Object Semantics and Heap Management: Equality vs. Identity

Almost all Java developers know how important it is to implement both equals() and hashCode() in their custom classes. However, there are important differences between "equality" and "identity" that can affect your implementation strategy. Read on to find out more!

by Emil Forslund MVB · Mar. 03, 16 · Java Zone · Analysis

When storing objects in a Set, it is important that the same object can never be added twice. That is the core definition of a Set. In Java, two methods are used to determine whether two referenced objects are the same or if they can both exist in the same Set; equals() and hashCode(). In this article, I will explain the difference between equality and identity and also take up some of the advantages they have over each other.

Java offers a standard implementation of both these methods. The standard equals()-method is defined as an "identity" comparing method. It means that it compares the two memory references to determine if they are the same. Two identical objects that are stored in different locations in the memory will, therefore, be deemed unequal. This comparison is done using the ==-operator, as can be seen if you look at the source code of the Object-class.

```

public boolean equals(Object obj) {
    return (this == obj);
}
```

The hashCode()-method is implemented by the virtual machine as a native operation so it is not visible in the code, but it is often realized as simply returning the memory reference (on 32-bit architectures) or a modulo 32 representation of the memory reference (on a 64-bit architecture).
One thing many programmers choose to do when designing classes is to override this method with a different equality definition where instead of comparing the memory reference, you look at the values of the two instances to see if they can be considered equal. Here is an example of that:

```java

import java.util.Objects;
import static java.util.Objects.requireNonNull;

public final class Person {

    private final String firstname;
    private final String lastname;

    public Person(String firstname, String lastname) {
        this.firstname = requireNonNull(firstname);
        this.lastname  = requireNonNull(lastname);
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 83 * hash + Objects.hashCode(this.firstname);
        hash = 83 * hash + Objects.hashCode(this.lastname);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        final Person other = (Person) obj;
        if (!Objects.equals(this.firstname, other.firstname)) {
            return false;
        } else return Objects.equals(this.lastname, other.lastname);
    }
}
```

This comparison is called "equality" (compared to the previous "identity"). As long as two persons have the same first- and last name, they will be considered equal. This can for an example be used to sort out duplicates from a stream of input. Remember that if you override the equals()-method, you should always override the hashCode()-method as well!

Equality

Now, if you choose equality over identity, there are some things you will need to think about. The first thing you must ask yourself is: are two instances of this class with the same properties necessarily the same? In the case of Person above, I would say no. It is very likely that you will someday have two people in your system with the same first- and last name. Even if you continue to add more personal information like a birthday or favorite color, you will sooner or later have a collision. On the other hand, if your system is handling cars and each car contains a reference to a "model", it can be safely assumed that if two cars both are black Tesla model S, they are probably the same model even if the objects are stored in different places in the memory. That is an example of a case when equality can be good.

```java

import java.util.Objects;
import static java.util.Objects.requireNonNull;

public final class Car {

    public static final class Model {

        private final String name;
        private final String version;

        public Model(String name, String version) {
            this.name    = requireNonNull(name);
            this.version = requireNonNull(version);
        }

        @Override
        public int hashCode() {
            int hash = 5;
            hash = 23 * hash + Objects.hashCode(this.name);
            hash = 23 * hash + Objects.hashCode(this.version);
            return hash;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null) return false;
            if (getClass() != obj.getClass()) return false;
            final Model other = (Model) obj;
            if (!Objects.equals(this.name, other.name)) {
                return false;
            } else return Objects.equals(this.version, other.version);
        }
    }

    private final String color;
    private final Model model;

    public Car(String color, Model model) {
        this.color = requireNonNull(color);
        this.model = requireNonNull(model);
    }

    public Model getModel() {
        return model;
    }
}
```

Two cars are only considered the same if they have the same memory address. Their models, on the other hand, are considered the same as long as they have the same name and

Two cars are only considered the same if they have the same memory address. Their models, on the other hand, are considered the same as long as they have the same name and version. Here is an example of this:

```java
final Car a = new Car("black", new Car.Model("Tesla", "Model S"));
final Car b = new Car("black", new Car.Model("Tesla", "Model S"));

System.out.println("Is a and b the same car? " + a.equals(b));
System.out.println("Is a and b the same model? " + a.getModel().equals(b.getModel()));

// Prints the following:
// Is a and b the same car? false
// Is a and b the same model? true
```

Identity

One risk of choosing equality over identity is that it can be an invitation to allocating more objects than necessarily on the heap. Just look at the car example above. For every car, we create we also allocate space in memory for a model. Even if java generally optimizes string allocation to prevent duplicates, it is still a certain waste for objects that will always be the same. A short trick to turn the inner object into something that can be compared using identity comparing method and at the same time avoid unnecessary object allocation is to replace it with an enum:

```java
public final class Car {

    public enum Model {

        TESLA_MODEL_S ("Tesla", "Model S"),
        VOLVO_V70     ("Volvo", "V70");

        private final String name;
        private final String version;

        Model(String name, String version) {
            this.name    = name;
            this.version = version;
        }
    }

    private final String color;
    private final Model model;

    public Car(String color, Model model) {
        this.color = requireNonNull(color);
        this.model = requireNonNull(model);
    }

    public Model getModel() {
        return model;
    }
}
```

Now we can be sure that each model will only ever exist in one place in memory and can therefore safely be compared using identity comparison. An issue with this, however, is that it really limits our extendability. Before with could define new models on the fly without modifying the source code in the Car.java-file, but now we have locked ourselves into an enum that should generally be kept unmodified. If those properties are desired, an equals comparison is probably better for you.

A finishing note, if you have overridden the equals() and hashCode()-methods of a class and later want to store it in a Map based on identity, you can always use the IdentityHashMap structure. It will use the memory address to reference its keys, even if the equals()- and hashCode()-methods have been overridden.

Implementing or Overriding Equals the Right Way

Free DZone Refcard

Getting Started With Kotlin

Topics: JAVA , SET , EQUALITY , IDENTITY

Java Partner Resources

Designing Reactive Systems: The Role Of Actors In Distributed Architecture
Lightbend

Level up your code with a Pro IDE
JetBrains

Microservices for Java Developers: A Hands-On Introduction to Frameworks & Containers
Red Hat Developer Program

Predictive Analytics + Big Data Quality: A Love Story
Melissa Data

Free, Fast, Open, Production-Proven, and All Java: OpenJ9

Take a deep dive into OpenJ9, a high-performance, production-ready, enterprise-grade JVM implementation that serves as an alternative to HotSpot.

by Markus Eisele MVB · Jun 09, 18 · Java Zone · Tutorial

This article is featured in the new DZone Guide to Java: Features, Improvements, & Updates. Get your free copy for more insightful articles, industry statistics, and more!

Eclipse OpenJ9 is a high-performance, scalable, Java virtual machine (JVM) implementation. It is not new. It has been part of the IBM Java Development Kit for many years and it is safe to say that it is an enterprise-grade, production-level component of many large Java-based systems. At the end of last year, it was contributed to the Eclipse Foundation by IBM. OpenJ9 is an alternative to the Hotspot JVM currently mostly used within OpenJDK.

The History

Although the Eclipse OpenJ9 project hasn't been around for very long, the VM itself has been around for years. It was launched during the fourth JavaOne and parts of the codebase can be traced back to Smalltalk.

It has been powering IBM middleware products for the last decade or more. IBM contributed the VM to the Eclipse Foundation end of 2017 and more than 70 IBM developers are actively involved in the project.

The long-term goal of the Eclipse OpenJ9 project is to foster an open ecosystem of JVM developers that can collaborate and innovate with designers and developers of hardware platforms, operating systems, tools, and frameworks. The Java community has benefited over its history from having multiple implementations of the JVM specification competing to provide the best runtime for your application. Whether adding compressed references, new cloud features, AOT (ahead-of-time) compilation, or straight up faster performance and lower memory use, the ecosystem has improved through that competition. Eclipse OpenJ9 aims to continue to spur innovation in the runtimes space.

As an essential part of this, the teams work closely with the AdoptOpenJDK efforts, which are led by the London Java Community. If you look closely, you realize that a large part of OpenJ9 has been living at the Eclipse Foundation for a couple of years now. Eclipse OpenJ9 embeds Eclipse OMR, which provides cross-platform components for building reliable, high-performance language runtimes.

Eclipse OpenJ9 takes OMR and adds extra code that turns it into a runtime environment for Java applications.

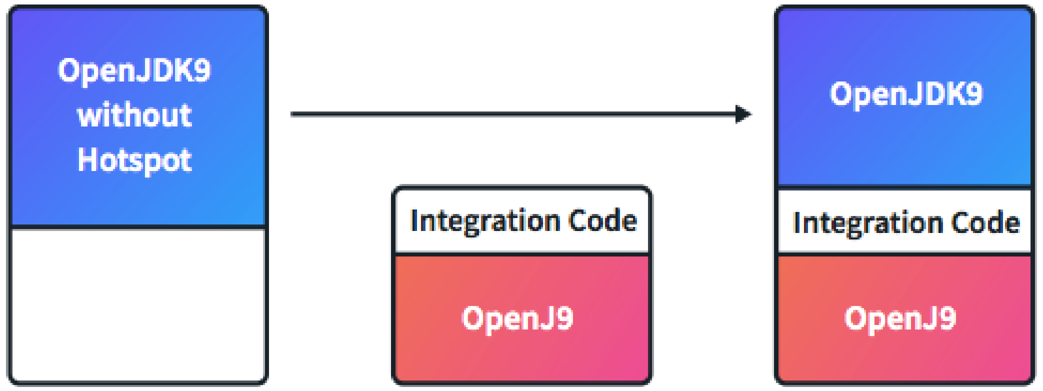
What Exactly Is Exactly Is OpenJ9 and How Does it Relate to OpenJDK?

OpenJ9 replaces HotSpot JVM in the OpenJDK build and becomes a new JVM runtime implementation that can be used with a wide range of system architectures and operating systems. Currently, there are pre-built OpenJDK binaries with OpenJ9 as the JVM available for Java versions 8, 9, and 10 via the AdopOpenJDK project website. A JVM is not the complete Java Development Kit (JDK). It is the execution engine. While the JDK provides tools, class libraries, and more general things that you need to build your application, the

JVM is responsible to run the actual bytecode. It is a replacement for the commonly used and known Hotspot JVM which was implemented by Oracle.

In order to create an OpenJDK build with OpenJ9 as the JVM, the OpenJDK project is mirrored into a separate repository that removes Hotspot and provides a place for the needed integration code. The integration code consists mostly of changes to Makefiles and configuration, some patches to the class library code (JCL), and some minor patches for additional OpenJ9 features.

The integration layer mitigates the few places where the interfaces from Hotspot and OpenJ9 are slightly different. All in all, it is roughly 4,000 lines of code, which is really small compared to the size of OpenJ9 or the OpenJDK. The primary goal is to keep this to a necessary minimum and also contribute as much as possible upstream to the OpenJDK.



With OpenJDK, with the integration code and OpenJ9 coming together via a build process, the end result is a binary that is available for a range of different platforms.

You can find more information about the build process and how to build your own OpenJDK with OpenJ9 on the project website.

Why OpenJ9 Is the Perfect Cloud Runtime

The requirements of cloud runtimes today are different to the characteristics employed by individual, physical machines in the datacenters from just a couple of years ago. A small memory footprint allows a higher application density for providers and reduces the cost per application for users as they can run more instances with less resources.

Another important feature is a fast startup, which lets application instances scale faster and more smoothly.

The OpenJ9 project uses the Daytrader application to monitor performance improvements in comparison with Hotspot. With a 66% smaller memory footprint after startup and a 42% faster startup time, it reaches a comparable throughput in a constraint environment like a Docker container.

Uinque Features and Advantages

The lower memory footprint and faster startup times demonstrate the practical impact of some of the advantages and implementation differences over the classical Hotspot JVM.

The lower memory footprint and faster startup times demonstrate the practical impact of some of the advantages and implementation differences over the classical HotspotVM.

Application Class Sharing and AOT Compilation

The first time an application runs, the JVM has to load all the classes that are required to start the application. This process takes time. If the needed classes are stored in cache, the second application run will be much faster. Another implementation difference is that the JVM doesn't store the Java bytecode classes but an optimized ROMClass version of it. These read-only, optimized versions are carrying symbolic data only. When the JVM is executing a class, it has to be able to do a bunch of lookups, e.g. finding the method it is actually invoking, and it also needs to be able to work on data and save this data. This is where the J9RAMClass comes in. It is a cache for the data at runtime and carries the live data for a particular class.

Further on, the ROMClasses can be shared between multiple JVMs on the same machine. OpenJ9 always shares both the bootstrap and application classes that are loaded by the default system class loader. A detailed view on how this works is available in this article. And there is something else that can be optimized with this shared class cache ahead-of-time compilation (AOT). Unlike just-in-time (JIT) compilation, AOT is dynamically compiling methods into ATO code at runtime and placing these ROMClasses into the shared cache. The VM automatically chooses which methods should be AOT-compiled based on heuristics that identify the start-up phase of large applications. It is enabled via the -Xshareclasses option, which is highly configurable.

As for AOT itself, it works straight out of the box when you enable shared classes and doesn't require any special tuning.

Tuning for Cloud Environments

The -Xtune:virtualized option is designed to configure OpenJ9 for typical cloud deployments where VM guests are provisioned with a small number of virtual CPUs to maximize the number of applications that can be run. The -Xquickstart option enables an ultra-fast startup of OpenJ9 and works best with short-lived tasks. But be aware that you may trade in the peak throughput capabilities. You can also specify -XX:+IdleTuningGcOnIdle on the command line. When set, OpenJ9 determines whether an application is idle based on CPU utilization and other internal heuristics. When an idle state is recognized, a GC cycle runs if there is significant garbage in the heap and releases unused memory back to the operating system. A more detailed overview is provided in this article.

Garbage Collection in OpenJ9

Eclipse OpenJ9 has a number of GC policies designed around different types of applications and workloads. The Generational Concurrent (-Xgcpolicy:gencon) GC policy is the default policy employed by the JVM. But there are four other alternatives: -Xgcpolicy:balanced, -Xgcpolicy:metronome, -Xgcpolicy:optavgpause, and -Xgcpolicy:optthruput. The Metronome GC (-Xgcpolicy:metronome) policy is especially interesting if your application depends on precise response times and you are running on x86 Linux. The main difference between Metronome and other policies is that garbage collection occurs in small interruptible steps rather than stopping an application completely while garbage is marked and collected.

By default, Metronome pauses for three milliseconds at a time. A full garbage collection cycle requires many pauses, which are spread out to give the application enough time to run. You can limit the amount of CPU that the GC process uses and you can control the pause time. A more complete overview about the individual policies can be found in this article.

Get Started

The AdoptOpenJDK project releases pre-built binaries for Linux x64, Linux s390x, Linux ppc64, and AIX ppc64. The easiest way to get started is to download the one for your operating system and start working with it — although you have the ability to build binaries for other platforms like Linux ARMv7 yourself. Docker images are available via DockerHub.

The JVM documentation is extensive and well-structured to get you started with optimizing for your application.

This article is featured in the new DZone Guide to Java: Features, Improvements, & Updates. Get your free copy for more insightful articles, industry statistics, and more!

Topics: JAVA, OPENJ9, JVM, TUTORIAL