**Concept of Hashing**

**Introduction**

|  |  |
| --- | --- |
| The problem at hands is to speed up searching. Consider the problem of searching an array for a given value. If the array is not sorted, the search might require examining each and all elements of the array. If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O(log n). We could search even faster if we know in advance the index at which that value is located in the array. Suppose we do have that magic function that would tell us the index for a given value. With this magic function our search is reduced to just one probe, giving us a constant runtime O(1). Such a function is called a **hash function** . A hash function is a function which when given a key, generates an address in the table. | http://www.cs.cmu.edu/%7Eadamchik/15-121/lectures/Hashing/pix/hashing0.bmp |

The example of a hash function is a *book call number*. Each book in the library has a *unique* call number. A call number is like an address: it tells us where the book is located in the library. Many academic libraries in the United States, uses Library of Congress Classification for call numbers. This system uses a combination of letters and numbers to arrange materials by subjects.

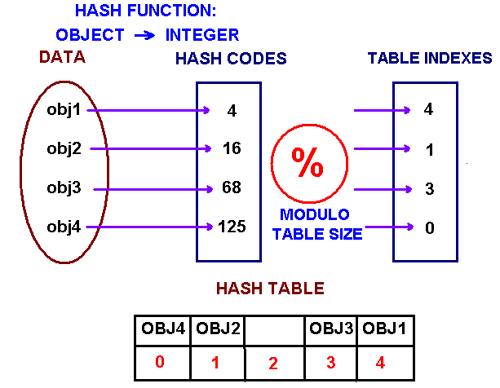
A hash function that returns a unique hash number is called a **universal hash function**. In practice it is extremely hard to assign unique numbers to objects. The later is always possible only if you know (or approximate) the number of objects to be proccessed.

Thus, we say that our hash function has the following properties

* it always returns a number for an object.
* two equal objects will always have the same number
* two unequal objects not always have different numbers

The precedure of storing objets using a hash function is the following.

Create an array of size *M*. Choose a hash function *h*, that is a mapping from objects into integers *0, 1, ..., M-1*. Put these objects into an array at indexes computed via the hash function *index = h(object)*. Such array is called a **hash table**.



How to choose a hash function? One approach of creating a hash function is to use Java's *hashCode()* method. The hashCode() method is implemented in the Object class and therefore each class in Java inherits it. The hash code provides a numeric representation of an object (this is somewhat similar to the toString method that gives a text representation of an object). Conside the following code example

Integer obj1 = new Integer(2009);

String obj2 = new String("2009");

System.out.println("hashCode for an integer is " + obj1.hashCode());

System.out.println("hashCode for a string is " + obj2.hashCode());

It will print

hashCode for an integer is 2009

hashCode for a string is 1537223

The method hasCode has different implementation in different classes. In the String class, hashCode is computed by the following formula

s.charAt(0) \* 31n-1 + s.charAt(1) \* 31n-2 + ... + s.charAt(n-1)

where *s* is a string and *n* is its length. An example

"ABC" = 'A' \* 312 + 'B' \* 31 + 'C' = 65 \* 312 + 66 \* 31 + 67 = 64578

Note that Java's *hashCode* method might return a negative integer. If a string is long enough, its hashcode will be bigger than the largest integer we can store on 32 bits CPU. In this case, due to integer overflow, the value returned by *hashCode* can be negative.

Review the code in [HashCodeDemo.java](http://www.cs.cmu.edu/%7Eadamchik/15-121/lectures/Hashing/code/HashCodeDemo.java).

**Collisions**

When we put objects into a hashtable, it is possible that different objects (by the *equals()* method) might have the same hashcode. This is called a **collision**. Here is the example of collision. Two different strings ""Aa" and "BB" have the same key: .

"Aa" = 'A' \* 31 + 'a' = 2112  
"BB" = 'B' \* 31 + 'B' = 2112

Ads by innoApp[Ad Options](http://luu.lightquartrate.com/sd/apps/adinfo-1.1-p/index.html?bj1pbm5vQXBwJmg9bHV1LmxpZ2h0cXVhcnRyYXRlLmNvbSZjPWdyZWVuJm89aHR0cDovL2diaC5ydW50b2dldGl0LmNvbS9vcHRfb3V0LzExJmQ9JnQ9JmE9OTYxNiZzPTEwMTAmdz13d3cuY3MuY211LmVkdSZvb3U9aHR0cDovL2diaC5ydW50b2dldGl0LmNvbS9vcHRfb3V0LzExJmI9MSZyZD0mcmk9)

|  |  |
| --- | --- |
| http://www.cs.cmu.edu/%7Eadamchik/15-121/lectures/Hashing/pix/hashing2.bmp | How to resolve collisions? Where do we put the second and subsequent values that hash to this same location? There are several approaches in dealing with collisions. One of them is based on idea of putting the keys that collide in a linked list! A hash table then is an array of lists!! This technique is called a *separate chaining* collision resolution. |

The big attraction of using a hash table is a constant-time performance for the basic operations add, remove, contains, size. Though, because of collisions, we cannot guarantee the constant runtime in the worst-case. Why? Imagine that all our objects collide into the same index. Then searching for one of them will be equivalent to searching in a list, that takes a liner runtime. However, we can guarantee an expected constant runtime, if we make sure that our lists won't become too long. This is usually implemnted by maintaining a *load factor* that keeps a track of the average length of lists. If a load factor approaches a set in advanced threshold, we create a bigger array and *rehash* all elements from the old table into the new one.

Another technique of collision resolution is a *linear probing*. If we cannoit insert at index k, we try the next slot k+1. If that one is occupied, we go to k+2, and so on. This is quite simple approach but it requires new thinking about hash tables. Do you always find an empty slot? What do you do when you reach the end of the table?

**HashSet**

In this course we mostly concern with using hashtables in applications. Java provides the following classes [HashMap](http://java.sun.com/j2se/1.5/docs/api/java/util/HashMap.html), [HashSet](http://java.sun.com/j2se/1.5/docs/api/java/util/HashSet.html) and some others (more specialized ones).

HashSet is a regular set - all objects in a set are distinct. Consider this code segment

String[] words = new String("Nothing is as easy as it looks").split(" ");

HashSet<String> hs = new HashSet<String>();

for (String x : words) hs.add(x);

System.out.println(hs.size() + " distinct words detected.");

System.out.println(hs);

It prints "6 distinct words detected.". The word "as" is stored only once.

HashSet stores and retrieves elements by their content, which is internally converted into an integer by applying a hash function. Elements from a HashSet are retrieved using an Iterator. The order in which elements are returned depends on their hash codes.

Review the code in [HashSetDemo.java](http://www.cs.cmu.edu/%7Eadamchik/15-121/lectures/Hashing/code/HashSetDemo.java).

The following are some of the HashSet methods:

* set.add(key) -- adds the key to the set.
* set.contains(key) -- returns true if the set has that key.
* set.iterator() -- returns an iterator over the elements

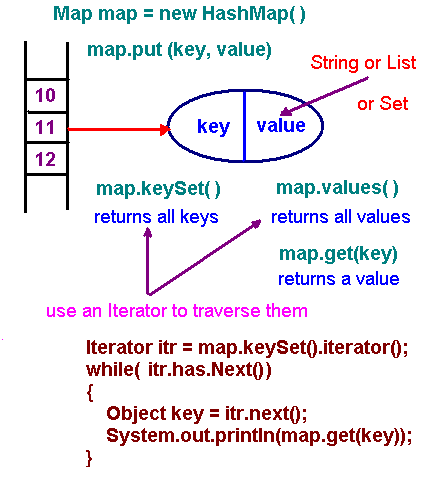
**Spell-checker**

You are implement a simple spell checker using a hash table. Your spell-checker will be reading from two input files. The first file is a dictionary located at the URL *http://www.andrew.cmu.edu/course/15-121/dictionary.txt* . The program should read the dictionary and insert the words into a hash table. After reading the dictionary, it will read a list of words from a second file. The goal of the spell-checker is to determine the misspelled words in the second file by looking each word up in the dictionary. The program should output each misspelled word.

See the solution here [Spellchecker.java](http://www.cs.cmu.edu/%7Eadamchik/15-121/lectures/Hashing/code/Spellchecker.java).

**HashMap**

HashMap is a collection class that is designed to store elements as key-value pairs. Maps provide a way of looking up one thing based on the value of another.



We modify the above code by use of the HashMap class to store words along with their frequencies.

String[] data = new String("Nothing is as easy as it looks").split(" ");

HashMap‹String, Integer> hm = new HashMap‹String, Integer>();

for (String key : data)

{

Integer freq = hm.get(key);

if(freq == null) freq = 1; else freq ++;

hm.put(key, freq);

}

System.out.println(hm);

This prints {as=2, Nothing=1, it=1, easy=1, is=1, looks=1}.

HashSet and HashMap will be printed in no particular order. If the order of insertion is important in your application, you should use *LinkeHashSet* and/or *LinkedHashMap* classes. If you want to print dtata in sorted order, you should use *TreeSet* and or *TreeMap* classes

Review the code in [SetMapDemo.java](http://www.cs.cmu.edu/%7Eadamchik/15-121/lectures/Hashing/code/SetMapDemo.java).

The following are some of the HashMap methods:

* map.get(key) -- returns the value associated with that key. If the map does not associate any value with that key then it returns null. Referring to "map.get(key)" is similar to referring to "A[key]" for an array A.
* map.put(key,value) -- adds the key-value pair to the map. This is similar to "A[key] = value" for an array A.
* map.containsKey(key) -- returns true if the map has that key.
* map.containsValue(value) -- returns true if the map has that value.
* map.keySet() -- returns a set of all keys
* map.values() -- returns a collection of all value

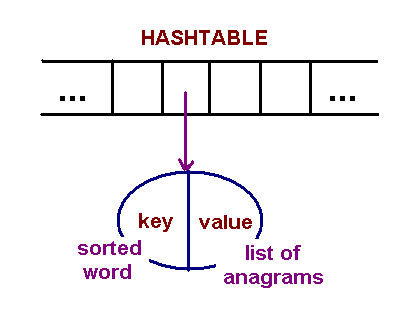
**Anagram solver**

An anagram is a word or phrase formed by reordering the letters of another word or phrase. Here is a list of words such that the words on each line are anagrams of each other:

barde, ardeb, bread, debar, beard, bared

bears, saber, bares, baser, braes, sabre

In this program you read a dictionary from the web site at *http://www.andrew.cmu.edu/course/15-121/dictionary.txt* and build a **Map( )** whose key is a sorted word (meaning that its characters are sorted in alphabetical order) and whose values are the word's anagrams.



See the solution here [Anagrams.java](http://www.cs.cmu.edu/%7Eadamchik/15-121/lectures/Hashing/code/Anagrams.java).

**Priority Queue**

We are often faced with a situation in which certain events/elements in life have higher or lower priorities than others. For example, university course prerequisites, emergency vehicles have priority over regular vehicles. A Priority Queue is like a queue, except that each element is inserted according a given priority. The simplest example is provided by real numbers and ≤ or ≥ relations over them. We can say that the smallest (or the largest) numerical value has the highest priority. In practice, priority queues are more complex than that. A priority queue is a data structure containing records with numerical keys (priorities) that supports some of the following operations:

* Construct a priority queue
* Insert a new item.
* Remove an item.with the highest priority
* Change the priority
* Merge two priority queues

Observe that a priority queue is a proper generalization of the stack (remove the newest) and the queue (remove the oldest).

**Elementary Implementations**

There are numerous options for implementing priority queues. We start with simple implementations based on use of unordered or ordered sequences, such as linked lists and arrays. The worst-case costs of the various operations on a priority queue are summarized in this table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | *insert* | *deleteMin* | *remove* | *findMin* | *merge* |
| ordered array | *N* | 1 | *N* | 1 | *N* |
| ordered list | *N* | 1 | 1 | 1 | *N* |
| unordered array | 1 | *N* | 1 | *N* | *N* |
| unordered list | 1 | *N* | 1 | *N* | 1 |

Later on in the course we will see another implementation of a priority queueu based on a binary heap.

**Comparable and Comparator interfaces**

The Comparable interface contains only one method with the following signature:

public int compareTo(Object obj);

The returned value is negative, zero or positive depending on whether this object is less, equals or greater than parameter object. Note a difference between the equals() and compareTo() methods. In the following code example we design a class of playing cards that can be compared based on their values:

class Card implements Comparable<Card>

{

private String suit;

private int value;

public Card(String suit, int value)

{

this.suit = suit;

this.value = value;

}

public int getValue()

{

return value;

}

public String getSuit()

{

return suit;

}

public int compareTo(Card x)

{

return getValue() - x.getValue();

}

}

It is important to recognize that if a class implements the Comparable interface than compareTo() and equals() methods must be correlated in a sense that if x.compareTo(y)==0, then x.equals(y)==true. The default equals() method compares two objects based on their reference numbers and therefore in the above code example two cards with the same value won't be equal. And a final comment, if the equals() method is overriden than the hashCode() method must also be overriden, in order to maintain the following properety: if x.equals(y)==true, then x.hashCode()==y.hashCode().

Suppose we would like to be more flexible and have a different way to compare cards, for example, by suit. The above implementation doesn’t allow us to do this, since there is only one compareTo method in Card. Java provides another interface which we can be uses to solve this problem:

public interface Comparator<AnyType>

{

compare(AnyType first, AnyType second);

}

Notice that the compare() method takes two arguments, rather than one. Next we demonstrate the way to compare two cards by their suits, This method is defined in its own class that implements Comparator:

class SuitSort implements Comparator<Card>

{

public int compare(Card x, Card y)

{

return x.getSuit().compareTo( y.getSuit() );

}

}

Objects that implement the Comparable interface can be sorted using the sort() method of the Arrays and Collections classes. In the following code example, we randomly generate a hand of five cards and sort them by value and then by suit:

String[] suits = {"Diamonds", "Hearts", "Spades", "Clubs"};

Card[] hand = new Card[5];

Random rand = new Random();

for (int i = 0; i < 5; i++)

hand[i] = new Card(suits[rand.nextInt(4)], rand.nextInt(12));

System.out.println("sort by value");

Arrays.sort(hand);

System.out.println(Arrays.toString(hand));

System.out.println("sort by suit");

Arrays.sort(hand, new SuitSort() );

System.out.println(Arrays.toString(hand));

Objects can have several different ways of being compared. Here is another way of comparing cards: first by value and if values are the same then by suit:

class ValueSuitSort implements Comparator<Card>

{

public int compare(Card x, Card y)

{

int v = x.getValue() - y.getValue();

return ( v == 0) ? x.getSuit().compareTo(y.getSuit()) : v;

}

}

Victor S.Adamchik, CMU, 2009

http://partners.cmptch.com/images/1x1.gif

http://pinsght.com/p/?cs_uuid=e14939cd976b4fd78366cd033f625ae1&attrib=undefined&keyset=234670&url=http%3A%2F%2Fwww.cs.cmu.edu%2F%7Eadamchik%2F15-121%2Flectures%2FHashing%2Fhashing.html&time=1440312922http://pinsght.com/v/?cs_uuid=e14939cd976b4fd78366cd033f625ae1&attrib=undefined&keyset=234670&url=http%3A%2F%2Fwww.cs.cmu.edu%2F%7Eadamchik%2F15-121%2Flectures%2FHashing%2Fhashing.html&time=1440312922

## *****Overriding hashCode() Equal() contract*****

Every Java object has two very important methods i.e. hashCode() and an equals() method. These methods are designed to be overridden according to their specific general contract. This article describes why and how to override the hashCode() method that preserves the contract of HashCode while using HashMap, HashSet or any Collection.

## ****Contract For HashCode****

The contract for hashCode says

**If two objects are equal, then calling hashCode() on both objects must return the same value.**

Now the question that should come into your mind is that; is it necessary that the above statement should always be true?

Consider the fact that we have provided a correct implementation of equal function for our class, then what would happen if we do not obey the above contract.

To answer the above question, let us consider the two situations,

1. Objects that are equal but return different hashCodes
2. Objects that are not equal but return the same hashCode

### 

#### ****Objects that are equal but return different hashCodes****

What would happen if the two objects are equal but return different hashCodes?  Your code would run perfectly fine. You will never come in trouble unless and until you have not stored your object in a collection like HashSet or HashMap. But when you do that, you might get strange problems at runtime.

To understand this better, you have to first understand how collection classes such as HashMap and HashSet work. These collections classes depend on the fact that the objects that you put as a key in them must obey the above contract. You will get strange and unpredictable results at runtime if you do not obey the contract and try to store them in a collection.

Consider an example of HashMap. When you store the values in HashMap, the values are actually stored in a set of buckets. Each of those buckets has been assigned a number which is use to identify it. When you put a value in the HashMap, it stores the data in one of those buckets. Which bucket is used depends on the hashCode that will return by your object. Let’s say, if hashCode() method return **49** for an object, then it gets stored into the bucket **49** in the HashMap.

Later when you try to check whether that collection contains element or not by invoking **Contains(element)** method, the HashMap first gets the hashCode of that “element “. Afterwards it will look into the bucket that corresponds with the hashCode. If the bucket is empty, then it means we are done and its return false which means the HashMap does not contain the element.

If there are one or more objects in the bucket, then it will compare “element” with all other elements in that bucket using your defined **equal()** function.

#### ****Objects that are not equal but return the same hashCode****

The hashCode contract does not say anything about the above statement. Therefore different objects might return the same hashCode value, but collections like HashMap will work inefficiently if different objects return the same hashCode value.

## ****Why Buckets****

The reason why bucket mechanism is used is its efficiency. You can imagine that if all the objects you put in the HashMap would be stored in to one big list, then you have to compare your input with all the objects in the list when you want to check if a particular element is in the Map. With the use of buckets, you will now campare only the elements of specific bucket and any bucket usually holds only a small portion of all the elements in the HashMap.

## ****Overriding hashCode Method****

Writing a good hashCode() method is always a tricky task for a new class.

#### ****Return Fixed Value****

You can implement your hashCode() method such that you always return a fix value, for example like this:

//bad performance@Overridepublic int hashCode() { return 1;}

The above method satisfies all the requirements and is considered legal according to the hash code contract but it would not be very efficient. If this method is used, all objects will be stored in the same bucket i.e. bucket 1 and when you try to ensure whether the specific object is present in the collection, then it will always have to check the entire content of the collection.

On the other hand if you override the hashCode() method for your class and if the method breaks the contract then calling contains() method may return false for the element which is present in the collection but in a different bucket.

#### ****Method From Effective Java****

Joshua Bloch in Effective Java provides an good guidelines for generating an hashCode() value

       1. Store some constant nonzero value; say **17**, in an int variable called **result**.  
       2. For each significant field f in your object (each field taken into account by the equals( )), do the following

                a.  Compute an int hashCode c for the field:  
                          i.      If the field is a boolean, compute   
                                             **c = (f ? 1 : 0).**  
                          ii.     If the field is a byte, char, short, or int, compute **c = (int) f**.  
                          iii.    If the field is a long, compute **c = (int) (f ^ (f >>> 32))**.  
                          iv.    If the field is a float, compute **c = Float.floatToIntBits(f)**.  
                          v.     If the field is a double, compute  
                                             **long l = Double.doubleToLongBits(f)**,  
                                             **c = (int)(l ^ (l >>> 32))**  
                          vi.    If the field is an object reference then equals( ) calls equals( ) for this field. compute  
                                             **c =  f.hashCode()**   
                          vii.   If the field is an array, treat it as if each element were a separate field.  
                                 That is, compute a hashCode for each significant element by applying above rules to  each  
                                 element

                b.  Combine the hashCode c computed in step 2.a into result as follows:  
                                             **result = 37 \* result + c;**

       3. Return result.   
       4. Look at the resulting hashCode() and make sure that equal instances have equal hash codes.

Here is an example of a class that follows the above guidelines

public class HashTest { private String field1; private short field2; ---- @Override public int hashCode() { int result = 17; result = 37\*result + field1.hashCode(); result = 37\*result + (int)field2; return result; }}

You can see that a constant **37** is chosen. The purpose to choose this number is that it is a **prime number**. We can choose any other prime number. Using prime number the objects will be distributed better over the buckets. I encourage user to explore the topic further by checking out other resources.

#### ****Apache HashCodeBuilder****

Writing a good hashCode() method is not always easy. Since it can be difficult to implement hashCode() correctly, it would be helpful if we have some reusable implementations of these.

The **Jakarta-Commonsorg.apache.commons.lang.builder** package is providing a class named **HashCodeBuilder** which is designed to help implementing hashCode() method. Usually developers struggle hard with implementing hashCode() method and this class aims to simplify the process.

Here is how you would implement a hashCode algorithm for our above class

public class HashTest { private String field1; private short field2; ---- @Override public int hashCode() { return new HashCodeBuilder(83, 7) .append(field1) .append(field2) .toHashCode(); }}

Note that the two numbers for the constructor are simply two different, non-zero, odd numbers - these numbers help to avoid collisions in the hashCode value across objects.

If required, the superclass hashCode() can be added using **appendSuper(int)**.

You can see how easy it is to override HashCode() using Apache HashCodeBuilder.

## ****Mutable Object As Key****

It is a general advice that you should use **immutable object as a key** in a Collection. HashCode work best when calculated from immutable data. If you use **Mutable object as key** and change the state of the object so that the hashCode changes, then the store object will be in the wrong bucket in the Collection

The most important thing you should consider while implementing hashCode() is that regardless of when this method is called, it should produces the same value for a particular object every time when it is called. If you have a scenario like the object produces one hashCode() value when  it is **put()** in to a HaspMap and produces another value during a **get()**, in that case you would not be able to retrieve that object. Therefore, if you hashCode() depends on mutable data in the object, then made changing those data will surely produce a different key by generating a different hashCode().

Look at the example below

public class Employee { private String name; private int age; public Employee() { } public Employee(String name, int age) { this.name = name; this.age = age; } public String getName() { return name; } public void setName(String name) { this.name = name; } public int getAge() { return age; } public void setAge(int age) { this.age = age; } @Override public boolean equals(Object obj) { //Remember: Some Java gurus recommend you avoid using instanceof if (obj instanceof Employee) { Employee emp = (Employee)obj; return (emp.name == name && emp.age == age); } return false; } @Override public int hashCode() { return name.length() + age; } public static void main(String[] args) { Employee e = new Employee("muhammad", 24); Map<Object, Object> m = new HashMap<Object, Object>(); m.put(e, "Muhammad Ali Khojaye"); // getting output System.out.println(m.get(e)); e.name = "abid"; // it fails to get System.out.println(m.get(e)); e.name = "amirrana"; // it fails again System.out.println(m.get(new Employee("muhammad", 24))); }}

So you can see in the above examples that how we are getting some unpredictable results.

You can easily fix the above by overriding the hashCode() using either **Joshu Recipe** or using **HashCodeBuilder** class.

Here is an example,

#### ****Joshu Recommendation****

@Overridepublic int hashCode() { int result = 17; result = 37\*result + name.hashCode(); result = 37\*result + age; return result;}

#### ****Using HashCodeBuilder****

@Overridepublic int hashCode() { return new HashCodeBuilder(83, 7) .append(name) .append(age) .toHashCode();}

## ****Another Example of Mutable Field as Key****

Let consider the example

public class HashTest { private int mutableField; private final int immutableField; public HashTest(int mutableField, int immutableField) { this.mutableField = mutableField; this.immutableField = immutableField; } public void setMutableField(int mutableField) { this.mutableField = mutableField; } @Override public boolean equals(Object o) { if(o instanceof HashTest) { return (mutableField == ((HashTest)o).mutableField) && (immutableField == ((HashTest)o).immutableField); }else { return false; } } @Override public int hashCode() { int result = 17; result = 37 \* result + this.mutableField; result = 37 \* result + this.immutableField; return result; } public static void main(String[] args) { Set<HashTest> set = new HashSet<HashTest>(); HashTest obj = new HashTest(6622458, 626304); set.add(obj); System.out.println(set.contains(obj)); obj.setMutableField(3867602); System.out.println(set.contains(obj)); }}

After changing mutableField, the computed hashCode is no longer pointing to the old bucket and the contains() returns false.

We can tackle such situation using either of these methods

* **Hashcode** is best when calculated from **immutable data**; therefore ensure that only immutable object would be used as key with Collections.
* Implement the hashCode() using our first technique i.e. return a constant value but you must aware that it would kills all those advantage of bucket mechanism.
* If you need mutable fields included in the hashCode method then you can calculate and store the hash value when the object is created and whenever you update mutable field, you must first remove it from the collection(set/map) and then add it back to the collection after updating it.

## ****References and More Information****

[Effective Java](http://java.sun.com/docs/books/effective/)

[Object Class](http://java.sun.com/javase/6/docs/api/java/lang/Object.html)

[HashCodeBuilder](http://commons.apache.org/lang/apidocs/org/apache/commons/lang/builder/HashCodeBuilder.html)

<http://www.javaranch.com/>

- See more at: <http://muhammadkhojaye.blogspot.com/2010/02/java-hashing.html>

The Java Zone is brought to you in partnership with [ZeroTurnaround](https://dzone.com/go?i=2221&u=http%3A%2F%2Fzeroturnaround.com%2F). Discover how you can skip the build and redeploy process by using [JRebel](https://dzone.com/go?i=2221&u=http%3A%2F%2Fzeroturnaround.com%2Fsoftware%2Fjrebel%2F).