# Programación II
# Como escribir el método equals en Java[1]

Este articulo describe una técnica para sobrescribir el método equals, que preserve el contrato de equals inclusive cuando las subclases de una clase concreta agreguen nuevos campos.

## Los errores mas comunes

La clase java.lang.Object define el método equals, sin embargo es un método sorprendentemente difícil de sobrescribir de manera correcta.

Esto resulta problemático, porque la igualdad es la base de muchas otras cosas.

Por ejemplo, una mala implementación de la igualdad, para una clase C, impediría colocar exitosamente, instancias de C en una colección:

-Sean $c_1$ y $c_2$ dos elementos iguales de tipo C.

-c1.equals(c2) devolvería true. Sin embargo podríamos tener el siguiente comportamiento:

```
Set<C> hashSet = new java.util.HashSet<C>();
hashSet.add(c1);
hashSet.contains(c2);    // returns false!
```

Existen cuatro errores comunes que pueden causar un comportamiento inconsistente del método equals cuando se sobrescribe:

1. Definir equals con aridad incorrecta
2. Cambiar equals son cambiar hashCode
3. ¡Definir equals basado en campos que pueden cambiar!
4. Hacer una implementación de equals que no se comporte como una relación de equivalencia

[1] Título original: "How to Write an Equality Method in Java" por Martin Odersky, Lex Spoon, and Bill Venners

# 1. Definir equals con aridad(signatura) incorrecta

Consideremos agregar la igualdad a la siguiente clase, que representa puntos:

```
public class Point {

    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    // ...
}
```

Una manera obvia pero incorrecta podría ser el Codigo$_{1a}$:

```
public boolean equals(Point other) {
  return (this.getX() == other.getX() && this.getY() == other.getY());
}
```

Codigo$_{1a}$: Equals$_1$

¿Que hay de malo con esta implementación? A primera vista parece ser correcta:

```
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);
Point q = new Point(2, 3);

System.out.println(p1.equals(p2)); // true
System.out.println(p1.equals(q)); // false
```

Codigo$_{1b}$: Ejemplo$_1$

Sin embargo los problema comienzan cuando queremos agregar puntos en una colección(Ver $Codigo_{1b}$ y $Codigo_{1c}$):

```
import java.util.HashSet;

HashSet<Point> coll = new HashSet<Point>();
coll.add(p1);

System.out.println(coll.contains(p2)); // false
```

$Codigo_{1c}$: equals1 falla

¿Como puede ser que coll parece no contener $p_2$, siendo que contiene $p_1$, y que $p_1$ y $p_2$ son iguales?

El motivo va a ser mas claro luego de ver $Codigo_{1d}$:

```
Object p2a = p2;
System.out.println(p1.equals(p2a)); // false
```

$Codigo_{1d}$: $Equals_1$ falla

¿qué esta mal? De hecho, la versión de Equals dada en $Codigo_{1a}$ no sobrescribe Object.equals, pues tiene diferente aridad.

Object.equals tiene la siguiente aridad:

```
public boolean equals(Object other)
```

Como el $Codigo_{1a}$ recibe un Point en lugar de un Object, no sobrescribe Object.equals, en cambio, es solo una **sobrecarga**.

Por lo tanto, tanto en el $Codigo_{1c}$ como en el en $Codigo_{1d}$ se utiliza la versión Object de equals.

Una versión mejorada de equals podría ser la siguiente:

```
@Override public boolean equals(Object other) {
    boolean result = false;
    if (other instanceof Point) {
        Point that = (Point) other;
        result = (this.getX() == that.getX()
                                    && this.getY() == that.getY());
    }
    return result;
}
```

Codigo$_{2a}$: equals$_2$

Ahora equals tiene el tipo correcto.

## 2. Cambiar equals sin cambiar hashCode

Sin embargo si repetimos el test del Codigo$_{1c}$, ***probablemente*** continúe fallando:

```
Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);

HashSet<Point> coll = new HashSet <Point>();
coll.add(p1);

System.out.println(coll.contains(p2)); // false (probablemente)
```

Codigo$_{2b}$: equals$_2$ tambien falla

¿Porque decimos probablemente?
La colección utilizada es un HashSet.

Eso significa que cada objeto obtendrá una posición de hash, basada en su código de hash.

En este caso, la asignación se realizara respecto del hashCode default(Object.hashCode) de p1 y p2.

En la mayoría de los casos se espera que p1 y p2 obtengan diferentes posiciones y por lo tanto el test falle.

Lo que fallo acá fue que redefinimos equals, pero no redefinimos hashCode.

El problema es que el hashCode utilizado para Point viola el contrato que tiene Object:

***Si dos objetos son iguales respecto de equals, hashCode tiene que devolver la misma posición para ambos objetos.***

Podríamos utilizar la siguiente definicion de hashCode para la clase Point:

```
public class Point {

    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof Point) {
            Point that = (Point) other;
            result = (this.getX() == that.getX()
                                  && this.getY() == that.getY());
        }
        return result;
    }

    @Override public int hashCode() {
        return (41 * (41 + getX()) + getY());
    }
}
```

### Codigo$_3$: hashCode

Multiplicando las coordenadas por el numero primo 41 nos da una distribución bastante uniforme del código.

Codigo$_3$ es una de las muchas posibilidades para hashCode. Además, la función se calcula rápidamente y devuelve valores acotados.

Redefinir hashCode arregla el problema de la igualdad para clases como Point.
Sin embargo también se pueden presentar otros problemas.

## 3. Definir equals basado en campos que pueden cambiar!

Consideremos la siguiente variación de la clase Point:

```
public class Point {

    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setX(int x) { // Problematica
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof Point) {
            Point that = (Point) other;
            result = (this.getX() == that.getX()
                        && this.getY() == that.getY());
        }
        return result;
    }

    @Override public int hashCode() {
        return (41 * (41 + getX()) + getY());
    }
}
```

Codigo$_{4a}$: Point sin el modificador ***final***.

La única respecto del Codigo$_3$ es que el Codigo$_{4a}$ no tiene el mdificador ***final***.

Es decir, podemos modificar el punto las veces que queramos.

Consideremos el Codigo$_{4b}$:

```
Point p = new Point(1, 2);

HashSet<Point> coll = new HashSet<Point>();
coll.add(p);

System.out.println(coll.contains(p)); // true

p.setX(p.getX() + 1);

System.out.println(coll.contains(p)); // false (probablemente)
```

Codigo$_{4b}$: equals/hashCode fallan

Esto se ve extraño. ¿A donde se fue p?

Lo que sucedió fue que al modificar p, su hashCode dejo de corresponder con hashCode utilizado al ingresar p en el HashSet.

Luego, al preguntar por p, este problamente ya no se encuentre en la misma posición del hash, por lo tanto, no será encontrado.

La moraleja de este ultimo ejemplo es no hacer depender la comparación entre objetos que van a ser utilizados dentro de un hash, de campos que pueden ser modificados.

Si es necesario, se deben utilizar otros campos para implementar equals/hashCode .

# 4. Hacer una implementación de equals que no se comporte como una relación de equivalencia

El contrato de Object.equals especifica que equals tiene que implementar una relación de equivalencia para todos los objetos que no sean null.

Para ser una relación de equivalencia se deben cumplir las siguientes propiedades(para cualquier valor no nulo de x, y,z):
-**Reflexividad:** x.equals(x) tiene que devolver true.
-**Simetria:** x.equals(y) == y.equals(x)
-**Transitividad:** Si x.equals(y) y y.equals(z), entonces x.equals(z)

Para cualquier valor no-null de x, x.equals(null) debe devolver false.
Hasta ahora la definición de equals nos viene funcionando.

¿Pero que sucede cuando agregamos subclases?

Consideremos la clase ColoredPoint, que extenderá la clase Point:
ColoredPoint sobrescribirá Point.equals, de manera de considerar los nuevos atributos agregados.

```
public enum Color {
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET;
}

public class ColoredPoint extends Point { // Problem: equals not
symmetric

    private final Color color;

    public ColoredPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof ColoredPoint) {
            ColoredPoint that = (ColoredPoint) other;
            result = (this.color.equals(that.color)
                            && super.equals(that));
        }
        return result;
    }
}
```

Codigo$_{5a}$: ColoredPoint

Notar que ColoredPoint no necesita sobrescribir Point.hashCode

Esto se debe a que ColoredPoint.equals(Ver $Codigo_{5a}$) es mas estricto que Point.equals, por lo tanto:

-Se sigue cumpliendo el contrato de hashCode

-Para cualquier de par ColoredPoint iguales respecto ColoredPoint.equal, Point.hashCode devolverá la misma posición.

Si trabajamos únicamente con instancias de ColoredPoint, ColoredPoint.equal funcionara bien.

Sin embargo, si mezclamos ColoredPoint y Point:

```
Point p = new Point(1, 2);

ColoredPoint cp = new ColoredPoint(1, 2, Color.RED);

System.out.println(p.equals(cp)); // true

System.out.println(cp.equals(p)); // false
```

$Codigo_{5b}$: ColoredPoint.equal y Point.equal violan la simetría

Aca hay dos interpretaciones posibles:

O bien las dos clases son comparables

En este caso, ambas funciones deberían haber devuelto true

O bien las dos clases no son comparables

En este caso, ambas funciones deberían haber devuelto false

En cualquier caso $Codigo_{5b}$ viola el principio de **simetría**:

- p.equals(cp) != cp.equals(p)

# El método canEqual

La solución es implementar el método canEqual a cada clase que implemente equals.

canEqual , para la semántica que queremos, garantiza que solo sean comparables clases del mismo tipo.

Versión final de Point y ColoredPoint:

```
public class Point {

    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof Point) {
            Point that = (Point) other;
            result = (that.canEqual(this)
                                    && this.getX() == that.getX()
                                    && this.getY() == that.getY());
        }
        return result;
    }

    @Override public int hashCode() {
        return (41 * (41 + getX()) + getY());
    }

    public boolean canEqual(Object other) {
        return (other instanceof Point);
    }
}
```

Codigo$_{6a}$: versión final de Point

```
// No longer violates symmetry requirement
public class ColoredPoint extends Point {
    private final Color color;

    public ColoredPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof ColoredPoint) {
            ColoredPoint that = (ColoredPoint) other;
            result = (that.canEqual(this)
                                    && this.color.equals(that.color)
                                    && super.equals(that));
        }
        return result;
    }

    @Override public int hashCode() {
        return (41 * super.hashCode() + color.hashCode());
    }

    @Override public boolean canEqual(Object other) {
        return (other instanceof ColoredPoint);
    }
}
```

Codigo$_{6b}$:Versión final de ColoredPoint

Se puede demostrar que Codigo$_{6a}$ y Codigo$_{6b}$ no violan el contrato de equals(relación de equivalencia).

Referencias

http://www.artima.com/lejava/articles/equality.html