Dependency-Only Interactive Java Interpreter

by

Michael Thomsen

A Project Report Submitted
in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Supervised by

Dr. James Heliotis

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences Rochester Institute of Technology Rochester, New York

May 2016
Abstract

Dependency-Only Interactive Java Interpreter

Michael Thomsen

Supervising Professor: Dr. James Heliotis

The Java programming language frequently is described as difficult to understand, especially for beginners. This object-oriented language is a staple in many learning environments and universities. However, because of its steep learning curve, it can prove to be frustrating for students.

This project is aimed at creating a well performing Java REPL (Read-Evaluate-Print-Loop) – or interpreter – so that this learning curve is lessened, all while offering unique features for beginners and experts alike. This is done by the concept of ‘dependency-only’ interpretation, namely that on each given iteration, only the current statement and the dependent statements are included during compilation.
# Contents

Abstract ................................................................. ii

1 Introduction .......................................................... 1

2 Related Work ......................................................... 3
   2.1 Dr. Java and Dynamic Java .................................. 3
   2.2 JavaREPL ...................................................... 4
   2.3 Dr. C# ......................................................... 4
   2.4 Java 9 and JShell .............................................. 5

3 Design ................................................................. 6
   3.1 Initial Attempts ................................................ 6
   3.2 Javassist ....................................................... 7
      3.2.1 Complications ........................................... 7
   3.3 Dependency Only .............................................. 8
      3.3.1 Introduction to REPL Phases ......................... 8
      3.3.2 Parsing with ANTLR ................................. 9
      3.3.3 Introduction to ANTLR and Custom Code ........ 9
      3.3.4 Custom Code with ANTLR .......................... 10
      3.3.5 The REPL Phases ..................................... 10
   3.4 Exception Handling .......................................... 17
   3.5 Saving and Loading Workspaces ........................... 19
   3.6 Design Advantages .......................................... 19

4 Implementation ....................................................... 20
   4.1 Modification of the ANTLR Grammar ...................... 20
   4.2 Handling User Input ........................................ 21
      4.2.1 Multi-Line Inputs ................................... 22
   4.3 Dependency Access .......................................... 22
4.3.1 Determining and Finding Identifiers ........................................ 23
4.3.2 Maps ..................................................................................... 23
4.4 Java Actions ............................................................................. 24
  4.4.1 Java Imports ......................................................................... 24
  4.4.2 Java Fields .......................................................................... 25
  4.4.3 Java Methods ....................................................................... 25
  4.4.4 Java Classes and Enums ......................................................... 25
  4.4.5 Java Expressions ................................................................. 26
  4.4.6 Java Dangling Expressions ....................................................... 27
  4.4.7 Java Loops and If Statements ................................................ 28
4.5 The evaluate() Method ............................................................... 28
4.6 Exception Handling ................................................................. 29
4.7 Load, Save, Uncatch – Reserved Methods ................................... 30
  4.7.1 Save Reserved Methods ......................................................... 30
  4.7.2 Load Reserved Method .......................................................... 31
  4.7.3 Uncatch Reserved Method ...................................................... 31
  4.7.4 Unimport Reserved Method ..................................................... 31

5 Conclusions .................................................................................. 33
  5.1 Advantages .............................................................................. 33
    5.1.1 Dependency Only ............................................................... 33
    5.1.2 Feature Set – Saving/Loading Workspaces .......................... 34
    5.1.3 Feature Set – User Defined Exceptions ............................... 34
    5.1.4 Feature Set – User Feedback ............................................... 34
  5.2 Limitations And Future Work .................................................... 35
    5.2.1 Saving and Loading ............................................................ 35
    5.2.2 Exception Handling ........................................................... 35
  5.3 Library Imports ....................................................................... 36
  5.4 Final Conclusion ...................................................................... 37

Bibliography .................................................................................... 38
List of Tables

4.1 Java Actions and the \textit{evaluate} Method ........................................ 29
List of Figures

3.1 Phase 1 – Custom Listeners .................................................. 11
3.2 Phase 2 – Dependency Gathering ............................................. 13
3.3 UML Diagram for Java Actions ............................................... 14
3.4 Phase 4 – Compilation ......................................................... 17
3.5 Overall Flow Diagram of REPL Phases .................................... 18
Chapter 1

Introduction

The object-oriented programming language of Java has long been the choice of universities for introduction to computer science and computing languages. Herbert Schildt, the author of the very popular textbook *Java: The Complete Reference*, points out that the driving force behind Java was not the world wide web but rather that small, cross-platform programs were the future of computing. He continues on to say that “most programmers learn early in their careers that portable programs are as elusive as they are desirable” [7] – thus the prevalence of Java in both the educational and workplace setting. Even though C# and the .NET frameworks are challenging Java for industry and marketplace usage, it is important to see Java is still “at the forefront of computer language design” [7]. Moreover, many universities use Java as a beginning language because it is a “modern language with direct commercial appeal” [5].

The current design specification for most (if not all) Java applications is to write the high-level, object-oriented code first. This code is compiled into byte-code, and then run on the machine through means of a “Java Virtual Machine” (JVM). As Schildt points out, this design does allow for portability and cross-platform usage, but it does also take time to complete the process (significantly sped up in recent years due to computing power becoming better and less expensive), especially iteratively. While this design specification is perfectly functional and meets a lot of developer needs, there is an alternative approach that this project targeted. The approach is that of interpretation – rather than compilation – by use of a REPL (read-evaluate-print-loop).

Part of the justification, rationale, and motivation for creating such an interpreter in this
project is because of the fact that Java is still so heavily used in the computing industry (even the makers boast over 3 billion devices run Java), and its popularity in colleges and universities. Although it is more geared towards student and educational learning, this REPL could easily be used by an expert Java user to avoid the hassle of establishing a full Java class for a simple method or functionality test.

In a compiled or iterative environment, most Java developers will write the code, compile the code, and then test the result once the JVM is spun up and running. In an interpretive environment, the developer inputs one statement at a time, interactively. This statement is immediately compiled and run. The compilation iteration does still happen, but appears seamless to the developer. Again, this approach is advantageous for experts and beginners alike because the code itself is running and compiling almost instantaneously. This drives down the need for iterations of code, build, test.

The ultimate goal of this project is to build an interactive Java interpreter as a way to learn, understand, and possibly experiment with Java code - a different paradigm from the traditional route of iterative compilation. Although similar and related research had been done in this area, the additional goals of this project were to build a unique solution that accomplishes more than existing implementations. Included in these goals were the ability to write the interpreter such that user-defined, custom exception handlers can be written. Additionally, the interpreter is able to save and load a users workspace on-demand in a robust and transparent way all contained within the language itself. Along with these goals, this REPL is built in a way that is incredibly robust and performant – contained within the concept of dependency-only interpretation which will be discussed at much greater length in the design and implementation sections of this paper.
Chapter 2

Related Work

In this section, related work in the area of REPL’s (most importantly Java REPL’s) are discussed and how they influenced and affected the work done in this project.

2.1 Dr. Java and Dynamic Java

Dr. Java is a Java REPL, designed with educational needs in mind, developed at Rice University [8] [4]. Although the project has been somewhat dormant at the time of writing, it still serves as an excellent template in many ways.

This REPL attacks compilation by shipping the product with known and pre-defined compilers for a user to leverage while interpreting. Additionally, the framework allows for user-written compilers to be added in real-time. The overall Dr. Java interpreter has an input editor component to it, and this pairs with the chosen compiler to provide a student with quick and easily visible compiler feedback to written code.

In terms of the REPL implementation itself, Dr. Java leverages an existing, open source framework called DynamicJava. This framework is its own Java interpreter, yet with some key quirks and deficiencies in classloading and return values during the REPL process. The writers of Dr. Java sought to use and augment DynamicJava and were able to do so since both tools were/are written in Java proper through strategic classloading and subclassing.

DynamicJava does full-on parsing of java input during its interpreting. In an examination of the open-source code, each node of the abstract syntax tree is visited by the visitor pattern after the parser has been run.
It is important to note that DynamicJava has been abandoned since 2000 and may not be able to interpret or parse brand new features and structures in newer versions of Java and may not be suitable to utilize in this project.

2.2 JavaREPL

JavaREPL is yet another approach to an interpreted Java environment with a REPL, except tailored for web application integration and offers unique features such as suggested completion for statements by pressing the tab key [2].

Upon examining the source code of JavaREPL, the compilation of user input is done through the use of the built-in compiler API in the JDK. The compilation is done by building a String version of a newly generated class and building in the user-input into that class and sending it to the system compiler. To build this new class, various (and previously input) Java fragments are built into one class – i.e. imports, class signature, evaluations, methods are appended in proper order to the String and then compiled.

The REPL itself makes use of the classes it generates and then has the system compile. Imports, variables, methods, and other statements are stored in an evaluation context and accessed during class generation. This allows the REPL to keep track of previously entered commands from the user so they will be able to be used and accessed later on. A consideration must be made here with this design decision for potential issues of the size of the class. Should the interpreter accept considerable input from a user and the class becomes oversized or bloated, it may lead to significant performance degradation. Great care should be taken to understand this possibility and prevent it if possible.

2.3 Dr. C#

Although not tailored for Java like the previous interpreters, Dr. C# is a REPL for C#, a well known compiled language in the .NET framework and has merit in terms of a template for building any kind of REPL, including Java. In this case, the makers of Dr. C# decided
to build an interactive compiler from scratch after an exhaustive search to utilize and make use of existing tools and technology [6]. No details are included except for the statement that the compilation step was built from scratch.

The REPL for Dr. C# is similarly implemented to Dr. Javas – full scale parsing leads to an abstract syntax tree, traversed by the visitor pattern. However, the value of this paper is not in the parsing and AST details, but rather in a list of specific requirements a REPL must meet to be a full interpreter.

These requirements served as an excellent starting point for this project, to be able to understand what kind of user input should be covered by parsing and interpreting. The author discusses how Dr. C# handles mathematical computation, statements and expressions, binary operators, loops, assignments, and method invocation among many other considerations [6].

2.4 Java 9 and JShell

At the time of writing, version 8 of Java is the most recent stable open-source release available. However, in the spring of 2017 it is expected that Java 9 will be generally available for use. In that release, a built-in interpreter for Java known as JShell will be included. In the OpenJDK documentation [1], it is stated that JShell will be a REPL for use on the command line, as well as an API hook point for developers to work with.

At the time of writing, the Java 9 source is unavailable to the general public, so it is difficult to ascertain what the implementation actually looks like. However, in minimal trials with JShell, there is a capability of saving and loading of a workspace. A save command will result in a text file that is the entirety of the user input for that given session. A load command reads the text file and presumably re-runs all of the commands through the normal process so that JShell is in the same state as of the time of saving.

Although details are somewhat fuzzy for Java 9 and JShell, going forward, it may be a benchmark and standard as far as Java REPLs are concerned when the code is finally released and available to the public.
Chapter 3
Design

In order to accomplish the goals of exception handling and saving/loading of user workspaces, it is necessary that the REPL be built from scratch. Without a concrete REPL, there can be no unique features built. This includes user input, parsing, compilation, and runtime (user feedback). This section is dedicated to understanding and explaining the design decisions made for the end result of this project – including other prototypes that did not pan out due to various reasons that will be discussed.

3.1 Initial Attempts

The initial prototype of this project was to consider the REPL as an “all-encompassing” class – namely that every (syntactically correct) user input was built into one single class so that all references were housed in the same place. This model is employed by the JavaREPL. However, this design does not perform well – in that this “backing” class can easily and quickly become bloated and too large for quick responses from the interpreter. Consider a class that a student may create over the course of a day. It could easily contain hundreds of methods, fields, or even classes. This can be difficult to expect quick response times if it becomes too large. When this was determined to a poor design for this project, another approach – with a library called Javassist – was tried.
3.2 Javassist

Built by the makers of JBoss, Javassist is a tool that allows developers to modify bytecode during runtime [3]. In doing so, the tool was thought to lend itself nicely to developing a Java REPL. The framework allows a user to create classes from scratch, and add new source code directly to a class file. Although built for more 'server-like' applications, this library’s main features overlapped significantly with what a Java interpreter might need.

There were a few attractive benefits of being able to modify bytecode of an already created and successfully compiled class in a REPL environment, especially if the assumption is that the REPL is one all-encompassing class that holds all inputs.

The first benefit was that Javassist would allow the REPL to directly accept string input from the user and easily append onto the overarching class file without needing to do much work. Along with this, the library could determine if user input was a field or a method simply from the input. However, this easy parsing from the input was limited to only that – fields and methods.

The second, going hand in hand with the first, was that some manner of incremental compilation must be built into Javassist in order to be able to modify bytecode and class files – by its very definition. It was thought that this may drastically decrease the amount of compilation work necessary each time a new input is accepted and included. However, as implementation continued, it was determined that this solution did not pan out as expected.

3.2.1 Complications

While Javassist did have those benefits as outlined, there were unfortunate and unexpected consequences along with them.

Classloading within the Javassist library was not as simple as was hoped. The underlying framework has a concept of ’frozen’ classes. Simply put, classes that had already been loaded by the classloader that could not be re-created or re-modified. Unless considerable work to bypass classloading schemes was done, there was no alternative to this feature of Javassist.
Secondly, the Java REPL should accept as many input types as possible – fields, methods, classes, expressions, etc. As previously noted, there is a significant limitation of Javassist in that the built-in, internal parsing from string inputs only can determine fields and methods.

The inability to parse more complex Java constructs paired with classloading complications led the project to steer away from Javassist and take a map-based, ‘dependency only’ approach involving an open source parsing library called ANTLR – as described in better detail in subsection 3.3.2.

### 3.3 Dependency Only

Both Javassist and the “all encompassing class” proved to be ineffective solutions for various reasons. The decided upon approach was similar to that of the “overall” class – but does not have the inefficiency problem of a bloated class. Simply put, this REPL uses a new class for each new input, containing only the compile-time dependencies as a result of parsing the initial statement.

This idea drove the design for the rest of the project – specifically that a mechanism needed to be determined to understand what dependencies exist for a given input, and how they would be included in the class each time. Moreover, a running list of successful inputs needed to be established so that those determined dependencies could be gathered and included at the time of compilation. The next sections will explain the proposed phases of this approach.

### 3.3.1 Introduction to REPL Phases

In order to be successful, this REPL had various phases to achieve the goal of performant Java interpretation with the ability to handle exceptions and workspace saving. These phases were ordered and carried out in the same way each time. However, before describing any of the phases, the use of a parsing library – ANTLR – will be discussed.
3.3.2 Parsing with ANTLR

Before any phases of this REPL begin, it was necessary to know the context of an individual user input. To do this, parsing was necessary. Since one of the goals of this project was not to reinvent the wheel, an open source framework called ANTLR was used to generate and understand the user input.

ANTLR is a tool meant to process and build parse trees as specified by a given grammar that can either be user defined for a specific purpose, or published for a language. In this case, a pre-existing grammar for Java 8 was used to generate parsing code for Java input. However, the published grammar expected that the root node would be a class declaration. For this project, the root node could be many things outside of a class declaration, so the grammar was modified to understand the needs of the REPL. The modified grammar expects a root statement to be a field declaration, a method declaration, a statement expression, a class declaration, or instructions on how to deal with an exception – a standalone “catch” expression.

The important information here is that the unmodified grammar expected that parsing would take place at the file level – that the input to the parsing algorithm would be a “.java” file, rather than potentially unconnected, fragmented Java statements. So the grammar was modified in this way so that parsing is more flexible.

To avoid enumerating the entire grammar to show what could be in all of the grammar, it can safely be assumed that between these nodes, any standard Java 8 user input can be captured and understood with parsing code logic.

3.3.3 Introduction to ANTLR and Custom Code

ANTLR provides the framework to parse input statements based on a grammar. Through a few simple steps and ANTLR tools, Java code (lexer, parser, etc.) can be generated from a given grammar – in this case, the Java 8 grammar. That generated Java code simply needs to be included in a project that wishes to use it, along with the ANTLR runtime library.

That being stated, the generated ANTLR code does not actually do any of the evaluation
of user input – it only provides the infrastructure. It is up to the implementer to design the necessary code around the infrastructure to meet the desired needs of the application. In this case, listeners needed to be created to interact with the ANTLR nodes.

Generally speaking, in the context of building a parser, the term (and pattern) visitor is used. Here, the ANTLR term is “listener” – but it can be assumed that a listener meets the needs of a visitor and the ANTLR framework indeed uses the visitor pattern under the hood.

### 3.3.4 Custom Code with ANTLR

To accomplish the goal of building a performant and correct REPL for Java, the parsing process needed to be as lightweight as possible. To do this, the custom code was limited to only the imperative and necessary places. To have ANTLR code actually run, the following steps had to be taken.

1. Create a Java 8 lexer for the user input string using generated antlr code.
2. Create a Java 8 parser for the lexer using generated antlr code.
3. Attach a listener to the parser and “walk” or “visit” each node with the listener.

For this project, the highest level visitations had code written for them – fields, methods, classes, statement expressions, and catch blocks.

### 3.3.5 The REPL Phases

Here we enumerate and explain (with diagrams) the phases of the REPL.

**Phase 1 – Custom Listeners**

The first phase of the REPL is the custom implementations for each of the high level nodes. In this phase, very minimal information is gathered from the nodes. Because of the dependency only design, the most important role of each listener is to determine the declared
name and the raw input from a single input. This information will allow for the input to be used as a dependency later on during interpretation. Figure 3.1 shows general information on what must be determined from each of these nodes.

![Flowchart](image)

**Figure 3.1: Phase 1 – Custom Listeners**

Once the proper names and input have been stored successfully, the dependency gathering phase begins.
Phase 2 – Dependency Gathering

When the custom implementations for the high-level nodes are being invoked, they have access to the subnodes of the tree at that time. This capability within ANTLR allows the gathering phase to begin. The gathering phase refers to determining what identifiers exist within a given Java statement. Identifiers – in the context of the Java language – are user created (non-keyword) variables, methods, or even types. These identifiers could refer to a previously stored input and may be crucial for the current statement to be compiled correctly.

Consider the following set of statements that are valid to a Java interpreter:

\[
\begin{align*}
\text{int } a &= 4; \\
\text{int } b &= a;
\end{align*}
\]

(3.1)
(3.2)

In this set of statements, equation 3.2 is clearly dependent on 3.1. It is necessary to capture this and be sure the dependency is known, established, and maintained. Similarly, consider the following set of statements:

\[
\begin{align*}
\text{public String myMethod() } \{ \text{return } \text{“String”; } \} \\
\text{String } s &= \text{myMethod();}
\end{align*}
\]

(3.3)
(3.4)

In this set of statements, 3.4 is dependent on 3.3. The invocation of myMethod is an identifier – and must be gathered as a dependency for both successful compiling and runtime evaluation.

To actually gather the dependencies, a recursive loop is used. For each of the main parse nodes, the tree is separately traversed looking for identifiers – terminal nodes in the context of the ANTLR grammar. When a terminal node is found, the name will be stored for later lookup.
Figure 3.2 shows the process for the gathering phase. Once all of the dependencies have been gathered, the next phase begins – the creation of java actions.

Phase 3 – Creation of Java Action

Even though the parser code will know the type of the Java input, in order to implement the REPL successfully, it is imperative to keep track of each of these in a custom way so they can be saved for future use. To do this, the concept of a Java action is introduced. An action is an object representation of each of the high level nodes – field, method, class, statement, and catch.

These objects were designed to be very lightweight, and need only the information
determined in phase 1 and phase 2 – the action name (field name, method name, etc.) and the required dependent actions. Figure 3.3 shows a UML diagram to better explain the class structure around these Java actions.

![UML Diagram for Java Actions](image)

The next phase to be discussed is compiling. Before that, however, it is important to note how identifier names are translated into dependent Java actions. To do this, once compilation is successful, the newly created Java action is stored in a map. There exists a map for field declarations, method declarations, and so on. The key to the map is the identifier name, and the value in the map is the Java action that corresponds to it.

Although phase 2 is crucial to create the Java action, there is a difference between the gathering phase and the actual dependent actions. In phase 2, identifiers will simply gathered and stored, phase 3 will gather the Java actions for those identifiers and establishes them as dependencies. Consider equations 3.5, 3.6, 3.7 as an example to help understand
the difference.

\[
\text{int } c = 5; \quad (3.5)
\]

\[
\text{int } d = 2 + c; \quad (3.6)
\]

\[
\text{int returnVal()} \{ \text{return } 3 + d; \} \quad (3.7)
\]

In 3.5, a JavaField will be created, storing \( c \) as the name, \( \text{int } c = 5 \) as the raw input, with no dependencies. This field will be stored in the map of fields to later be accessed. In 3.6, the gathering phase will determine that \( c \) is an identifier. In phase 3, \( c \) will be looked up in the field map, found, and saved as a dependency. Finally, in 3.7, a JavaMethod will be created, storing \( \text{returnVal} \) as the name, and will determine that \( d \) is a dependency and subsequently \( c \). All of the information in any given Java action is utilized in the next phase – compiling.

**Phase 4 – Compilation**

Before an interpreter statement can be evaluated, it clearly must adhere to both syntax standards and correct compilation rules. To do this, two things were needed – the current statement and the dependent actions. These are provided from the Java action resulting from phase 3. With any given Java action, a new class can be generated for compilation.

To construct this class, a shell is used. Within this shell class (defined as a string), each of the dependent actions and the current input are enumerated by accessing the raw input they were created with. Note that if there are nested dependencies, the class must have those enumerated as well to avoid compilation failures of an otherwise valid class.

Once this shell class was built up with the current statement and the dependent actions, the system compiler API validates the input. This compiler API allows the caller to compile a Java class as a string and return real-time results.

As previously discussed, here, if compilation is successful, the action is stored in the proper map to possibly be a dependent action of a future input. If the compilation fails, the input is not stored in any capacity, and the interpreter waits for the next input.
An important design decision was that no type checking is done with the dependent actions in the identifier-gathering phase. The lookup for dependencies is simply based on the name of the identifier that is being referenced. Consider 3.3 and 3.4 again.

During the scanning phase of the declaration of the field in 3.4, the method `newMethod` would be found and included in the new class to be compiled – all perfectly valid Java syntax. However, during the compilation phase, there would be a compilation failure due to mismatching types. In this case, the user would be notified and 3.4 would not be saved. Figure 3.4 is a flow diagram to help understand how the class is generated as a string and then compiled.

**Phase 5 – Evaluation**

The last phase of a given user input is the evaluation of that input. Whether a computation, assignment, or an acknowledgment of the creation of a method, it is important for the user to be notified of proper acceptance of the input. There are two different cases for evaluation of input – an actual “value” result, and an acknowledgment of a creation of something new. In either case, a single `evaluate()` method will be called on the newly compiled class for feedback to the user.

A value result is the result of an expression, variable creation, method invocation, and so on. This is an actual evaluation done by the Java code. The `evaluate()` method in this case will return the value result of the user input to the console.

For method declarations, class declarations, and other non-value return statements (like a catch block), the `evaluate()` method will simply accept the user input and confirm that it successfully compiled and was saved, to be used later if necessary.

**Overall Flow Diagram**

Figure 3.5 shows an overall flow diagram for the REPL and all of its phases.
3.4 Exception Handling

This interpreter has the goal of being able to design and understand exception handling as defined by a given user. This feature will be included by allowing the user to type in a “catch” block – in standard Java syntax. All catch declarations will be stored and enumerated.

A necessary feature of the REPL and the runtime user feedback is that the code is wrapped in the `evaluate` method (explained later in the implementation section). This
**evaluate** method is where an exception *could* be encountered and thus needs to be handled.

These lines of code will be appended in a try/catch block within the code that will be executed so that if the given exception is encountered, the user defined code will run as it was specified. These user input statements are subject to the same compilation standards the same as any other user input during “regular” interpreter processing.
3.5 Saving and Loading Workspaces

Similar to exception handling, the ability to save/load workspaces will be done with standard Java syntax. To do this, save() and load() methods will become reserved and the user will not be able to override them.

When the save is invoked, the stored maps will be serialized to a directory specified by the name given by the user. In a similar fashion, when a load is invoked, the previously serialized files will be deserialized back into maps from a given directory as specified by the user – ready for use.

3.6 Design Advantages

The advantages of this dependency only design is that the REPL will only compile what is needed. Where previous approaches had the potential for large or bloated classes (leading to poor performance) when all inputs were stored together, this approach needs only the smallest amount of code to operate correctly. Performance is excellent during user operation.

Additionally, this interpreter has the advantage of being very lightweight. Because of the ability to only compile what's necessary through a simple lookup on a given input, only a few classes were necessary to understand/parsing user input. This leads to a small amount of source code that is both very maintainable and easily modified to accept any changes.
Chapter 4

Implementation

After design specifications were completed, the implementation of the REPL and its features began. The implementation was written and runs in Java (version 8), with dependencies only on the ANTLR runtime libraries. The user interacts with the REPL by virtue of the command line – a statement is entered one at a time (through single or spanning multiple for method or class declarations) and sent through the REPL phases immediately after a complete statement. Single line statements are complete with a carriage return, while multiple line statements are considered complete when matching open/close symbols are encountered. The specific implementation choices are discussed next, including modification of the ANTLR grammar, how user input was accepted and passed to the parser, how dependencies were accessed, different Java actions were handled, and how the `evaluation()` method was constructed on each interpreted statement. Note that in this section, the meaning of the phrase “compiled class” is in reference to the class that is generated as a result of the current statement and all of its dependencies in raw input form.

4.1 Modification of the ANTLR Grammar

ANTLR, being open source, contains a lot of code for any developer to look at. In addition to the source code, grammars for various languages are also published as open source. For this project, a Java 8 grammar was needed as it was the most major version to be released at the time of implementation. As discussed in the design phase, this grammar expected a fully formed Java file as its input and thus needed to be modified to allow fragmented Java
Before explaining the modification of the grammar, it must be noted that there are two phases of using ANTLR. The first is to use the ANTLR framework ahead of time – using the grammar. Using ANTLR framework tools, the grammar is input and corresponding Java objects are produced. The second is to incorporate those generated objects from the first phase *in code* and utilize them to parse user inputs by means of the ANTLR runtime libraries.

All ANTLR grammars start with a single root object regardless of language – *compilationUnit*. The original, unmodified version of the Java 8 grammar looked as follows.

\[
\text{compilationUnit} : \text{packageDeclaration}? \text{importDeclaration} \ast \text{typeDeclaration} \ast \text{EOF}; \quad (4.1)
\]

To change this, it was necessary to understand the expected inputs and how they correlate to the Java 8 grammar as a whole. The *compilationUnit* grammar after modification resulted as shown in equation 4.2.

\[
\text{compilationUnit} : \text{classBodyDeclaration} | \text{blockStatements} | \text{importDeclaration} | \text{catchClause}; \quad (4.2)
\]

In 4.2, the pipe character represents or in the logical sense – namely that any of the root objects in the *compilationUnit* definition are the start of valid input to the REPL. Note that a *blockStatement* definition contains many of the possible fragmented Java statements. There was no need to enumerate the grammar any narrower in this case, since that level of granularity was handled in the “expression” listener as described in chapter 3.

### 4.2 Handling User Input

The resulting string from a given input is lexically analyzed, parsed (in tandem with the ANTLR runtime libraries), and transformed into the proper Java action for later runtime. In some cases, it is necessary to have user input span multiple lines – this is a case that needed to be handled in a special way.
4.2.1 Multi-Line Inputs

For method, class, or enum declarations, it cannot be assumed that a user would input this in a single line. The standard for most Java code is to write in multiple lines for more complex statements, so this REPL needed to be able to conform to that standard – this was done by use of a stack.

The stack (first-in, first-out data structure) for REPL input was simple – it kept track of parentheses and braces, both open and closed. Whenever an open brace or parenthesis on the first line of input, the stack began to keep track of input across multiple carriage returns.

From that point, When a open brace or parenthesis was encountered, it was pushed onto the stack. When a close brace or parenthesis was encountered, a peek was done to ensure that the valid open/closed sequence was input (i.e. an opening brace and close parenthesis would be invalid). If so, the corresponding open brace or parenthesis was popped off of the stack.

If an incorrect sequence of closing and opening characters were encountered, the interpreter does not accept the input as valid, displays the error the user, and continues on waiting for the next round of input. If the sequence is valid, the multiple lines are converted into one single string, and parsing begins as normal. Note that single and double quotes were not handled by the stack, as the user would need to correct the code that resulted in unbalanced quotes in any way.

4.3 Dependency Access

In the description of phase 2 of the REPL design, dependency gathering was discussed. This technique and feature is the heart of the interpreter and this section is dedicated to explaining the implementation details of how the phase is carried out.
4.3.1 Determining and Finding Identifiers

When the current statement is being evaluated through ANTLR and custom listeners, identifiers must be discovered for potential dependencies. This is carried out through means of reflection and a feature of the ANTLR framework.

While parsing a current statement in the REPL, there can be multiple nodes and subtrees. In these nodes and subtrees is where identifiers may live and must be extracted. To do this, a feature of the ANTLR grammar is used. For a given parse tree, many of the nodes can resolve down to an identifier. ANTLR specifies these identifiers with a single method on many different nodes – called Identifier.

For every “root” node, the subtrees (the ANTLR generated objects) are recursively searched for the Identifier method, using reflection. If the given object has an Identifier method on it, the text value is gathered and put into a list of possible identifiers for extraction out of the next discussion topic – the maps.

4.3.2 Maps

Once all of the identifiers are determined from the current statement by means of the Identifier method, those identifiers are queried against a series of maps. Each map is separated out by its given Java action (discussed at greater length in the next section), and contains a key of the action name (method, field, or class name, etc.). The action name maps to the corresponding Java action for that name.

Each identifier found will translate into zero to multiple dependent Java action objects. For example, consider the following inputs to the interpreter.

\[
\text{int someIdentifier} = 5; \quad (4.3)
\]

\[
\text{public String someIndentifier()} \{ \text{return ”Value” } \}; \quad (4.4)
\]

\[
\text{int newIdentifier} = \text{someIdentifier}; \quad (4.5)
\]

Both 4.3 and 4.4 are perfectly valid Java syntax even though they share the same name.
Should the next input to the interpreter be 4.5, the dependency gathering phase will determine a **field** and a **method** as both the field map and the method map will contain an entry with the name “someIdentifier”.

Moreover, as stated, the resulting class would contain a reference to both 4.3 and 4.4. However, only 4.3 would be used in actual compilation and runtime.

This approach can lead to extraneous inputs to a given class being compiled, but the overwhelming majority of cases result in negligible performance degradation, if any at all. The dependency only feature even in this “extraneous” case is still far better performing than a class that contains and tracks all inputs for each statement encountered.

The REPL tracks and has maps for the following Java actions: field, method, class, enum, and expression. The Java actions and their implementations are discussed further in the next section.

### 4.4 Java Actions

This section is intended to describe the means by which each type of Java action is created and stored in the maps. Refer to figure 3.3 for the class (UML) diagram for this class and its subclasses. Note that each of the Java actions is created and stored with *its* specific dependencies.

#### 4.4.1 Java Imports

A Java import is simply a statement in which a user wishes to include a class to be used in subsequent REPL actions. To accomplish this, the ANTLR grammar was modified to allow an *importStatement* – a string prefixed by “import”, followed by the package or class that should be imported and used during subsequent compilations.

This import Java action stores only the raw input from the user and is kept in a separate list. The objects in this list are enumerated in the “import” section of each iterative compile step.
4.4.2 Java Fields

In the context of this REPL, a Java field refers to a class-level variable only. While it could be expected that a field could be declared within the scope of a method, this REPL assumes that all field declarations are global and are included in the iterative compile step in the “field declaration” section of the compiled class.

The field Java action contains the field name, and the raw input it was declared with. These are stored in a map, keyed by the field name, mapping to the action itself.

4.4.3 Java Methods

A Java method is fairly straightforward – a unit of work defined by the user that is repeatable. Due to the dependency-only feature of this REPL and the two phase (parse and then compile) approach, the only important metadata off of a method creation is the name of the method itself. When a method is later referenced (after successful declaration), it is included (as raw input) at the class level, in the same scope as the fields.

The method Java action contains the method name, and the raw method body it was declared with. These are stored in a map, which is keyed by the method name and maps directly to the Java action method object itself for easy retrieval.

4.4.4 Java Classes and Enums

In the context of a “typical” Java development environment, Java classes more often that not are declared within separate .java files with the same name as the class itself. Enums are also “typically” declared within separate files for better code separation and maintainability. However, Java syntax rules allow for classes and enums to be declared within another class – a nested reference. This REPL takes advantage of that feature when handling classes and enums.

Similar to method processing, a class or enum is written by the user and accepted by the REPL. Because of the dependency-only architecture, the class or enum name is the only
important data of this declaration. If the newly created class or enum is referenced in a later statement, it is included (as raw input) in the class as a nested class/enum declaration.

The class Java action contains only the class name, and the raw class body. The enum Java action similarly only contains the enum name. Both are stored in separate maps, keyed by the class/enum name, pointing to the Java action object itself.

### 4.4.5 Java Expressions

Java expressions are by far the most complicated and far reaching set of statements within the Java 8 grammar. They can range from a simple assignment, to the most complex of lambda expressions. This REPL was designed to handle all of these.

Within the context of a Java expression, there is always a variable being changed, modified, or invoked. When collecting information through various ANTLR parse nodes, the expression variable is always the target piece of information. Once that expression variable was found, a Java expression action could be created and used.

Once the variable was extracted, and the expression action was created (containing the raw input of the expression), it was stored in a map for later use. An important design feature was that this map was that the string variable maps to a linked list of Java expressions. This list of expressions is enumerated in the `evaluate` method (which will be described later) so that all modifications to this variable are kept track of.

Note that within the map, using a list ensures that a given variable can be acted upon as many times as necessary, and the nature of a linked list is that order is maintained. For example, consider equations 4.6, 4.7, 4.8 and assume a field declaration exists to declare this variable already as an integer.

\[
x = 5; \quad (4.6) \\
x = 57; \quad (4.7) \\
System.out.println(x); \quad (4.8)
\]
In this group of statements, the user would expect that 4.8 would print out 57, since that is the last assignment of the REPL. To be sure that this is the order in which the variable is modified, the linked list is enumerated in the same way so that 57 is shown to the user. While this may result in unnecessary code (the argument could be made that storing and running 4.6 is not needed), it is necessary in some cases. Consider 4.9, 4.10, 4.11 as a previously created list of integers.

\[
\text{intList.add}(5); \quad (4.9)
\]
\[
\text{intList.add}(6); \quad (4.10)
\]
\[
\text{System.out.println(intList);} \quad (4.11)
\]

If only the latest (4.10) modification expression was kept, the addition of 5 to the list would be lost and not printed when the list was.

It is important to note that Java expressions as just described – while complicated in and of themselves – do not enumerate all possibilities of user input. A “dangling” expression can also be input, which are handled very similarly, with a slight difference.

### 4.4.6 Java Dangling Expressions

A dangling expression in the context of this REPL means an expression which does not necessarily have any “need to be saved” actions. For example, a simple method call, or just an inquiry into the value of a given field are considered dangling expressions. In both cases, neither the method nor the field are modified. The method is called (the value, if any, is returned), and the field value is returned to the user.

To capture this type of action within the REPL, an addition to the ANTLR grammar was created that a basic `blockStatement` can be input at any time and in any place. This allows for simple statements like \( x \); to be input to the interpreter and the value to be returned.

These dangling expressions live within the scope of the current iteration and current compiled class. They are not stored (as they have no bearing on future compilations), and
are not used after the iteration is complete.

### 4.4.7 Java Loops and If Statements

A similar case to a dangling expression are Java loops and if statements. These are perfectly valid inputs to the interpreter and allow a user to experiment and understand how loops and if statements are constructed – on the fly. Like a dangling expression, however, they have no real bearing on future iterations and are thrown away after the iteration has been completed.

The REPL captures an if statement or a loop based on the ANTLR grammar. By virtue of adding `blockStatement` to the grammar, if statements and loops can be referenced outside of a method body – where they normally reside. Loops and if statements follow the pattern of a normal input, and just gather dependencies to be sure the execution completes successfully if a dependency is referenced.

As discussed in the details of multi-line inputs, Should a loop or an if statement span multiple lines, it is expected that it has been terminated when a combination of start and end braces has been encountered successfully.

The full body of a loop or an if statement is put into the `evaluation` method (discussed in the next section) and run when the `evaluation` method is called.

### 4.5 The `evaluate()` Method

An important part of any REPL is the feedback that returns to the user. It is important that the user is aware of the fact that the interpreter understood the input and successfully processed (or didn’t process) it. This is especially important if the REPL is geared towards students learning how to code with Java for the first time.

This REPL makes use of inheritance within the “compiled class” to make this feedback happen. When constructing the class to be compiled, a base class is used, `InterpreterSuperClass`. This class has an abstract method, called `evaluate`, which returns an object. By virtue of inheritance, every subclass must implement this method.
When the raw input (as a string) is put together from the current statement and all of the necessary dependencies, it is put into a class that extends InterpreterSuperClass, called CompileClass. This means that CompileClass must implement `evaluate`, which it does differently each iteration. Consider table 4.1 for each of the values that are output from the `evaluate` method in each case of a Java action.

### 4.6 Exception Handling

Aside from the architectural decision to be “dependency only”, this REPL also design goals in mind for additional features. One of these features is that of user-defined exception handling. To do this, as mentioned, the ANTLR grammar was modified so that a `catch` block is perfectly valid input to the REPL, at any given time.

Once a `catch` block is encountered, the caught exception name is determined through parsing. The exception and the exception body is stored in a map, the key being the name, and the body being the value.

Each time a new input is accepted, the exceptions are enumerated in separate catch blocks of an overall `try – catch – finally` block. To be able to handle exceptions, the `evaluate` method always contains a `try – finally` block, where the `finally` block is empty, and the `try` block contains the code necessary to perform actions and return values as
mentioned in table 4.1. In the event when user-defined exceptions have been created, the block becomes a \texttt{try -- catch -- finally}, and the catch blocks are enumerated as the user defines.

When the behavior of a given exception needs to be overwritten, a \texttt{catch} block with the same defined exception can be input to overwrite the previously stored one. The map will be updated with the new code and the old code will be thrown away.

### 4.7 Load, Save, Uncatch – Reserved Methods

Another feature of this REPL is the ability to save and load workspaces on demand. This can be useful to allow a user to save current work and continue at a later date. Additionally, a user may determine that a previously caught exception should no longer be caught – i.e. uncaught. To accomplish both saving/loading and uncatching, certain methods were deemed to be “reserved” in this REPL. To be reserved means that the REPL itself will intercept calls to these methods and not allow them through to the compilation step. Should a user create a method with a reserved method name, it will not actually be called when invoked later on – the system-defined reserved method will be instead.

#### 4.7.1 Save Reserved Methods

To save a given workspace, a user has two method options – either invoke \texttt{save()}, or invoke \texttt{save(saveName)}. In the former, the save will be done and saved in a folder with the current date and time. In the latter, the save will be saved in a folder of the user’s choosing. A feature of the latter is that a user can use a previously defined string variable within the REPL context to choose a save name. Consider this case in equations 4.12 and 4.13. Once completed, a save will be located in a folder called “SaveBackup”.

\begin{align*}
\text{String } \textit{newSave} & \ = \ “\textit{SaveBackup}”; \\
\text{save(newSave)}; & \tag{4.12} \\
\end{align*}
To actually complete the save, each of the maps and lists that have been created up to the save point are serialized to separate binary files. This makes use of the object output stream methods within the Java JDK. The opposite (de-serialize) operation is done during the load operation.

4.7.2 Load Reserved Method

Similar to save, the load reserved method has the capability of accepting no parameters, or one string parameter which is the file to load. If the no-parameter version is called, then a list of saves will be enumerated, and prompt the user to pick one through a number assigned to each previous save. If the one parameter version is called, the system will try and find the save with that name, and either load it, or alert the user that no save exists for that given name.

If the load method is invoked successfully, the operation uses the serialized files from the save operation, and uses object input streams to de-serialize the maps and lists and assigning them to the “current” or in-memory maps to be used from that point forward. This means that the load operation will overwrite the workspace values entirely and replace them with the saved values.

4.7.3 Uncatch Reserved Method

In some cases, a user may determine that a given custom exception may have been incorrect (syntactically or otherwise) and will need to remove it to avoid a compilation error each time. To do this, the reserved method uncatch is used. To uncatch an exception, the user needs to invoke the `uncatch(String exceptionName)` method. The interpreter will determine what the exception name is, and remove it from the list of user-defined exceptions.

4.7.4 Unimport Reserved Method

Similarly, a user may create a syntactically incorrect import statement, or attempt to import a class or package that does not exist. Similar to exceptions, there must be a way to undo
the invalid import rather than display an error back to the user each time. Like *uncatch*, *unimport* exists and can be invoked with the import to remove – *unimport(String import)*.
Chapter 5

Conclusions

This REPL is implemented in a way that is both performant and user-friendly. It allows any developer – from beginner to expert – to experiment with the Java programming language in a unique, and rapid environment. There are advantages and limitations to any software, and this REPL is no exception to that case.

5.1 Advantages

This REPL has many advantages – both in its architecture and its feature set.

5.1.1 Dependency Only

There are many advantages of the “dependency-only” model of a REPL. The first is performance – the idea that only what is necessary is included in a given iteration of compilation. No extraneous, irrelevant input is included. This allows the REPL to provide rapid feedback to the user both in terms of successful inputs and failures during compilation or otherwise.

Where previous attempts at a Java REPL use a similar approach of taking a string value and converting to a class file on the fly – this REPL is superior in that the class file generated has a significantly smaller footprint. This allowed for easier debugging both in the development phase of the REPL itself, but also allows for a narrowed set of potential failures from a user standpoint.
5.1.2 Feature Set – Saving/Loading Workspaces

Being able to save and load workspaces is another significant advantage of this REPL over those previously developed. This allows for work to be saved as a user inputs it, so that it is not all lost should the user navigate away or accidentally close the interpreter.

It is worth noting again that with the advent of JShell and Java9 [1], a save and load operation will be included for a user to take advantage of. However, there may be limitations of that model as it re-plays all of the user’s previous input through the REPL, which may be time consuming. This REPL has a tailored, simple approach to saving and loading which allows prompt restoration of a previously saved state.

5.1.3 Feature Set – User Defined Exceptions

A feature that is not implemented by any current Java REPL models is that of exception handling. This allows a user to determine what needs to be done in the case of certain exceptions, and can be very useful for a beginning programmer to help understand how exceptions work.

Moreover, where a REPL without exception handling may crash and burn when encountering an exception, the choice is given to the user to recover from an exception (or not) based on how the exception body is declared. This flexibility is a very important and unique feature of this REPL.

5.1.4 Feature Set – User Feedback

Although not unique to this interpreter, feedback to the user is another important feature. This allows the user to see real-time what the result of the statement was. Whether a declaration, or a computation, the user can see how the REPL understood and processed the statement. This is especially important if the REPL is geared towards less experienced, or beginning Java programmers.
5.2 Limitations And Future Work

Although this REPL has a lot of excellent features and well performing, it is not without limitations and future enhancements.

5.2.1 Saving and Loading

While saving and loading is (mainly) unique to this project, the design/implementation decision of using serialization does lead to certain pitfalls. If a saved workspace (and serialized files) exist, they can be invalidated if any of the source code of the REPL is changed. For example, if a method is added to say, the JavaField class, it will invalidate any saved workspace up until that point because the class signature has changed from the time it was saved.

While this is somewhat of an edge case, it is possible in an environment where the REPL is used in a centrally hosted place. If updates are rolled out to the source code in that environment, all previously saved workspaces for a given user could become corrupt and unusable – which is clearly undesirable.

For future work, this could be fixed by means of a different save algorithm – potentially at the cost of performance and easy code maintenance. This REPL could follow a similar pattern to that of JShell [1], where each individual input is again run through the interpreter one at a time. Although slow in performance, it avoids the problem of invalidating serialized files due to a source code file change.

5.2.2 Exception Handling

When considering exception handling, a Java developer must take great care to be sure that exception hierarchies are adhered to, and that caught exceptions are actually thrown from the given set of code in the try block.

Unfortunately, in this REPL, this case is not handled and is left entirely up to the user.
Because of the flexibility of handling exceptions, the user must take precaution when defining exceptions and especially the order in which they are defined. Since the list of exceptions is a linked list, the exceptions will be enumerated in the order they are input into the interpreter. To illustrate this, consider the following catch blocks in equation 5.1 and 5.2, input in that order.

\[
\text{catch(NullPointerException e)} \ldots \tag{5.1}
\]

\[
\text{catch(Exception e)} \ldots \tag{5.2}
\]

If 5.1 is truly input before 5.2, the try − catch − finally block will compile correctly. However, if the inputs were switched, it would not. Since NullPointerException extends Exception, the compiler would indicate that NullPointerException was already handled by means of catching Exception. This case is not handled by the REPL as-is, but the reserved uncatch method can be useful in the case where the exceptions are not handled in the right order.

This could be enhanced in future work by interrogating each exception as the user inputs it. Although tedious, the REPL could investigate and look up the class as it is input, to determine its exception inheritance. In doing that, the list could be sorted by the place the caught exception appears in the hierarchy to avoid the previously shown compile error.

### 5.3 Library Imports

One additional feature that may be useful in the future is the ability to import a library into the interpreter. Many times, a Java developer will find a library (as a .jar file) and wish to include it in a project. In a lot of these cases, the developer may just wish to experiment with the library, rather than immediately use it. Having the REPL be able to handle this case would be very useful for a more experienced Java developer. The compiler API has a feature where the REPL could add additional classpath, and since imports are already handled as-is, this may be fairly simple enough to include as an enhancement.
5.4 Final Conclusion

As a final conclusion, this project created a REPL that is very useful and flexible for experienced and beginner Java programmers alike. The unique feature of being “dependency-only” makes this interpreter stand out above the rest in terms of performance while maintaining correctness of interpretation. Although .NET may be gaining on Java in industry, there will be a place for Java in both academia and the marketplace for the foreseeable future. This project built a well-performing, robust, and very extendable Java REPL that can be useful to both for years to come.
Bibliography


