**Trabajo Práctico 2: Programación II**

**Parte teórica**

**Renzo Di Laudo – Comisión 3**

**1. ¿Qué es la herencia en java? ¿Cuáles son sus beneficios?**

La herencia en Java es un concepto que te permite crear nuevas clases basadas en clases ya existentes. La nueva clase hereda automáticamente los campos y métodos de la clase existente, y luego puedes agregar nuevos campos y métodos o modificar los existentes según sea necesario.

Los beneficios de la herencia en Java son:

**Reutilización de código:** No necesitas volver a escribir el código que ya está en la clase base. Puedes simplemente heredarlo y luego agregar nuevas funcionalidades si es necesario.

**Mantenimiento:** Si necesitas realizar cambios en la funcionalidad común a varias clases, solo necesitas hacerlo en la clase base, y los cambios se propagarán automáticamente a todas las clases que heredan de ella.

**Organización:** La herencia te permite organizar tus clases de manera jerárquica, lo que hace que tu código sea más estructurado y fácil de entender.

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

Se representa con la relación denominada “generalización”, representada por una flecha de punta blanca que sale de la clase hija y apunta a la clase padre.

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

Hay varias situaciones en las que es útil la herencia.

**Cuando existe una relación de "es un"**: Si puedes decir que una clase es un tipo específico de otra clase, es probable que la herencia sea adecuada. Por ejemplo, "Perro" es un tipo de "Animal".

**Para reutilizar código común**: Cuando varias clases tienen características o comportamientos comunes, puedes colocar ese código común en una clase base y luego hacer que otras clases hereden de ella. Esto evita la duplicación de código y facilita el mantenimiento.

**Para extender la funcionalidad de una clase existente**: Si necesitas agregar nuevas características o comportamientos a una clase existente sin modificar su comportamiento original, la herencia te permite hacerlo de manera limpia y estructurada.

**Para establecer una jerarquía de clases**: La herencia te permite organizar tus clases en una jerarquía, lo que puede hacer que tu código sea más fácil de entender y mantener.

**4. ¿Qué es una interfaz en java?**

Una interfaz es una colección de declaraciones de métodos (sin definirlos) y también puede incluir constantes, pero no variables. El papel de una interfaz es describir algunas de las características de una clase, por ejemplo: una persona que sea futbolista no define su personalidad completa, pero si tiene ciertas características que lo distinguen de otras. El polimorfismo hace que las interfaces sean aplicables a clases que no tienen nada que ver unas con otras.

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

Se representa con la relación denominada “realización”, representada por una flecha punteada y punta blanca que sale de la clase que implementa y apunta a la interfaz.

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

Las interfaces en Java son recomendables en varias situaciones:

**Cuando necesitas definir un contrato**: Las interfaces te permiten definir un conjunto de métodos que una clase debe implementar. Esto es útil cuando quieres establecer un contrato común para varias clases, pero no necesariamente quieres imponer una jerarquía de herencia.

**Para lograr la abstracción**: Las interfaces te permiten abstraer el comportamiento común de diferentes clases. Por ejemplo, si varias clases tienen métodos similares pero no comparten una relación de herencia, puedes definir una interfaz común para esos métodos.

**Para lograr la modularidad y flexibilidad**: Las interfaces permiten que una clase implemente múltiples interfaces, lo que ofrece una mayor flexibilidad en el diseño de tu aplicación. Esto te permite adaptarte a diferentes escenarios y requisitos sin tener que depender de una única jerarquía de clases.

**Para facilitar la interoperabilidad y la programación orientada a interfaces**: Las interfaces son ampliamente utilizadas en Java y en muchos frameworks y bibliotecas. Al programar orientado a interfaces, puedes escribir código que dependa de las interfaces en lugar de implementaciones concretas, lo que hace que tu código sea más flexible y fácil de mantener.

**Cuando necesitas implementar múltiples herencias**: Java no permite la herencia múltiple de clases, pero sí permite que una clase implemente múltiples interfaces. Esto es útil cuando necesitas que una clase tenga comportamientos de varias fuentes diferentes.

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

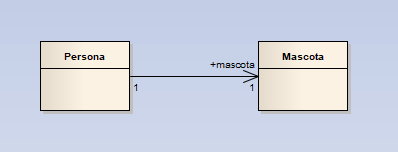
Una interfaz es simplemente una lista de métodos no implementados, además puede incluir la declaración de constantes. Una clase abstracta puede incluir métodos implementados y no implementados o abstractos, miembros dato constantes y otros no constantes. La diferencia es más profunda en java, ya que no permite la herencia múltiple de clases, pero si permite la implementación de varias interfaces

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

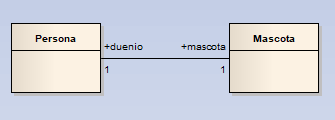
Pueden existir 3 tipos de relaciones: asociación (que a su vez puede ser unidireccional o bidireccional), agregación y composición.

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

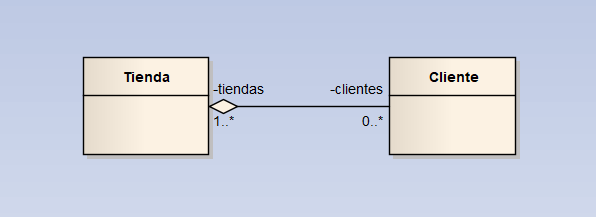
**Asociación unidireccional**



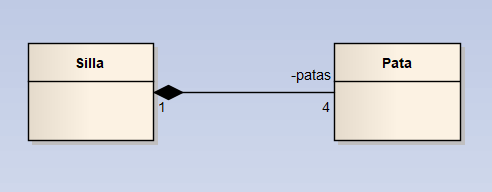
**Asociación bidireccional**

****

**Agregación**

****

**Composición**

****

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

**Asociación unidireccional:** En una asociación unidireccional, una clase (llamada clase origen) tiene una referencia a otra clase (llamada clase destino), pero la clase destino no tiene una referencia de vuelta a la clase origen. Esto es útil cuando una clase necesita acceder a otra clase, pero la segunda clase no necesita conocer sobre la primera. Por ejemplo, en un sistema de gestión de empleados, una clase "Departamento" podría tener una lista de empleados, pero los empleados no necesitan conocer sobre el departamento al que pertenecen.

**Asociación bidireccional:** En una asociación bidireccional, dos clases tienen referencias mutuas entre sí. Esto es útil cuando necesitas que ambas clases se comuniquen entre sí. Por ejemplo, en un juego de ajedrez, una clase "Jugador" puede tener una referencia al "Tablero" y viceversa, ya que ambos necesitan interactuar entre sí durante el juego.

**Agregación:** La agregación es una relación en la que una clase (llamada clase contenedora) contiene referencias a otras clases (llamadas clases contenidas), pero las clases contenidas pueden existir independientemente de la clase contenedora. Esto es útil cuando una clase necesita contener otras clases, pero las clases contenidas pueden ser utilizadas por otras partes del sistema. Por ejemplo, en un sistema de reservas de vuelos, una clase "Aerolínea" puede contener una lista de "Vuelos", pero los vuelos también pueden ser reservados por otras clases como "Clientes".

**Composición:** La composición es una forma más fuerte de agregación, donde las clases contenidas son partes integrales de la clase contenedora y no pueden existir independientemente de ella. Esto es útil cuando una clase necesita contener otras clases y es responsable de su ciclo de vida. Por ejemplo, en un juego de video, una clase "Juego" puede contener una instancia de la clase "Nivel", y el nivel solo existe dentro del contexto del juego y se crea y destruye junto con él.