



[New Guide] Download the 2018 Guide to Open Source: Democratizing Development

[Download Guide](#) ▶

Java Memory Management

by Constantin Marian · Jan. 07, 18 · Java Zone · Tutorial

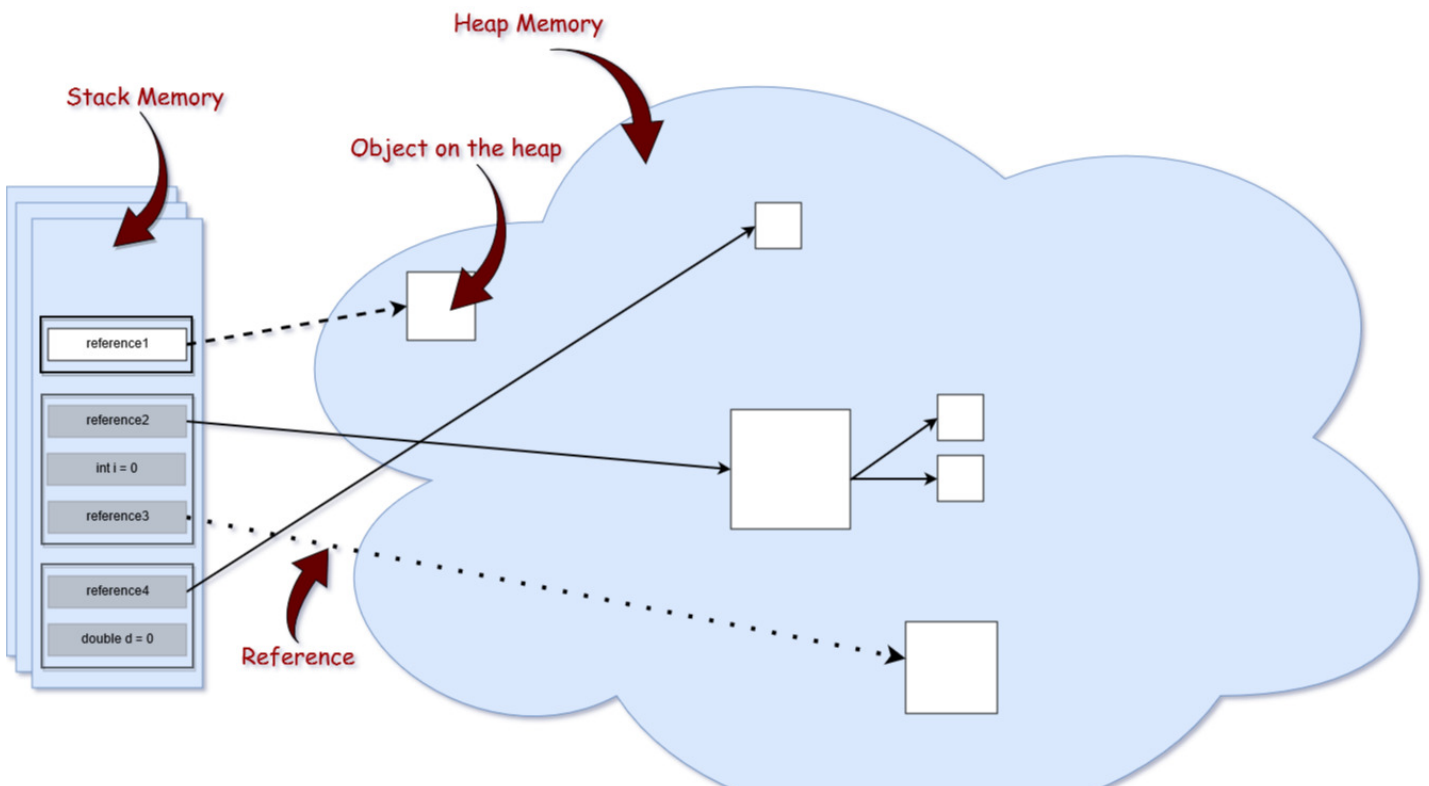
Build vs Buy a Data Quality Solution: Which is Best for You? Gain insights on a hybrid approach. Download white paper now!

You might think that if you are programming in Java, what do you need to know about how memory works? Java has automatic memory management, a nice and quiet garbage collector that quietly works in the background to clean up the unused objects and free up some memory.

Therefore, you as a Java programmer do not need to bother yourself with problems like destroying objects, as they are not used anymore. However, even if this process is automatic in Java, it does not guarantee anything. By not knowing how the garbage collector and Java memory is designed, you could have objects that are not eligible for garbage collecting, even if you are no longer using them.

So knowing how memory actually works in Java is important, as it gives you the advantage of writing high-performance and optimized applications that will never ever crash with an `OutOfMemoryError`. On the other hand, when you find yourself in a bad situation, you will be able to quickly find the memory leak.

To start with, let's have a look at how the memory is generally organized in Java:



Memory Structure

Generally, memory is divided into two big parts: the **stack** and the **heap**. Please keep in mind that the size of memory types in this picture are not proportional to the memory size in reality. The heap is a huge amount of memory compared to the stack.

The Stack

Stack memory is responsible for holding references to heap objects and for storing value types (also known in Java as primitive types), which hold the value itself rather than a reference to an object from the heap.

In addition, variables on the stack have a certain visibility, also called **scope**. Only objects from the active scope are used. For example, assuming that we do not have any global scope variables (fields), and only local variables, if the compiler executes a method's body, it can access only objects from the stack that are within the method's body. It cannot access other local variables, as those are out of scope. Once the method completes and returns, the top of the stack pops out, and the active scope changes.

Maybe you noticed that in the picture above, there are multiple stack memories displayed. This is due to the fact that the stack memory in Java is allocated per Thread. Therefore, each time a Thread is created and started, it has its own stack memory — and cannot access another thread's stack memory.

The Heap

This part of memory stores the actual object in memory. Those are referenced by the variables from the stack. For example, let's analyze what happens in the following line of code:

```
1  StringBuilder builder = new StringBuilder();
```

The `new` keyword is responsible for ensuring that there is enough free space on heap, creating an object of the `StringBuilder` type in memory and referring to it via the “builder” reference, which goes on the stack.

There exists only one heap memory for each running JVM process. Therefore, this is a shared part of memory regardless of how many threads are running. Actually, the heap structure is a bit different than it is shown in the picture above. The heap itself is divided into a few parts, which facilitates the process of garbage collection.

The maximum stack and the heap sizes are not predefined — this depends on the running machine. However, later in this article, we will look into some JVM configurations that will allow us to specify their size explicitly for a running application.

Reference Types

If you look closely at the *Memory Structure* picture, you will probably notice that the arrows representing the references to the objects from the heap are actually of different types. That is because, in the Java programming language, we have different types of references: **strong**, **weak**, **soft**, and **phantom** references. The difference between the types of references is that the objects on the heap they refer to are eligible for garbage collecting under the different criteria. Let's have a closer look at each of them.

1. Strong Reference

These are the most popular reference types that we all are used to. In the example above with the `StringBuilder`, we actually hold a strong reference to an object from the heap. The object on the heap it is not garbage collected while there is a strong reference pointing to it, or if it is strongly reachable through a chain of strong references.

2. Weak Reference

In simple terms, a weak reference to an object from the heap is most likely to not survive after the next garbage collection process. A weak reference is created as follows:

```
1 WeakReference<StringBuilder> reference = new WeakReference<>(new StringBuilder());
```

A nice use case for weak references are caching scenarios. Imagine that you retrieve some data, and you want it to be stored in memory as well — the same data could be requested again. On the other hand, you are not sure when, or if, this data will be requested again. So you can keep a weak reference to it, and in case the garbage collector runs, it could be that it destroys your object on the heap. Therefore, after a while, if you want to retrieve the object you refer to, you might suddenly get back a `null` value. A nice implementation for caching scenarios is the collection **WeakHashMap**<K,V>. If we open the `WeakHashMap` class in the Java API, we see that its entries actually extend the `WeakReference` class and uses its **ref** field as the map's key:

```
1 /**
2  * The entries in this hash table extend WeakReference, using its main ref
3  * field as the key.
4  */
5
6 private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V> {
7
8     V value;
```

Once a key from the `WeakHashMap` is garbage collected, the entire entry is removed from the map.

3. Soft Reference

These types of references are used for more memory-sensitive scenarios, since those are going to be garbage collected only when your application is running low on memory. Therefore, as long as there is no critical need to free up some space, the garbage collector will not touch softly reachable objects. Java guarantees that all soft referenced objects are cleaned up before it throws an `OutOfMemoryError`. The Javadocs state, “*all soft references to softly-reachable objects are guaranteed to have been cleared before the virtual machine throws an `OutOfMemoryError`.*”

Similar to weak references, a soft reference is created as follows:

```
1 SoftReference<StringBuilder> reference = new SoftReference<>(new StringBuilder());
```

4. Phantom Reference

4. Phantom Reference

Used to schedule post-mortem cleanup actions, since we know for sure that objects are no longer alive. Used only with a reference queue, since the `.get()` method of such references will always return `null`. These types of references are considered preferable to **finalizers**.

How *Strings* Are Referenced

The `String` type in Java is a bit differently treated. Strings are immutable, meaning that each time you do something with a string, another object is actually created on the heap. For strings, Java manages a string pool in memory. This means that Java stores and reuse strings whenever possible. This is mostly true for string literals. For example:

```
1  String localPrefix = "297"; //1
2  String prefix = "297";      //2
3
4  if (prefix == localPrefix)
5  {
6      System.out.println("Strings are equal" );
7  }
8  else
9  {
10     System.out.println("Strings are different");
11 }
```

When running, this prints out the following:

Strings are equal

Therefore, it turns out that after comparing the two references of the `String` type, those actually point to the same objects on the heap. However, this is not valid for Strings that are computed. Let's assume that we have the following change in line `//1` of the above code

```
1  String localPrefix = new Integer(297).toString(); //1
```

Output:

Strings are different

In this case, we actually see that we have two different objects on the heap. If we consider that the computed `String` will be used quite often, we can force the JVM to add it to the string pool by adding the `.intern()` method at the end of computed string:

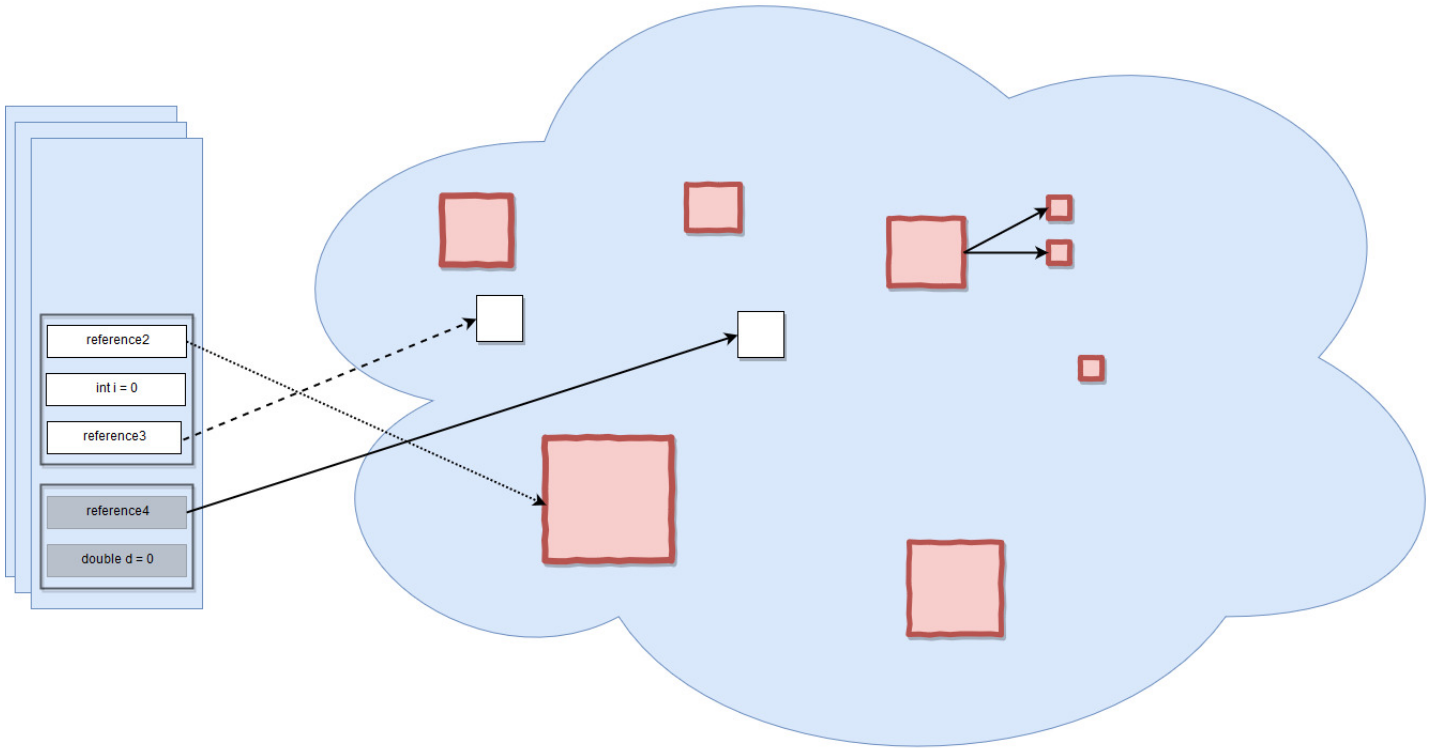
```
1  String localPrefix = new Integer(297).toString().intern(); //1
```

Adding the above change creates the following output:

Strinas are equal

Garbage Collection Process

As discussed earlier, depending on the type of reference that a variable from the stack holds to an object from the heap, at a certain point in time, that object becomes eligible for the garbage collector.



Garbage-eligible objects

For example, all objects that are in red are eligible to be collected by the garbage collector. You might notice that there is an object on the heap, which has strong references to other objects that are also on the heap (e.g. could be a list that has references to its items, or an object that has two referenced type fields). However, since the reference from the stack is lost, it cannot be accessed anymore, so it is garbage as well.

To go a bit deeper into the details, let's mention a few things first:

- This process is triggered automatically by Java, and it is up to Java when and whether or not to start this process.
- It is actually an expensive process. When the garbage collector runs, all threads in your application are paused (depending on the GC type, which will be discussed later).
- This is actually a more complicated process than just garbage collecting and freeing up memory.

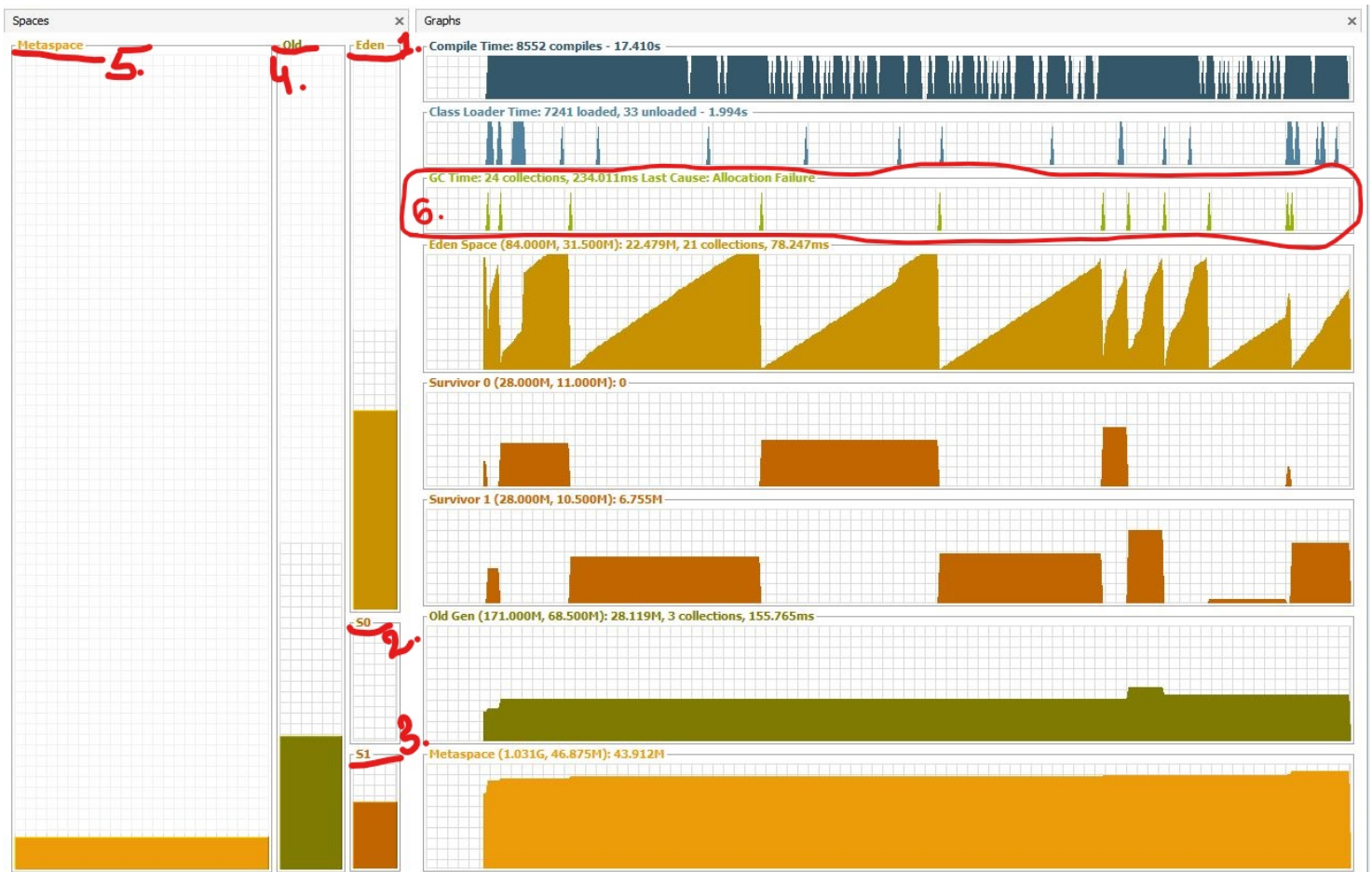
Even though Java decides when to run the garbage collector, you may explicitly call `System.gc()` and expect that the garbage collector will run when executing this line of code, right?

this is a wrong assumption.

You only kind of ask Java to run the garbage collector, but it's, again, up to it whether or not to do that. Anyway, explicitly calling `System.gc()` is not advised.

Since this is a quite complex process, and it might affect you performance, it is implemented in a smart way. A so-called "Mark and Sweep" process is used for that. Java analyzes the variables from the stack and "marks" all the objects that need to be kept alive. Then, all the unused objects are cleaned up.

So actually, Java does not collect any garbage. In fact, the more garbage there is, and the fewer that objects are marked alive, the faster the process is. To make this even more optimized, heap memory actually consists of multiple parts. We can visualize the memory usage and other useful things with **JVisualVM**, a tool that comes with the Java JDK. The only thing you have to do is install a plugin named **Visual GC**, which allows you to see how the memory is actually structured. Let's zoom in a bit and break down the big picture:



Heap memory generations

When an object is created, it is allocated on the **Eden(1)** space. Because the Eden space is not that big, it gets full quite fast. The garbage collector runs on the Eden space and marks objects as alive.

Once an object survives a garbage collecting process, it gets moved into a so-called survivor space **S0(2)**. The second time the garbage collector runs on the Eden space, it moves all surviving objects into the **S1(3)** space. Also, everything that is currently on **S0(2)** is moved into the **S1(3)** space.

If an object survives for X rounds of garbage collection (X depends on the JVM implementation, in my case it's 8), it is most likely that it will survive forever, and it gets moved into the **Old(4)** space.

Taking everything said so far, if you look at the **garbage collector graph(5)**, each time it has run, you can see that the objects switch to the survivor space and that the Eden space gained space. And so on and so forth. The old generation can be also garbage collected, but since it is a bigger part of the memory compared to Eden space, it does not happen that often. The **Metaspace(5)** is used to store the metadata about your loaded classes in the JVM.

The presented picture is actually a Java 8 application. Prior to Java 8, the structure of the memory was a bit different. The metaspace is called actually the PermGen. space. For example, in Java 6, this space also stored the memory for the string pool. Therefore, if you have too many strings in your Java 6 application, it might crash.

Garbage Collector Types

Actually, the JVM has three types of garbage collectors, and the programmer can choose which one should be used. By default, Java chooses the garbage collector type to be used based on the underlying hardware.

1. Serial GC – A single thread collector. Mostly applies to small applications with small data usage. Can be enabled by specifying the command line option: `-XX:+UseSerialGC`

2. Parallel GC – Even from the naming, the difference between Serial and Parallel would be that Parallel GC uses multiple threads to perform the garbage collecting process. This GC type is also known as the throughput collector. It can be enabled by explicitly specifying the option: `-XX:+UseParallelGC`

3. Mostly concurrent GC – If you remember, earlier in this article, it was mentioned that the garbage collecting process is actually pretty expensive, and when it runs, all thread are paused. However, we have this mostly concurrent GC type, which states that it works concurrent to the application. However, there is a reason why it is “mostly” concurrent. It does not work 100% concurrently to the application. There is a period of time for which the threads are paused. Still, the pause is kept as short as possible to achieve the best GC performance. Actually, there are 2 types of mostly concurrent GCs:

3.1 Garbage First – high throughput with a reasonable application pause time. Enabled with the option: `-XX:+UseG1GC`

3.2 Concurrent Mark Sweep – The application pause time is kept to a minimum. It can be used by specifying the option: `-XX:+UseConcMarkSweepGC`. As of JDK 9, this GC type is deprecated.

Tips and Tricks

- To minimize the memory footprint, limit the scope of the variables as much as possible. Remember that each time the top scope from the stack is popped up, the references from that scope are lost, and this could make objects eligible for garbage collecting.
- Explicitly refer to `null` obsolete references. That will make objects those refer to eligible for garbage collecting.
- Avoid finalizers. They slow down the process and they do not guarantee anything. Prefer phantom references for cleanup work.
- Do not use strong references where weak or soft references apply. The most common memory pitfalls are caching scenarios, when data is held in memory even if it might not be needed.

- JVisualVM also has the functionality to make a heap dump at a certain point, so you could analyze, per class, how much memory it occupies.
- Configure your JVM based on your application requirements. Explicitly specify the heap size for the JVM when running the application. The memory allocation process is also expensive, so allocate a reasonable initial and maximum amount of memory for the heap. If you know it will not make sense to start with a small initial heap size from the beginning, the JVM will extend this memory space. Specifying the memory options with the following options:
 - Initial heap size `-Xms512m` – set the initial heap size to 512 megabytes.
 - Maximum heap size `-Xmx1024m` – set the maximum heap size to 1024 megabytes.
 - Thread stack size `-Xss128m` – set the thread stack size to 128 megabytes.
 - Young generation size `-Xmn256m` – set the young generation size to 256 megabytes.
- If a Java application crashes with an `OutOfMemoryError` and you need some extra info to detect the leak, run the process with the `-XX:HeapDumpOnOutOfMemory` parameter, which will create a heap dump file when this error happens next time.
- Use the `-verbose:gc` option to get the garbage collection output. Each time a garbage collection takes place, an output will be generated.

Conclusion

Knowing how memory is organized gives you the advantage of writing good and optimized code in terms of memory resources. In advantage, you can tune up your running JVM, by providing different configurations that are the most suitable for your running application. Spotting and fixing memory leaks is just an easy thing to do, if using the right tools.

Build vs Buy a Data Quality Solution: Which is Best for You? Maintaining high quality data is essential for operational efficiency, meaningful analytics and good long-term customer relationships. But, when dealing with multiple sources of data, data quality becomes complex, so you need to know when you should build a custom data quality tools effort over canned solutions. Download our whitepaper for more insights into a hybrid approach.

Like This Article? Read More From DZone



Java Garbage Collector and Reference Objects



Java Version Upgrades: GC Overview



**Potential Java Garbage Collection
Interview Questions**



**Free DZone Refcard
Getting Started With Kotlin**

Topics: JAVA , JAVA MEMORY MANAGEMENT , JAVA MEMORY LEAK , GARBAGE COLLECTION IN JAVA , REFERENCE TYPE , TUTORIAL

Opinions expressed by DZone contributors are their own.

Get the best of Java in your inbox.

Stay updated with DZone's bi-weekly Java Newsletter. [SEE AN EXAMPLE](#)

[SUBSCRIBE](#)

Java Partner Resources

jQuery UI and Auto-Complete Address Entry

Melissa Data



Learn more about Kotlin

JetBrains



Microservices for Java Developers: A Hands-On Introduction to Frameworks & Containers

Red Hat Developer Program



Akka A to Z: The Java Architect's Guide To Reactive Microservices Architecture

Lightbend



Exploring Java 9's Module System and Reactive Streams

by **Eugen Paraschiv** MVB · May 17, 18 · [Java Zone](#) · [Tutorial](#)

Get the Edge with a Professional Java IDE. 30-day free trial.

Java 9, one of the more noticeable releases of the Java platform in recent years – coming out with a number of prominent features:

- The module system
- Reactive Streams
- JShell
- The Stack Walking API

This article focuses on the Module System and Reactive Streams; you can find an in-depth description of JShell [here](#), and of the Stack Walking API [here](#).

Naturally, Java 9 also introduced some other APIs, as well as improvements related to internal implementations of the JDK; you can follow this link for the entire list of Java 9 characteristics.

The Module System

The Java Platform Module System (JPMS) – the result of Project Jigsaw – is the defining feature of Java 9. Simply put, **it organizes packages and types in a way that is much easier to manage and maintain.**

In this section, we'll first go over the driving forces behind JPMS, then walk you through the declaration of a module. Finally, you'll have a simple application illustrating the module system.

Driving Forces

Up to Java 8, applications face two issues related to the type system:

- **All the artifacts, most being JARs, are in the classpath without any explicit dependency declarations.** A build tool, such as Maven, can help out with organizing them during development. However, there's no such a supporting facility at runtime. You may end up with a classpath where a type is missing, or more seriously, two versions of the same type are present; and errors like that are difficult to diagnose.
- **There's no support for encapsulation at the API level.** All public types are accessible throughout the whole application, despite the fact that most of them are intended to be used by just a few other types. On the other hand, private types and type members aren't private as you can always use the Reflection API to bypass the access restriction.

This is where the Java module system comes to the rescue. Mark Reinhold, Oracle's Java platform chief architect, described the goals of the module system:

- *Reliable configuration – to replace the brittle, error-prone class-path mechanism with a means for program components to declare explicit dependences upon one another.*
- *Strong encapsulation – to allow a component to declare which of its public types are accessible to other components, and which are not.*

Java 9 allows you to define modules with module descriptors.

Module Descriptors

Module descriptors are the key to the module system. **A descriptor is the compiled version of a module declaration – specified in a file named *module-info.java* at the root of the module's directory hierarchy.**

A module declaration starts with the *module* keyword, followed by the name of the module. The declaration ends with a pair of curly braces wrapping around zero or more module directives.

You can declare a simple module with an empty body like this:

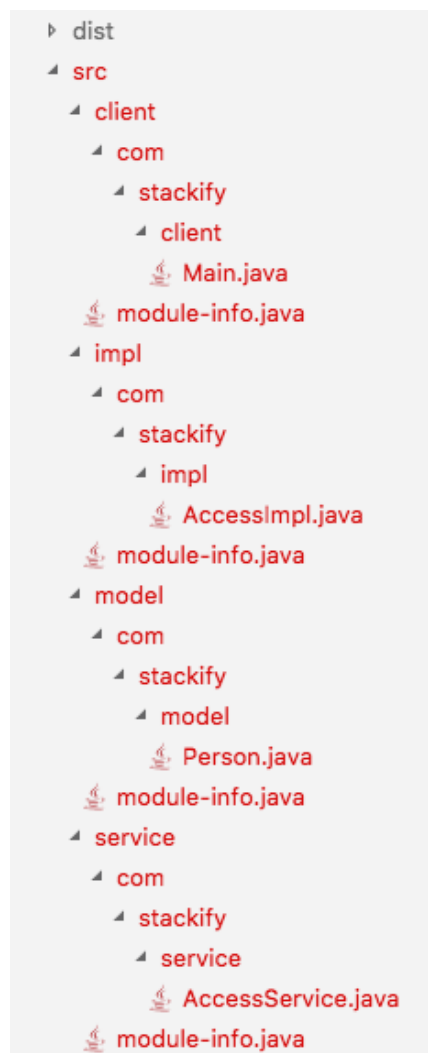
```
1 module com.stackify { }
```

Here are the directives you can lay out in a module declaration:

- *requires* – indicates the module it depends on, also called dependency
- *transitive* – only used with the *requires* directive, indicating that the specified dependency is also accessible to those requiring this module
- *exports* – declares a package accessible to other modules
- *opens* – exposes a package at runtime for introspection with the Reflection API
- *uses* – specifies the fully qualified name of a service this module consumes
- *provides ... with* – denotes an implementation, specified by the *with* keyword, for a service, indicated by *provides*

The following sample application illustrates all these directives.

A Sample Modular Application



The sample application you'll be going through consists of four modules – namely *model*, *service*, *impl*, and *client*. In a real-world project, you should name modules with the reverse domain name pattern to avert name conflicts. This article uses simple names to avoid lengthy commands, making them easier to grasp.

The source code of each module is included in its directory, which is located inside the *src* root directory. The *dist* directory for the compiled files at the top of the attached image will be generated during compilation. That means there's no need to create it manually.

Let's start with the *model* module. This module has only one package with a single class:

```
1  package com.stackify.model;
2
3  public class Person {
4      private int id;
5      private String name;
6
7      public Person(int id, String name) {
8          this.id = id;
9          this.name = name;
10     }
11 }
```

And here's the declaration of the module:

```
1  module model {
2      exports com.stackify.model;
3      opens com.stackify.model;
4  }
```

This module exports the *com.stackify.model* package and opens it for introspection.

Next comes the *service* module – defining a service interface:

```
1  package com.stackify.service;
2
3  import com.stackify.model.Person;
4
5  public interface AccessService {
6      public String getName(Person person);
7  }
```

Since the *service* module makes use of the *com.stackify.model* package, it must require access to the *model* module:

```
1  module service {
2      requires transitive model;
3      exports com.stackify.service;
4  }
```

Notice the *transitive* keyword in the declaration. This indicates that all modules requiring the *service* module will automatically gain access to the *model* module. To make *AccessService* available for other modules, you must export its package via the *exports* directive.

The *impl* module provides the access service with an implementation:

```
1  package com.stackify.impl;
2
3  import com.stackify.service.AccessService;
4  import com.stackify.model.Person;
5  import java.lang.reflect.Field;
6
7  public class AccessImpl implements AccessService {
8      public String getName(Person person) {
9          try {
10             return extract(person);
11          } catch (Exception e) {
12             throw new RuntimeException(e);
13          }
14      }
15
16      private String extract(Person person) throws Exception {
17          Field field = person.getClass().getDeclaredField("name");
18          field.setAccessible(true);
19          return (String) field.get(person);
20      }
21 }
```

Thanks to the *opens* directive in the declaration of the *model* module, *AccessImpl* can introspect the *Person* class.

This is the declaration of the *impl* module:

```
1  module impl {
2      requires service;
3      provides com.stackify.service.AccessService with com.stackify.impl.AccessImpl;
4  }
```

The import of *model* into the *service* module is transitive, thus *impl* just needs to require *service* to have access to both of those modules.

The *provides ... with* directive signals that the *impl* module provides an implementation for the *AccessService* interface. That implementation is the *AccessImpl* class.

And here's how the *client* module consumes *AccessService* – with this module declaration:

```
1  module client {
2      requires service;
3      uses com.stackify.service.AccessService;
4  }
```

Here's how a client uses the access service:

```

1  package com.stackify.client;
2
3  import com.stackify.service.AccessService;
4  import com.stackify.model.Person;
5  import java.util.ServiceLoader;
6
7  public class Main {
8
9      public static void main(String[] args) throws Exception {
10         AccessService service = ServiceLoader.load(AccessService.class).findFirst().get();
11         Person person = new Person(1, "John Doe");
12         String name = service.getName(person);
13         assert name.equals("John Doe");
14     }
15 }

```

You can see that the *Main* class doesn't use the *AccessImpl* implementation class directly. Instead, the module system locates that implementation automatically at runtime based upon the *uses* and *provides ... with* directives in the module descriptor.

Compilation and Execution

This subsection presents the steps to compile and execute the modular application you have just seen. Note that you must run all the commands in the root directory of the project – the parent directory of *src* – in the order, they show up.

This command compiles the *model* module and places the generated class files into the *dist* directory:

```

1  javac -d dist/model src/model/module-info.java src/model/com/stackify/model/Person.java

```

Since the *service* module requires the *model* module, you must specify the location of that dependency module in the *-p* option – which denotes the module path – when compiling the *service* module:

```

1  javac -d dist/service -p dist src/service/module-info.java src/service/com/stackify/service

```

Likewise, the following are how to compile the *impl* and *client* modules:

```

1  javac -d dist/impl -p dist src/impl/module-info.java src/impl/com/stackify/impl/AccessImpl.
2  javac -d dist/client -p dist src/client/module-info.java src/client/com/stackify/client/Mai

```

The *Main* class uses an *assert* statement, hence you need to enable assertions when executing the program:

```
1 java -ea -p dist -m client/com.stackit.client.main
```

Notice that you must precede the main class with the name of its module – before passing to the *-m* option.

When the application runs, you won't see any error, implying that it works as expected.

Backward Compatibility

Before Java 9, all packages were declared without any idea about modules. However, that doesn't keep you from deploying those packages on the new module system. All you need to do is simply add them to the classpath as you would on Java 8; the packages in the classpath will then become part of the *unnamed* module.

The *unnamed* module reads all the other modules – whether they are in the classpath or module path. As a result, an application that compiles and runs on Java 8 will work the same way on Java 9. Nevertheless, a module with an explicit declaration doesn't have access to the *unnamed* module. This is where you need another kind of modules – automatic modules.

You can convert a legacy JAR file – which doesn't have a module descriptor – to an automatic module by placing it in the module path.

This will define a module whose name is derived from the name of the JAR file. Such an automatic module has access to all the other modules in the module path and exposes all of its packages, enabling seamless interoperations between packages with and without an explicit module.

Reactive Streams

Reactive Streams is a programming paradigm – allowing for processing asynchronous data streams in a non-blocking manner with back-pressure. Essentially, this mechanism puts the receiver in control, enabling it to determine the amount of data to be transferred, while not having to wait for the response after each request.

The Java platform has integrated Reactive Streams as part of Java 9. This integration permits you to leverage Reactive Streams in a standard way, whereby various implementations can be working together.

The *Flow* Class

The Java API wraps all the interfaces for Reactive Streams – including *Publisher*, *Subscriber*, *Subscription*, and *Processor* – inside the *Flow* class.

A *Publisher* produces items and related control messages. This interface defines a single method – namely *subscribe* – which adds a subscriber that will be listening for data the publisher transmits.

A *Subscriber* receives messages from a publisher; this interface defines four methods:

- *onSubscribe* – invoked when the subscriber successfully subscribes to a publisher; this method has a *Subscription* parameter that allows the subscriber to control its communication with the publisher
- *onNext* – invoked when the next item comes in
- *onError* – invoked upon an unrecoverable error the publisher or subscription encountered
- *onComplete* – invoked when the conversation with the publisher is complete

A *Subscription* controls the communication between a publisher and a subscriber. This interface declares two methods: *request* and *cancel*. The *request* method asks the publisher to send a specific number of items, whereas the *cancel* method causes the subscriber to unsubscribe.

Sometimes you may want to perform operations on the data items while they are transferring from the publisher to the subscriber. This is where a *Processor* comes into play. This interface extends both *Publisher* and *Subscriber*, enabling it to act as a subscriber from the publisher's perspective, and as a publisher from the subscriber's.

Built-In Implementations

The Java platform provides an implementation for the *Publisher* and *Subscription* interface out of the box. The implementation for *Publisher* is a class named *SubmissionPublisher*.

In addition to the method defined in the *Publisher* interface, this class has some others, including:

- *submit* – publishes an item to each subscriber
- *close* – sends an *onComplete* signal to each subscriber and forbids subsequent subscriptions

The *Subscription* implementation of the platform is a private class – which is intended to be used by *SubmissionPublisher* only. When you call the *subscribe* method on a *SubmissionPublisher* with a *Subscriber* argument, a *Subscription* object is created and passed to the *onSubscribe* method of that subscriber.

A Simple Application

With *Publisher* and *Subscription* implementations in hand, you only need to declare a class implementing the *Subscriber* interface to have a Reactive Streams application. This class expects messages of the *String* type:

```
1 public class StringSubscriber implements Subscriber<String> {
2     private Subscription subscription;
3     private StringBuilder buffer;
4
5     @Override
6     public void onSubscribe(Subscription subscription) {
7         this.subscription = subscription;
8         this.buffer = new StringBuilder();
9         subscription.request(1);
10    }
11
12    public String getData() {
13        return buffer.toString();
14    }
15
16    // other methods
17 }
```

As you can see, a *StringSubscriber* object stores the *Subscription* it takes when subscribing to a publisher in a

private field. Also, it uses the *buffer* field to keep the received *String* messages. You can retrieve the value of this buffer using the *getData* method.

Apart from initializing private fields, the *onSubscribe* method requests the publisher to issue a single data item.

Here's the definition of the *onNext* method:

```
1  @Override
2  public void onNext(String item) {
3      buffer.append(item);
4      if (buffer.length() < 5) {
5          subscription.request(1);
6          return;
7      }
8      subscription.cancel();
9  }
```

This method demands the publisher to send a new string upon the arrival of the previous one. After getting five items, the subscriber stops receiving.

The two other methods – *onError* and *onComplete* – are trivial:

```
1  @Override
2  public void onError(Throwable throwable) {
3      throwable.printStackTrace();
4  }
5
6  @Override
7  public void onComplete() {
8      System.out.println("Data transfer is complete");
9  }
```

This quick test verifies our implementation class:

```
1  @Test
2  public void whenTransferringDataDirectly_thenGettingString() throws Exception {
3      StringSubscriber subscriber = new StringSubscriber();
4      SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
5      publisher.subscribe(subscriber);
6
7      String[] data = { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" };
8      Arrays.stream(data).forEach(publisher::submit);
9
10     Thread.sleep(100);
11     publisher.close();
12
13     assertEquals("01234", subscriber.getData());
}
```

```
14 }
```

In the above test, the *sleep* method does nothing but to wait for the asynchronous data transmission to complete.

An Application With *Processor*

Let's make our sample application a bit more complicated with a processor. This processor transforms published strings into integers, throwing an exception if the conversion fails. After the transformation, the processor forwards the resulting numbers to the subscriber.

The following is the new *Subscriber* implementation class:

```
1  public class NumberSubscriber implements Subscriber<Integer> {
2      private Subscription subscription;
3      private int sum;
4      private int remaining;
5
6      @Override
7      public void onSubscribe(Subscription subscription) {
8          this.subscription = subscription;
9          subscription.request(1);
10         remaining = 1;
11     }
12
13     public int getData() {
14         return sum;
15     }
16
17     // other methods
18 }
```

The fields and methods you have just seen are pretty much the same as those of the custom *StringSubscriber* class – except for the existence of the *remaining* field. You can use this field to control the requests to the publisher:

```
1  @Override
2  public void onNext(Integer item) {
3      sum += item;
4      if (--remaining == 0) {
5          subscription.request(3);
6          remaining = 3;
7      }
8  }
```

Rather than asking for items one by one, the subscriber now solicits three at once. It only sends a new demand after collecting all the unfulfilled items

after collecting all the unhandled items.

The *onError* and *onComplete* methods are the same as before, hence we'll leave it off for simplicity.

The last piece of the application is a *Processor* implementation:

```
1 public class StringToNumberProcessor extends SubmissionPublisher<Integer> implements Subscr
2     private Subscription subscription;
3
4     @Override
5     public void onSubscribe(Subscription subscription) {
6         this.subscription = subscription;
7         subscription.request(1);
8     }
9
10    // other methods
11 }
```

The processor class, in this case, extends *SubmissionPublisher*, thus only needs to implements abstract methods of the *Subscriber* interface. Here are the three other methods:

```
1  @Override
2  public void onNext(String item) {
3      try {
4          submit(Integer.parseInt(item));
5      } catch (NumberFormatException e) {
6          closeExceptionally(e);
7          subscription.cancel();
8          return;
9      }
10     subscription.request(1);
11 }
12
13 @Override
14 public void onError(Throwable throwable) {
15     closeExceptionally(throwable);
16 }
17
18 @Override
19 public void onComplete() {
20     System.out.println("Data conversion is complete");
21     close();
22 }
```

Note that when the publisher is closed, the processor should also be closed, issuing an *onComplete* signal to the subscriber. Likewise, when an error occurs – whether on the processor or the publisher – the processor itself ought to inform the subscriber of that error.

You can implement that notification flow by calling the *close* and *closeExceptionally* methods.

Here's a test confirming the accuracy of our application:

```
1  @Test
2  public void whenProcessingDataMidway_thenGettingNumber() throws Exception {
3      NumberSubscriber subscriber = new NumberSubscriber();
4      StringToNumberProcessor processor = new StringToNumberProcessor();
5      SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
6
7      processor.subscribe(subscriber);
8      publisher.subscribe(processor);
9
10     String[] data = { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" };
11     Arrays.stream(data).forEach(publisher::submit);
12
13     Thread.sleep(100);
14     publisher.close();
15
16     assertEquals(45, subscriber.getData());
17 }
```

API Usage

The applications you have gone through should provide a better understanding of Reactive Streams. They by no means serve as a guideline for you to build a reactive program from scratch, however.

Implementing the Reactive Streams specification isn't easy since the problem it resolves isn't simple at all. **You should leverage a library available out there** – such as RxJava or Project Reactor – to write an effective application.

In the future, when many Reactive libraries support Java 9, you can even combine various implementations from different tools to get the most out of the API.

Summary

This article went over the new Module System as well as Reactive Streams – two of the core technologies in Java 9.

The module system is new and not expected to be widely adopted soon. However, the movement of the whole Java world into modular systems is inevitable, and you should certainly prepare yourself for that.

On the contrary, Reactive Streams has been around for a while. The introduction of Java 9 helped standardize the paradigm, which may speed up its adoption.

Get the Java IDE that understands code & makes developing enjoyable. Level up your code with IntelliJ IDEA.

Download the free trial.

Like This Article? Read More From DZone



Real World Java 9 [Webinar]



Java 9 Modules (Part 1): Introduction




What Are Reactive Streams in Java?



**Free DZone Refcard
Getting Started With Kotlin**

Topics: JAVA, JPMS, JAVA 9 MODULES, REACTIVE STREAMS, TUTORIAL

Published at DZone with permission of Eugen Paraschiv , DZone MVB. [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.
