extremos de la línea, simplemente una línea punteada.

Estas representaciones son estándar en los diagramas de clases UML y son utilizadas para visualizar las relaciones entre las clases en un sistema de software.

* 1. **¿Cuándo es recomendable usar cada tipo de relación?**

Asociación:

-Se utiliza para representar una relación estructural entre dos clases, donde una clase conoce o utiliza a la otra.

-Es recomendable cuando dos clases tienen una relación débil o temporal, y ninguna de ellas es parte integral de la otra.

Agregación:

-Se utiliza para representar una relación en la que una clase (objeto todo) contiene a otra clase (objeto parte), pero las partes pueden existir independientemente de la clase contenedora.

-Es recomendable cuando una clase contiene a otra clase como parte de su estructura, pero las partes pueden existir sin la clase contenedora.

Composición:

-Se utiliza para representar una relación en la que una clase (objeto todo) contiene a otra clase (objeto parte), pero las partes no pueden existir sin la clase contenedora.

-Es recomendable cuando una clase contiene a otra clase como parte de su estructura, y las partes no tienen sentido sin la clase contenedora.

Herencia:

-Se utiliza para representar una relación entre una clase base (superclase) y una clase derivada (subclase), donde la subclase hereda características y comportamientos de la superclase.

-Es recomendable cuando una clase comparte atributos y métodos con otra clase, pero tiene sus propios atributos y métodos adicionales.

Dependencia:

-Se utiliza para representar una relación en la que una clase (cliente) depende de otra clase (proveedor) en algún aspecto.

-Es recomendable cuando una clase utiliza los servicios de otra clase temporalmente, como cuando un método de una clase toma un parámetro de otra clase.

La elección del tipo de relación adecuado depende del contexto y de la naturaleza de la relación entre las clases en el sistema de software que estás diseñando.

**A CODIFICAR!**

1. Desarrollar un sistema para gestionar una tienda de mascotas que vende diferentes tipos de animales: perros, gatos, aves.
   1. Clases:
      1. Animal: nombre (String), edad (int), tipoAnimal (String), precio (double). Método Alimentar(String nombre). Sobreescribir el método toString() para que devuelva una pequeña presentación del animal.
      2. Perro: raza (String), vacunado (boolean). Hereda de Animal.
      3. Gato: raza (String), esterilizado (boolean). Hereda de Animal.
      4. Ave: especie (String), habla(boolean). Hereda de Animal.
      5. Crear también una clase Pez con los atributos que creas necesarios. Hereda de Animal.

Desarrollá las clases especificadas anteriormente y crea una nueva clase TiendaMascotas para gestionar todas las demás. Esta debe tener los métodos:

* addAnimal(Animal animal): Agrega un animal a la tienda.
* venderAnimal(String nombre): Vende un animal por nombre.
* alimentarAnimal(String nombre): Alimenta a un animal por nombre.
* listarAnimales(): Lista todos los animales de la tienda.

En el main crea una tienda, un animal de cada clase y utilizá todos los métodos definidos.

1. Desarrollar un sistema de gestión para una biblioteca.
   1. Definir una clase Libro con los atributos título (String), autor (String), añoDePublicacion (int), prestado (boolean). Podés crear más atributos si lo considerás necesario.
   2. Crea una interfaz llamada Prestable con los métodos prestar() y devolver().
   3. Libro debe implementar esta interfaz y se deben definir los métodos para cambiar el estado del libro.
   4. Crear las clases Novela, Universitario, Infantil que hereden de Libro. Agregá a cada clase un atributo propio (o más, si te parece necesario).
   5. Definir la clase Biblioteca, que contenga una lista de libros y métodos para agregar un libro, listar los libros disponibles, prestar un libro, devolver un libro.

En el main crea una biblioteca, un libro de cada clase y utilizá todos los métodos definidos.