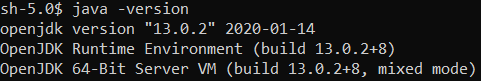
# Getting Started

## Installing Java

1. Download and run the latest Java JDK 13 installer at the following [link](https://www.oracle.com/java/technologies/javase-jdk13-downloads.html) (as of 3/26/2020)
2. Open your system’s console application (Terminal, Command Prompt, etc.) and run the following and confirm your version is “13.x.x”:



1. If you don’t get that output, look up how to add Java to your PATH environment variable in your specific operating system

## Installing LAPS

1. Make a directory called “LAPS” (your LAPS directory) where you install other programs
   1. For Windows, that is “C:\Program Files”
   2. For Mac OS X, that is “/Applications”
   3. For Linux, that is “/opt”
2. Download and extract LAPS.zip (TODO insert link)
3. Just like you did for the Java JDK, add this directory to your PATH environment variable
4. For Mac OS X and other Unix-like systems run “chmod a+x laps”

# Your First Language Made With LAPS

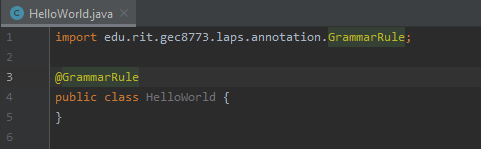
## Things to Consider

* LAPS uses Java Annotations to under what your code means, but if LAPS can’t see your code it can’t determine what your code means. TL; DR: any code you expect LAPS to see make it public.
* LAPS does NOT construct an Abstract Syntax Tree (AST) for you. You must save all the data you want to save in your constructors in your code.
* All the classes you write are yours. You can implement any interface and extend any class. Just code like you normally would in Java. Use your favorite IDE.
* The LAPS script expects the current working directory when called to be in the default package of your Java .class files (not your Java .java source code)
* All your compiled classes should be compiled with the compile flag “-parameter” to save the names of parameters in your methods and constructors

## Creating a New Project

1. Make a new empty Java project in your IDE of choice
2. Add the LAPS.jar from your LAPS directory as a library in your project
3. Add the “-parameters” compilation flag to your build configuration
4. Add a run configuration which to run the LAPS script which should be in your PATH
   1. Arguments for the script should be “-c <fully-qualified-class-name>”

## Getting Down to Business

1. Let’s start out with root node in your AST
   1. Create a new public class and call it “HelloWorld”
   2. Annotate the class with @GrammarRule to indicate this class is a grammar rule when parsing
   3. At this point you should have the following:
2. To run execute your language you can run “laps -c HelloWorld”
   1. On Windows, the LAPS command is a batch file
   2. On Unix-like systems, the LAPS command is a bash script
   3. At this point, LAPS will do nothing with exit code 0

## Adding Tokens

1. Let’s add 2 tokens to start
2. In your class (HelloWorld), add 2 “public static final String” fields
   1. These fields will hold Java regular expression String values
   2. The first field can be HELLO = “hello”
   3. And the second can be WORLD = “world”
3. Once you have your fields, annotated them with @Token

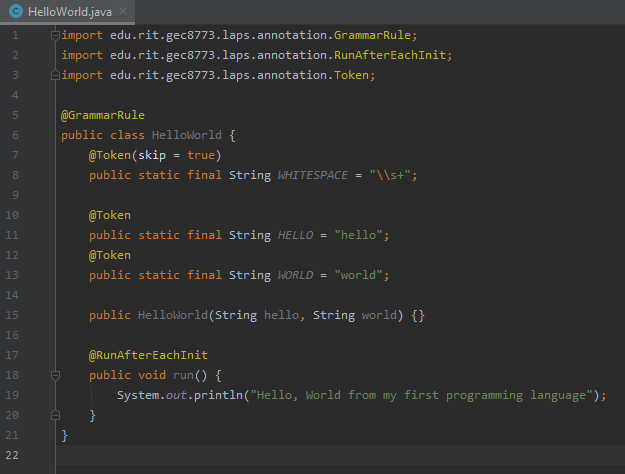
## Defining a Grammar Rule

1. In LAPS, public constructors give definitions for grammar rules
2. So, let’s make a new constructor to accept the tokens HELLO then WORLD
   1. To do this, the parameters of the constructor should be of type String which will store the value of the accepted token from the input
   2. It is very important that the names of these parameters align with the defined fields (case-insensitive); this is how LAPS determines which token to accept in a grammar rule
   3. Code update:

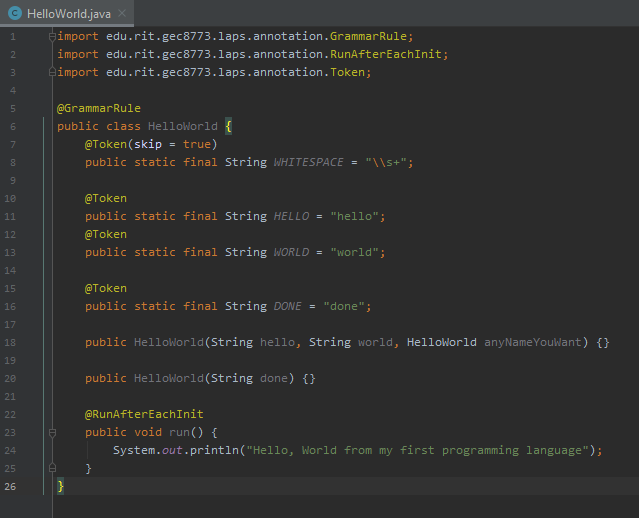


1. At this point, you have a language which accepts the input string “helloworld” and that’s it
   1. I know kind of useless, right?

## Making a More Useful Language

1. Something that always comes in handy in language development is the ability to skip whitespace
   1. So, let’s add a new token to accomplish that
   2. To define a skip tokens, instead of annotating your fields with @Token, annotate your fields with @Token(skip=true)
   3. Now to skip whitespace, annotate the field with @Token(skip=true), name it WHITESPACE (but LAPS doesn’t actually care about the name), and set its value to “\\s+” (the Java regular expression for 1 or more characters of whitespace)
   4. Great your language now skips whitespace
2. Let’s define some semantics for your language to make it finally do something
   1. LAPS has 3 annotations for methods to help define semantics
      1. @RunBeforeEachInit selects static methods to run before every constructor call
      2. @RunBeforeFirstInit selects static methods to run before the first constructor call
      3. @RunAfterEachInit selects instance methods to run after every constructor call
      4. Note: there is no guarantee for the order in which the selected methods run by the same annotation
   2. Define a new instance method in your class and annotate it with @RunAfterEachInit and print out “Hello, World from my first programming language” in the method
3. Code Check:

## Making a List Style Grammar Rule

1. Let’s start out with a naïve approach by adding the HelloWorld type as a parameter to our constructor and nothing else
   1. After a bit of testing, you will notice that the language does accept “hello world” however many times you add it with however many white-space characters in between (including none) as input to your language, but it never terminates unless you put in an unknown or unexpected token resulting in a failed parse
2. To fix this, let’s try to add a new empty grammar rule in the same class
   1. To do this, we must add a constructor without parameters
   2. Now give this language a test
3. You will notice the language still doesn’t terminate unless you put in an unknown or unexpected token resulting in a successful parse. What? Why? What changed?
   1. Note: This is just an edge case for the root node in the AST. In most other cases, this works
4. Before we explain this successful parse, let’s try to understand what LAPS is doing
   1. When parsing, LAPS goes through each constructor of a class containing grammar rules and tries to parse that rule based on the selected constructor
   2. If there is a constructor with no parameters, it is always parsed last
   3. If the parsing fails on a particular constructor, it goes to the next constructor
   4. If parsing fails on all the constructors in a class, it returns a failed state to the grammar rule which wanted a grammar rule from the class
5. The explanation
   1. LAPS accepts “hello world” an indefinite number of times
   2. Then when it doesn’t see a HELLO token it tries the accept nothing which always succeeds, and leaves the rest of the input unread
6. The best way to handle a list as a root node in your AST
   1. Add another token DONE = “done”
   2. And Instead of having a parameterless constructor have the constructor accept a DONE token
7. Final Product: