


# Java's Object Methods: hashCode()



Adam McQuistan 



## Introduction

This article is a continuation of a series of articles describing the often forgotten about methods of the Java language's base Object class. The following are the methods of the base Java Object which are present in all Java objects due to the implicit inheritance of Object.

- [toString](#)
- [toClass](#)
- [equals](#)
- `hashCode` (*you are here*)
- [clone](#)
- [finalize](#)
- [wait & notify](#)

The focus of this article is the `hashCode()` method which is used to generate a numerical representation of the contents of an object and is used heavily in the collections framework.

## Why the hashCode() Method is Important

The purpose of the `hashCode()` method is to provide a numeric representation of an object's contents so as to provide an alternate mechanism to loosely identify it.

By default the `hashCode()` returns an integer that represents the internal memory address of the object. Where this comes in handy is in the creation and use of an important computer science data structure called a [hash table](#). Hash tables map keys, which are values that result from a hash function (aka, `hashCode()` method), to a value of interest (i.e., the object the `hashCode()` method was executed on). This becomes a very useful feature when dealing with moderate-to-large collections of items, because it is usually a lot faster to compute a hash value compared to linearly searching a collection, or having to resize and copy items in an array backing a collection when it's limit is reached.

The driving feature behind an efficient hash table is the ability to create a hash that is adequately unique for each object. Buried in that last sentence is the reason why I emphasized the need to override both `equals(Object)` and `hashCode()` in the prior article.

If an object has implementation characteristics that require it to be logically distinct from others based on its content then it needs to produce as distinct a hash as reasonably possible. So two objects that are logically equivalent should produce the same hash, but it is sometimes unavoidable to have two logically different objects that may produce the same hash which is known as a **collision**. When collisions happen the colliding objects are placed in a metaphorical bucket and a secondary algorithm is used to differentiate them within their hash bucket.



## Demonstrating Hash Table Usage

In Java the concept of a hash table is conceptualized in the `java.util.Map` interface and implemented in the `java.util.HashMap` class.

We'll demonstrate a hash table and why it is important to have a reasonably unique hash value computed by `hashCode()` when a class implementation warrants the notion of logical equality consider the following class and program.

Person.java

```
import java.time.LocalDate;

public class Person {
    private final String firstName;
    private final String lastName;
    private final LocalDate dob;

    public Person(String firstName, String lastName, LocalDate dob) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.dob = dob;
    }

    // omitting getters for brevity

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Person)) {
            return false;
        }
        Person p = (Person)o;
        return firstName.equals(p.firstName)
            && lastName.equals(p.lastName)
            && dob.equals(p.dob);
    }
}
```

Main.java

```
import java.time.LocalDate;
import java.util.HashMap;
import java.util.Map;

public class Main {
```

```

public static void main(String[] args) {
    Map<Person, String> peopleMap = new HashMap<>();
    Person me = new Person("Adam", "McQuistan", LocalDate.parse("1987-09-23"));
    Person me2 = new Person("Adam", "McQuistan", LocalDate.parse("1987-09-23"));
    System.out.println("Default hash: " + me.hashCode());
    System.out.println("Default hash: " + me2.hashCode());

    peopleMap.put(me, me.toString());
    System.out.println("me and me2 same? " + me.equals(me2));
    System.out.println("me2 in here? " + peopleMap.containsKey(me2));
}
}

```

Output:

```

Default hash: 1166726978
Default hash: 95395916
me and me2 same? true
me2 in here? false

```

As you can see from the output the default hash of `me` and `me2` are not equal even though the custom implementation of `equals(Object)` indicates that they are logically the same. This results in two distinct entries in the hash table even though you would expect only one, which opens the doors to some nasty bugs in a program if it were to implement this code.



Let me improve the `Person` class by ensuring that the `hashCode()` method returns the same value for the equal instance objects `me` and `me2`, like so:

Person.java

```

public class Person {
    // omitting all other stuff for brevity

    @Override
    public int hashCode() {
        return 31;
    }
}

```

Main.java

```

public class Main {
    public static void main(String[] args) {
        Map<Person, String> peopleMap = new HashMap<>();
        Person me = new Person("Adam", "McQuistan", LocalDate.parse("1987-09-23"));
        Person me2 = new Person("Adam", "McQuistan", LocalDate.parse("1987-09-23"));
        Person you = new Person("Jane", "Doe", LocalDate.parse("1999-12-25"));
        System.out.println("Default hash: " + me.hashCode());
        System.out.println("Default hash: " + me2.hashCode());
    }
}

```

```

    peopleMap.put(me, me.toString());
    System.out.println("me and me2 same? " + me.equals(me2));
    System.out.println("me2 in here? " + peopleMap.containsKey(me2));

    peopleMap.put(me2, me2.toString());
    peopleMap.put(you, you.toString());
    for(Person p : peopleMap.keySet()) {
        String txt = peopleMap.get(p);
        System.out.println(txt);
    }
}
}

```

Output:

```

Default hash: 31
Default hash: 31
me and me2 same? true
me2 in here? true
<Person: firstName=Adam, lastName=McQuistan, dob=1987-09-23>
<Person: firstName=Jane, lastName=Doe, dob=1999-12-25>

```

Ok, so now I have equal hash values for equal objects, but it is also clear that non-equal objects will also always have the same hash values.

First I will explain what is happening as the equal objects `me` and `me2` are added to the HashMap. When the `me2` `Person` instance is added to the HashMap already containing the `me` instance, the HashMap notices that the hash is the same and then it determines that they are also logically equivalent via the `equals(Object)` method. This results in the HashMap simply replacing the first `me` with the second `me2` at that location in the hash table.

Next comes the `you` instance, which again has the same hash value, but this time the HashMap identifies that it is logically different from the existing hash in that bucket `me2`. This leads to the HashMap adding the `you` instance to the bucket, turning that bucket into a list-like collection. For small numbers of collisions this doesn't have too great an impact, but in my example above, where every instance is guaranteed to have the same hash value, the bucket representing 31 in the HashMap will rapidly degrade to a poor implementation of a list for the entire HashMap.

At this point in time I would like to further demonstrate the ineffectiveness of this solution with concrete data to compare against the final implementation that will follow.

## Free eBook: Git Essentials

Check out our hands-on, practical guide to learning Git, with best-practices, industry-accepted standards, and included cheat sheet. Stop Googling Git commands and actually *learn* it!

Below is a program that builds two equally sized collections, `peopleList` and `peopleMap`, of `Person` instances with equally sized random names and birthdays selected. I will measure the amount of time it takes to build the collections for a first comparison measurement. Next I will measure the amount of time it takes to search each collection for the existence of an equally placed known instance, `me`.

```
import java.time.Duration;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.stream.Collectors;

public class Main {
    private static final char[] alphabet = "abcdefghijklmnopqrstuvwxyz".toCharArray();

    public static void main(String[] args) {
        Person me = new Person("Adam", "McQuistan", LocalDate.parse("1987-09-23"));

        LocalDateTime start = LocalDateTime.now();
        List<Person> peopleList = new ArrayList<>();
        for (int i = 0; i < 10000; i++) {
            if (i == 4999) {
                peopleList.add(me);
            }
            peopleList.add(new Person(getRandomName(), getRandomName(), getRandomDate()));
        }
        System.out.println("Microseconds to build list: " + getTimeElapsed(start, LocalDateTime.now()));

        start = LocalDateTime.now();
        Map<Person, String> peopleMap = new HashMap<>();
        for (int i = 0; i < 10000; i++) {
            if (i == 4999) {
                peopleMap.put(me, me.toString());
            }
            Person p = new Person(getRandomName(), getRandomName(), getRandomDate());
            peopleMap.put(p, p.toString());
        }
        System.out.println("Microseconds to build map: " + getTimeElapsed(start, LocalDateTime.now()));

        start = LocalDateTime.now();
        boolean found = peopleList.contains(me);
        System.out.println("Microseconds to search list is " + getTimeElapsed(start, LocalDateTime.now()));

        start = LocalDateTime.now();
        found = peopleMap.containsKey(me);
        System.out.println("Microseconds to search map is " + getTimeElapsed(start, LocalDateTime.now()));
    }

    public static String getRandomName() {
        int size = alphabet.length;
        Random rand = new Random();
        List<Character> chars = Arrays.asList(
            alphabet[rand.nextInt(size)],
            alphabet[rand.nextInt(size)],
            alphabet[rand.nextInt(size)],
            alphabet[rand.nextInt(size)]
        );
    }
}
```

```

    );
    return chars.stream().map(String::valueOf).collect(Collectors.joining());
}

public static LocalDate getRandomDate() {
    Random rand = new Random();
    int min = (int) LocalDate.of(1980, 1, 1).toEpochDay();
    int max = (int) LocalDate.of(2018, 10, 14).toEpochDay();
    long day = min + rand.nextInt(max - min);
    return LocalDate.ofEpochDay(day);
}

public static long getTimeElapsed(LocalDateTime start, LocalDateTime end) {
    Duration duration = Duration.between(start, end);
    return Math.round(duration.getNano() / 1000);
}
}

```

Output:

```

Microseconds to build list: 53789
Microseconds to build map: 892043
Microseconds to search list is 450
Microseconds to search map is 672

```

Wow that is grossly inefficient! This great hash table implementation in HashMap has been completely degraded to a terrible implementation of a list-like structure. Even worse is that arguably one of the primary reasons for using a hash table is to have rapid  $O(1)$  searching and retrieval of values via key access, but as you can see that is actually performing worse than searching a standard list linearly due my implementation of a `hashCode()` that has no differentiating capability. Yikes!

Let me fix this. There are a few ways that I know of to approach implementing a reasonably functioning `hashCode()` method and I will explain them below.

### A. `hashCode()` by Hand

In the book [Effective Java: best practices for the Java platform, 3rd edition](#) Java guru Joshua Bloch describes the following algorithm for implementing your own `hashCode()` method.

- i) compute the hash of the first deterministic class field used in the implementation of `equals(Object)` and assign that to a variable I'll call `result`.
- ii) for each remaining deterministic field used the `equals(Object)` implementation multiply `result` by 31 and add the hash value of the deterministic field.

In my `Person` example class this approach looks something like this:

```

public class Person {
    private final String firstName;
    private final String lastName;
    private final LocalDate dob;

    // omitting all other stuff for brevity

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Person)) {
            return false;
        }
    }
}

```

```

    }
    Person p = (Person)o;
    return firstName.equals(p.firstName)
        && lastName.equals(p.lastName)
        && dob.equals(p.dob);
}

@Override
public int hashCode() {
    int result = dob == null ? 1 : dob.hashCode();
    result = 31 * result + firstName == null ? 0 : firstName.hashCode();
    result = 31 * result + lastName == null ? 0 : lastName.hashCode();
    return result;
}
}

```



Now if I rerun the same program that builds the `List` and `HashMap` measuring execution time I should see a significant difference.

Output:

```

Microseconds to build list: 54091
Microseconds to build map: 35528
Microseconds to search list is 582
Microseconds to search map is 20

```

Pretty shocking right!? The `HashMap` itself is built in nearly half the time, plus the time required to find the `me` object is on an entirely different level of magnitude.

## B. Using `Objects.hash(...)`

If you are looking for a simpler way to implement a custom hash value and are not extremely averse to not having the most performant implementation then its a good idea to reach for the `Objects.hash(...)` utility and pass it the deterministic fields of your object. This is a generally well performing method, and if you are like me and favor being able to quickly ship code rather than prematurely optimizing for performance, this is a great route to solving this problem.

Below is an example of this implementation for the `Person` class:

```

public class Person {
    // omitting all other stuff for brevity

    @Override
    public int hashCode() {
        return Objects.hash(dob, firstName, lastName);
    }
}

```

Here is the output for the analysis program:

```
Microseconds to build list: 56438
Microseconds to build map: 38112
Microseconds to search list is 733
Microseconds to search map is 24
```

As you can see it is essentially identical to the hand rolled implementation.



### C. Autogeneration with IDE

My preferred method to implementing both the `equals(Object)` and `hashCode()` methods are actually to use the autogeneration functionality in my Java IDE of choice Eclipse. The implementation that Eclipse provides is shown below.

```
public class Person {

    // omitting all other stuff for brevity

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((dob == null) ? 0 : dob.hashCode());
        result = prime * result + ((firstName == null) ? 0 : firstName.hashCode());
        result = prime * result + ((lastName == null) ? 0 : lastName.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Person other = (Person) obj;
        if (dob == null) {
            if (other.dob != null)
                return false;
        } else if (!dob.equals(other.dob))
            return false;
        if (firstName == null) {
            if (other.firstName != null)
                return false;
        } else if (!firstName.equals(other.firstName))
            return false;
        if (lastName == null) {
            if (other.lastName != null)
                return false;
```



```
    } else if (!lastName.equals(other.lastName))  
        return false;  
    return true;  
}  
}
```

And the output from the analysis program is this:

```
Microseconds to build list: 53737  
Microseconds to build map: 27287  
Microseconds to search list is 1500  
Microseconds to search map is 22
```

Again this implementation is nearly identical in performance.

## Conclusion

In this article I have, to the best of my ability, explained the importance of co-implementating the `hashCode()` method along with `equals(Object)` in order to efficiently work with data structures that apply the notion of a hash table. In addition to explaining why it is important to implement the `hashCode()` method I also demonstrated how to implement a few reasonably performant and robust hashing algorithms.

As always, thanks for reading and don't be shy about commenting or critiquing below.

#java

Last Updated: August 30th, 2019

Was this article helpful? ☆☆☆☆☆



## Improve your dev skills!

Get tutorials, guides, and dev jobs in your inbox.

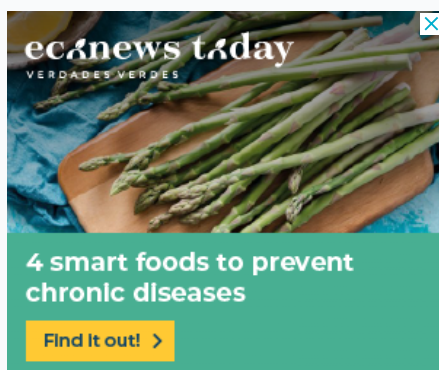
Sign Up

No spam ever. Unsubscribe at any time. Read our [Privacy Policy](#).

**Adam McQuistan** *Author*



I am both passionate and inquisitive about all things software. My background is mostly in Python, Java, and JavaScript in the areas of science but, have also worked on large ecommerce and ERP apps.



### Senior Go Developer

**Toptal** a month ago

### Senior DevOps Engineer (m/f/x)

**refurbed** a month ago

### Full Stack Engineer

**Refinable** 2 months ago

### Senior React Developer - Remote

**Toptal** 3 months ago

➔ More Jobs

Jobs by [HireRemote.io](https://hireremote.io)

## Prepping for an interview?

Improve your skills by solving one coding problem every day

Get the solutions the next morning via email

Practice on **actual problems** asked by top companies, like:

📄 Daily Coding Problem

The advertisement features a dark blue background with a silhouette of a person holding a tablet. At the top right is a 'G2 Leader 2021' badge. The Dynatrace logo is in the top left. The main text reads 'Dynatrace is the G2 APM leader outscoring Splunk'. Below this, it states 'G2 satisfaction score:' followed by two bars: 'Dynatrace 99%' in a blue bar and 'Splunk Enterprise 72%' in a grey bar. A 'Read the report' button is at the bottom right.

**dynatrace**

**Dynatrace is the G2 APM leader outscoring Splunk**

G2 satisfaction score:

Company	Satisfaction Score
Dynatrace	99%
Splunk Enterprise	72%

[Read the report](#)

## Better understand your data with visualizations

---

With over 330+ pages, you'll learn the ins and outs of visualizing data in Python with popular libraries like Matplotlib, Seaborn, Bokeh, and more.

[Learn more →](#)

The advertisement features a dark blue background with a silhouette of a person holding a tablet. At the top right is a G2 'Leader' badge for Q4 2021. The Dynatrace logo is in the top left. The main headline reads 'Dynatrace is the G2 APM leader outscoring Splunk'. Below this, it states 'G2 satisfaction score:' followed by two horizontal bars: a blue bar for 'Dynatrace 99%' and a grey bar for 'Splunk Enterprise 72%'. At the bottom is a blue button that says 'Read the report'.

**dynatrace**

**Dynatrace is the G2 APM leader outscoring Splunk**

G2 satisfaction score:

Product	Satisfaction Score
Dynatrace	99%
Splunk Enterprise	72%

[Read the report](#)

