Java REPL

This project teaches you about dynamic compilation, class loaders, and simple lexical analysis (matching nested structures). The project demonstrates that Java can be extremely dynamic. Servers use this technology all the time to reload classes without having to restart the server program. We'll build a read-eval-print loop (REPL) interface for Java code similar to Python's interactive shell.

```
> int i = 3;
> System.out.println(i+4);
7
> print i*2; // we have a print command too :)
6
>
```

Goal

Your goal is to create an interactive "shell" tool that mimics the behavior of the Python interpreter shell. In other words you should be able to type the following, hit return, and see the result immediately:

```
$ java cs652.repl.JavaREPL
> System.out.println("Hi");
Hi
^D
$
```

where ^D is "control-D". That sends end-of-file to standard input on UNIX. It's ^Z on Windows. Assumes your repl-1.0.jar is in the `CLASSPATH` .

You should print a line with a "> " prompt for the user.

One of the tricky bits is figuring out when the user has finished typing a complete declaration or statement. For example, we would not want to have to type complete functions all on a single line. You'll notice that Python tracks the nesting level of parentheses, brackets, etc. and executes whenever it sees a newline and it is not nested or comment. Your should behave in a similar fashion:

```
> // do nothing
> void f() {
    System.out.println("hi");
}
> f();
hi
> print (
3
+
4
);
7
>
```

You do not have to worry about `/*...*/` comments, just the line comments that start with `//`.

You must allow variable definitions, function definitions, and class definitions as well as executable statements:

```
> class T {
    int y;
    void f() {
        System.out.println("T:f");
    }
}
> T t = new T();
> t.f();
T:f
>
```

Pretty cool, eh?!

Assume that the code entered by the user is restricted to use classes in:

```
import java.io.*;
import java.util.*;
```

Before you freak out that you have to build a Java interpreter, note that my solution is less than 200 lines of code for the core functionality. Then, I have 100 lines of code for the scanner that figures out when the user has completed a statement or function or whatever.

For simplicity, I will use the term *line* to mean whatever complete declaration or statement the user types in from here on.

How to build an interactive Java shell

Here's the trick you need to make this work without having to do much. When the user enters a statement or declaration (*line*), generate a class definition with that line and then compile it using Java's built-in compiler API; see packages `javax.tools.*`, `com.sun.source.util.JavacTask`. Once you compile the class just conjured up, use the `ClassLoader` to bring back code into memory and execute it.

Each line results in a new class definition, such as `class` name Interp_*i*. Each class inherits from the previous class, except for the first one of course which has no explicit superclass. For declarations, such as variables and functions, make them `static` members of the generated class. Put statements into a method some called such as `exec()`. For example, let's translate the following commands.

```
int i = 3;
void f() { System.out.println(i); }
f();
```

We'd get the following class definitions:

```
import java.io.*;
import java.util.*;
public class Interp_0 {
    public static int i = 3;
    public static void exec() {
    }
}
```

```
import java.io.*;
import java.util.*;
public class Interp_1 extends Interp_0 {
    public static void f() { System.out.println(i); }
    public static void exec() {
    }
}
```

```
import java.io.*;
import java.util.*;
public class Interp_2 extends Interp_1 {
    public static void exec() {
        f();
    }
}
```

There are 3 key elements here:

1. Inheritance allows us to see previously defined symbols.
2. The use of `static` variables and functions means that we don't have to create instances of objects.
3. We inject the user line in the proper place within the generated class depending on whether it is a declaration or an executable statement.

To distinguish between declarations and statements, we need a final trick. Using the Java compiler API, try to parse the input line as a declaration. If it fails to parse, then it is either a syntax error or a statement. We assume it's a valid statement and then simply try to compile the line as a statement within the generated `exec()` . For example, if the user enters a print statement, try to parse it as a declaration:

```
import java.io.*;
import java.util.*;
public class Bogus {
    public static System.err.println("hi");
    public static void exec() {
    }
}
```

(The name `Bogus` is just what I happen to use when generating code in an effort to distinguish between statements and declarations.)

The compiler will puke on this input obviously and your program should then try to compile it as a statement:

```
import java.io.*;
import java.util.*;
public class Interp_0 {
    public static void exec() {
        System.err.println("hi");
    }
}
```

Finally, before you try to analyze the Java code and execute it, translate statements such as `print` *expr* `;` to `System.out.println(` *expr* `);`. For simplicity, assume that this print statement only works as the first characters of a line and only as a complete statement, not nested within a function body for example. Assume it has a space after the `print` and before the expression.

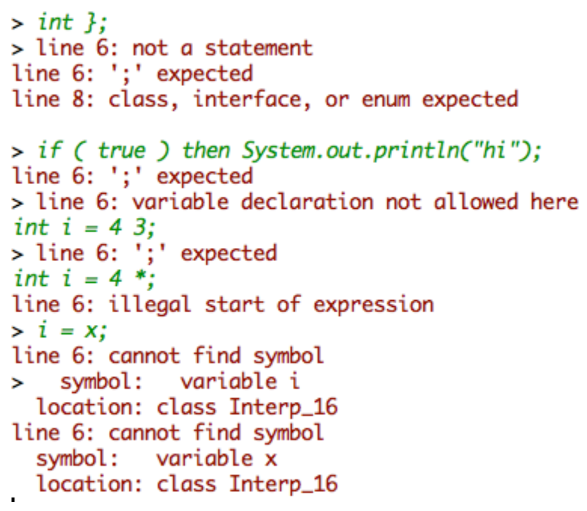
Handling stderr and stdout

For invalid Java, or at least what we can't handle, the compiler will generate errors. The Java compiler API does not emit errors automatically to standard error so you must collect these messages and emit them yourself to `stderr` using `System.err.println()`.

For automated testing purposes we need to capture that stuff. Please examine the `TestREPL` class and its methods that redirect stdout/stderr to buffers it can examine:

```
@Before
public void setup() {
    save_stdout = System.out;
    save_stderr = System.err;
    stdout = new ByteArrayOutputStream();
    stderr = new ByteArrayOutputStream();
    System.setOut(new PrintStream(stdout));
    System.setErr(new PrintStream(stderr));
}

@After
public void teardown() {
    System.setOut(save_stdout);
    System.setErr(save_stderr);
}
```

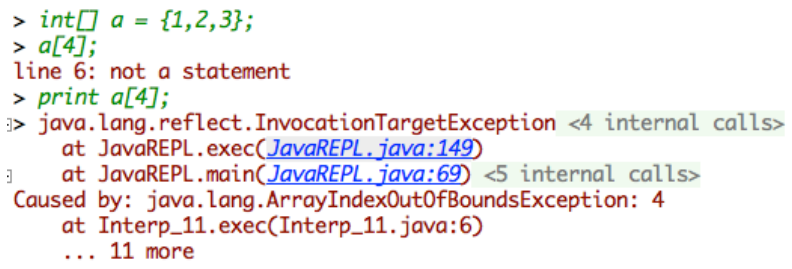
Here are some examples (notice that the errors are not necessarily intuitive because of the way we generate classes with the user input.):



```
> int };
> line 6: not a statement
line 6: ';' expected
line 8: class, interface, or enum expected

> if ( true ) then System.out.println("hi");
line 6: ';' expected
> line 6: variable declaration not allowed here
int i = 4 3;
> line 6: ';' expected
int i = 4 *;
line 6: illegal start of expression
> i = x;
line 6: cannot find symbol
>   symbol:   variable i
  location: class Interp_16
line 6: cannot find symbol
  symbol:   variable x
  location: class Interp_16
.
```

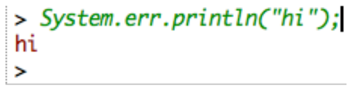
That colorization comes from the fact that I'm running it within the intellij console. From the regular command line running `java JavaREPL` , you would not see such colorization.

If there is an error during execution, the Java virtual machine will emit errors to `stderr` automatically and you don't have to do anything.



```
> int[] a = {1,2,3};
> a[4];
line 6: not a statement
> print a[4];
> java.lang.reflect.InvocationTargetException <4 internal calls>
    at JavaREPL.exec(JavaREPL.java:149)
    at JavaREPL.main(JavaREPL.java:69) <5 internal calls>
Caused by: java.lang.ArrayIndexOutOfBoundsException: 4
    at Interp_11.exec(Interp_11.java:6)
    ... 11 more
```

The Java code entered by the user might also generate `stderr` or `stdout` . You have to make sure that this output is still sent to the user. Fortunately, you don't have to do anything to make that happen. Because we are operating within the same process, and indeed the same thread, standard streams will go to their usual places. For example, you might see something like this:



```
> System.err.println("hi");
hi
>
```

Recognizing nested character streams

When the user types in a simple line like `int i;` it's easy to recognize they want you to execute that at the first newline, but we also want to handle the case of nested character streams with embedded newline's like:

```
> int[] a = {
1,
2
};
> print a[0];
1
>
```

I created a class called `NestedReader` that tracks three things:

```
public class NestedReader {
    StringBuilder buf;    // fill this as you process, character by character
    BufferedReader input; // where are we reading from?
    int c; // current character of lookahead; reset upon each getNestedString() call

    public NestedReader(BufferedReader input) {
        this.input = input;
    }
    public String getNestedString() throws IOException {
        ...
    }
}
```

A tricky issue is avoiding forcing the user to hit more than a single newline to indicate the interpreter should execute. You will find that the best way to handle this is to keep a current character of *lookahead*, which I call field `c`. This is always the next character to process. When we consume a character, I add the character to my buffer and refill the lookahead character:

```
void consume() throws IOException {
    buf.append((char)c);
    c = input.read();
}
```

The basic idea is as follows. When asked to do so, my code looks at each character in turn and processes it accordingly. If it is an opening curly brace, bracket, or parenthesis, I push the appropriate closing character onto a stack. Upon closing character, I check the top of the stack to make sure that it's the right symbol. If not, I declare an improperly nested piece of code and return the current buffer to my JavaREPL. When I see the start of a string or character literal, I consume characters until the closing character. Upon `//`, I consume characters until the end of line and in fact I strip away these comments and do not send them to the compiler. Upon `\n` and an empty stack, I return the current buffer to the invoking code otherwise I keep consuming characters.

Requirements

Just to summarize, keep in mind the following requirements.

1. Your tool must be an interactive Java shell that executes user code when they hit newline when not nested inparentheses, square brackets, or curly brackets. Those characters are not counted when they are inside single or double quoted strings or within single-line comments.
2. Accept as a declaration, anything that will compile with `public static` in front of it as a field of a class, which would include variable, function, and class definitions.
3. Your program must not require more than a single newline character after the end of a valid statement or declaration. In other words, I shouldn't have to hit extra newlines to make your program execute my code.
4. Allow a blank line as a "do nothing" statement
5. Accept `print` *expr* `;` as a statement and converted to a call to `System.out.println()` before processing so the unit test rig can test the output.
6. Comments on the end of the line should not present execution of the line: `print "hi"; // a comment`.
7. Anything that does not parse as a valid declaration, should be assumed to be a statement and compiled/executed as such.
8. All code execution must occur within the same user thread and within the same process; i.e., you **cannot** use `Runtime.exec()` or anything like it to launch Java and another process. It won't do you any good but I wanted to

prevent you from wasting time going down that path.

9. You must use the Java compiler API to parse and compile code.
10. You must use a `ClassLoader` to bring in the compiled class that you generate and then use standard reflection to execute that code (i.e., call `exec()` on the class object you bring in).
11. Users must see standard output and standard error as they would normally expect from compiler errors and run-time errors.
12. For improperly nested input, such as `(3+4]`, it's fine to just pass it off to the compiler and let it complain. But, you need to handle this case in your input scanner so it knows when to return input for processing.
13. Multiple statements should work simply because of the way we inject code, such as `i=3; y=i;` but it should not handle a line that includes both two declarations or both a declaration and a statement because that would force the declaration to be a local variable. Anything that will work after `public static` should work as a declaration for this project.
14. Do not attempt to execute statements are declarations with compile errors.
15. Upon invalid and incomplete input such as `int {;`, keep waiting for the user to close with an improper symbol or with the proper symbol or hit eof. Your program should wait even if there are multiple new lines afterwards.
16. Upon an improperly nested string, consume until the end of line before attempting to read another statement from the user.
17. Exit your program when the user hits EOF (^D or ^Z, depending on the OS).
18. Ask Java for a temp directory to store your java files.

You are free to use Java 8, as that is how I will test your code.

As usual, you should try sending in all sorts of random input in an effort to make your program robust. If I were you, I would try sending in lots of random Java code to ensure your program operates correctly.

Resources

You need to learn about compilation on-the-fly using Java's compiler API. **You are free to copy and use in your project any example code you find including Oracle javadoc but you must clearly delineate it in your code and provide the source from which you derived the bits of your solution.** Here are some pointers

- JavaCompiler interface
- Article on Java Compiler API
- How to compile java using javax
- Compile a Java file with JavaCompiler

For example, I use this to get an in-memory Java compiler:

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
```

Then, using a `DiagnosticCollector` and a `StandardJavaFileManager`, I get a specific `CompilationTask` called `JavacTask` and then compile by calling `call()`. I pull any error messages out of the DiagnosticCollector. There are lots of simple versions of compilation like this on the net that you are free to copy and mangle for your projects (assuming you give proper attribution):

```
...
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
JavacTask task = (JavacTask)
    compiler.getTask(null, fileManager, diagnostics,
    compileOptions, null, compilationUnits);
boolean ok = task.call();
```

To check whether or not something is *syntactically* (but not *semantically*) valid, use the `parse()` method of the `JavacTask` class.

```
public static boolean isDeclaration(String line) throws IOException {
   ...

   task.parse();
   return diagnostics.getDiagnostics().size() == 0;
}
```

Here's some information on dynamic class loading:

- Dynamic class loading
- What is the difference between Class.forName() and ClassLoader.loadClass()?
- ClassLoader API

I use `URLClassLoader` to load the compiled class then used `Class.getDeclaredMethod()` to find the `exec()` method. Then `invoke()` to call our `exec()` method. Call `exec()` even if it is blank because the user typed a declaration like `int i=1;` .

You must use a single instance of a `ClassLoader` during a single run of your REPL.

If it helps, here is my list of methods

- main()
- isDeclaration()
- getCode()
- compile()
- exec()
- writeFile()

Getting started

I have provided a cs652 starter kit and cs345 starter kit that you can pull into your repository. From the command line, clone your project repo and then pull in my starter kit:

```
$ git clone git@github.com:USF-CS652-S16/USERID-repl.git
Cloning into 'USERID-repl'...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
$ cd USERID-repl
$ git checkout -b master
Switched to a new branch 'master'
$ git remote add starterkit git@github.com:USF-CS652-starterkits/parrt-repl.git
$ git pull starterkit master
...
From github.com:USF-CS652-starterkits/parrt-repl
 * branch            master     -> FETCH_HEAD
 * [new branch]      master     -> starterkit/master
$ git push origin master
```

```
...
To git@github.com:USF-CS652-S16/USERID-repl.git
 * [new branch]      master -> master
```

NOTE: If you're doing this project as part of CS345 not CS652, note that your repo will live in USF-CS345-XX and you will pull the starterkit from `git@github.com:USF-CS345-starterkits/parrt-repl.git`. The package will be `cs345.repl`. Here is what the "pull" looks like to get started for CS345:

```
$ git clone git@github.com:USF-CS345-S16/USERID-repl.git
Cloning into 'USERID-repl'...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
$ cd USERID-repl
$ git checkout -b master
Switched to a new branch 'master'
$ git remote add starterkit git@github.com:USF-CS345-starterkits/parrt-repl.git
$ git pull starterkit master
...
From github.com:USF-CS345-starterkits/parrt-repl
 * branch            master     -> FETCH_HEAD
 * [new branch]      master     -> starterkit/master
$ git push origin master
...
To git@github.com:USF-CS345-S16/USERID-repl.git
 * [new branch]      master -> master
```

Building and testing

The build assumes Java 8.

I suggest that you use Intellij, which knows how to deal with maven (`mvn`) builds. You can run the unit tests with a simple click. It should work on UNIX (including Mac) and Windows.

You can build and test from the command line too:

```
$ mvn compile
...
$ java -cp target/classes:$CLASSPATH cs345.repl.JavaREPL
> print "hi";
hi
> ^D
$
```

You will need to add

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/lib/*`
```

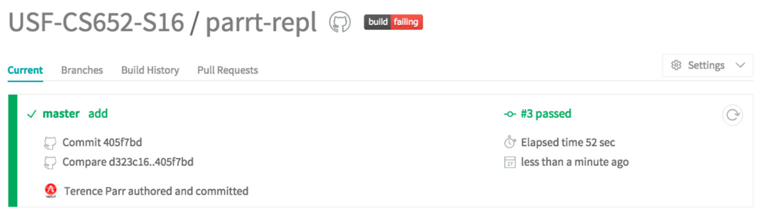
to your `CLASSPATH` so that the java compiler tools classes are available.

Or to run all tests with maven (w/o worrying about `CLASSPATH`):

```
$ mvn test
...
```

```
Running cs652.repl.TestREPL
> line 7: not a statement
line 7: ';' expected

> > line 7: ';' expected

> line 7: illegal start of expression

> > line 7: illegal start of expression

> > line 7: ';' expected

> > > > > 8> 60> > > > > > > > > > > > T:f> > > > > > 123456789> > > > > > > > fruits 1 : Pineapplefruits

Results :

Tests run: 18, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.174 s
[INFO] Finished at: 2016-01-13T18:15:10-08:00
[INFO] Final Memory: 17M/367M
[INFO] ------------------------------------------------------------------------
```

Every time you commit to your repository, your software will automatically be downloaded and tested on the Travis continuous integration server using maven. Success looks like:


USF-CS652-S16 / parrt-repl

build failing

✓ master add

#3 passed

Commit 405f7bd
Compare d323c16..405f7bd

Elapsed time 52 sec
less than a minute ago

Terence Parr authored and committed

You must have a `.travis.yml` file (already included in starter kit):

```
jdk:
  - oraclejdk8
script: mvn clean verify
language: java
```

Check out https://travis-ci.com/USF-CS652-S16/USERID-repl or https://travis-ci.com/USF-CS345-S16/USERID-repl where USERID is your github user id. Mine is parrt, for example. You will not be able to see the repositories of other students.

Deliverables

- `JavaREPL.exec()`
- Any supporting classes you write or pull in from the starterkit

Make sure your repository has all classes you build for this project.

Do not add `.class` files or any other build artifacts to your repository.

Submission

You must submit your project via github using your account and the repository I've created for you in organization USF-CS652-S16.