Contenido

Repaso de POO en Java

```
package ClasesConncretas;

public class Ferrari extends Coche {

   int varMismoNombre=12;

  public Ferrari(String varPrivada1) {
     super(varPrivada1);
   }
/*
   public Ferrari() {
```

```
   }*/
//   static String varStatica;

 /*   @Override
   public int getVarMismoNombre() {
      return varMismoNombre;
   }

   @Override
   public void setVarMismoNombre(int varMismoNombre) {
      this.varMismoNombre = varMismoNombre;
   }
*/

}
```

```
package ClasesConncretas;

public class Coche {

   static String varStatica;
public int varPublica;

   private String varPrivada1;
private int varPrivada;

int varMismoNombre=23;

public Coche(String varPrivada1)
{
   this.varPrivada1=varPrivada1;

}
//public Coche()
//{}


  @Override
   public String toString() {
      return this.varPrivada1;
   }

   public int getVarPublica() {
      return varPublica;
   }

   public void setVarPublica(int varPublica) {
      this.varPublica = varPublica;
   }

   public int getVarPrivada() {
      return varPrivada;
   }

   public void setVarPrivada(int varPrivada) {
      this.varPrivada = varPrivada;
   }

   public int getVarMismoNombre() {
      return varMismoNombre;
   }
```

```
    public void setVarMismoNombre(int varMismoNombre) {
        this.varMismoNombre = varMismoNombre;
    }

    public static String metodoStatico()
    {

        return varStatica;
    }


}
```

Clases concretas y Herencia

Cuando tenemos clases que heredan de otras clases(extends) las clases hijas "heredan" el estado y comportamiento de la clase padre. Vamos a analizar algunas consideraciones:

- Por convención las variables de instancia deberían ser private y los métodos public.
- En el caso de que una clase hija tenga una variable de instancia con el mismo nombre que la clase padre, la clase hija accederá a esa nueva variable.
- Siguiendo el caso anterior, si la clase hija hereda el método GET de esa variable, ese método GET accederá a la variable heredada de la clase padre y no así a su propia variable.
- Siguiendo el caso anterior, si la clase hija sobreescribe el GET de esa variable de instancia, ahora sí el GET accederá a la variable de instancia hija.
- Cuando solo tenemos los constructores por defecto, no notamos el procedimiento de una clase hija al instanciarse que llama al constructor por defecto de la clase padre, pero sucede.
- Cuando en la clase padre solo tenemos un constructor con un parámetro, si quisiéramos instanciar un objeto de la clase hija con el constructor por defecto, estaríamos obligados en el constructor por defecto de la clase hija, colocar en la primer línea la palabra super(parámetros), llamando explícitamente al único constructor de la clase padre.

```
public static void main(String args[])
    {
        Ferrari varFerrari=new Ferrari("esteban");

            varFerrari.getVarMismoNombre();

System.out.println(varFerrari);



//1-------mismo nombree publicas
//System.out.println(varFerrari.varMismoNombre);
        //2------mismo nombree por el get
//System.out.println(varFerrari.getVarMismoNombre());
        //3-------variable static con valor en la hija y mostramos el padre
        Ferrari.varStatica="ferrari";
      //  System.out.println(Coche.varStatica);
        //accedo al meotod statico creado un objeto
        Ferrari.varStatica="ferrari";
```

```
        System.out.println(varFerrari.metodoStatico());
    }
}
```

Sobreescritura de métodos

- Los métodos heredados pueden sobreescribirse.
- Si queremos utilizar la definición del método de la clase padre, debemos usar la palabra super.
- Podemos sobreescribir algunos métodos de la clase Object(ejemplo toString).

This

- Cada objeto puede referenciarse a sí mismo mediante la palabra this, cuando se llama a un método no-static y a una variable no-static en this está implícito.
- Se puede usar la palabra this explícitamente en un método no-static.
- No se puede usar la palabra this explícitamente en un método static.
- Es importante usar la palabra this dentro de un método, cuando este tiene una variable local del mismo nombre que una variable de instancia, para referenciarla usamos el this.nombredelavariable.
- Es un error usar al this para invocar a un constructor desde otro constructor de la misma clase, salvo que este línea de código se la primer línea dentro del constructor que llama.
- Otro error es tratar de invocar desde un método a un constructor con la palabra this.

Variables static

- Una variable estática (static) en Java es una variable que pertenece a la clase en que fue declarada y se inicializa solo una vez al inicio de la ejecución del programa, la característica principal de este tipo de variables es que se puede acceder directamente con el nombre de la clase sin necesidad de crear un objeto, a continuación otros detalles que tiene una variable static en Java.
- Es una variable que pertenece a la clase y no al objeto.
- Las variables static se inicializan solo una vez, al inicio de la ejecución. Estas variables se inicializarán primero, antes de la inicialización de cualquier variable de instancia.
- No puedo declarar una variable static dentro de un método ya sea static o no.
- Si puedo acceder a una variable static de clase desde un método static.
- También puedo acceder a una variable de instancia static desde un objeto, aunque no se recomienda, de hecho los IDE, al poner el "." Luego del objeto, no van a mostrar esa variable de instancia.

- Cuando manejamos variables de instancia STATIC, sabemos que podemos acceder a esa variable con el nombre de la (clase."*la variable*").
- En el caso de tener una variable de instancia STATIC en la clase padre y modificamos el valor desde la clase hija, el cambio de estado será aceptado si accedemos como (Coche.var).
- En el caso de que la clase hija tenga la misma variable STATIC, si es modificado el estado de esta, al acceder como (Coche.var) el valor será null.

```
import ClasesConncretas.Ferrari;
public class Main2 {
public static void main(String args[])
   {
Coche.varStatic++;
      Coche  coche1 =new Coche();
coche1.varStatic++;
System.out.println(Coche.varStatic);
      System.out.println(Coche.devuelveVariableStaticPrivada());
   }
}
```

Métodos static

- Un método static en Java es un método que pertenece a la clase y no al objeto. Un método static solo puede acceder a variables o tipos de datos declarados como static.
- Un método static sólo puede acceder a datos static. No puede acceder a datos no static (variables de instancia).
- Un método static puede llamar solo a otros métodos static y no puede invocar un método no static a partir de él.
- Un método static se puede acceder directamente por el nombre de la clase y no se necesita crear un objeto para acceder al método (aunque se puede hacerlo).
- Un método static no puede hacer referencia a «this» o «super».
- Los métodos static no pueden sobreescribirse.
- Los métodos static no pueden ser abstracts, ya que no pueden sobreescribirse.

Bloque static

- El bloque static es un bloque de instrucción dentro de una clase Java que se ejecuta cuando una clase se carga por primera vez en la JVM. Básicamente un bloque static inicializa variables de tipo static dentro de una clase, al igual que los constructores ayudan a inicializar las variables de instancia, un bloque static inicializa las variables de tipo static en una clase.

```
static {  //claramente las variables deben ser static
                        var1 = 10;
                        var2 = 20;
            }
```

Clases finales

- Las clases finales tienen la característica que no pueden ser heredadas.
- String es una clase típica en JAVA final.
- Las clases Wrapper(Integer, Float, etc) son finales también.
- Una clase final tiene la característica de ser inmutable, es decir aquellas cuyas instancias no pueden ser modificadas una vez que su información ha sido definida. No habrá ninguna modificación a la misma durante su ciclo de vida.

Variables final

- Las variables finales son las constantes en JAVA, aunque existe la palabra reservada const, no se usa.
- Siguen el concepto de inmutabilidad.
- Por convención deben escribirse en mayúscula.
- Si una variable de instancia es final, las únicas dos formas de inicializarlas es directamente al momento de declararla, o también en la primer línea del o los constructores definidos en la clase.
- Una variable estática final en blanco se puede inicializar dentro de un bloque estático.
- Si la variable final es una referencia, esto significa que la variable no se puede volver a vincular para hacer referencia a otro objeto, pero el estado interno del objeto apuntado por esa variable de referencia se puede cambiar, es decir, los campos de ese objeto pueden modificarse, o si fuera una colección, los elementos de la colección se puden agregar, eliminar o modificar.
- En el caso de que la variable esté dentro de un método, solo se puede inicializar al momento de ser declarada o también por única vez posteriormente al ser declarada(esto es una gran diferencia con otros lenguajes de programación).

Métodos final

- Cuando se declara un método con la palabra clave final, se denomina método final. Un método final no puede ser anulado.
- En otras palabras éstos métodos no pueden ser sobreescritos.

```
package paqueteFinal;

public  class CocheFinal {

int var;

}
```

```
package paqueteFinal;
public class Main3 {
   public static void main(String args[])
   {
final  int VARIABLE_FINAL;
VARIABLE_FINAL=23;
Ferrari fer=new Ferrari();
fer.metodoFinal();
final CocheFinal coche =new CocheFinal();
//coche=new CocheFinal();
coche.var=89;
System.out.println(coche.var);
   }
}
```

Métodos abstractos

- Los métodos abstractos solo pueden ser declarados pero no implementados o definidos.

Los podemos encontrar en Interfaces o clases abstractas.

Clase Abstracta y Herencia

- Una clase abstracta es una clase concreta con la particularidad que puede tener métodos abstractos o no.
- Una clase abstracta no puede instanciarse, aunque existe el caso de clases anónimas generadas por interfaces o clases abstractas.
- Si una clase declara métodos abstractos, entonces sí, debe ser declarada como abstracta.
- Si tenemos una clase padre abstracta, se mantienen los casos descriptos en el apartado anterior de clases concretas y herencia.
- En el caso de que la clase padre abstracta tenga algún método abstracto, la clase hija deberá obligatoriamente sobreescribir a ese método.

Interfaz

- Todos los métodos de una interfaz son implícitamente public abstract.
- Podemos definir métodos con tipo de acceso private. A partir de java 9.
- Podemos definir un método predeterminado.

```
default String getAdmin(){
      return “hola”;
   }
```

- Las clases que implementan la interfaz no necesitan implementarlo.
- Las clases que implementan la interfaz podrían sobreescribirlo.
- Podemos definir métodos static, a partir de JAVA 8.

- Todas las variables y atributos de una interfaz son implícitamente constantes (public static final), no es necesario especificarlo en la declaración del mismo, esto quiere decir que no puede tener variables de instancia.
- Al ser declaradas las variables deben tener un valor asignado.
- El método predeterminado resuelve este problema suministrando una implementación que se usará si no se proporciona explícitamente otra implementación. Por lo tanto, la adición de un método predeterminado no causará la ruptura del código preexistente.
- Otra motivación para el método predeterminado fue el deseo de especificar métodos en una interfaz que son, esencialmente, opcionales, dependiendo de cómo se usa la interfaz. Por ejemplo, una interfaz puede definir un grupo de métodos que actúan sobre una secuencia de elementos.
- Una interfaz no puede heredar de otro elemento que no sea una interfaz.
- Una interfaz no puede implementar (implements) otra interfaz.
- Una interfaz debe ser declarada con la palabra clave interface.
- Los tipos de las interfaces pueden ser utilizados polimórficamente.
- Una interfaz puede ser public o package (valor por defecto).
- Los métodos toman como ámbito el que contiene la interfaz.
- Hay que decir finalmente que una interfaz también se puede heredar utilizando la palabra clave extends. La sintaxis es la misma que para la herencia de clases. Cuando una clase implementa una interfaz que hereda de otra interfaz, debe proporcionar las implementaciones para todos los métodos definidos en la cadena de herencia de la interfaz.

```
package pqueteAbstracto;
public interface InterfaceJAVA   {
   public abstract void metodoAbstracto();
   private void metodoPrivate()
   {
      System.out.println("MetodoPrivado");
   }

}
//en ambas columnas se cumple la interfaz funcional, ya que solo
hay un método abstracto.
```

```
package pqueteAbstracto;

public interface InterfaceJAVA2 {

   public abstract void
metodoAbstracto2();

   public static void metodoStatico()

   {
      System.out.println("metodo Statico");

   }
   default String getAdmin() {
      return "hola";
   }

   }
```

```
package pqueteAbstracto;

public interface InterfaceJAVA3 extends InterfaceJAVA2{



   public abstract void metodoAbstracto3();

}//acá vemos como una interface puede heredar de otra, tb se cumpliría en este caso la interfaz funcional
```

Clases anónimas

Una clase interna sin nombre y para la que solo necesitamos crear un único objeto se conoce como clase interna anónima de Java. Es útil cuando tenemos que crear una instancia del objeto cuando tenemos que hacer algunas cosas adicionales como sobrecargar métodos de una clase o una interfaz.

Podemos hacer esto usando la clase interna anónima de Java sin crear la subclase de la clase.

En otras palabras, una clase sin nombre se denomina clase interna anónima de Java. Por lo general, cada vez que creamos una nueva clase, esta tiene un nombre.

Por lo tanto, la clase interna anónima de Java no se utiliza para crear nuevas clases. Más bien, lo usamos para anular métodos de una clase o interfaz.

```
Demo d = new Demo()
{
public void demo_method()
{

//definition

}
};
```

Aquí, demo se refiere a una clase abstracta/concreta o una interfaz. Para comprender mejor el concepto de una clase interna anónima, veamos en qué se diferencia de una clase Java normal.

Podemos implementar cualquier cantidad de interfaces usando una clase normal, pero la clase interna anónima implementa solo una interfaz a la vez.

Una clase regular puede extender otra clase e implementar simultáneamente varias interfaces. Mientras que, por otro lado, una clase interna anónima solo puede hacer una de estas cosas a la vez.

El nombre del constructor es el mismo que el nombre de una clase. Implica que podemos escribir cualquier número de constructores en una clase regular, pero esto no es cierto para una clase interna anónima.

Las clases internas anónimas se clasifican en 3 tipos según la declaración y el comportamiento.

- Clase interna anónima que amplía una clase.
- Clase interna anónima que implementa una interfaz.
- La clase interna anónima se define dentro del argumento del método/constructor.

Clase interna anónima que amplía una clase en Java

Usamos la palabra clave new aquí para crear un objeto que hace referencia al tipo de clase principal. Mira el ejemplo de abajo.

```
class Demo {
public void example() {
System.out.println("Types of Anonymous classes");
}
}
public class AnonymousDemo {
public static void main(String args[]) {
Demo d1 = new Demo();
d1.example();
Demo d2 = new Demo() {
@Override
public void example() {
System.out.println("Type - 1");
}
};
d2.example();
}
}
```

Clase interna anónima que implementa una interfaz en Java

Usamos la palabra clave new para crear un objeto de clase interna anónimo que hace referencia a un tipo de interfaz.

```
interface Demo {
public void demo();
}
public class AnonymousDemo {
public static void main(String args[]) {
Demo d = new Demo() {
@Override
public void demo() {
System.out.println("Type - 2");
}
};
d.demo();
System.out.println(d.getClass().getName());
}
}
```

Clase interna anónima dentro de un argumento de método/constructor en Java

Aquí, usamos la clase interna anónima como argumento y la pasamos a métodos o constructores.

```
abstract class Demo {
public abstract void demo();
}
class Example {
public void example(Demo d) {
d.demo();
```

```
}
}
public class AnonymousDemo {
public static void main(String args[]) {
Example e = new Example();
e.example(new Demo() {
@Override
public void demo() {
System.out.println("Type - 3");
}
});
}
}
```

Interfaz Funcional

- Una interfaz funcional solo puede tener un método abstracto, pudiendo tener métodos por defecto, privados o estáticos.
- La anotación @FunctionalInterface no es obligatoria, pero comprueba en tiempo de compilación si se cumplen las condiciones detalladas en el punto anterior.

```
@FunctionalInterface
Interface Calcula{
public int suma(int a, int b);
}
Public class ImplementaCalcula implements Calcula


{
Public int suma(int a, int b)
{
return a+b;
}
}

Public class Prueba {
Public static void main(String[] args)
{
Calcula cal=new ImplementCalcula();
Cal.suma(2,3);

//podríamos haber hecho
Calcula cal1=new Calcula ()
{
@Override
Public int suma(int a,int b)
{
Return a+b;
}};
}
}


}
```

- No estamos creando una clase concreta, sino que estamos creando una clase anónima, una instancia de esa, que solo se va a instanciar una vez.
- Comparator es otro ejemplo, bastante conocido.

```
List<Persona> lista=new ArrayList<Persona>();

      Persona per=new Persona(1,4);
      Persona per1=new Persona(1,6);
      Persona per2=new Persona(1,2);

      lista.add(per);
      lista.add(per1);
      lista.add(per2);

//Collections.sort(lista);

      Collections.sort(lista,new Comparator<Persona>() {

         public int compare(Persona f1, Persona f2) {
            return f1.compareTo(f2);
         }
      });
for(Persona p: lista)
{
   System.out.println(p.edad);
}
   }
}
```

El mismo ejemplo del comparator anterior, utilizando una interfaz funcional, se podría haber desarrollado con expresiones lambda.

Vale aclarar que este caso de comparator, la forma tradicional sería, crear una clase que implemente Comparator, en esta clase sobreescribir el método compareTo y luego aplicar esta clase como parámetro del método sort.

```
      lista.stream().sorted(Comparator.comparing(Persona::getEdad)).forEach(System.out::println);
```