# EECS 345: Programming Language Concepts

# Interpreter Project, Part 4

## Due Monday, April 20

In the final interpreter, you will expand the interpreter of part 3 adding classes and exceptions. Because of the level of work needed, this last interpreter is split into two parts. Part 4 will add classes, static functions and fields, and try/catch/finally. Part 5 will add objects and new.

Because there are no objects at first, all values are either integer or boolean. In Part 5, values may also be objects.

Here is an example of a program of two classes, using only static fields and methods:

```
class Rectangle {
  static var width = 10;
  static var height = 12;

  static area() {
    var a = width * height;
    return a;
  }

  static setSize(x, y) {
    width = x;
    height = y;
  }
}

class Square extends Rectangle {
  static setSize(x) {
    super.setSize(x, x);
  }

  static main() {
    setSize(20);
    return area();
  }
}
```

Your interpreter should now take two parameters, a *file* and a *classname*. For example, `(interpret "MyProgram.j" "Square")`, where *file* is the name of the file to be interpreted, and *classname* is the name of the class whose main method you are to run. The function should call `parser` on the file *file*, and then lookup `(string->symbol` *`classname`*`)` in the environment to get the desired class, and then lookup the `main` method of this class. The final value returned by your interpreter should be whatever is returned by `main`.

**For those seeking an extra challenge**, try allowing method/function overloading. Since we are only checking types dynamically, an overloaded function must have a different number of parameters.

### Details

- Variables and methods can now be static (class) or non-static (instance).

- The `main` method should be static.
- The language supports use of `super`.
- The top level of the program is only class definitions.
- Each class definition consists of variable declarations and function definitions (just like the top level of part 3 of the interpreter).
- Although it will not be useful until the part 5 of the interpreter, note that nested uses of the `dot` operator will be allowed.
- The language supports `try`, `catch`, and `finally` blocks, and the `throw` statement.

**Parser Constructs**

```
class A {                   =>   (class A () body)
  body

class B extends A {         =>   (class B (extends A)  body)
  body

static var x = 5;           =>   (static-var x 5)

static function main() {    =>   (static-function main () body)
  body

A.x                         =>   (dot A x)

A.f(3,5)                    =>   (funcall (dot A f) 3 5)

throw e;                    =>   (throw e)

try {                       =>   (try body (catch (e) body) (finally body))
  body
}
catch (e) {
  body
}
finally {
  body
}
```

Note that either the finally or the catch block may be empty:

```
try {                      =>  (try body (catch (e) body) ())
  body
}
catch (e) {
  body
}
```

As there are no types, only one catch statement per try block is allowed.

**Basic Task**

Write an interpreter that correctly handles classes, static variables, static methods, and try/catch/finally. You should be able to set value for variables, call methods, and use `super`.

Here is a suggested order to attack the classes.

1. Use an environment that separates the names from the values, and have the values stored in reverse order, using the index of the name to look up the value. (This will be needed in part 5.)
2. Create helper functions to create a new class and instance (will be needed for part 5) and to access the portions of a class and instance.
3. All M_state and M_value functions will need to pass parameters for the class (the compile-time type) and instance ("this" - needed for part 5).
4. Change the top level interpreter code that you used in part 3 to return a class instead of returning an environment.
5. Change the top level interpreter code that you wrote for part 3 to expect static and non-static declarations for variables and functions.
6. Update your code that interprets a function definition to add a new function to the closure that looks up the function's class in the environment.
7. Create a new global level for the interpreter that reads a list of class definitions, and stores each class with its definition in the environment.
8. Create a function that takes a variable, a class, and an instance, and checks if the variable is in the list of class or instance variables and returns its value.
9. Create a function that takes a variable, and environment, a class, and an instance, if the variable is in the environment, look it up, otherwise look in the class and instance variables. (Why both this function and the one above it? To deal with when you have a dot and when you don't.)
10. Create a pair of functions (or a single function that returns a pair) that takes the left hand side of a dot expression and returns the class and the instance of the left hand side.
11. Update the code that evaluates a function call to deal with objects and classes. (Follow the denotational semantics sketched in lecture.)
12. Update the code that interprets an assignment statement so that it looks for the variable in the environment, class and instance variables.
13. Create a new `interpret` function.

Attacking the try/catch/finally is more straightfoward. The logic will be similar to how you did blocks with break and continue.

**Other language features**

Everything we did previously in the interpreter is still allowed: functions inside funtions, call-by-reference (if you chose to implement it). A function that is inside a static function will not have the `static` modifier, but it will be static simply by the nature of the containing function.