SOBREESCRITURA (REDEFINICION, OVERWRITE) DE MÉTODOS

Hay dos términos que "suenan" parecido(Los dos empiezan por "Sobre") pero son diferentes conceptualmente:

- **Sobrecargar un método:** generar una nueva versión de un método(o constructor). La nueva versión debe de tener una firma diferente a las existentes. Es un concepto que ya vimos.
- **Sobrescribir un método:** se produce cuando en una jerarquía de clase, un método escrito en una subclase tiene el mismo tipo de retorno y la misma firma que un método de la superclase.

La versión sobrescrita de la subclase oculta a la versión de la superclase

Observa y ejecuta el siguiente ejemplo que no aporta ninguna novedad:

```
class A{
    int i,j;
    A(int a, int b){
        i=a;
        j=b;
    }
    void mostrar_ij(){
    System.out.println("i y j son: "+ i +" y "+ j);
    }
}

class B extends A{
    int k;
    B(int a, int b, int c){
        super(a,b);
        k=c;
    }
    void mostrar_k(){
        System.out.println("k es: "+k);
    }
}
class App {
    public static void main(String[] args) {
        B obj=new B(1,2,3);
        obj.mostrar_ij();
        obj.mostrar_k();
    }
}
```

A continuación vamos a probar el efecto de la sobrescritura sustituyendo mostrar_ij() y mostrar_k() por mostrar(), es decir, provocamos sobreescritura.

```
class A{
    int i,j;
    A(int a, int b){
        i=a;
        j=b;
    }
    void mostrar(){
        System.out.println("i y j son: "+ i +" y "+ j);
    }
}
class B extends A{
    int k;
    B(int a, int b, int c){
        super(a,b);
```

```
        k=c;
    }
    void mostrar(){
        System.out.println("k es: "+k);
    }
}
class App {
    public static void main(String[] args) {
        B obj_b=new B(1,2,3);
        A obj_a=new A(7,8);
        obj_a.mostrar();
        obj_b.mostrar();
    }
}
```

el resultado de obj_a.mostrar() es obvio, en cambio, el resultado de obj_b.mostrar() hay que aclararlo:

obj_b posee dos métodos con nombre *mostrar()*, uno propio y otro heredado, como obj_b es de tipo B, ante este "conflicto" el compilador utiliza la versión sobrescrita, es decir la de B

conclusión: el método sobrescrito "oculta" al de la superclase.

Uso de super para ejecutar la versión oculta de la superclase.

Como ya vimos en anteriores ejemplos, se puede usar *super* para invocar miembros de la superclase "ocultados". Si lo que queremos es que mostrar() imprima todos los atributos de B incluyendo los heredados, podemos hacer:

```
class A{
    int i,j;
    A(int a, int b){
        i=a;
        j=b;
    }
    void mostrar(){
        System.out.println("i y j son: "+ i +" y "+ j);
    }
}

class B extends A{
    int k;
    B(int a, int b, int c){
        super(a,b);
        k=c;
    }
    void mostrar(){
        super.mostrar();
        System.out.println("k es: "+k);
    }
}
class App{
    public static void main(String[] args) {
        B obj_b=new B(1,2,3);
        obj_b.mostrar();
    }
}
```

Observa main() es ahora un "usuario" de objetos de tipo B y le resulta transparente la jerarquía entre A y B, simplemente sabe que si crea un objeto B le pasa tres enteros y los puede mostrar por pantalla con el método mostrar().

Sobrecargar y sobrescribir son conceptos diferentes.

Vimos que una subclase puede sobrescribir un método de su superclase. También puede sobrecargarlo. Veamos un ejemplo.

```
class A{
    int i,j;
    A(int a, int b){
        i=a;
        j=b;
    }
    void mostrar(){
        System.out.println("i y j son: "+ i +" y "+ j);
    }
}
class B extends A{
    int k;
    B(int a, int b, int c){
        super(a,b);
        k=c;
    }
    void mostrar(String saludo){ //tiene una firma diferente que en el padre, es sobrecarga no sobreescritura
        System.out.println(saludo);
    }
}
class App{
    public static void main(String[] args) {
        B obj_b=new B(1,2,3);
        obj_b.mostrar(); //llama al mostrar() de superclase
        obj_b.mostrar("hola");
    }
}
```

observa como la sobrecarga no oculta otra versión del método ya que al existir firmas diferentes el compilador puede distinguir según el contexto una versión de otra.

Sobrescritura y modificadores de acceso.

El método que sobreescribimos tiene que ser igual o más permisivo que el del padre.

Prueba el siguiente ejemplo, observa errores y haz variantes. El caso más raro es cuando el método del padre es private. En este caso en realidad no hay sobreescritura ya que el método está inaccesible, simplemente, si en el hijo escribo un método con misma firma es como si hubiera creado un nuevo método.

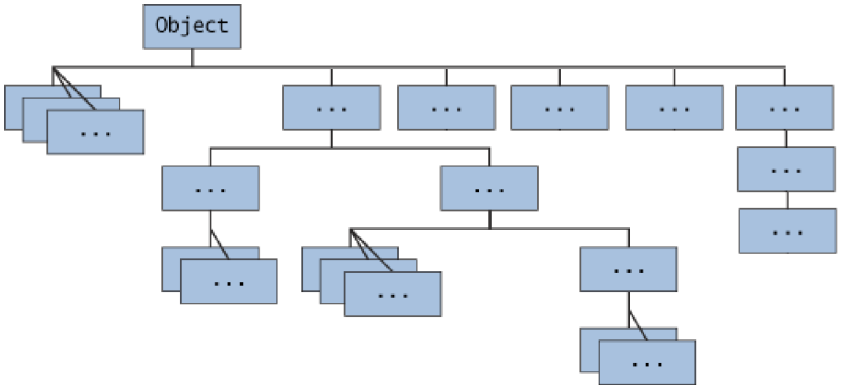
```
class Padre{
    private void m1(){
        System.out.println("m1() de padre");
    }
    void m2(){
        System.out.println("m2() de padre");
    }
    protected void m3(){
        System.out.println("m3() de padre");
    }
    public  void m4(){
        System.out.println("m4() de padre");
    }

}
class Hijo extends Padre{
    private void m1(){//como m1() en padre es private esto no es sobreescritura si no creacion de un m1 nuevo
        System.out.println("m1() hijo");
```

```
    }
    private void m2(){//error compilación por tener menos privilegios que en padre
        System.out.println("m2() hijo");
    }
    public   void m3(){//ok por tener más privilegios
        System.out.println("m3() hijo ");
    }
    public void m4(){//ok por tener mismos  privilegios
        System.out.println("m4() hijo ");
    }
}
```

LA CLASE OBJECT Y SOBRESCRITURA DE MÉTODOS.

Las clases predefinidas de java también conocidas por los términos *API Java* y *biblioteca(librería) java*, es una compleja estructura en árbol en las que unas clases extienden *"extends"* a otras. La raíz del árbol es la clase object.



Por ejemplo, la clase Math es una clase "hija directa" de Object. También las clases que creamos nosotros , como la clase Coche o la clase Racional derivan directamente de la clase object. Toda clase java predefinida o de usuario "cuelga" directa o indirectamente de la clase object.

Por lo tanto

```
class Trabajador{ …..
```

Es en realidad

```
class Trabajador extends Object{ …..
```

es decir, si no escribimos de que clase se extiende se asume que se extiende de Object

La clase Object ofrece una serie de métodos que serán heredados por todas las clases. Vemos ahora como ejemplo dos métodos de la clase Object, otros irán saliendo en ejemplos futuros.

- hashcode()-> Devuelve un número asociado al objeto que se invoca, este número hash es una especie de DNI del objeto que diferencia a un objeto de otro, pero ojo, lo de "unico" es su comportamiento por defecto. Veremos que esto se puede cambiar en sublcases si se sobrescribe este método.
- equals()-> determina si un objeto es igual a otro. Para ello comprueba si los hash de cada objeto son iguales.

```
class X{
    int a;
```

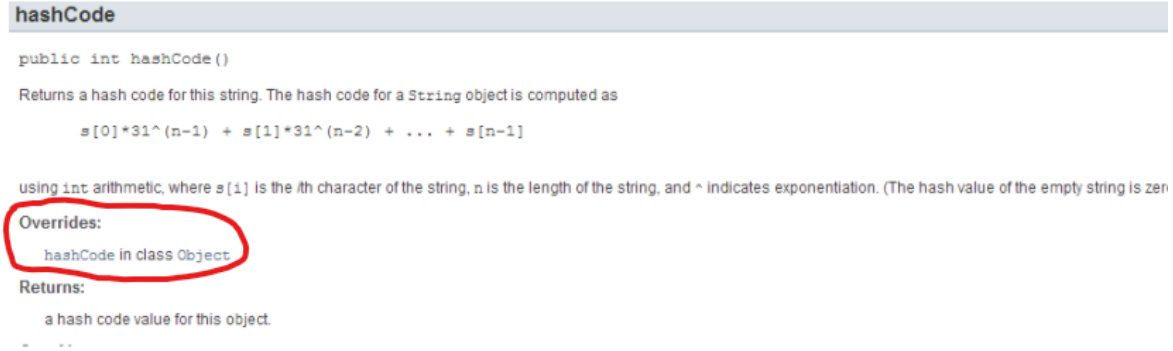
```
   X(int i){a=i;}
}
class App {
   public static void main(String[] args) {
      X x1 = new X(2);
      X x2 = new X(2);
     X x3;
      System.out.println("código hash de x1 "+x1.hashCode());
      System.out.println("código hash de x2 "+x2.hashCode());
      System.out.println("x1 equals x2? "+x1.equals(x2));
      x3=x1;
      System.out.println("código hash de x3 "+x3.hashCode());
      System.out.println("x1 equals x3? "+x1.equals(x3));
   }
}
```

LA CLASE STRING SOBRESCRIBE hashCode() y equals() de OBJECT

Sabemos que todas las clases java "cuelgan" de object y heredan sus métodos, pero hay que tener en cuenta que muchas subclases sobrescriben métodos de sus superclases. Por ejemplo, la clase predefinida java *String* tienen sobrescritas las clases anteriores hashCode() y equals(), de forma que el método equals() se basa en el contenido, es decir, dos objetos son iguales si su contenido es igual. En el siguiente objeto observa como hashCode ya no devuelve un número único por objeto, ahora el valor de hashcode se basa en el contenido del objeto no en su posición de memoria, por tanto, dos Strings con el mismo contenido, aun siendo objetos diferentes, tienen el mismo hashcode. Por otro lado usamos en el ejemplo
System.identityHashCode()
Este método nos permite acceder al hashCode() del Object superclase embebido, para poder acceder a él aunque fuera sobrescrito y lo usamos para constatar que trabajamos realmente con objetos diferentes en el siguiente ejemplo

```
class App {
   public static void main(String[] args) {
      String s1 = new String("hola");
      String s2 = new String("hola");
      System.out.println("hashCode() del subobjeto Object de s1 " + System.identityHashCode(s1));
      System.out.println("hashCode() del subobjeto Object de s2 " + System.identityHashCode(s2));
      System.out.println("código hash de s2 " + s2.hashCode());
      System.out.println("código hash de s1 " + s1.hashCode());
      System.out.println("s1 equals s2? " + s1.equals(s2));
      System.out.println("s1==s2? " + (s1==s2));//esto demuestra que son objetos diferentes
   }
}
```

Observamos que el valor que devuelve hashcode() en Strings se basa en el contenido del String, no es un identificador único. Si consultamos en API hashCode de clase String vemos que efectivamente sobrescribe al hashCode() del Object y nos dice como:



- Observa como nos indica que este método sobrescribe el método hashCode de la clase Object
- Observa como en returns se aprecia que el valor que devuelve es función del contenido(s[0], s[1], …)

Un método que sobrescribimos frecuentemente: toString()

Es muy habitual querer que una clase devuelva un String con la información fundamental de la clase correspondiente a su estado, es decir, de sus atributos, y por tanto, es muy típico que muchas clases tengan un método toString() sobrescrito ya que el toString() de la clase object no nos devuelve el String deseado "por contenido"

Ejemplo: Recuerda el siguiente ejemplo de La clase rectángulo que tiene un método datosAString().

```
class Punto {
   int x;
   int y;
   Punto(int x, int y) {
      this.x = x;
      this.y = y;
   }
 }

class Rectangulo {
   Punto origen;
   int ancho;
   int alto;

   Rectangulo(Punto p, int w, int h) {
      origen = p;
      ancho = w;
      alto = h;
   }
   String datosAString(){
      return "Origen:(" + origen.x+","+origen.y+")"+
            "ancho:"+ancho+
            " alto:"+alto;

   }
 }

public class App{
   public static void main(String[] args) {
      Rectangulo r1= new Rectangulo(new Punto(20,20),5,8);
      Punto p=new Punto(20,20);
      Rectangulo r2 = new Rectangulo(p,5,8);
      Rectangulo r3 = new Rectangulo(p,5,8);

      System.out.println("los tres rectángulos tiene los mismos valores");
      System.out.println("r1=>"+ r1.datosAString());
      System.out.println("r2=>"+r2.datosAString());
      System.out.println("r3=>"+r3.datosAString());
   }
}
```

Cambiando datosString por toString()

```
class Punto {
   int x;
   int y;
   Punto(int x, int y) {
      this.x = x;
      this.y = y;
   }
 }

class Rectangulo {
   Punto origen; !
   int ancho;
   int alto;

   Rectangulo(Punto p, int w, int h) {
      origen = p;
      ancho = w;
      alto = h;
   }
   public String toString(){
      return "Origen:(" + origen.x+","+origen.y+")"+
             "ancho:"+ancho+
             " alto:"+alto;

   }
 }

public class App{
   public static void main(String[] args) {
      Rectangulo r1= new Rectangulo(new Punto(20,20),5,8);
      Punto p=new Punto(20,20);
      Rectangulo r2 = new Rectangulo(p,5,8);
      Rectangulo r3 = new Rectangulo(p,5,8);

      System.out.println("los tres rectángulos tiene los mismos valores");
      System.out.println("r1=>"+ r1.toString());
      System.out.println("r2=>"+r2.toString());
      System.out.println("r3=>"+r3.toString());
   }
}
```

Si alguien usara nuestra clase rectángulo preferirá usar la clase toString() que datosAString() ya que sólo con ver el nombre se hace una idea de lo puede hacer. Recuerda que si no redefinimos toString() en la clase rectángulo lo podemos usar ya que es un método heredado, pero se ejecuta según el código de la superclase Object y la información que nos proporciona no es apropiada en este caso.

Sin sobrescribir toString() debemos de entender la salida:

```
class Punto {
   int x;
   int y;
   Punto(int x, int y) {
      this.x = x;
      this.y = y;
   }
 }

class Rectangulo {
   Punto origen;
   int ancho;
   int alto;

   Rectangulo(Punto p, int w, int h) {
```

```
        origen = p;
        ancho = w;
        alto = h;
    }
 }

public class App{
    public static void main(String[] args) {
        Rectangulo r1= new Rectangulo(new Punto(20,20),5,8);
        Punto p=new Punto(20,20);
        Rectangulo r2 = new Rectangulo(p,5,8);
        Rectangulo r3 = new Rectangulo(p,5,8);

        System.out.println("los tres rectángulos tiene los mismos valores");
        System.out.println("r1=>"+ r1.toString());
        System.out.println("r2=>"+r2.toString());
        System.out.println("r3=>"+r3.toString());
    }
}
```

```
run:
los tres rectángulos tiene los mismos valores
r1=>Rectangulo@15db9742
r2=>Rectangulo@6d06d69c
r3=>Rectangulo@7852e922
```

Ahora lo que ocurre es que como no sobrescribimos, estamos ejecutando el toString() heredado de la clase Object. Si consultamos el API de Object nos dice que el toString() (si no se sobrescribimos) devuelve nombreClase@hasCode del objeto.

Observa también que el main anterior es equivalente a

```
public class App{
    public static void main(String[] args) {
        Rectangulo r1= new Rectangulo(new Punto(20,20),5,8);
        Punto p=new Punto(20,20);
        Rectangulo r2 = new Rectangulo(p,5,8);
        Rectangulo r3 = new Rectangulo(p,5,8);

        System.out.println("los tres rectángulos tiene los mismos valores");
        System.out.println("r1=>"+ r1);
        System.out.println("r2=>"+r2);
        System.out.println("r3=>"+r3);
    }
}
```

Para entender porqué debes consultar el API de println() y observarás que la versión de println(Object o) lo que hace es obtener el String que va a imprimir haciendo un o.toString()