SOFTENG 701
Advanced Software Engineering Development Methods Assignment #2 Report

Ruoyi (Zoe) Cai
6007959 – rcai861

I. Map Literals

Map literals is a much requested feature and is often asked about by developers who do not know that it is not supported in Java, hence it is a feature that attracts a substantial amount of interest. (Evans, Nov 24, 2015)

Adding support for map literals brings many advantages, including improvement in developer productivity, code maintainability, software performance, and code safety.

A. Reasoning

1) Developer productivity

The current way of initializing a HashMap instance is to first create the HashMap, and then adding the key value pairs one by one using the put() method. This makes creating a HashMap object with more than a few key-value pairs tedious and time consuming. By allowing the key-value pairs to be added in one block, the developer could potentially take the key-value pairs from another file format – for example, a csv file – format them automatically, then copy and paste them into the code. Although one could do the same with the current way of creating HashMaps by formatting the key-value pairs to include the word "put()", the new feature makes formatting much simpler.
The new way of initialising HashMaps also removes the need for the developer to specify type arguments (they are inferred from the types of the map entries in the literal expression). This also saves developer time. As the types supported by this feature are primarily primitive types, the map entries would make it clear for the reader of the code what the types of the keys and the values are.

2) Code maintainability

The new feature "reduces verbosity and adds clarity when creating small [HashMap objects] and whose contents are known at compile time. This feature is well-liked in the many languages that support it (e.g., Perl, Python)." (J. Bloch, Mar 30, 2009)

The new feature reduces lines of code that must be written by combining the initialising of the map and the addition of entries into one method. Less lines of code reduces class size, which has a positive impact on class maintainability, reducing effort and cost. (Al Dallal, 2013)
Other than reducing the lines of code, the new feature also improves the clarity of code, making the finished code easier to read and therefore to maintain. The new feature makes it clear that the entries are added at initialisation.
The clarity of code diminishes during maintenance. Software maintenance is often done under stressful and time-constrained situations that discourage the developer to carefully observe and understand the construction of the original code before making modifications. (Smith, 1999) It is therefore quite possible for a developer to insert code between the initialisation of the HashMap and the addition of the map entries, separating the two and potentially causing confusion. The new feature prevents this from happening as it keeps all of the entries together and makes it easy for the developer to see which entries should be added together.

3) Performance

Performance is improved with the new feature as the size of the HashMap is known at initialization, so an implementation that is more space-efficient can be selected immediately at initialization.
When a HashMap is created using the existing method, if the initial capacity and the load factor are not specified, the HashMap defaults to an initial capacity of 16 and a load factor of 0.75. Each time a new map entry is added using put(), the function checks if it needs to resize the HashMap by checking if the number of entries in the HashMap exceeds the threshold (which equals to the number of buckets * the load factor). If resizing is needed, the number of buckets is doubled and all the existing entries are redistributed. (Kalenzaga, October 14, 2015) The process of resizing the inner array of the HashMap results in a higher time complexity compared to if the size of the HashMap were known when the HashMap was created, especially if the number of entries later added to the HashMap were large.

4) Code safety

The new feature also allows a HashMap object to be made immutable. Immutability has many advantages and strengthens code safety. As an immutable object can only be in the state that it was created in, it is simple to use reliably. Immutable objects are also inherently thread-safe and do not

require synchronisation. They are the easiest approach to achieving thread safety, as their state cannot be modified and corrupted by multiple threads accessing them concurrently. (J. Bloch, 2008)

B. Design Decision

An empty HashMap initialization is not supported as the same result could be accomplished using the original way, `HashMap map = new HashMap();` Adding the curly braces afterwards with no key value pair defeats the purpose of the feature, which is to make code simpler and cleaner.

Only primitive types and String are supported in this new feature, as it is unlikely for a developer to manually initialise a map with objects of reference types – it is much more likely for these objects to be added to the map in a loop using the put() method, after the map has been initialised.

C. Implementation

1) Parsing

Two classes were created in the src/japa/parser/ast/expr/ directory to accommodate the parsing and processing of the new HashMap feature. HashMapCreationExpr.java extends the Expression class and stores the type of the entire expression, i.e., a class, as well as the type of the keys and values inside the map. It also stores an instance of the other class, HashMapInitializerExpr.java, which represents the initialization of the HashMap, i.e., the part between the two curly braces containing the actual key and value pairs.

This is consistent with the way the other expression classes are organized.

The java_1_5.jj file was modified to parse the new grammar. Expression AllocationExpression contained most of the modifications, and HashMapInitializerExpr HashMapInitializer() was added as well to assist with the parsing. AllocationExpression is responsible for the creation of new instances of classes and for the creation of arrays, which makes it perfect for parsing the new HashMap syntax. A ClassOrInterfaceType type is first assigned, followed by the expression itself, which is handled by HashMapInitializer. The HashMapInitializer parses the key and value pairs between the curly braces, and creates and returns an instance of HashMapInitializerExpr containing the key value pairs. AllocationExpression then creates a new HashMapCreationExpr instance containing the type of the expression and the key value pair expression.

A lookahead of 3 was added to the array creation option inside of AllocationExpression in order to prevent conflict.

2) Visitors

In semantic analysis, the CreateScopesVisitor first adds the scope to the HashMapCreationExpr node so that the types of the keys and values could be resolved later on.

The DefineTerminalVisitor creates symbols and adds scopes to the terminal nodes such as binary expressions. It does not modify the HashMap-related nodes.

The CheckTerminalVisitor then checks through the keys and values of the HashMap to make sure that all keys are of the same type, and that all values are of the same type. If it comes across a key or a value that differs in type from the previous keys and values, it throws an exception with the line number.

Finally, the DumpVisitor prints out the correct, transformed code to the final java file. It does so by visiting the HashMapCreationExpr node directly from VariableDeclarationExpr, first printing out the class of the HashMap – as it could be either Map or HashMap – and then checking that the node contains a HashMap creating expression. This is done so that the type of the key and the value could be added to the output code. If the node indeed contains a HashMap expression, the HashMapCreationExpr is immediately visited.

In the HashMapCreationExpr visit method the name of the HashMap instance is printed first, using the argument passed from the VariableDeclarationExpr, and then the types of the key and value pairs are printed in brackets. As it is certain at this stage that the expression represents the initialization of a HashMap, the rest of the line, "= new HashMap<>()", is hardcoded in the visitor code and is printed. The visitor then gets the HashMapInitializerExpr expression and iterates through the key-value pairs stored in the values field in the HashMap, and prints out a map.put(key, value) accordingly.

D. Conclusion

In conclusion, adding HashMap literals addresses an important issue in Java. It allows developers to be more productive by making code more concise and therefore more clear and easy to read. It improves code maintainability by grouping the HashMap creation and the addition of map entries that are known at compile time. It improves performance, removing the need to increase the inner array size for the buckets in the HashMap when the initial size could be inferred from the entries added during the initialisation of the map. And finally, it strengthens code safety, allowing the HashMap objects to be made immutable and therefore thread-safe and reliable.

II. References

Al Dallal, J. (2013). Object-oriented class maintainability prediction using internal quality attributes. *Information and Software Technology, 55*(11), 2028-2048.

Bloch, J. (Mar 30 16:30:14 PDT 2009). *Proposal: Collection literals* Retrieved 4/16/2016, 2016, from http://mail.openjdk.java.net/pipermail/coin-dev/2009-March/001193.html

Bloch, J. (2008). *Effective jav*a (2nd ed.. ed.). Upper Saddle River, NJ; Upper Saddle River, N.J.: Upper Saddle River, NJ : Addison-Wesley c2008.

Evans, B. (Nov 24, 2015). *Java: The missing features* Retrieved 4/15/2016, 2016, from http://www.infoq.com/articles/Java-The-Missing-Features

Kalenzaga, C. (October 14, 2015). *How does a HashMap work in JAVA - coding geek* Retrieved 4/16/2016, 2016, from http://coding-geek.com/how-does-a-hashmap-work-in-java/

Smith, D. D. (1999). *Designing maintainable softwar*e. New York, NY: New York, NY : Springer New York : Imprint : Springer 1999.