Java vs. Scala:

First Impressions of Scala Through Huffman Coding

By Zachary Mikel

Introduction

Java is one of the most popular programming languages in the programming community. It was invented at the beginning of the object-oriented era, and has continued to be one of the most widely-used languages to this day. It's easy enough for most people to learn. Colleges and universities primarily teach Java primarily in their computer science programs to keep students engaged; it's easy enough to learn that it doesn't scare many people off. Java has had excellent funding since its birth, and continues to receive huge updates every couple of years. It's no wonder that it is so successful.

Functional programming hasn't been hugely popular, because of its harsh recursive and mathematical structure. A functional program is thought of as a function that takes inputs, does work on those inputs and produces a value. The function can call itself or other functions, but it still is responsible for simply taking an input and producing an output. This means that there aren't really "singleton" or static members in a functional program, there is only input and output. Programming languages like Lisp, Scheme, and Haskell have all been limited to the academic world for a long time. They're hard to learn and more difficult to understand, and it takes much longer to become fluent enough in one of those languages than an object-oriented language.

In recent years, however, elements of functional programs are seeping into mainstream languages. Both Java 8 and C# include "lambda expressions", which allow the creation of typeless, anonymous functions to be used alongside objects and structures. This rise in popularity can be mostly attributed to the growing need for multi-core and distributed computing efficiency. Functional programming eliminates side effects, preventing multi-core systems from walking over each other's work.

Small tidbits of lambda expressions in Java, C#, and other object-oriented languages don't really solve the problem of eliminating side effects. These languages just aren't designed to work well without modifying things indirectly. What if we could have a language that was a true blend of functional and object-oriented programming? A true half-step between these two worlds that allowed us to program functionally, but use objects when we needed to? This is how Scala came to be.

Scala can be a purely functional language, but a program is "allowed" to have objects that can be a collection of values, properties, and functions. It handles abstraction and interfaces just like Java, but is really meant to be a functional language first, utilizing objects when it needs to. The best part about Scala is that is fully-compatible with one of the most popular programming languages: Java! When it was first introduced, it seemed that Scala was going to be the next big thing in programming. The inventor of Scala actually helped write some of the JVM code, and Scala runs on the JVM. This makes it just as portable and mobile as its parent. There are many things that Scala does much more elegantly than Java, including more concise syntax and deliberate attempts to eliminate "boilerplate" code.

There is a lot of "hype" about Scala being much faster than Java because of its functionality. Since it is designed to utilize much more recursion and functions, in theory it could be quicker than its object-oriented parent. Scala has many features that make it seem like it should perform just as well or better than Java. The reason this language was selected in this project was to put it to the test; to see if it really lives up to the talk about its performance and

superior syntactic flow in comparison to Java. One of the best ways to try out a new language is to pick a project that is moderate in difficulty, and to design and implement it with the new language's documentation close by. In this project, a Huffman encoder is implemented in both languages to compare the them and observe the unique characteristics of Scala.

Scala: Overview of Unique Characteristics

Scala is indeed a different beast altogether from Java. It shares many similarities with Java in its underlying structure (in fact, both implementations of this Huffman encoder were written, tested, and debugged in the same IDE.) Simple variable assignments are close enough, shown in the declaration of an integer variable:

```
Java:                int i = 1;
Scala:               val i: Int = 1;
```

Figure 1.1

Curiously enough, the Scala definition of a variable looks more verbose than Java. In Scala, any variable assignment can be immutable or mutable, depending on what the programmer specifies. In Java, one would only use an immutable variable for a class-level member or singleton object that would remain in the same state over the life of the program. Scala allows the programmer to explicitly set *anything* as mutable or immutable upon its declaration. It does not care if a "static" or immutable member is referenced via static or non-static context; an action that, performed in a Java program would surely result in the Java compiler screaming at the programmer.

Scala treats everything as an object. The declaration in figure 1.1 shows a Java primitive type of "int" being declared, whereas Scala's variable declarations always use object wrappers (e.g. Java's Integer). Because of this tendency to treat everything as an object, functions are objects too! They can be defined and passed around to other functions as parameters. Here is a Scala function that takes a parameter called "param", which is an anonymous function that should evaluate to an Int.

```
def myFunc(param: () => Int)
```

Figure 1.2

By default, most data structures used in Scala are immutable. This is meant to facilitate situations that are parallel-computing friendly, because immutable data structures don't change after their declaration. Because anonymous functions are so easily available, it is much easier in Scala to write a function that puts all desired outputs into an immutable data structure than it would be in Java. This functionality is nice, but it is very difficult to read. For example, consider the following code from the Huffman encoder:

```
fileContent.foreach(c => {
 if (frequencyTable.getOrElse(c, 0) == 0) {
   frequencyTable.put(c, 0)
 }
 frequencyTable.update(c, frequencyTable(c) + 1)
})
frequencyTable
```

Figure 1.3

Here, we have a loop that iterates the file content, inserting character frequencies into a map that will be used to build the code tree. We declare an anonymous member, "c", that will be the iteratee of each iteration of the foreach loop. Its type is inferred by the compiler, and since we are inserting it into a map of type [Char, Integer], the loop executes successfully.

A good example of Scala's conciseness over Java's is displayed in an online Java blog[1]. This is a simple program that takes in a list of integers and sums any of the integers that are multiples of 2.

```
Java:
List<Integer> iList = Arrays.asList(2, 7, 9, 8, 10);
List<Integer> sumTwo = new ArrayList<Integer>();
for(Integer number: iList){
    if(number % 2 == 0){
       sumTwo.add(number, 2);
    }
}
// output: 20
```

```
Scala:
val iList = List(2, 7, 9, 8, 10);
val sumTwo = iList.filter(_ % 2 == 0).map(_ + 2)
// output: 20
```

Figure 1.4, taken from Javarevisited: Blog about Java Programming

Here we have a Java program that sets up an ArrayList and declares some integer values. Then, each member of the list is visited. If the member is divisible by 2, the member value + 2 is added to the new list. The Scala version has the same end result, but does the same problem in two lines. This is because we are able to assign the output of a function directly to a value, rather than modify a variable in a for-loop. The underscore character "_" is a wildcard. It can represent anything, or can be used in pattern matching.

Scala tries desperately to eliminate the "null" value from its code altogether. Instead, things like the _ wildcard and the Option[A] type were invented. Option is simply a wrapper for a result that could possibly be empty. This forces the programmer to explicitly handle situations where an empty value could be returned. While it feels tricky and messy when learning this in Scala, many developers claim that Option is actually a beautiful thing when mastered.

Scope, Names, and Binding

Scoping and binding in Scala are strikingly similar to Java. They are both statically-scoped languages. Method-level variables are only accessible inside of the current method, and are destroyed after the method's execution is finished. If a local variable has the same name as a class-level variable, the local variable in scope will override the class variable unless the class level variable is called out specifically with the "this" keyword.

Visibility of variables in Scala is slightly different from Java's in that all variables and classes are public by default. There are protected members, but they are not visible from anywhere besides top-level members inside the same package. The private modifier can be used in a couple of ways. If a variable is marked private in Scala, it can still be accessed by other instances of the same class. However, there is the option to mark something as private[this], which makes a property only visible to the current class. This is a level of encapsulation deeper than what is available in Java, aligning with the side-effect elimination goals of this language.

Scala's declarations are a little different than Java's, but both languages have similar components to their naming. Everything in Scala is named with the following pattern:

```
name : Type = value/function/etc
```

Method names and parameters follow a similar pattern. In lambda and anonymous functions, type does not need to be specified as the compiler infers data types in these situations. It should be noted that Scala does not allow a variable to be instantiated as a null type. If a variable must be declared that doesn't have an immediate value, the situation should be redesigned, or the wildcard underscore character should be used.

Scala also has a slightly deeper level of specificity when it comes to objects. In Scala, there are abstract classes (like an interface or abstract class in Java), and concrete classes that can implement their abstract parents or extend their concrete parents. The step up from this is a Scala object, which is like a class, but all of its values are declared and immutable - they cannot change unless there are functions within the object that change their values. This made a difficult time of designing classes in the Huffman project, because they are used so differently in Scala. However, there were enough situations in the encoder that all types of Scala classes and objects were used.

Project Overview

In order to understand the project, we should quickly recap what a Huffman encoder does. At a high level, a Huffman encoder takes in a file and counts the frequency of each character in the file. This means that we must read through the entire file in order to build a table of each character's frequency. Then the two lowest-occurring characters are picked from the table and added as children to an internal node. The internal node is placed back into the table. This process is repeated until there is only one entry in the table, with all of the internal and leaf nodes connected to it. From this tree, we can encode every character in the file with less bytes than the original file. We iterate every character in the file, writing its path through the binary tree as its encoding (0 for left, 1 for right.) When decoding the file, we must have a reference to

the tree so that we can navigate the tree as we read in the bits from the encoded file. The formal algorithm for encoding characters is shown in the following snippet:

Steps to build Huffman Tree

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Figure 1.5, taken from Geeksforgeeks/greedy-algorithms/huffman-encoding

In scouring the internet for examples of Huffman encoding (one of the most popular file-compression algorithms), there are almost no examples of "real-life" implementations. Most of the projects that exist are only models of Huffman encoders where the input files and trees are already designed into the program - not ones that actually write out to files and decode them. One of the goals of this project was to build a "real-life" encoder in both languages. Scala doesn't attempt to redo any of the Java file I/O operations, so writing to the files is similar between both languages.

Implementing a real Huffman encoder proved more challenging than the initial research would have indicated. The actual bitwise operations were not apparently obvious. After encoding a file, the encoding table for a set of characters might appear like this:

Character	Encoding
A	101
B	10
C	100
D	11

Figure 1.6

In a typical text file, each character should take up one byte (8 bits.) When encoding down, the end result should use much less than one byte to encode a character as seen above. This means

that some of our characters may span multiple bytes. If we were able to view each bit of an encoded file for the string “ABCD”, it would appear as follows:

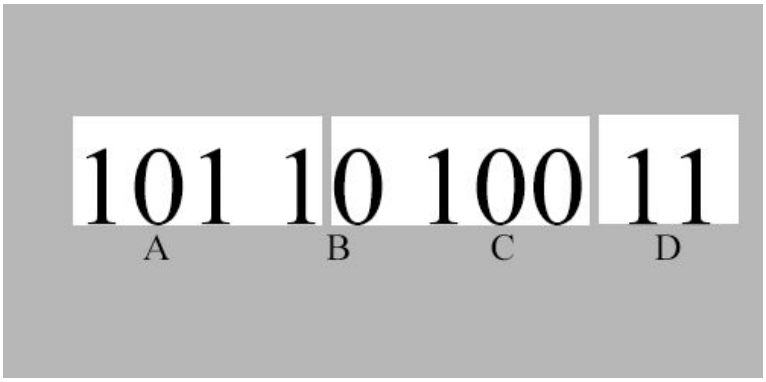

Figure 1.7

This is not an issue for encoding; we simply build up a byte of encoded bit until it is full. We write the full byte to the file, and move along to the next byte. If we get to the end of our characters and we haven’t filled out the last byte, we just write zeros into it at the end. The bit shifting expression for this act of “building up” the current byte is as follows:

```
def write(i: Int, output: FileOutputStream) {
 currentByte = (currentByte << 1) | i
 bitsFilled += 1

 if(bitsFilled == 8) {
   output.write(currentByte)
   currentByte = 0
   bitsFilled = 0
 }
}
```

Figure 1.8

Decoding requires us to “extract” the least significant bit off of the current byte, navigate in the direction down the tree that the bit indicates (0 for left, 1 for right), and remove this bit from the current byte. This must be done with a sort of bit mask expression, where we isolate the bit at the end of the current byte.

```
def readBit(): Int = {

   if(currentByte == -1) return -1

   if(bitsLeft == 0) {
     currentByte = inputStream.read()
     if(currentByte == -1) return -1
     bitsLeft = 8
   }
```

```
  bitsLeft -= 1
  (currentByte >>> bitsLeft) & 1
 }
```

Figure 1.9

During Huffman encoding, the encoder's tree must be visible by the decoder somehow or the decoder won't know how to decode the characters in the encoded file. This can be achieved in implementation by writing the tree to the file, or developing a Canonical Huffman code (where the codes are ordered lexically by length.) For ease of implementation and due to time constraints, neither of these solutions were implemented with these huffman encoders. The encoder simply passes along its code tree to the decoder instantly.

This program should reduce input files to about 60% of their original size. When tested on the *Alice In Wonderland* excerpt, both programs took the original file (approximately ~160 kilobytes) and reduced it to around 120 kilobytes. The surprising part of this project was that the Java implementation was about twice as fast as the Scala implementation on most files.

The code between the two projects is structured the same. Almost every function in the Scala project is written with the Scala API, except the file input and output streams. With many hours invested, I felt that I only had a small grasp on how Scala is supposed to work. Most of the development on the Scala side of the project consisted of me lapsing into my old object-oriented habits, only to find that there are better solutions to do things functionally in Scala. Future development may include redoing the Scala program in a way that uses more advantages of the language.

Implementation Observations

When comparing the two programs side-by-side, we can see that the Scala program took less total lines of code than its Java counterpart. There are also no placeholder values in the Scala program. Anything that is declared immediately receives an assignment from either function output or some value. The exception handling in Scala is slightly cleaner for the calling functions, but looks awkward in the functions that use the Option wrapper.

Figure 1.10

```
def nextSymbol(): Option[Char] = {
 var currentNode = root
 while(true) {
   currentNode match {
     case internal: InternalNode => {
       var next: Node = internal.left
       val direction = readBit()
       if (direction == 1) next = internal.right
       currentNode = next
     }
     case leaf: LeafNode => return if(leaf.c != '□') Option(leaf.c) else None
   }
 }
 None
}
```

This is the Option-wrapped method that traverses the tree to decode the file. This is reminiscent of a switch statement in Java. There is a loop that either builds up an Internal Node, or returns a Leaf node's character, or None if the node is null. This function is looking for the corresponding character that we're navigating to in the Huffman tree. If the character is the special end-of-file character, the method returns None so that the caller knows that it's time to finish decoding the file.

Recap

Scala is a very useful language that has some incredible advantages. Its portability and two-way compatibility with Java allows developers to use bits of Scala code inside of Java applications (or the other way around), which I may consider doing more with server/back end projects in my day job. I have become (in a very short time) a fan of the approach Scala takes toward exception handling, it is enjoyable how much less code it takes to accomplish menial tasks.

Creating reusable code is a huge commitment that we all try to keep as developers. However, Scala allows too much margin for squashing readable code into cryptic symbols. If it were being used in an organization where code maintainability is key, some strict regulations would need to be put in place in order to keep things readable and reusable.

The reason that Scala will probably never be used as a mainstream language in the industry is that most computer scientists are not strictly academically or mathematically-minded. Scala uses mathematical type theory, implicits, and macros that can cause the flow-of-control to behave unpredictably if a programmer is used to Java or C#. It has been observed that some teams productivity actually decreases when they switch to Scala, even after a year or longer.[2] The learning curve that it would take to become proficient in this language just isn't worth the cost of "converting" from a traditional language. It is arguable, however that Scala sparked the addition of functional programming features in Java 8. It is an excellent language nonetheless, and has very specific examples where its use is much more practical than Java (like big data science.) While it won't become the next mainstream language like its parent, Scala paved the way to the era that we are in now; hybrid languages consisting of object-oriented environments with elements of functional programming.

Sources

1. "Scala vs Java - Differences and Similarities." *JavaRevisited: Blog about Java programming*. Web. 17 Apr. 2017.
2. Kranc, Moshe. "The Rise and Fall of Scala - DZone Java." *Dzone.com*. N.p., 14 Mar. 2017. Web. 17 Apr. 2017.