

Java Memory Management

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

Rapidly [provision TLS certificates from any certificate authority](#) within your DevOps CI/CD pip

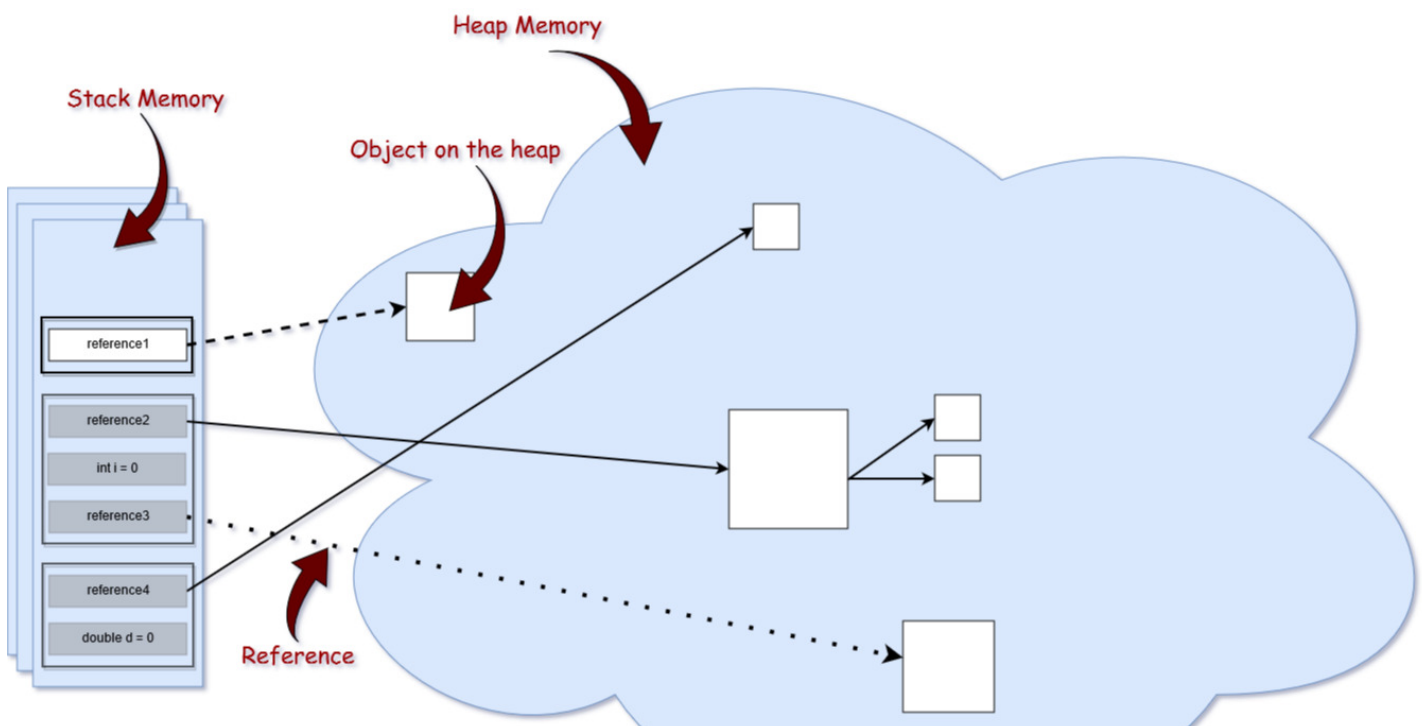
Presented by Venafi

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 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 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 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

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

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:

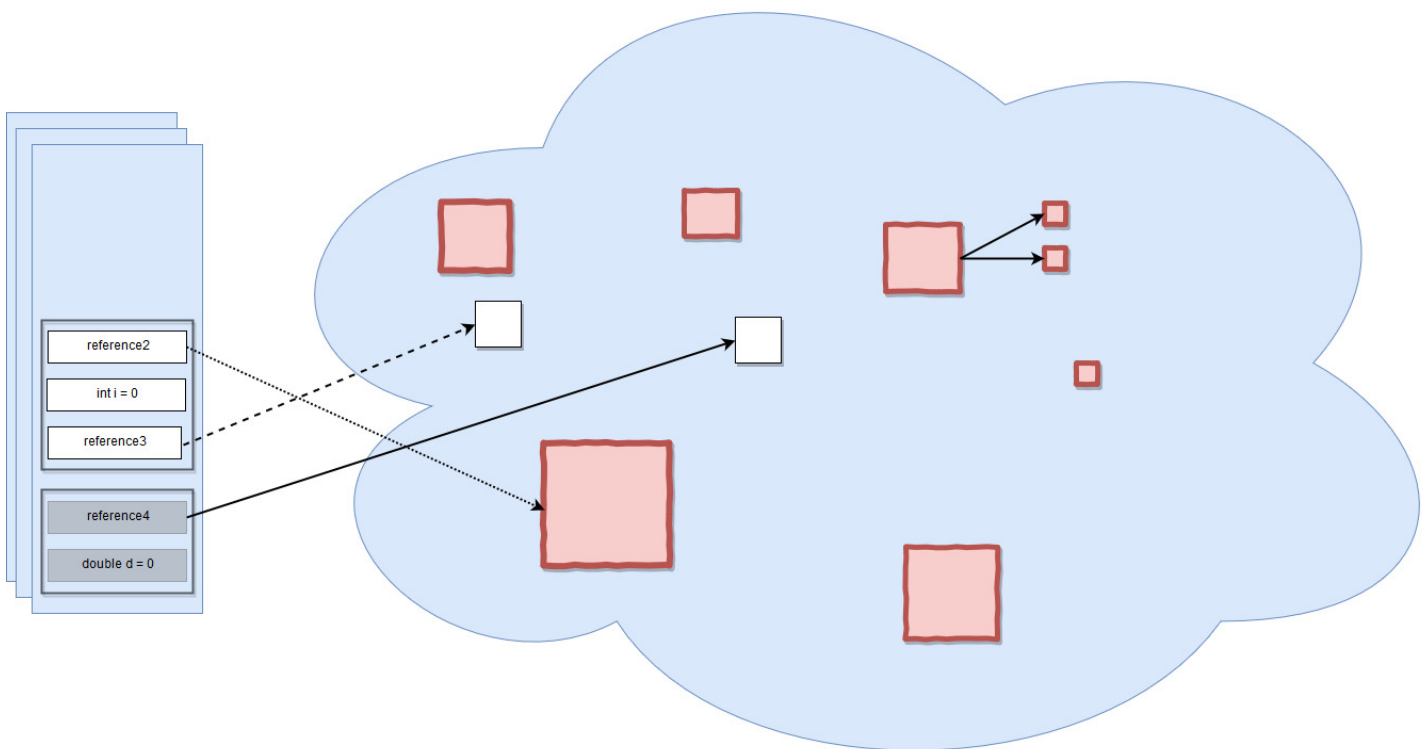
```
1 String localPrefix = new Integer(297).toString().intern(); //1
```

Adding the above change creates the following output:

Strings 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)

(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.

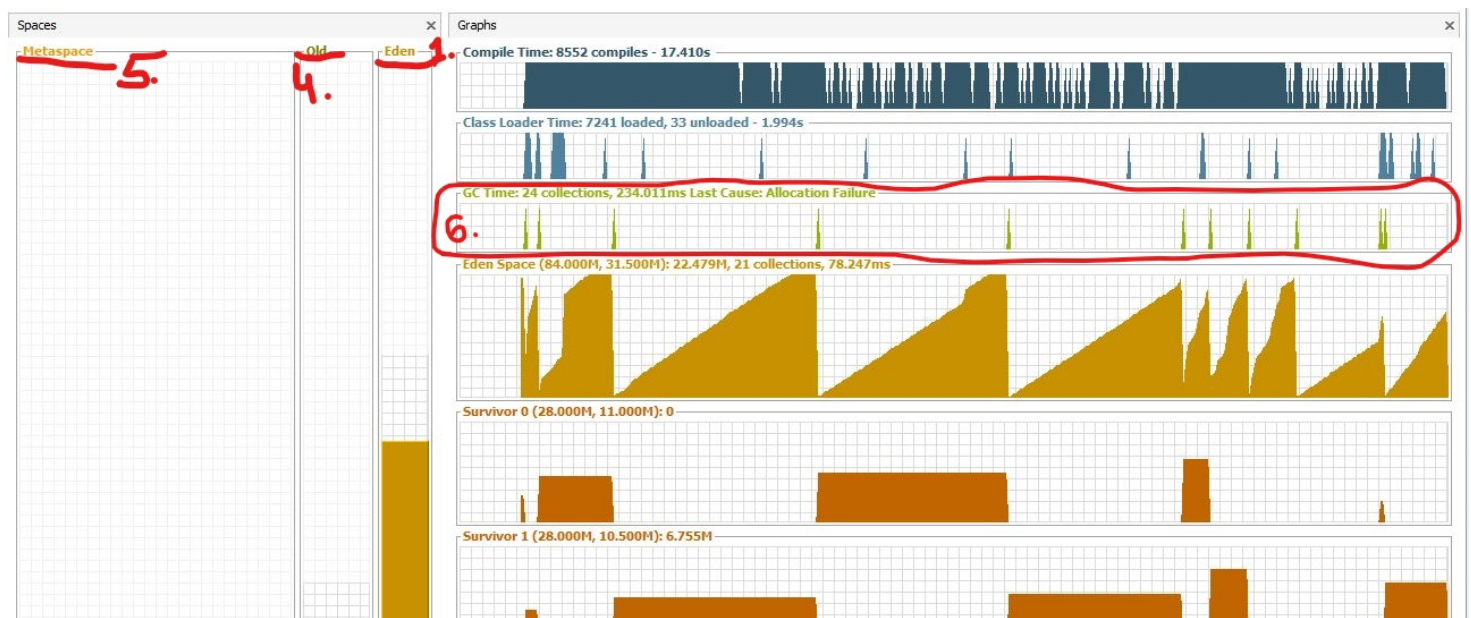
Even though Java decides when to run the garbage collector, you may explicitly call `System.gc()` and expect that the garbage collector will run when executing this line of code, right?

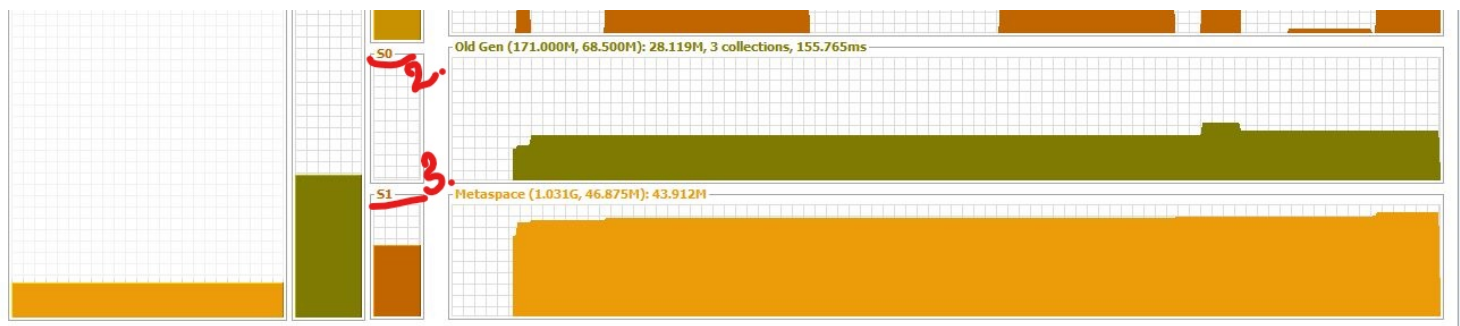
This is a wrong assumption.

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 your 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(6)**, 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.

A Machine and Deep Learning Primer. [Download Now.](#)

Presented by GridGain

Like This Article? Read More From DZone



Java Garbage Collector and Reference Objects



Java Garbage Collection: Best Practices, Tutorials, and More



Java Version Upgrades: GC Overview



Free DZone Refcard Java Containerization

Topics: [JAVA](#) , [JAVA MEMORY MANAGEMENT](#) , [JAVA MEMORY LEAK](#) , [GARBAGE COLLECTION IN JAVA](#) , [REFERENCE TYPE](#) , [TUTORIAL](#)

Opinions expressed by DZone contributors are their own.

Focus on Your Data Structures With Scala Lenses

by [Francisco Alvarez](#) MVB · Jul 16, 19 · [Java Zone](#) · [Tutorial](#)

With new programming techniques come new problems and new patterns to solve them.

In functional programming, immutability is a must. As a consequence, whenever it is needed to modify the

content of a data structure, a new instance with updated values is created. Depending on how complex the data structure is, creating a copy may be a verbose task.

To simplify the process, a set of functions, generically called Optics, have been designed to access/modify parts of a whole in an easy way. Those functions must obey some laws that make their behavior predictable and intuitive (for instance, if we modify a value and then read it back, we should obtain the modified value).

This post presents some examples of how to use an Optics library in Scala called Monocle.

Monocle Examples

To illustrate the use of Monocle, let's start by creating a simple domain model:

```

1 import monocle.macros.Lenses
2 sealed trait RoomTariff
3 case class NonRefundable(fee: BigDecimal) extends RoomTariff
4 case class Flexible(fee: BigDecimal) extends RoomTariff
5
6
7 @Lenses("_") case class Hotel(name: String, address: String, rating: Int, rooms: List[Room], facilities: List[String])
8 @Lenses("_") case class Room(name: String, boardType: Option[String], price: Price, roomTariff: RoomTariff)
9 @Lenses("_") case class Price(amount: BigDecimal, currency: String)
```

The annotation `@Lenses` generates automatically a lens for each attribute of the case class.

Let's create an imaginary hotel:

```

1 val rooms = List(
2     Room("Double", Some("Half Board"), Price(10, "USD"), NonRefundable(1)),
3     Room("Twin", None, Price(20, "USD"), Flexible(0)) ,
4     Room("Executive", None, Price(200, "USD"), Flexible(0))
5 )
6 val facilities = Map("business" -> List("conference room"))
7 val hotel = Hotel("Hotel Paradise", "100 High Street", 5, rooms, facilities)
```

And now, the fun part:

Room changes based on the room position in the List

```

1 test("double price of even rooms") {
2
3     val updatedHotel = (_rooms composeTraversal filterIndex{i: Int => i/2*2 == i} composeLens .
4
```

```

5      assert(updatedHotel.rooms(0).price.amount == hotel.rooms(0).price.amount * 2)
6      assert(updatedHotel.rooms(1).price.amount == hotel.rooms(1).price.amount)
7      assert(updatedHotel.rooms(2).price.amount == hotel.rooms(2).price.amount * 2)
8  }
9
10     test("set price of 2nd room") {
11
12         val newValue = 12
13         val roomToUpdate = 1
14
15         assert(hotel.rooms(roomToUpdate).price.amount != newValue)
16
17         val updatedHotel = (_rooms composeOptional index(roomToUpdate) composeLens _price composeL
18         val updatedRoomList = (index[List[Room], Int, Room](roomToUpdate) composeLens _price compo
19
20         assert(updatedHotel.rooms(roomToUpdate).price.amount == newValue)
21         assert(updatedRoomList(roomToUpdate).price.amount == newValue)
22     }

```

Modifying a non-existing room

```

1  test("no changes are made when attempting to modify a non-existing room") {
2
3      val newValue = 12
4      val roomToUpdate = 3
5
6      assert(hotel.rooms.length == 3)
7
8      val updatedHotel = (_rooms composeOptional index(roomToUpdate) composeLens _price composeL
9
10     assert(hotel == updatedHotel)
11 }
12
13 test("hotel 'disappears' when attempting to modify a non-existing room") {
14
15     val newValue = 12
16     val roomToUpdate = 3
17
18     assert(hotel.rooms.length == 3)
19
20     val updatedHotel = (_rooms composeOptional index(roomToUpdate) composeLens _price composeL

```

```
20
21
22     assert(updatedHotel.isEmpty)
23 }
```

Changing an optional value

```
1  test("set a value inside an Option") {
2
3      val newValue = "New Board Type"
4      val roomToUpdate = 0
5
6      assert(!hotel.rooms(roomToUpdate).boardType.contains(newValue))
7
8      val updatedHotel = (_rooms composeOptional index(roomToUpdate) composeLens _boardType compo
9
10     assert(updatedHotel.rooms(roomToUpdate).boardType.contains(newValue))
11 }
12
13 test("no changes are made when attempting to modify an empty Option") {
14
15     val newValue = "New Board Type"
16     val roomToUpdate = 1
17
18     assert(hotel.rooms(roomToUpdate).boardType.isEmpty)
19
20     val updatedHotel = (_rooms composeOptional index(roomToUpdate) composeLens _boardType compo
21
22     assert(updatedHotel.rooms(roomToUpdate).boardType.isEmpty)
23 }
24
25 test("hotel 'disappears' when attempting to modify an empty Option") {
26
27     val newValue = "New Board Type"
28     val roomToUpdate = 1
29
30     assert(hotel.rooms(roomToUpdate).boardType.isEmpty)
31
32     val updatedHotel = (_rooms composeOptional index(roomToUpdate) composeLens _boardType compo
33
34     assert(updatedHotel.isEmpty)
```

```
35 }
```

Changes with an applicative function

```
1  test("divide prices by 10"){
2
3      assert(hotel.rooms(0).price.amount == 10)
4      assert(hotel.rooms(1).price.amount == 20)
5
6      val updatedHotel = (_rooms composeTraversal each composeLens _price composeLens _amount modify {
7
8          assert(updatedHotel.rooms(0).price.amount == 1)
9          assert(updatedHotel.rooms(1).price.amount == 2)
10     })
11
12     test("divide prices by 0"){
13
14         assert(hotel.rooms(0).price.amount == 10)
15         assert(hotel.rooms(1).price.amount == 20)
16
17         val updatedHotel = (_rooms composeTraversal each composeLens _price composeLens _amount).modify {
18
19             assert(updatedHotel.isEmpty)
20         }
```

Modifying the number of rooms

```
1  test("append a room"){
2
3      assert(hotel.rooms.length == 3)
4
5      val newRoom = Room("Triple", None, Price(1, "USD"), Flexible(0))
6
7      val updatedHotel = (_rooms set _snoc(hotel.rooms, newRoom))(hotel)
8
9      assert(updatedHotel.rooms.length == 4)
10     assert(updatedHotel.rooms(3) == newRoom)
11 }
12
13 test("prepend a room"){
14
15     assert(hotel.rooms.length == 3)
```

```

15  assert(hotel.rooms.length == 3)
16
17  val newRoom = Room("Triple", None, Price(1, "USD"), Flexible(0))
18
19  val updatedHotel = (_rooms set _cons(newRoom, hotel.rooms))(hotel)
20
21  assert(updatedHotel.rooms.length == 4)
22  assert(updatedHotel.rooms(0) == newRoom)
23  }

```

Using prisms to modify the room tariff

```

1  test("set prices of Flexible rooms"){
2
3      val prism = Prism.partial[RoomTariff, BigDecimal]{case Flexible(x) => x}(Flexible)
4
5      val newValue = 100
6
7      assert(hotel.rooms(0).roomTariff == NonRefundable(1))
8      assert(hotel.rooms(1).roomTariff == Flexible(0))
9      assert(hotel.rooms(2).roomTariff == Flexible(0))
10
11     val updatedHotel = (_rooms composeTraversal each composeLens _roomTariff composePrism prism)
12
13     assert(hotel.rooms(0).roomTariff == updatedHotel.rooms(0).roomTariff)
14     assert(updatedHotel.rooms(1).roomTariff == Flexible(newValue))
15     assert(updatedHotel.rooms(2).roomTariff == Flexible(newValue))
16 }

```

Manipulating a Map

```

1  test("modifying business facilities") {
2
3      val updatedHotel = (_facilities composeLens at("business") set Some(List("")))(hotel)
4
5      assert(updatedHotel.facilities("business") == List(""))
6  }
7
8  test("removing business facilities") {
9
10     val updatedHotel = (_facilities composeLens at("business") set None)(hotel)
11     val updatedFacilities = remove("business")(hotel.facilities)

```

```

12
13     assert(updatedHotel.facilities.get("business").isEmpty)
14     assert(updatedFacilities.get("business").isEmpty)
15 }
16
17 test("adding entertainment facilities") {
18
19     val updatedHotel = (_facilities composeLens at("entertainment") set Some(List("satellite tv", "internet")))
20
21     assert(updatedHotel.facilities("entertainment") == List("satellite tv", "internet"))
22 }

```

Folding over the room list

```

1 test("folding over room prices to add them up") {
2
3     assert(hotel.rooms(0).price.amount == 10)
4     assert(hotel.rooms(1).price.amount == 20)
5     assert(hotel.rooms(2).price.amount == 200)
6
7     assert((_rooms composeFold Fold.fromFoldable[List, Room] foldMap(_price.amount))(hotel) == 230)
8 }

```

Modifying rooms that meet specific criteria

```

1 val unsafePrism = UnsafeSelect.unsafeSelect[Room](_.name == "Double")
2 test("double price of Double rooms using unsafe operation") {
3
4     val updatedHotel = (_rooms composeTraversal each composePrism unsafePrism composeLens _price)
5
6     assert(hotel.rooms.filter(_.name == "Double").map(_price.amount*2) == updatedHotel.rooms.filter(_.name == "Double"))
7 }

```

This last example makes use of an unsafe prism (it is unsafe because does not comply with all Prism laws). Let's verify this statement by testing the laws:

```

1 val roomGen: Gen[Room] = for {
2     name <- Gen.oneOf("Double", "Twin", "Executive")
3     board <- Gen.option(Gen.alphaStr)
4     price <- for{

```



```

5      price <- Gen.posNum[Double]
6      currency <- Gen.oneOf("USD", "GBP", "EUR")
7      } yield Price(price, currency)
      tariff <- Gen.oneOf(Gen.posNum[Double].map(NonRefundable(_)), Gen.posNum[Double].map(Flexi
8  } yield Room(name, board, price, tariff)
9
10
11  implicit val roomArb: Arbitrary[Room] = Arbitrary(roomGen)
12
13  implicit val arbAA: Arbitrary[Room => Room] = Arbitrary{
14    for{
15      room <- roomGen
16    } yield (_: Room) => room
17  }
18
19  checkAll("unsafe prism", PrismTests(unsafePrism))

```

When running the above test, the following tests fail:

- Prism.compose modify
- Prism.round trip another way

So, what is wrong? Let's check the law "round trip other way." Here's its definition on PrismLaws :

```

1  def roundTripOtherWay(a: A): IsEq[Option[A]] =
2    prism.getOption(prism.reverseGet(a)) <==> Some(a)

```

And this is how the law is broken:

```

1  val a = Room(Twin,None,Price(1.0,USD),Flexible(1.0))
2  val b = unsafePrism.reverseGet(a) = Room(Twin,None,Price(1.0,USD),Flexible(1.0))
3  val c = unsafePrism.getOption(b) = None
4
5  None != Some(a)

```

So, our `unsafePrism` is unsafe when used to make changes on the attribute included in the predicate to create the prism.

All the above examples can be found on this repo.

Like This Article? Read More From DZone



The Developer's Guide to Collections



cleanframes: A Data Cleansing Library for Apache Spark!




The Developer's Guide to Collections: Queues



**Free DZone Refcard
Java Containerization**

Topics: JAVA, SCALA, LENSES, TUTORIAL, DATA STRUCTURES, MONOCLE

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

IN PROGRESS