Professional Development Program Series, Vol. I, No. 1, 1-5, 2014

Java Apprentice Badge Report

Marko Viitanen[1]*

Abstract
As part of the Company Professional Development Program we are provided wasy to learn and demonstrate our development skills by earning badges. There are several badges, Java being one of them. This document describes my learnings and answers to the requirements for earning the Apprentice Java Badge.

Keywords
Java — Professional Development Program — Badges

[1] *FamilySearch, Salt Lake City*

Contents

Introduction 1
1 Core Java 1
1.1 Object Life cycle 1
1.2 Basic Data Types 3
1.3 List of Strings 4
1.4 StringBuffer vs StringBuilder 4
1.5 StringBuffer vs StringBuilder 4
1.6 Sorting in Order 4
1.7 Sorting in Reverse Order 4
1.8 Exceptions 4
1.9 Enums 4
2 Methods, Encapsulation and Inheritance 4
2.1 Composition, Inheritance, and Static Method Calls 5
2.2 Constructors 5
2.3 Encapsulation 5
2.4 Object References and Primitives 5
2.5 Access Modifiers 5
2.6 Virtual Method Invocation 5
2.7 Casting 5
2.8 Overridden Methods 5
2.9 Overloaded Constructors and Methods 5
3 Library 5
3.1 Logging Directly 5
3.2 Logging Configuration 5
Appendix 6
Appendix A 6
References 6

Introduction

The Java Apprentice Badge is the first badge in the Java series. It covers intermediate concepts of Java programming. It is not a 101 course in programming, so many of the common language features are not covered. The requirements don't include installation,

High-level requirements[1] include:

- Object life cycle
- Exceptions
- Polymorphism
- Collections
- Using a library

Each apprentice is allowed to creatively demonstrate their knowledge on the required topics. I chose to write a report about it.

1. Core Java

The first section deals with Java primitives, objects and their life cycle, and some JDK-provided classes for String manipulation and Collections. It includes writing applications to sort Strings. It also deals with Exceptions and Enums.

1.1 Object Life cycle

Describe the life cycle of an object instance in Java and how garbage collection works

Procedural programming preceded object-oriented programming. Basically, The code was written line by line, like a recipe. The computer would execute the lines in the order they appeared in the program. There were constructs to jump from one place to another, called gotos. But very quickly a program can become quite complicated, and specially the gotos would make it very hard to follow.

With many lines of code, the program becames difficult to maintain. We can split the program into several files and

[1] See confluence for full description of the requirements at https://almtools.ldschurch.org/fhconfluence/display/Product/Core+Skills+-+Java+-+Apprentice .

import the pieces when compiling. That helps humans to organize the data but also fulfills the need for the compiler to have all the pieces of the program available. It doesn't solve the problem of scope, though. In procedural programming the data is accessible to the entire program. There is no clear ownership of data.

Procedural programming also provided constructs called subroutines. They are blocks of code, collections of instructions. Subroutines have their own scope, but any data they need to access outside the subroutine is still exposed to the entire program.

Java is an object-oriented language. In object-oriented languages, instead of having data and subroutines, we deal with objects that have data and behavior. WIth object-oriented programming we can easily model the real world. For example we can have an object of a dog that has data (color, breed) and behavior (barks, runs, drools).

When developers write java programs, they write classes. A class is like a blueprint of an object, it defines the object. A class becomes an object when we instanciate it, we create an instance of it. Obects live when the program is executing, at runtime.

With objects it is easy to encapsulate behavior and data. We can restrict access to the data to only the members inside of a class. Nobody outside the class can access the data, if we don't want to (and we shouldn't want to.) We can define interfaces that provide indirect, controlled access to the data inside the class.

Organizing code becomes easier too, because each file can only have one public class per file. Since each class encapsulate one "thing", that has a well-defined interface that determines its behaviors, the code becomes very logical.

Everything in Java is made of another object. We call this inheritance. Java provides the mother of all objects, called `Object`, from which any other class must inherit.

Instantiating a class. We create an object from a class with the Java keyword `new`. Before instantiation, the object doesn't exist. After instantiation it exists. Classes have a special method called a constructor. After creating the class the Java Virtual machine (JVM) calls the object's constructor. If you don't provide a constructor for your class, the its parent (eventually the `Object` constructor is called. The constructor is a place where you coudl initialize the object or start resources.[5]

Strongly Referenced. When the constructor has been called, your program has a strong reference to it.[6] It means you can access the non-private methods and data on it. It is usable by your program.

```java
Dog pepper = new Dog();
```

In the above example, `pepper` is the handle to your object, or a reference. It is a strong reference (as opposed to a weak reference) because you can use it to do things with the `pepper` object:

```java
pepper.bark();
```

You can have several references to the same object:

```java
// create an instance of Dog
Dog pepper = new Dog();

// pepperClone also points to the same object
pepperClone = pepper;

// set pepper's name to "pepper"
pepper.setName("pepper");

// returns "pepper"
pepperClone.getName();
```

In the above example we created an instance of a `Dog` and got back a reference to it called `pepper`. Then we made `pepperClone` also point to the same object. After that we set the name of `pepper` to "pepper". Because `pepper` and `pepperClone` point to the exact same object, when we ask `pepperClone` for its name, we get "pepper".

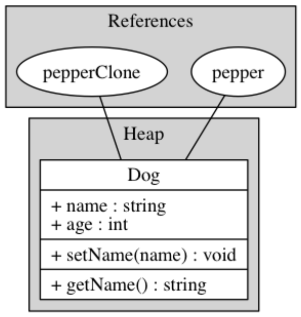

Figure 2. Object References

Other references. Once you let go of all the references to an object, it becomes eligible for garbage collection. The JVM still holds a weak or soft reference to the object (so it can manage it), but eventually, when it detects that memory needs to be cleaned up, it will finalize the object.[6]

Garbage Collection. When the JVM determines that it needs to free memory, it will perform a garbage collection. The soft and weak references will be cleared before throwing an `OutOfMemoryException`.

Garbage collection is controlled by the JVM. There are tweaks you can do to suggest a certain behavior to the garbage collector, and you can even suggest that it will do garbage collection (generally not a good idea), but eventually the garbage collector will decide when to run.

The benefit of garbage colection is that the programmer doesn't need to think about finalizing objects. When they are not needed, they may be thrown into garbage automatically.

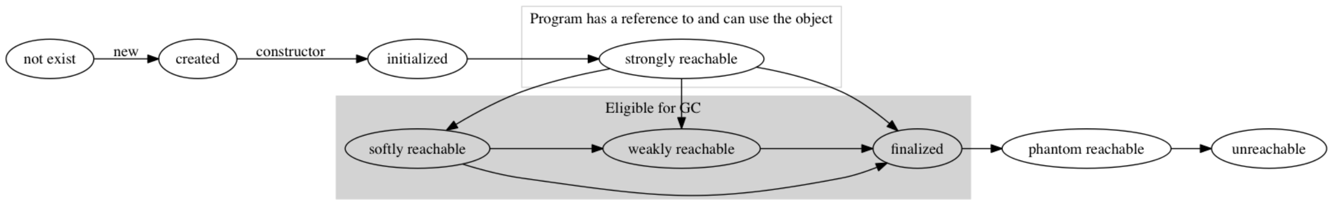

Figure 1. Object Life Cycle

There are times when this thinking can get you into troublem though. If you don't release the references the objects will never be garbage collected. An object is released when the program no longer holds a reference to it. You can either set the reference to null or it will automatically be released when your object goes out of scope.

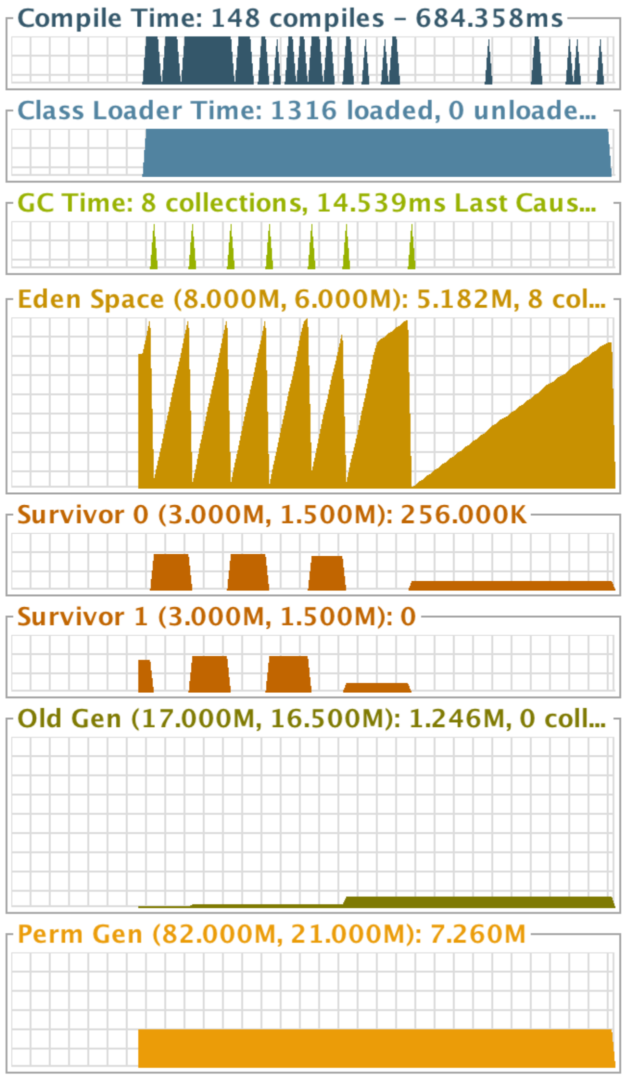
Compile Time: 148 compiles – 684.358ms
Class Loader Time: 1316 loaded, 0 unloade...
GC Time: 8 collections, 14.539ms Last Caus...
Eden Space (8.000M, 6.000M): 5.182M, 8 col...
Survivor 0 (3.000M, 1.500M): 256.000K
Survivor 1 (3.000M, 1.500M): 0
Old Gen (17.000M, 16.500M): 1.246M, 0 coll...
Perm Gen (82.000M, 21.000M): 7.260M

Figure 3. Garbage Collection

The above image shows garbage collection in action. I wrote a little program that creates objects and puts them in a collection. I used the visualvm tool provided in the Oracle JDK distribution[7]. After every 10000 objects I clear the collection. I wrote created a total of 306,480,000 objects, so I cleared the collection over 30,000 times. Garbage collection, though only kicked in 7 times. I had set my heap size to 25MB.

In the image you can also see the movement of objects from one generation to another.

1.2 Basic Data Types

Describe how the basic data types are represented in memory (boolean, int, long, String, array of ints, array of Objects, class with fields)

Java's primitive types can be divided into two[2] main groups: numeric primitives, and boolean. Numeric primitives consist of integral and floating point primitives.[1]

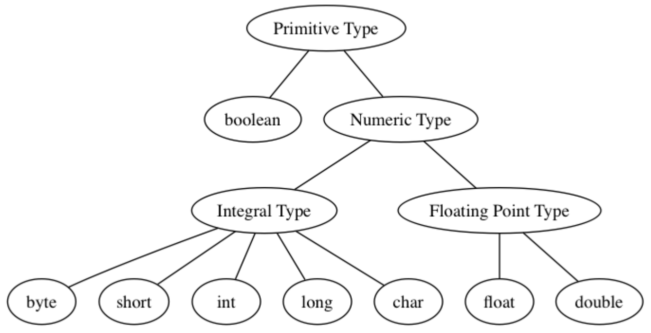

Figure 4. Primitive Classification

Java specification[1] defines sizes and ranges for primitive types. The integer types are clear cut, but the floating point types are not so simple. They follow the ANSI/IEEE Standard 754-1985[3]. The values in table 1 are from my MacBookPro, printing out the max value for float and double. They might be different on a different architecture.

Also to note that although Java defines the primitive sizes, on different architectures might actually use different sizes. For example, although an int is defined as 32 bits, it might take 64 bits on a 64 bit computer. The primitive sizes defined in the Java Specification is how the programmer sees the types, not necessarily how they are stored.

Here is the list of primitives in Java[1]:

[2]Really, there are three. The third one, `returnAddress`, is not available for the programmer

[3]see http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3 for details

Type	Size (bits)	Range
byte	8	from -128 to 127, inclusive
short	16	from -32768 to 32767, inclusive
int	32	from -2147483648 to 2147483647, inclusive
long	64	from -9223372036854775808 to 9223372036854775807, inclusive
char	16	from '\u0000' to '\uffff' inclusive, that is, from 0 to 65535
float	32	from -3.4028235E38 to 3.4028235E38[3]
double	64	from -1.7976931348623157E308 to 1.7976931348623157E308[3]
boolean	32[4]	true or false

Table 1. Java Primitive Types

class with fields

String in Java is represented by sequences of Unicode code points. String is a sequence of characters, which each is represented by two bytes.

array of ints

array of Objects

1.3 List of Strings

Write an application to find out how many total characters can be held in a list of strings before you run out of memory

some text

1.4 StringBuffer vs StringBuilder

Compare and contrast StringBuffer and StringBuilder and when to use each Notes: What does it mean to be concurrency resistant? Example? write sample application to show concurrent insert and non concurrent. Check for execution times. show concurrency failure. string+ is actually a better comparison point (who writes to strings from different threads?)

StringBuffer is designed to be thread-safe and all public methods in StringBuffer are synchronized. StringBuilder does not handle thread-safety issue and none of its methods is synchronized.

http://stackoverflow.com/questions/16653119/imagine-a-real-concurrent-scenario-where-stringbuffer-should-be-used-than-string (last examnple) Test with aaaaaa and bbbb strings of equal length, or different lengths.

[3] Java float and double follow the ANSI/IEEE Standard 754-1985, see http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3

[4] `boolean` is internally implemented like an `int`

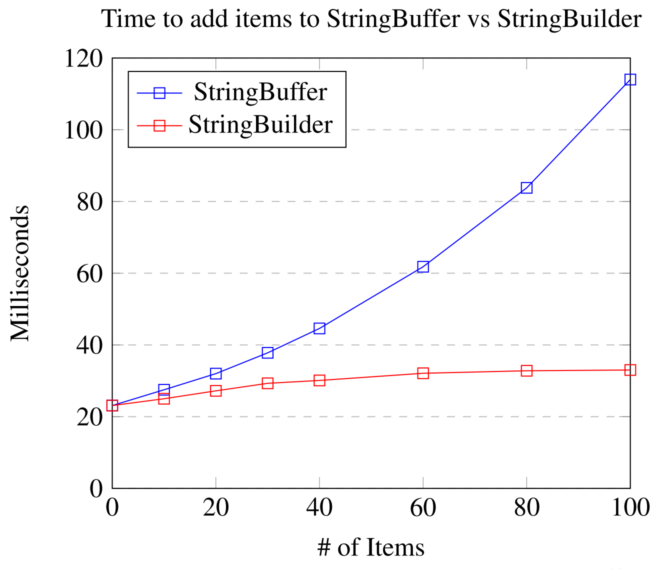

Figure 5. Insert time into a StringBuilder vs. a StringBuffer

Reference to Figure 4.

1.5 StringBuffer vs StringBuilder

Compare/contrast use of ArrayList / LinkedList / HashMap / HashSet / TreeSet Notes: write sample application to show the use of the collections. Check for execution times Make a table for comparison

some text

1.6 Sorting in Order

Write an application to read a file with 10k lines of text, and output another file with the lines in sorted order.

some text

1.7 Sorting in Reverse Order

Write an application to read a file with 10k lines of text, and output another file with the lines in reverse sorted order

some text

1.8 Exceptions

Write code to show exception handling including examples of checked, unchecked, and Error exceptions

some text

1.9 Enums

Write your own enum type. Describe when you would use it.

some text

2. Methods, Encapsulation and Inheritance

some description

2.1 Composition, Inheritance, and Static Method Calls

Show how to use a common piece of logic from two different classes, in three different ways: 1) by composition, 2) by inheritance, and 3) by static method calls, discuss the tradeoffs for example:

- *two different classes that write a message to a file, one in XML, one in line-oriented text, but both need to reuse logic to open the file in the same way*

some text

2.2 Constructors

Create and overload constructors – Create a class that has 4 fields and construct the class with variations of one required field and the others are optional. Use constructor chaining as an example.

some text

2.3 Encapsulation

Apply encapsulation principles to a class – Show an example of good encapsulation. Show a bad example of encapsulation and explain why. Additionally explain access modifiers and how they can be used as part of the class encapsulation.

some text

2.4 Object References and Primitives

Determine the effect upon object references and primitive values when they are passed into methods that change the values – Create a method 3 parameters, one is parameter is pass by value, one is passed by reference and one with the keyword final. Explain each and what the effects in side the method that changes each one.

some text

2.5 Access Modifiers

Write code to show how access modifiers work: private, protected, and public, talk about why you would use each of these.

some text

2.6 Virtual Method Invocation

Write code to show how virtual method invocation lets one implementation be swapped for another.

some text

2.7 Casting

Write code that uses the instanceof operator and show how casting works.

some text

2.8 Overridden Methods

Show how to override a method in a subclass, talk about plusses and minuses in doing so.

some text

2.9 Overloaded Constructors and Methods

Show how to overload constructors and methods, talk about plusses and minuses in doing so.

some text

3. Library

Some description

Creating a Library some text

Using a Library some text classpath

3.1 Logging Directly

Write an application that uses the slf4j logging library directly (can also choose log4j if you want)

some text

3.2 Logging Configuration

Do the following:

- *configure the logging using an accepted department log statement format (see Application Logging)*
- *log at different logging levels (error, warn, info, debug), to see the effect of the default logging level setting*
- *turn on DEBUG in the logging config to show DEBUG output*
- *configure logging to go to both the console and a log file)*

some text

Appendix

Appendix A

```
// create an instance of Dog
Dog pepper = new Dog();

// pepperClone also points to the same object
pepperClone = pepper;

// set pepper's name to "pepper"
pepper.setName("pepper");

// returns "pepper"
pepperClone.getName();
```

References

[1] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java® Virtual Machine Specification.* The Java® Virtual Machine Specification. Oracle America, Inc, 28 Feb. 2013. Web. 12 Aug. 2014. <https://docs.oracle.com/javase/specs/jvms/se7/html/>.

[2] Lindholm, Tim, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java® Virtual Machine Specification.* The Java® Virtual Machine Specification. Oracle America, Inc., 28 Feb. 2013. Web. 12 Dec. 2014. <https://docs.oracle.com/javase/specs/jvms/se7/html/>.

[3] *Java Platform SE 7.* Java Platform SE 7. Oracle, n.d. Web. 12 Dec. 2014. <http://docs.oracle.com/javase/7/docs/api/>.

[4] *Trail: Learning the Java Language.* The Java™ Tutorials. Oracle, n.d. Web. 12 Dec. 2014. <https://docs.oracle.com/javase/tutorial/java/index.html>.

[5] Nicholas, Ethan. *Understanding Weak References.* Understanding Weak References. Java.net, 4 May 2006. Web. 13 Dec. 2014. <https

[6] *Package java.lang.ref.* Java.lang.ref (Java Platform SE 7). Oracle, n.d. Web. 13 Dec. 2014. <http://docs.oracle.com/javase/7/docs/api/java/lang/ref/ package-summary.html>.

[7] *Java Garbage Collection Basics.* Java Garbage Collection Basics. Oracle, n.d. Web. 13 Dec. 2014. <http://www.oracle.com/webfolder/technetwork/tutorials/ obe/java/gc01/index.html>.