CURSO DE PROGRAMACIÓN FULL STACK

HERENCIA

PARADIGMA ORIENTADO A OBJETOS

GUÍA DE HERENCIA

HERENCIA

La herencia es una relación fuerte entre dos clases donde una clase es padre de otra.
La herencia es un pilar importante de POO. Es el mecanismo mediante el cual una clase es capaz de heredar todas las características (atributos y métodos) de otra clase.

Las propiedades comunes se definen en la superclase (clase padre) y las subclases heredan estas propiedades (Clase hija). En esta relación, la frase “Un objeto es un-tipo-de una superclase” debe tener sentido, por ejemplo: un perro es un tipo de animal, o también, una heladera es un tipo de electrodoméstico.

La herencia apoya el concepto de "reutilización", es decir, cuando queremos crear una nueva clase y ya existe una clase que incluye parte del código que queremos, podemos utilizar esa clase que ya tiene el código que queremos y hacer de la nueva clase una subclase. Al hacer esto, estamos reutilizando los campos y métodos de la clase existente.

La manera de usar herencia es a través de la palabra **extends**.

```
class subclase extends superclase {
   // atributos y métodos
}
```

Herencia y atributos

La subclase (Hija) como hemos dicho recibe todos los atributos de la superclase (Madre), y además la subclase puede tener atributos propios.

```
Class Persona {
protected String nombre;
protected Integer edad;
protected Integer documento;
}
Class Alumno extends Persona {
private String materia;
}
```

El siguiente programa crea una superclase llamada Persona, que crea personas según su nombre, edad y documento, y una subclase llamada Alumno, que recibe todos los atributos de Persona. De esta manera se piensa que los atributos de alumno son nombre, edad y documento, que son propios de cualquier Persona y matera que sería especifico de cada Alumno. Usualmente la superclase suele ser un concepto muy general y abstracto, para que pueda utilizarse para varias subclases.

En la superclase podemos observar que los atributos están creados con el modificador de acceso protected y no private. Esto es porque el modificador de acceso protected permite que las subclases puedan acceder a los atributos sin la necesidad de getters y setters.

Los atributos se trabajan como protected también, porque una subclase no hereda los miembros privados de su clase principal. Sin embargo, si la superclase tiene métodos públicos o protegidos (como getters y setters) para acceder a sus campos privados, estos también pueden ser utilizados por la subclase. Entonces, para evitar esto, usamos atributos protected.

Visibilidad	Public	Private	Protected	Default
Desde la misma Clase	SI	SI	SI	SI
Desde cualquier Clase del mismo Paquete	SI	NO	SI	SI
Desde una Subclase del mismo Paquete	SI	NO	SI	SI
Desde una Subclase fuera del mismo Paquete	SI	NO	SI, a través de la herencia	NO
Desde cualquier Clase fuera del Paquete	SI	NO	NO	NO

Herencia y Constructores

Una diferencia entre los constructores y los métodos es que los constructores no se heredan, pero los métodos sí. Todos los constructores definidos en una superclase pueden ser usados desde constructores de las subclases a través de la palabra clave super. La palabra clave super es la que me permite elegir qué constructor, entre los que tiene definida la clase padre, es el que debo usar. Si la superclase tiene definido el constructor vacío y no colocamos una llamada explícita super, se llamará el constructor vacío de la superclase.

```
Class Persona {

public Persona(String nombre, Integer edad, Integer documento){

this.nombre = nombre;

this.edad = edad;

this.documento = documento;

      }

}

Class Alumno extends Persona {

public Alumno(String materia, String nombre, Integer edad, Integer documento){

super.(nombre, edad, documento);

this.materia = materia;

      }

}
```

En el ejemplo podemos ver que el constructor de la clase Alumno utiliza la palabra clave super para llamar al constructor de la superclase y de esa manera utilizarlo como constructor propio y además sumarle su atributo materia.

La palabra clave super nos sirve para hacer referencia o llamar a los atributos, métodos y constructores de la superclase en las subclases.

```
super.atributoClasePadre;

super.metodoClasePadre;
```

Herencia y Métodos

Todos los métodos accesibles o visibles de una superclase se heredan a sus subclases. Pero, ¿qué ocurre si una subclase necesita que uno de sus métodos heredados funcione de manera diferente?

Los métodos heredados pueden ser redefinidos en las clases hijas. Este mecanismo se lo llama sobreescritura. La sobreescritura permite que las clases hijas sumen sus particulares en torno al funcionamiento y agrega coherencia al modelo. Esto se logra poniendo la anotación @Override arriba del método que queremos sobreescribir, el método debe llamarse igual en la subclase como en la superclase.

```
Class Persona {

public void codear(){

System.out.println(“Una persona comun no codea”);

      }

}

Class Alumno extends Persona {

@Override
public void codear(){

System.out.println(“Está aprendiendo”);

      }

}
```

En el ejemplo podemos ver que tenemos el mismo método en la clase Persona, que en la clase Alumno, el método nos va a informar cuales son sus capacidades para codear según la clase que llamemos. Por lo que en la clase Alumno, cuando heredamos el método lo sobreescribimos para cambiar su funcionamiento, y hacer que diga algo distinto al método de Persona.

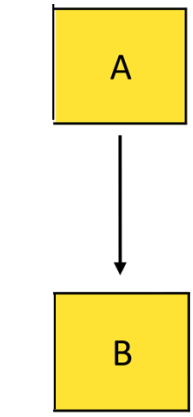
En los métodos de la superclase, también podemos hacer que tengan un modificador de acceso protected, esto hace que los únicos que puedan invocar a ese método sean las subclases.

Polimorfismo

El término polimorfismo es una palabra de origen griego que significa "muchas formas". Este término se utiliza en POO para referirse a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos, es decir, que la misma operación se realiza en las clases de diferente forma. Estas operaciones tienen el mismo significado y comportamiento, pero internamente, cada operación se realiza de diferente forma. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía. Esto hace referencia a la idea de que podemos tener un método definido en la superclase y que las subclases tengan el mismo método, pero con distintas funcionalidades

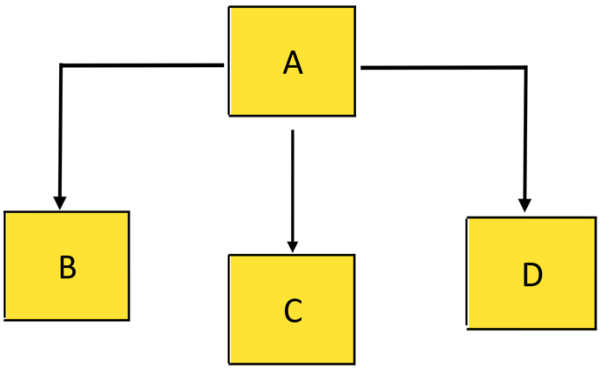
TIPOS DE HERENCIA

Herencia única: en la herencia única, las subclases heredan las características de solo una superclase. En la imagen a continuación, la clase A sirve como clase base para la clase derivada B.



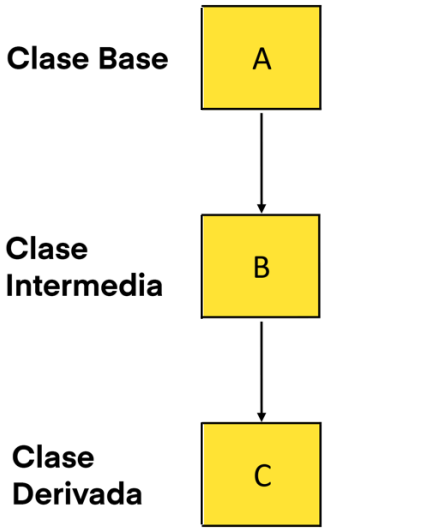
Herencia Simple

Herencia Jerárquica: en la herencia jerárquica, una clase sirve como una superclase (clase base) para más de una subclase. En la imagen inferior, la clase A sirve como clase base para la clase derivada B, C y D.



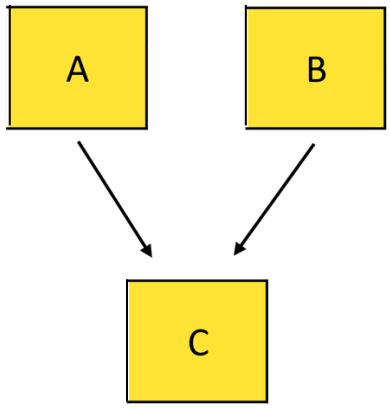
Herencia Jerárquica

Herencia Multinivel: en la herencia multinivel, una clase derivada heredará una clase base y, además, la clase derivada también actuará como la clase base de otra clase. En la imagen inferior, la clase A sirve como clase base para la clase derivada B, que a su vez sirve como clase base para la clase derivada C. En Java, una clase no puede acceder directamente a los miembros de los "abuelos".



Herencia Multinivel

Herencia Múltiple (a través de interfaces): en Herencia múltiple, una clase puede tener más de una superclase y heredar características de todas las clases principales. Tenga en cuenta que Java no admite herencia múltiple con clases. En Java, podemos lograr herencia múltiple **solo a través de Interfaces**. En la imagen a continuación, la Clase C se deriva de la interfaz A y B.



Herencia Multiple

MODIFICADORES DE CLASES Y METODOS

Clases Finales

El modificador final puede utilizarse también como modificador de clases. Al marcar una clase como final impedimos que se generen hijos a partir de esta clase, es decir, cortamos la jerarquía de herencia.

```
public final class Animal{ }
```

Métodos Finales

El modificador final puede utilizarse también como modificador de métodos. La herencia nos permite reutilizar el código existente y uno de los mecanismos es la crear una subclase y sobrescribir alguno de los métodos de la clase padre. Cuando un método es marcado como final en una clase, evitamos que sus clases hijas puedan sobrescribir estos métodos.

```
public final void método(){ }
```

Clases Abstractas

En java se dice que una clase es abstracta cuando no se permiten instancias de esa clase, es decir que no se pueden crear objetos. Nosotros haríamos una clase abstracta por dos razones. Usualmente las clases abstractas suelen ser las superclases, esto lo hacemos porque creemos que la superclase o clase padre, no debería poder instanciarse. Por ejemplo, si tenemos una clase Animal, el usuario no debería poder crear un Animal, sino que solo debería poder instanciar solo objetos de las subclases

```
public abstract class Animal { }
```

Otra razón es porque decidimos hacer métodos abstractos en nuestra superclase. Cuando una clase posee al menos un método abstracto esa clase necesariamente debe ser marcada como abstracta.

Métodos Abstractos

Un método abstracto es un método declarado, pero no implementado, es decir, es un método del que solo se escribe su nombre, parámetros, y tipo devuelto, pero no su código de implementación. Estos métodos se heredan y se sobreescriben por las clases hijas quienes son las responsables de implementar sus funcionalidades. ¿Qué utilidad tiene un método abstracto? Podemos ver un método abstracto como una palanca que fuerza dos cosas: la primera, que no se puedan crear objetos de una clase. La segunda, que todas las subclases sobreescriban el método declarado como abstracto.

```
public abstract void codear();

Hija:

@Override
Public void codear(){

System.out.println(“Está aprendiendo”);

}
```

INTERFACES

Una interfaz en Java es una clase que contiene todos sus métodos abstractos y sus atributos son sólo constantes. En las interfaces se especifica qué se debe hacer, pero no su implementación o funcionalidad. Las clases que implementen una interface serán las responsables de describir la lógica del comportamiento de los métodos, las interfaces se implementan usando la palabra **implements**. La principal diferencia entre interface y abstract es que una interface proporciona un mecanismo de encapsulación de los métodos sin forzar a la utilización de una herencia.

```
public interface Interfaz {

public final constante = 10;

public void metodo();

}

public class Clase implements Interfaz {

@Override
Public void metodo(){

System.out.println(“Implementacion del método”);

System.out.println(“La constante tiene un valor de ” + constante);

      }

}
```

PREGUNTAS DE APRENDIZAJE

1) ¿Qué palabra se usa para generar la herencia entre clases?

   a) implements
   b) super
   c) this
   d) extends

2) ¿Qué código de los siguientes tiene que ver con la herencia?

   a) public class Componente extends Producto
   b) public class Componente inherit Producto
   c) public class Componente implements Producto
   d) public class Componente belong to Producto

3) ¿La superclase es la clase?

   a) Madre
   b) Hija
   c) Nieta
   d) Es una interfaz

4) Cual de estos componentes de la superclase no hereda la subclase

   a) Atributos
   b) Métodos
   c) Getter/Setters
   d) Constructores

5) La superclase debe ser abstracta

   a) Siempre
   b) Nunca
   c) Depende de la situación
   d) Cuando implementa una interfaz

6) ¿Cuál tipo de clase debe tener al menos un método abstracto?

   a) final
   b) abstract
   c) public
   d) Todas las anteriores

7) ¿De cuántas clases se puede derivar Java?

   a) Tres clases
   b) Dos clases
   c) Una clase
   d) Cinco clases

8) Una clase que termina la cadena de una herencia se la puede declarar como:

a) final
b) abstract
c) public
d) Ninguna de las anteriores

9) ¿Qué palabra se usa para implementar una interfaz?

a) implements
b) super
c) this
d) extends

10) ¿Qué código asociarías a una Interfaz en Java?

a) public class Componente interface Product
b) Componente cp = new Componente (interfaz)
c) public class Componente implements Printable
d) Componente cp = new Componente.interfaz