Conceptos y ejemplos de uso de this, super, Herencia, Subclases con ejemplos en codificación Java

this

Al acceder a variables de instancia de una clase, la palabra clave this hace referencia a los miembros de la propia clase en el objeto actual; es decir, this se refiere al objeto actual sobre el que está actuando un método determinado y se utiliza siempre que se quiera hace referencia al objeto actual de la clase. Volviendo al ejemplo de MiClase, se puede añadir otro constructor de la forma siguiente:

```
public class MiClase {
   int i;
   public MiClase() {
      i = 10;
      }
   // Este constructor establece el valor de i
   public MiClase( int valor ) {
      this.i = valor; // i = valor
      }
   // Este constructor también establece el valor de i
   public MiClase( int i ) {
      this.i = i;
      }
   public void Suma_a_i( int j ) {
      i = i + j;
      }
   }
```

Aquí **this**.**i** se refiere al entero **i** en la clase **MiClase**, que corresponde al objeto actual. La utilización de **this** en el tercer constructor de la clase, permite referirse directamente al objeto en sí, en lugar de permitir que el ámbito actual defina la resolución de variables, al utilizar **i** como parámetro formal y después **this** para acceder a la variable de instancia del objeto actual.

La utilización de **this** en dicho contexto puede ser confusa en ocasiones, y algunos programadores procuran no utilizar variables locales y nombres de parámetros

formales que ocultan variables de instancia. Una filosofía diferente dice que en los métodos de inicialización y constructores, es bueno seguir el criterio de utilizar los mismos nombres por claridad, y utilizar **this** para no ocultar las variables de instancia. Lo cierto es que es más una cuestión de gusto personal que otra cosa el hacerlo de una forma u otra.

La siguiente aplicación de ejemplo, java509.java, utiliza la referencia **this** al objeto para acceder a una variable de instancia oculta para el método que es llamado.

```
class java509 {
   // Variable de instancia
   int miVariable;

   // Constructor de la clase
   public java509() {
      miVariable = 5;
      }

   // Metodo con argumentos
   void miMetodo(int miVariable) {
      System.out.println( "La variable Local miVariable contiene "
         + miVariable );
      System.out.println(
         "La variable de Instancia miVariable contiene "
         + this.miVariable );
      }

   public static void main( String args[] ) {
      // Instanciamos un objeto del tipo de la clase
      java509 obj = new java509();
      // que utilizamos para llamar a su unico metodo
      obj.miMetodo( 10 );
      }
   }
```

super

Si se necesita llamar al método padre dentro de una clase que ha reemplazado ese método, se puede hacer referencia al método padre con la palabra clave **super**:

```
import MiClase;
public class MiNuevaClase extends MiClase {
   public void Suma_a_i( int j ) {
      i = i + ( j/2 );
      super.Suma_a_i( j );
   }
}
```

En el siguiente código, el constructor establecerá el valor de **i** a 10, después lo cambiará a 15 y finalmente el método *Suma_a_i()* de la clase padre **MiClase** lo dejará en 25:

```
MiNuevaClase mnc;
mnc = new MiNuevaClase();
mnc.Suma_a_i( 10 );
```

super es un concepto que no existe en C++, al menos no con una implementación similar a Java. Si un método **sobreescribe** un método de su superclase, se puede utilizar la palabra clave **superpara** eludir la versión **sobreescrita** de la clase e invocar a la versión original del método en la **supreclase.** Del mismo modo, se puede utilizar **super** para acceder a variables miembro de la superclase.

En el ejemplo java510.java, la aplicación utiliza **super** para referirse a una variable local en un método y a una variable de la superclase que tiene el mismo nombre. El programa también utiliza **super** para invocar al constructor de la superclase desde en constructor de la subclase.

Herencia

La ***herencia*** es el mecanismo por el que se crean nuevos objetos definidos en términos de objetos ya existentes. Por ejemplo, si se tiene la clase **Ave**, se puede crear la subclase **Pato**, que es una especialización de **Ave**.

```
class Pato extends Ave {
   int numero_de_patas;
```

```
}
```

La palabra clave **extends** se usa para generar una subclase (especialización) de un objeto. Un Pato es una subclase de Ave. Cualquier cosa que contenga la definición de Ave será copiada a la clase **Pato**, además, en Pato se pueden definir sus propios métodos y variables de instancia. Se dice que Pato deriva o hereda de Ave.

Además, se pueden sustituir los métodos proporcionados por la clase base. Utilizando nuestro anterior ejemplo de MiClase, aquí hay un ejemplo de una clase derivada sustituyendo a la función ***Suma_a_i()*:**

```
import MiClase;
public class MiNuevaClase extends MiClase {
   public void Suma_a_i( int j ) {
      i = i + ( j/2 );
      }
   }
```

Ahora cuando se crea una instancia de MiNuevaClase, el valor de **i** también se inicializa a 10, pero la llamada al método ***Suma_a_i()*** produce un resultado diferente:

```
MiNuevaClase mnc;
mnc = new MiNuevaClase();
mnc.Suma_a_i( 10 );
```

Java se diseñó con la idea de que fuera un lenguaje sencillo y, por tanto, se le denegó la capacidad de la herencia múltiple, tal como es conocida por los programadores C++. En este lenguaje se añade cierta complejidad sintáctica cuando se realiza herencia múltiple de varias clases, las cuales comparten una clase base común (hay que declarar dicha clase como virtual y tener bastante cuidado con los constructores), o también cuando las clases base tienen miembros de nombre similar (entonces hay que utilizar especificadores de acceso).

Por ejemplo, de la clase *aparato con motor* y de la clase *animal* no se puede derivar nada, sería como obtener el objeto *toro mecánico* a partir de una *máquina motorizada* (aparato con motor) y un *toro* (aminal). En realidad, lo que se pretende es copiar los métodos, es decir, pasar la funcionalidad del toro de verdad al toro mecánico, con lo cual no sería necesaria la herencia múltiple sino simplemente la

compartición de funcionalidad que se encuentra implementada en Java a través de ***interfaces*.**

Subclases

Como ya se ha indicado en múltiples ocasiones en esta sección, cuando se puede crear nuevas clases por herencia de clases ya existentes, las nuevas clases se llaman ***subclases***, mientras que las clases de donde hereda se llaman ***superclases***.

Cualquier objeto de la subclase contiene todas las variables y todos los métodos de la superclase y sus antecesores.

Todas las clases en Java derivan de alguna clase anterior. La clase raíz del árbol de la jerarquía de clases de Java es la clase **Object**, definida en el paquete **java.lang**. Cada vez que se desciende en el árbol de jerarquía, las clases van siendo más especializadas.

Cuando se desee que nadie pueda derivar de una clase, se indica que es final; y lo mismo con los métodos, si no se desea que se puedan **sobreescribir**, se les antepone la palabra clave final.

Lo contrario de **final** es **abstract**. Una clase marcada como abstracta, únicamente está diseñada para crear subclases a partir de ella, no siendo posible instanciar ningún objeto a partir de una clase abstracta.

Modificadores de acceso (public, protected y private)

Al principio los modificadores de acceso pueden resultar un tanto inútiles puesto que, si puedes acceder directamente a un atributo, ¿para que vas a modificarlos mediante un método?, pues bien, como norma general los atributos de nuestras clases serán **private** o **protected** si utilizaremos herencia. Para modificar atributos o consultar su valor utilizaremos métodos, siendo esta la base de la encapsulación que consiste en hacer visible los atributos o métodos que sean realmente necesarios.

Para controlar el acceso a nuestros atributos y métodos se utilizan los modificadores de acceso que no son más que palabras reservadas del lenguaje

que se encargarán de controlar desde dónde serán accesibles los miembros de una clase, estos modificadores son:

- private (Acceso solo dentro de la clase)
- protected (Acceso desde la clase y sus hijos "herencia")
- Vacío (Sin escribir nada, denominado acceso de paquete)
- public (Acceso publico desde cualquier lugar)

Estos modificadores de acceso **se colocan justo delante del valor de retorno de un método, o del tipo de un atributo**, así que de momento y hasta que no expliquemos el concepto de herencia, utilizaremos los más sencillos, **private** y **public**. A continuación podemos ver un ejemplo de su uso en una clase:

```
public class Vehiculo {
   //Atributos con acceso private
   private String modelo;
   private int velocidad;
   private boolean arrancado;
   //El constructor siempre debe de ser public
   public Vehiculo(String modelo, int velocidad, boolean arrancado) {
      this.modelo = modelo;
      this.velocidad = velocidad;
      this.arrancado = arrancado;
   }
   //Atributos getter con acceso public
   public String getModelo() {
      return modelo;
   }
```

```
    public int getVelocidad() {
        return velocidad;
    }

    public boolean isArrancado() {
        return arrancado;
    }
}
```

Como vemos, hemos definido los atributos **private**, así nadie podrá modificarlos sin nuestro consentimiento, el constructor es **public** para permitir instancias de la clase desde fuera del archivo Vehiculo.java, y los métodos **get** serán también **public**, para permitir a cualquiera consultar el valor de nuestros atributos.

Con los modificadores de acceso, evitamos que nuestros programas tengan un uso no deseado, como modificar atributos introduciendo valores inválidos o llamando a métodos que pueden tener efectos no deseados.

El cuanto al modificador **protected**, este hace nuestro miembro (atributo o método) accesible desde la propia clase, y también desde las clases que hereden de esta o subclases.

Sobrecarga

La sobrecarga es cuando **en una misma clase** se redefine un mismo método. Esta técnica tiene una restricción y es:

- Un método sobrecargado debe tener diferentes parámetros (tipos de dato)

Un ejemplo correcto:

Código

```
1. public int multiplicar(int x, int y) {
2.   return x * y;
3. }
4.
5. public int multiplicar(float x, float y) {
6.   return x * y;
7. }
```

Un ejemplo incorrecto:

Código

```
1. public int multiplicar(int x, int y) {
2.   return x;
3. }
4.
5. public float multiplicar(int x, int y) {
6.   return new Float(String.valueOf(x));
7. }
```

A pesar que el tipo de retorno es diferente, los parámetros son iguales, por lo que el compilador lo detectará como el mismo método.

Videos Tutoriales Con sobrecarga de métodos, uso de This , crear constructores Cedidos por el Prof, Daniel Balbi.

Parte 1

https://www.youtube.com/watch?v=ZUMJwTN4i2M

Parte 2

https://www.youtube.com/watch?v=Da8i-_D8BPk