[**Java Hashing**](http://www.dosideas.com/noticias/java/758-java-hashing.html)



Todos los objetos en Java tiene dos métodos muy importantes: el método hashCode () y el método equals (). Estos métodos están diseñados para ser sobrescritos de acuerdo a su contrato general.

**El contrato de un HashCode**

El contrato del método hashCode () dice:

**"Si dos objetos son iguales usando equals (), entonces la invocación a hashCode () de ambos objetos debe retornar el mismo valor"**

Entonces, la pregunta que surge es: ¿es necesario que siempre se cumpla esa oración?

Considerar una clase que tiene una implementación correcta del método equals(), ¿qué pasaría si no se obedece el contrato anterior?

Para responder a esa pregunta, se deben considerar dos situaciones:

* + - Objetos que son iguales, pero retornan diferentes hashCode
		- Objetos que no son iguales, pero retornan el mismo hashCode

**Objetos que son iguales, pero retornan diferentes hashCodes**

¿Qué pasaría si dos objetos son iguales (invocando equals()) pero retornan diferentes hashCodes? El código se ejecutará a la perfección. Nunca se va a encontrar problemas... hasta que suceda que se pide almacenar un objeto dentro de una colección como un HashSet o un HashMap. Cuando se realice esto, es posible encontrarse con problemas raros durante la ejecución.

Primero se debe comprender cómo funcionan las clases del tipo HashSet y HashMap. Estas clases de colecciones dependen de que los objetos que son agregados cumplan con el contrato del hashCode. Se obtienen resultados impredecibles en tiempo de ejecución si no se obedece el contrato y se quiere almacenar estos objetos en la colección.

Observar por ejemplo el HashMap. Cuando se guardan valores en un HashMap, estos valores en realidad se almacenan dentro de "lotes". Cada uno de estos lotes tiene asignado un número que lo identifica. Cuando agregamos un valor al HashMap, almacena el dato en uno de esos lotes. El lote que se usa depende del hashCode que devuelva el objeto a ser almacenado. Por ejemplo, si el método hashCode () del objeto retorna 49, entonces se almacena en la posición 49 dentro del HashMap.

Más tarde, cuando se verifique si la colección contiene el elemento invocando, el método contains(elemento), el HashMap primero obtiene el hashCode de ese "elemento". Luego buscará la posición que corresponde a ese hashCode. Si la posición está vacía, significa que el HashMap no contiene al elemento y devuelve false.

Si hay un objeto o más dentro de la posición del HashMap, entonces se compara al "elemento" con todos los elementos en ese lugar usando el método equals().

**Objetos que no son iguales, pero retornan el mismo hashCode**

El contrato del hashCode no dice nada sobre este caso. Por lo tanto, objetos distintos pueden devolver el mismo hashCode, pero las colecciones como los HashMap van a ser más ineficientes si se almacenan objetos diferentes con el mismo valor de hashCode.

**¿Por qué almacenar en lotes?**

Se utiliza este mecanismo de "lotes" por un tema de eficiencia. Pueden imaginarse que si todos los objetos que se agregan a un HashMap se almacenaran en una única lista grande, entonces tendríamos que comparar la entrada con todos los objetos de la lista para determinar si un elemento en particular está contenido en el Map. Como se usan lotes, sólo se comparan los elementos del lote específico, y en general cada lote sólo almacena una pequeña cantidad de elementos en el HashMap.

**Sobrescribir el método hashCode ()**

Puede resultar complejo escribir un buen método de hashCode () para una clase nueva.

**Retornar un valor fijo (es una mala idea...)**

Se puede implementar un método de hashCode () que devuelva un valor fijo, como por ejemplo:

**//no hagan esto, genera mal rendimiento**

**@**[**Override**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Override.html)

**public int hashCode() {**

 **return 1;**

**}**

Este método satisface todos los requerimientos y es "legal" de acuerdo al contrato del hashCode, pero no va a resultar muy eficiente. Si se usa este método, todos los objetos se almacenarán dentro del mismo lote (el correspondiente al "1"), y cuando queramos comprobar si un objeto específico está dentro de la colección, entonces siempre se tendrá que verificar el contenido completa de dicha colección.

Por otro lado, si se sobrescribe el método hashCode() y rompemos el contrato ("dos objetos iguales con equals deben devolver el mismo hashCode"), entonces cuando se invoque el método contains() podría devolver false para un elemento que se encuentra dentro de la colección, pero en un balde diferente.

**Método de Effective Java**

Joshua Bloch en su libro Effective Java brinda una buena guía para generar un valor de hashCode():

1. Guardar alguna constante con un valor distinto al cero; por ejemplo **17**, en una variable int llamada **result**.
2. Para cada campo significativo f en el objeto (es decir, cada campo que se tiene en cuenta al ejecutar un equals()), hacer lo siguiente:
	* Calcular un int de hashCode para el campo:
		+ Si esl campo es un booleano, calcular  **c = (f ? 1 : 0)**
		+ Si el campo es un byte, char, short o int, calcular **c = (int) f**
		+ Si el campo es un long, calcular **c = (int) (f ^ (f >>> 32))**
		+ Si el campo es un float, calcular **c = Float.floatToIntBits(f)**
		+ Si el campo es un double, calcular **long l = Double.doubleToLongBits(f); c = (int) (l ^ (l >>> 32));**
		+ Si el campo es una referencia a un objeto, calcular **c = f.hashCode()**
		+ Si el campo es un array, tratar a cada elemento por separado. Es decir, calcular el hashCode de cada elemento significativo usando las reglas anteriores.
	* Combinar el hashCode calculado c en el paso 2.1 en un resultado como sigue: **result = 37 \* result + c;**
3. Retornar **result**
4. Mirar al hashCode () resultante y asegurarse que instancias iguales tengan el mismo hashCode.

Veamos un ejemplo de este algoritmo:

public class HashTest {

 private [String](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) campo1;

 private short campo2;

  //resto de la clase...

  @[Override](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Override.html)

 public int hashCode() {

 int result = 17;

 result = 37\*result + campo1.hashCode();

 result = 37\*result + (int)campo2;

 return result;

 }

**}**

Como vemos elegimos la constante 37. La idea es elegir un número que sea un **número primo**. Podemos elegir cualquier número primo. Al usar un número primo los objetos se distribuirán mejor en los lotes. Pueden aprender más sobre este algoritmo y la distribución que genera buscando en Internet.

**Apache HashCodeBuilder**

Como estamos aprendiendo, no es siempre fácil retornar un buen valor de hashCode. Por suerte existen clases que nos pueden ayudar.

El paquete **org.apache.commons.lang.builder** de Jakarta-Commons contiene la clase **HashCodeBuilder** que está diseñada para ayudarnos a implementar el método hashCode(). Muchos desarrolladores luchan por escribir sus hashCode cuando existe esta clase que nos simplifica el proceso.

Así es como quedaría la clase de prueba anterior usando la clase HashCodeBuilder:

public class HashTest {

 private [String](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) campo1;

 private short campo2;

  //resto de la clase...

  @[Override](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Override.html)

 public int hashCode() {

 return new HashCodeBuilder(83, 7)

 .append(campo1)

 .append(campo2)

 .toHashCode();

 }

}

Noten que los dos números del constructor del HashCodeBuilder son dos números impares distintos a cero - estos números ayuda a evitar la colisión de valores de hashCode en otros objetos.

Si se necesita, se puede invocar al hashCode() de la superclase usando appendSuper(int).

Resulta muy fácil escribir el método hashCode() usando la clase Apache HashCodeBuilder.

**Objetos mutables como clave**

Como consejo general, se debería **usar objetos inmutables como clave** en una colección. El hashCode funciona mejor cuando se calcula con datos inmutables. Si usamos objetos mutables como clave y estos objetos cambian su estado de manera que el hashcode también cambia, entonces el objeto almacenado quedará ubicado en un lote incorrecto dentro de la colección.

La cosa más importante a considerar cuando se implementa el hashCode () es que, sin importar cuándo se invoca a este método, tiene que producir el mismo valor para un objeto en particular cada vez que se invoca. Si tenemos un escenario en donde el objeto produce un valor de hashCode () cuando se invoca al put() del HashMap y luego produce otro valor durante un get(), en ese caso no podremos recuperar este objeto. Por lo tanto, si nuestro hashCode () depende de datos mutables en el objeto, cambiar estos datos con seguridad producirán una nueva clave al generar un hashCode () diferente.

Veamos el siguiente ejemplo:

**public class Empleado {**

**private** [**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) **nombre;**

 **private int edad;**

**public Empleado(**[**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) **nombre, int edad) {**

 **this.nombre = nombre;**

 **this.edad = edad;**

 **}**

**public** [**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) **getNombre() {**

 **return nombre;**

 **}**

**public void setNombre(**[**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) **nombre) {**

 **this.nombre= nombre;**

 **}**

**public int getEdad() {**

 **return edad;**

 **}**

**public void setEdad(int edad) {**

 **this.edad = edad;**

 **}**

**@**[**Override**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Override.html)

 **public boolean equals(**[**Object**](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+Object) **obj) {**

 **if (obj instanceof Empleado) {**

 **Empleado emp = (Empleado)obj;**

 **return (emp.nombre.equals(nombre) && emp.edad == edad);**

 **}**

 **return false;**

 **}**

**@**[**Override**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Override.html)

 **public int hashCode() {**

 **return nombre.length() + edad;**

 **}**

**public static void main(**[**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html)**[] args) {**

 **Empleado e = new Empleado("muhammad", 24);**

[**Map**](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Map.html) **map = new** [**HashMap**](http://java.sun.com/j2se/1.5.0/docs/api/java/util/HashMap.html)**();**

 **map.put(e, "Muhammad Ali Khojaye");**

**// encuentra el resultado**

[**System**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html)**.out.println(map.get(e));**

**e.nombre = "abid";**

**// el map devolverá null porque no lo encuentra**

[**System**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html)**.out.println(map.get(e));**

**// otra vez devolverá null**

[**System**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html)**.out.println(map.get(new Empleado("muhammad", 24)));**

 **}**

**}**

En el ejemplo anterior se observa que algunos resultados obtenidos son extraños. Después de cambiar el campo nombre, el cálculo del hashCode () devuelve un nuevo número y apuntará a un nuevo lote, por lo que el contains () devolverá false.

Podemos arreglar esta situación usando alguna de estas alternativas:

* **El hashcode es mejor calcularlo con datos inmutables**; por lo tanto, asegurarnos que sólo usaremos objetos inmutables como claves de una colección.
* Implementar el hashCode () usando la primera técnica: devolver un valor constante. En este caso tenemos que ser conscientes que estamos quitando todas las ventajas del mecanismo de lotes de las coleccione
* Si necesitamos incluir campos mutables en el método de hashCode (), entonces podemos calcular y almacenar el valor del hash cuando se crea el objeto, y cada vez que se actualiza alguno de los campos mutables, primero debemos quitarlo de la colección y luego agregarlo nuevamente una vez hecho el cambio.