# Dependency Injection in Java

## An extension for ExtendJ

Niklas Jönsson
D13, Lund University, Sweden
dat13njo@student.lu.se

Kim Fransson
D13, Lund University, Sweden
dat13kfr@student.lu.se

## Abstract

In this paper we discuss the problems of slow execution times as a result of sub-optimal choices for the underlying data structures. We present our dependency injection extension for Java that collects information about library collection type uses and makes an intelligent choice on what data structure to use.

Our solution lets the programmer decide if he wants help from the compiler to make the choice by specifying an interface type on the right hand side of an assignment when declaring a variable.

Our solution scratches the surface of dependency injection and achieves performance gains. However there is still a lot to be discovered and these gains could be greater with more work.

**ACM Reference Format:**
Niklas Jönsson and Kim Fransson. January 28, 2018. Dependency Injection in Java: An extension for ExtendJ. In *Proceedings of Project in computer science (Course paper, EDAN70).* ACM, New York, NY, USA, 5 pages. https://doi.org/

## 1 Introduction

In software programs the data is stored in various type of data structures, such as *lists, sets* and *maps*. In Java, and many other programming languages, there exist a collections framework (library) containing sets of classes that implement these commonly used data structures. These data structures work in different ways and they all have their limitations and advantages[5]. The task of deciding what implementation of these interface to use can be a boring task for a programmer that just want things to work.

Today the computers are very powerful and can execute most of the time consuming operations in a fraction of a second and a relative slow execution time wont raise any eyebrows as the absolute execution time is still very low. This leads to that the programmer will most likely use the option he is most comfortable with. In larger programs using the optimal underlying data structure is always of uttermost importance, as the execution time could multiply and result in a very long execution time if a non-optimal choice is made. According to a study made in 2009 [4], expert software engineers at Google made sub-optimal choices for underlying data structures in C++ applications in an internal benchmark suite. A speedup of 17% was measured when a single line change to an underlying data structure was made.

It is hard to extensively test a program with data of various sizes and for the programmer to realize that one certain data structure might not be optimal to use in a certain situation. It therefore would be preferable if the programmer had the option to let the compiler decide which data structure that should be used if the programmer is in doubt of which is optimal.

There already exists different type of solutions to this problem, but they are not yet established. The analysis tools Brainy[1] and Chameleon[7] use offline inspections in order to produce suggestions to the programmer of which data structure is optimal. The tool CoCo[8] offers online dynamic replacement of data structures depending on the situation and context of a program.

What separates our solution from the above is that we provide the programmer an option to let the compiler decide which data structure to use. The actual replacement is done on the language level and is therefore not dependant on any external java virtual machine as it can run on the stock java virtual machine.

The solution is based on the dependency injection technique, where the programmer can specify when he requires a variable to be injected with the optimal type. Therefore the programmer can hand over the tedious task of choosing the optimal class to someone that knows better (allegedly). Our dependency injection lets the programmer skip the task of creating boilerplate code for factories and manual dependency injection. This will result in more modular, more clean and more test friendly code as well as it will save time for the programmer. The injection is done on the language level before the code is compiled into bytecode which results in the same portability as Java.

To evaluate our solution we used the three different benchmarks. The first stage of evaluation was done with micro-benchmark. The main evaluation was done with the SPECjvm2008 benchmark suite. The results can be seen in section 5. Our results from the micro-benchmark showed that our solution did increase the performance significantly, however the result from SPECjvm2008 only showed minor performance gains.

*Course paper, EDAN70, Lund University, Sweden*
January 28, 2018.

## 2 ExtendJ

Our solution uses the extensible Java compiler ExtendJ[2] which is built using the declarative attribute grammar system JastAdd[1]. ExtendJ allows modules to be added which extends the support of the compiler. An informative image of the structure and the modular architecture can be seen in figure 1.

ExtendJ was developed at Lund University as a research project and has since then been updated rigorously and support for new versions of Java has been added. As of today the compiler supports Java 8 but the bytecode generation for Java 8 is not yet fully supported.

ExtendJ is open source and available under the modified BSD license.

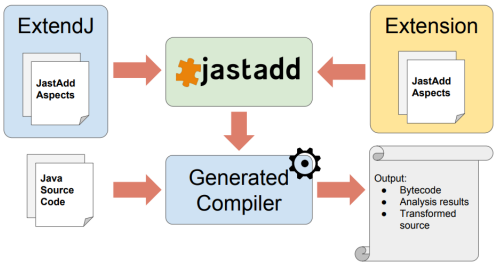


**Figure 1.** Overview of ExtendJ [3]

## 3 Language extensions

Our solution extends the Java language with new functionality, it is now possible initiate certain interfaces by writing:

```
Interface i = new Interface();
```

At this time we have support for the common interfaces `List`, `Set` and `Map` that are found in the standard library.

The ExtendJ compiler will parse an interface initialization and decide whether it is able to perform an injection or not by mapping the interface type to a whitelist of supported interfaces. If the interface is supported it will run an algorithm that replaces the interface initialization with the deemed most fitting class that implements that interface.

We have also added variable declaration annotations to the java language to allow the programmer to enforce behaviour of the data structures. These annotations are placed before declaring a new variable:

```
@Annotation Interface i = new Interface();
```

All code replacement is done before any byte-code is generated. Our solution is portable as the generated code is base java compliant.

[1]http://jastadd.org/web/

## 4 Implementation

To decide which implementation of the interface to replace with, we have designed and implemented a decision tree based static analysis along with a dynamic analysis aspect. The dynamic analysis is only needed if no solution can be found in the static analysis.

### 4.1 Static analysis

In the static semantic analysis the compiler looks for specific annotations written by the programmer. These annotations describe requirements for the order of elements in the data structure. We have added two new annotations `@Sorted` and `@InsertionOrder`. These annotation are added to the variable declaration. `@Sorted` indicates that the elements should be sorted according to the elements natural sorting order. `@InsertionOrder` indicates that the elements should be ordered in the order they were insorted. Depending on which annotation is used, a data structure is automatically selected because the programmer enforces some certain behavior.

### 4.2 Dynamic analysis

In the dynamic analysis the compiler compiles a version of the program for each implementation of the data structure in question. Each of these versions are executed several times and a mean execution time is calculated. The implementation with the lowest mean execution time is deemed the best choice for the context and is chosen as the replacement. If the analysis would fail a default implementation is chosen.

### 4.3 List replacement

The classes that can be injected from a List interface is `ArrayList` and `LinkedList`. Table 1 shows the difference in time complexity in big O notation for the two implementations. To decide which implementation to select the compiler makes a dynamical analysis. If the dynamical analysis fails, the compiler uses `ArrayList` as the default choice, although it may not be optimal.

**Table 1.** Time complexity comparison for `List`[6]

| | get | add | contains | next | remove(0) | iterator.remove |
|---|---|---|---|---|---|---|
| ArrayList | O(1) | O(1) | O(n) | O(1) | O(n) | O(n) |
| LinkedList | O(n) | O(1) | O(n) | O(1) | O(1) | O(1) |

### 4.4 Map replacement

For `Map` the class choices is between `EnumMap`, `TreeMap`, `LinkedHashMap` and `HashMap`.

The compiler first check the key element type. If the key is of type `Enum` the choice is `EnumMap`. `EnumMap` is implemented for handling enum keys extremely efficient[2], As seen in table 2.

[2]https://docs.oracle.com/javase/7/docs/api/java/util/EnumMap.html

After that the compiler checks against available annotations. If the programmer specifies the `@InsertionOrder` annotation, `LinkedHashMap` is chosen as this implementation is the only implementation that preserves the insertion order. If `@Sorted` is present the compiler chooses `TreeMap` as this is the only implementation that sorts the elements in the natural ordering of the element type. These implementations may not be optimal but are chosen as they satisfy the programmers demand.

If the compiler has not yet decided which implementation to use it moves on to the dynamic analysis. If the dynamical analysis would fail, `HashMap` is the default choice.

**Table 2.** Time complexity comparison for `Map`[6]

| | get | containsKey | next | notes |
|---|---|---|---|---|
| HashMap | O(1) | O(1) | O(s/n) | s is the table capacity |
| LinkedHashMap | O(1) | O(1) | O(1) | |
| EnumMap | O(1) | O(1) | O(1) | |
| TreeMap | O(log n) | O(log n) | O(log n) | |

### 4.5 Set replacement

The `Set` classes that are available for selection is `EnumSet`, `HashSet`, `TreeSet` and `LinkedHashSet` and the time complexity values can be seen in table 3.

If the element type is `Enum`, `EnumSet` is selected because it is implemented using bit vectors which are very compact and efficient[3].

If the `@Sorted` or `@InsertionOrder` annotation is used the `TreeSet` respectively `LinkedHashSet` is selected.

If the dynamical analysis would fail, `HashSet` is the default choice.

**Table 3.** Time complexity comparison for `Set`[6]

| | add | contains | next | notes |
|---|---|---|---|---|
| HashSet | O(1) | O(1) | O(s/n) | s is the table capacity |
| LinkedHashSet | O(1) | O(1) | O(1) | |
| EnumSet | O(1) | O(1) | O(1) | |
| TreeSet | O(log n) | O(log n) | O(log n) | |

## 5 Evaluation

All the tests are executed on a quad-core machine with Intel 8th generation i5 1.60Ghz processor, 8GB RAM, running Ubuntu 17.10.

### 5.1 Micro benchmark

In the early stages of development we tested our solution on a self-written program that simply iniated two sets of integers and floats and added one million values into each. The program then iterated through each set and computed the sum of these values. The values that were added were deterministic so that each run produced the same sum output.

In the modified version the sets were initialized as `HashSets`. However after running it through our compiler, it suggested that these sets should actually be `LinkedHashSets`. We run both versions of the program 50 times each and calculate the mean execution time and 90% confidence intervals. To take JVM warm-up time into consideration we discard the first 10% of the runs when calculating the mean and confidence interval. For the original version we got a mean time of 2.041 and the 90% confidence interval (1.217, 2.864). For the modified program that was outputted by our compiler we got the mean time of 1.735 and the 90% confidence interval (1.144, 2.325). The modified program had a 15% decreased mean execution time. The confidence intervals do overlap but the worst case execution time of the confidence interval is 19% lower in the modified version.

### 5.2 SPECjvm2008

We used the SPECjvm2008[4] benchmark suite as the real test for evaluating our solution. SPECjvm2008 is a benchmark suite for measuring the performance of a JRE (Java Runtime Environment). It contains several sub-benchmarks focusing on java functionality.

First we analyzed the sub-benchmarks in order to find possible data structures that could be replaced. We ran the benchmark suite with all sub-benchmarks that contained a possible data structure replacement and saved the result. Then the sub-benchmarks which had data structures that could be replaced was used as input to our compiler. The compiler generated new class files which we used to replace the original files in the benchmark suite. We ran the benchmark again for the modified files with identical options as for the original files.

Both the unmodified and modified benchmark suite was ran for 10 times each, each run took around 30 minutes. Each sub-benchmark has a 2 minutes warm-up time and 4 minutes measured runtime, the result is measured in operations per minute. The higher operations per minute the better.

Figure 2 shows, for each sub-benchmark test, the average execution time for the benchmark in milliseconds.

As seen in figure 2 the results from the two versions are almost identical, with a slight favor to the modified version. However the 90% confidence intervals of the two versions was almost equal and therefor this does not prove that the modified version is better. This is most likely because the benchmark suite did not have many data structures that

[3] https://docs.oracle.com/javase/7/docs/api/java/util/EnumSet.html

[4] https://www.spec.org/jvm2008

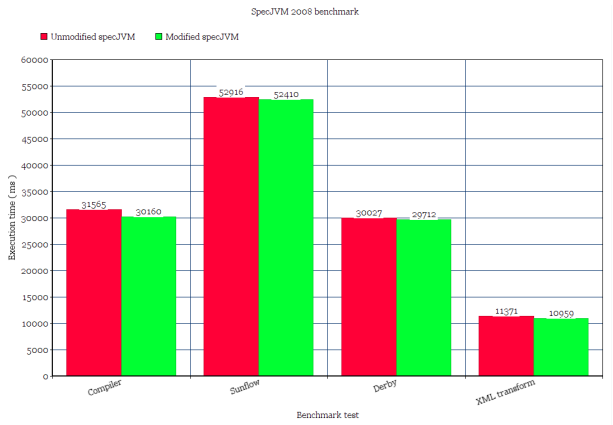


**Figure 2.** Evaluation result

could be replaced by our compiler and therefor our modifications was not enough to affect the end result in a noticeable manor.

### 5.3 JaCoP

We did some evaluation testing on the constraint solver library, JaCoP[5]. The library contains a kind of benchmark suite in the form of many example programs that are included in the library jar file.

Our compiler did produce decreases in execution time of 2% for some of the example programs but the modification was done in a core component of the library, which changed a certain `HashSet` to a `TreeSet`. This modification caused other example programs to crash as the type used in the data structures did not always implement `Comparable`, which is demanded by a `TreeSet`.

## 6 Limitations & future work

Albeit our solution is able to dynamically compare the execution time between implementations of a data structure, and should theoretically choose the optimal choice for the context, it still has alot of improvements that could be done.

Our solution is at this moment limited to `Map, Set` and `List`. As seen in the result section this does limit the result of our compiler in a negative way. In order for our compiler to be more successful, support for more interfaces to inject must be added. We would also like to extend the dependency injection to allow for user specified interfaces and their implementations to be injected.

The static analysis in our solution is quite limited and could be improved in many ways. An idea for future work for the static analysis could be to do more extensive type checking in order to eliminate certain options for the dynamic analysis. E.g. if the type used in the data structure is not `Comparable` then the options of `TreeSet` and `TreeMap` should be eliminated. We would also like to add more annotations, such as `@Synchronized/ThreadSafe` to just tell the compiler to wrap the data structure in a thread safe collection.

An interesting addition to the dynamic analysis would be to add a wrapper class for the data structure in order to count the number each operation is used on the data structure. This information could be used to come up with an replacement instead of running the program several times to get an mean execution time. This could be advantageous if the run time of a single instance is quite long.

Lastly, inspired by Chameleon[7], we could add an aspect of optimizing the memory gains instead of the execution time performance. This type of optimization's is desirable for e.g. IoT[6] machines as they run on low memory hardware.

## 7 Related work

There exists some research project in this area [1][7][8]. First out was Chameleon which is an adaptive technique that focuses on memory usage optimization. It collects runtime statistics and heap-related information from the garbage collector. Chameleon then fed the data into a hand-constructed model to determine if any data structure should be changed. The output is a file with a list of data structure replacement recommendations.

Brainy is similar to Chameleon but instead it focuses on execution time optimization. Instead of a constructed model it uses an offline machine learning model. An important aspect is that Brainy is not restricted to a language with runtime features such as Javas garbage collector. Both Brainy and Chameleon analyses the programs and makes recommendations for the programmer but does not do any actual code replacement. This implies that the programmer needs to do manual changes, which can be a tedious task if there is a lot of replacement suggestions.

In 2013, inspired by Brainy and Chameleon, Xu Guoqing at University of California developed an online adaptive replacement technique for Java collections, called CoCo. This technique uses an own written library to wrap data structures into so called combo structures to be able to dynamically identify optimal choices and perform run-time replacement.

One of CoCo's limitations is that they introduce space overhead with their combo structures aswell as you will need the whole CoCo library and their own compiler to run use the CoCo solution. With our solution the replacement is done at language level which means that it can run on any Java Virtual Machine (JVM), however the replacement is not done at run-time as by CoCo.

[5] jacop.osolpro.com

[6] Internet of Things

## 8 Conclusions

This paper presents a language-level optimization solution using the dependency injection technique. We added a new language extension in Java using the extensible Java compiler ExtendJ, by writing `new Interface` the compiler will replace the interface with the most optimal class that implements that interface.

The current version of our solution does optimize programs, however not as well as we would have though. As discussed in section 7 there are many possible improvements that would make the analysis better and most likely increase the performance gain. From our research we have concluded that data structure selection optimization's are useful and have the potential to increase performance for most types of programs. Because of the flexibility of a language-level based solution compared to solutions that are reliant on third party libraries we believe that it is a good idea to research this area further.

## Acknowledgments

We would like to thank our supervisor Christoph Reichenbach, for providing valuable feedback throughout the project.

## References

[1] Brian P. Railing Nathan Clark Santosh Pande Changhee Jung, Silvius Rus. 2011. Brainy: Effective Selection of Data Structures. 1, 1 (2011), 86–97. https://dl.acm.org/citation.cfm?id=1993509

[2] Torbjörn Ekman and Görel Hedin. 2007. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA '07)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/1297027.1297029

[3] Emma Söderberg Jesper Öqvist, Görel Hedin. 2015. *Build your own Java Code Analysis with ExtendJ [PowerPoint slide 6]*. Retrived from https://bitbucket.org/extendj/extendj/downloads/2015%20SPLASH%20Tutorial%20ExtendJ.pdf.

[4] S.Rus L.Liu. 2009. A Context Sensitive Performance Advisor for C++ Programs. *Proc. of the 2009 International Symposium of Code Generation and Optimization* 1, 1 (2009). https://dl.acm.org/citation.cfm?id=1545076

[5] William McAllister. 2009. *Data Structures and Algorithms using Java*. Jones and Bartlett Publishers.

[6] Philip Wadler & Maurice Naftalin. 2009. *Speed Up the Java Development Process*. O'Reilly Media.

[7] Eran Yahav Ohad Shacham, Martin Vechev. 2009. Chameleon: Adaptive Selection of Collections. 1, 1 (2009), 408–418. https://dl.acm.org/citation.cfm?id=1542522

[8] Guoqing Xu. 2013. CoCo: Sound and Adaptive Replacement of Java Collections. 1, 1 (2013), 1–26. https://link.springer.com/chapter/10.1007/978-3-642-39038-8_1