ULP Virtual

Tecnicatura Universitaria en Desarrollo de Software

Programación II

Guía 4

Herencia

Universidad de LA PUNTA

GOBIERNO DE SAN LUIS

HERENCIA:

La herencia es una relación fuerte entre dos clases donde una clase es padre o madre 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” 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 int edad;
        protected int documento;
}
Class Alumno extends Persona {
        private String materia;
}
```

El siguiente programa crea una superclase llamada Persona, la clase 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 materia 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, int edad, int documento){

        this.nombre = nombre;

        this.edad = edad;

        this.documento = documento;
}
}
```

```
Class Alumno extends Persona {
public Alumno(String materia, String nombre, int edad, int  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 cuáles 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.

Consideraciones especiales de la sobreescritura:

- El método sobreescrito tiene que tener el *mismo nombre, número y tipo de parámetros (firma)* y tipo de retorno que el método que sobreescribe.
- El método sobreescrito puede retornar un subtipo del tipo retornado por el método que sobreescribe. Esto es llamado tipo de retorno covariante.
- La especificación de acceso para los métodos sobreescritos no debe ser más restrictiva que el método que sobreescribe.
- No puedo sobreescribir un método **final** o **private.**

NO CONFUNDIR ASERRIN CON PAN RALLADO!!!

En la Guía 1, incorporamos el concepto de sobrecarga, haciendo un poco de memoria, decíamos de la sobrecarga: “que puede haber dos métodos en una clase, con el mismo nombre que realicen dos funciones distintas.”

Por lo tanto, no debemos confundir sobrecarga de métodos con sobreescritura; la sobrecarga (overloading) y la sobrescritura (overriding) son dos conceptos diferentes pero relacionados en Java.

La sobrecarga se refiere a la capacidad de una clase de tener múltiples métodos con el mismo nombre pero con diferentes parámetros. Es decir, la sobrecarga permite tener dos o más métodos con el mismo nombre pero con diferentes tipos de parámetros o número de parámetros. En otras palabras, en la sobrecarga se crea un método adicional con el mismo nombre pero con una lista de argumentos diferente.

Por ejemplo, imagina que tienes una clase llamada "Calculadora" y quieres que tenga diferentes métodos para sumar dos números. Puedes crear dos métodos con el mismo nombre "sumar", pero con diferentes tipos de parámetros:

```java
public int sumar(int a, int b) {
   return a + b;
}

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

La sobrecarga permite que la clase "Calculadora" tenga dos métodos "sumar", uno para sumar enteros y otro para sumar números con decimales.

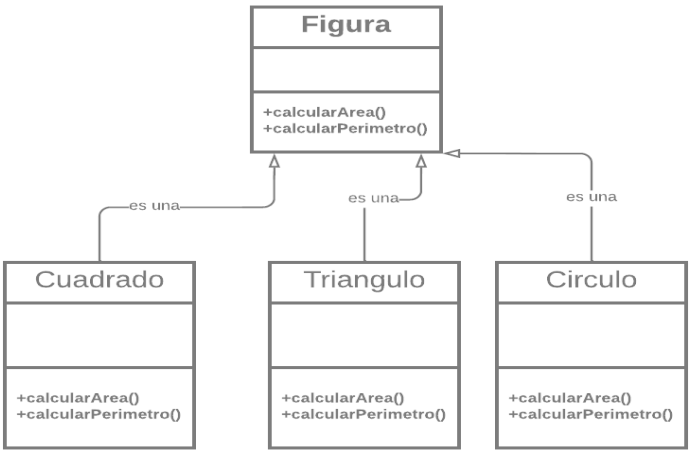
En resumen, la sobrecarga se refiere a tener múltiples métodos con el mismo nombre pero con diferentes parámetros, mientras que la sobrescritura se refiere a la capacidad de una subclase de proporcionar su propia implementación de un método que ya está definido en la clase padre.

Ver Video

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.

Por Ejemplo:



Como un Cuadrado es una Figura, es decir, Cuadrado hereda de Figura, podríamos asignar a una variable de tipo Figura (superclase) un Cuadrado (subclase);

Figura fig=new Cuadrado();

Si bien la variable "fig" es de tipo Figura, apunta a una instancia de tipo Cuadrado y si hacemos:

fig.calcularArea();

Se ejecutará el comportamiento de calcularArea() de Cuadrado, ya que en tiempo de ejecución estará apuntando a una instancia de este.

Si luego, a la variable "fig" le asignamos un Circulo y hacemos lo mismo:

fig= new Circulo();

fig.calcularArea();

Ahora el comportamiento de calcularArea() que se ejecutará es el de Circulo, ya que en tiempo de ejecución está apuntando a un objeto de tipo Circulo.

Entonces podemos concluir que, cuando ante un mismo mensaje "calcularArea" el comportamiento que se ejecuta es el de la instancia a la que está apuntando la variable de superclase Figura, lo llamaremos polimorfismo.

El operador instanceof

Como vimos anteriormente, en una variable de una superclase, podemos asignarle una instancia de una subclase o clase hija y por lo tanto podríamos querer saber de qué tipo es la instancia que guarda dicha variable; para ello tenemos un operador booleano llamando **instanceof**. Su sintaxis es la siguiente:

referencia **instanceof** Clase

Por ejemplo:

Figura fig=new Triangulo();

fig instanceof Triangulo // retornará true.

MODIFICADORES DE CLASES Y MÉTODOS 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 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 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 sobrescriben 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 es sintácticamente similar a una clase abstracta, en la que puede especificar uno o más métodos que no tienen cuerpo. Esos métodos deben ser implementados por una clase para que se definan sus acciones.

Por lo tanto, una interface especifica qué se debe hacer, pero no cómo hacerlo. Una vez que se define una interface, cualquier cantidad de clases puede implementarla. Además, una clase puede implementar cualquier cantidad de interfaces.

Para implementar una interfaz, una clase debe proporcionar cuerpos (implementaciones) para los métodos descritos por la interfaz. Cada clase es libre de determinar los detalles de su propia implementación. Dos clases pueden implementar la misma interfaz de diferentes maneras, pero cada clase aún admite el mismo conjunto de métodos.

Para decirle a Java que estamos trabajando sobre una interfaz vamos a tener que utilizar la palabra **interface**, esto se vería así:

```
public interface nombreInterfaz { }
```

Para una interfaz, el modificador de acceso es **public** o no se usa. Cuando no se incluye ningún modificador de acceso, los resultados de acceso serán los predeterminados y la interfaz solo están disponibles para otros miembros de su paquete. Si se declara como public, la interfaz puede ser utilizada por cualquier otra clase. El nombre de la interfaz puede ser cualquier identificador válido.

INSTANCIAR UNA INTERFAZ

Aunque las interfaces van a ser implementadas por clases y van a tener métodos, al igual que una clase abstracta, esta, no se va a poder instanciar. La definición de un interfaz no tiene constructor, por lo que no es posible invocar el operador new sobre un tipo interfaz, por lo que no podemos crear objetos del tipo interfaz.

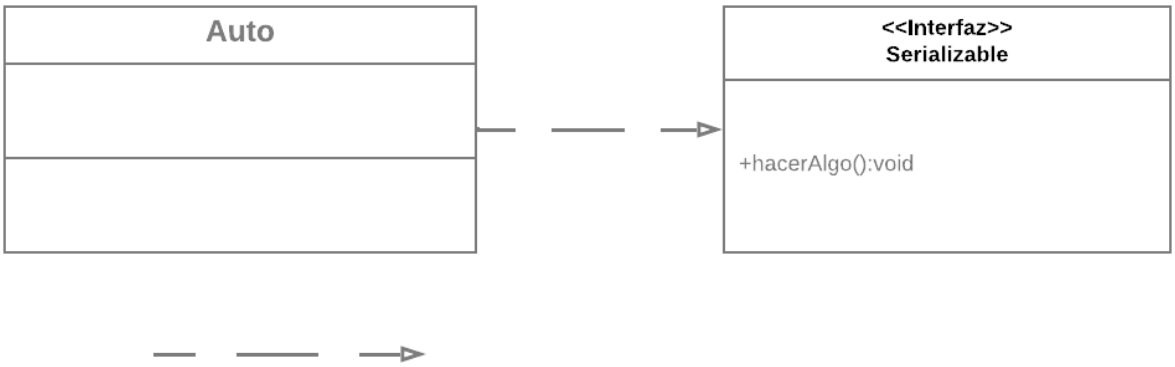
IMPLEMENTACIÓN DE INTERFACES

Una vez que se ha definido una interfaz, una o más clases pueden implementar esa interfaz. Para implementar una interfaz, incluya la palabra reservada **implements** en la definición de clase.

```
public class Clase implements Interfaz { }
```

Los métodos de la interfaz los podemos implementar en nuestra clase y se van a sobrescribir desde lainterfaz, métodos que recordemos, no tendrán cuerpo. Nuestra tarea será completar esos métodos que implementamos de la interfaz.

En UML para indicar que una clase implementa una interface lo haremos de la siguiente forma:



MÉTODOS

Los métodos de una interfaz se declaran utilizando solo su tipo de devolución y firma. Son, esencialmente, métodos abstractos. Por lo tanto, cada clase que incluye dicha interfaz debe implementar todos sus métodos. En una interfaz, los métodos son implícitamente públicos.

```
public interface Interfaz {
        public void metodo();
        public int sumar();
}
public class Clase implements Interfaz {
            @Override
            public void metodo(){
                System.out.println("Implementacion del método");
             }
            }
            @Override
            public int sumar(){
                int suma = 2 + 2;
                return suma;
             }
}
```

Como podemos ver en la interfaz teníamos dos métodos sin cuerpo y al implementar la interfaz en nuestra clase, los sobrescribimos y les dimos una funcionalidad a dichos métodos.

VARIABLES ATRIBUTOS:

Las variables declaradas en una interfaz no son variables de instancia. En cambio, son implícitamente **public y final**, además, deben inicializarse. Por lo tanto, son esencialmente constantes, recordemos que se escriben en mayúsculas para diferenciarlas de variables.

```
public interface Interfaz {
        public final int CONSTANTE = 10;
        public void metodo();
}
public class Clase implements Interfaz {
        @Override
        Public void metodo(){
        System.out.println("La constante tiene un valor de " + CONSTANTE);  }
}
```

Las constantes definidas en nuestra interfaz pueden ser llamadas en la clase, solo con el nombre y de esa manera podemos utilizar los valores definidos en la interfaz.

BIBLIOGRAFIA:

Christopher Alexander, “A Pattern Language”, 1978

Erich Gamma, “Design Patterns: Elements of Reusable OO Software”

Martin Fowler, “Analysis Patterns: Reusable Object Models”, Addison Wesley,

1997

Kathy Sierra, “OCA/OCP JAVA SE 7 PROGRAMMER I & II STUDY GUIDE (EXAMS 1Z0-803 & 1Z0-8”, McGraw Hill, 2014