Specification

```
import std.IO;

class Main
{
    main(): int
    {
        IO.print("Hello, World!\n");

        return 0;
    }
}
```

The ROOPL language is a minimalist programming language by design, as it is an exercise to construct my first Source -> ASM compiler. A lot of choices in the design of ROOPL come from the COOL programming language, as well as early Java and C.

Contents

1. General Structure

The general structure of ROOPL is a self-enclosed `Main` class containing methods and properties.

```
import std.IO;

class Main
{
    main(): int
    {
        IO.print("Hello, World!\n");

        return 0;
    }
}
```

Listing 1: A hello world program.

Note how:

1. `import std.IO;` imports the `IO` class.
2. Statements are separated by semicolons.
3. The `IO` class has a print() function that takes a string.
4. The return value of the `Main.main()` function is an integer, much like C (and other programming languages).
5. Blocks are structured in Allman Style and thus get their own lines.

1.1. Examples

1.1.1. FizzBuzz

```
import std.IO;

class Main
{
    main(): int
    {
        for (int i = 0; i < 5; i++)
        {
            if (i % 5 == 0 && i % 3 == 0) IO.print("FizzBuzz\n");
            else if (i % 5 == 0) IO.print("Fizz\n");
            else if (i % 3 == 0) IO.print("Buzz\n");
        }

        return 0;
    }
}
```

Listing 2: A FizzBuzz example.

The FizzBuzz example briefly introduces the structure of for loops. ROOPL takes inspiration here from C and the C family of languages wherein a for loop is three statements separated by semicolons. The first statement being the setup; the second statement being the condition, and the third statement being the statement that runs after each iteration. [TODO: LINK THE LOOPING SECTION]

We also get a glimpse into the some math with the MOD operator. In ROOPL, much of the math stays consistent to C. [TODO: LINK THE MATH SECTION]

1.1.2. Fibonacci

```
import std.IO;

class Main
{
    main(): int
    {
        int in = IO.getInt();
        int out = fib(in);

        IO.print(String.from(out)->append("\n"));

        return 0;
    }

    fib(int n): int
    {
        if (n == 1) return 1;
        if (n == 2) return 1;
        return fib(n - 1) + fib(n - 2);
    }
}
```

Listing 3: A Fibonacci example in ROOPL.

Clearly ROOPL supports reading from and writing to the standard in/out. The output method is only a `print` and as such must take a string. Because the `print` method only prints the given string to the standard out, we must append a `\n` to the end so that the cursor is moved to the next line. We will delve into the semantidcs of `.` versus `->` property accessing semantics later.

2. Classes

ROOPL is an object oriented progamming language so everything is, unfortunately, wrapped in a class. The main function must have its own class, and even helper/utility functions must also be contained within some class.

Although this specification (yes THIS specification) is open to change, I opted to disregard any complex OOP properties and allow only for extension of classes. i.e. `class Square < Rectangle {}`, wherein the class `Square` is an subclass of `Rectangle` (and rectangle a superclass of square).

An example implementaion of both of these classes:

```
import std.Math;

class Rectangle
{
    width : int = 0;
    height : int = 0;

    Rectangle(width: int, height: int)
    {
        this->width = width;
        this->height = height;
    }

    setWidth(width: int): void
    {
        this->width = width;
    }

    setHeight(height: int): void
    {
        this->height = height;
    }

    area(): int
    {
        return width * height;
    }
}

class Square < Rectangle
{
    Square(side : int)
    {
        super(side, 0);
    }

    getSide(): int {
        return this->width;
    }

    setSide(side: int) {
        this->width = side;
    }
```

```
    diagonal(): float
    {
        return this->width * Math.sqrt(2);
    }
}
```

Where `Square` is just a rectangle with the added `diagonal()` method that returns the length of the diagonal of the instantiated square.

2.1. Instantiated Classes as Reference Varibles

A class, when instantiated, is treated as though it is a reference. This is more akin to how C might treat a reference to a struct. To access a method or property of a given object, we use the `->` operator. To access a method or property of a given class, we use the `.` operator.

Basically, a static access is done using the `.` operator. For example, `IO.print()`; `Math.sqrt()`; `IO.getInt()`. Conversely, accessing a non-static member of an object is done using the `->` operator. For example: `rect->area()`; `rect->setWidth(100)`; and so on.

Note that in the above examples, `this` is accessed using the `->` operator. As such, `this` is effectively an instance of the class and must be treated as a reference.

2.2. Classes as Types

An object oriented language would be useless without being able to pass objects as method arguments. This brings us to our next topic, using an object as a type. The following example demonstrates how a `Square` object can be passed as an argument.

```
import std.IO;

class SquarePrinter
{
    SquarePrinter() {}

    Print(in : Square): void
    {
        int s = in->getSize();

        if (s == 1)
        {
            IO.print("\u2588\n");
            return;
        }

        // Print top
        IO.print("\u250C");
        for (int x = 0; x < s - 2; x++) IO.print("\u2500");
        IO.print("\u2510\n");

        // Print middle rows
        for (int y = 0; y < s - 2; y++)
        {
            IO.print("\u2502");
            for (int x = 0; x < s - 2; x++) IO.print(" ");
            IO.print("\u2502\n");
        }

        // Print bottom
```

```
        IO.print("└");
        for (int x = 0; x < s - 2; x++) IO.print("─");
        IO.print("┘\n");
    }
}
```

2.3. Properties of Objects

Generally, objects can support properties. ROOPL respects this and thereby supports properties, as can be seen above.

Properties are of the form `NAME : TYPE (= VALUE);`. Even if declaration isn't required at this point, the compiler should error if a property is left undeclared at the completion of a constructor.

An example of this is the Rectangle class.

```
class Rectangle
{
    width : int;
    height : int;

    Rectangle(width: int, height: int)
    {
        this->width = width;
        this->height = height;
    }

    // ...
}
```

Had the width and height not been set at the end of the execution of `Rectangle()`, then the compiler should error.

As it stands, ROOPL has no property modifiers. This might be changed in the future.

3. Primitives

Primitives in ROOPL are limited to a few data types. We have, in order of increasing size/ complexity, `boolean`, `char`, `int`, `float`.

3.1. Boolean

Booleans represent `true` or `false` values.

3.2. Char

Chars represent UTF-8 characters. This gives a total of 256 different values supported. Read more about UTF-8 here.

3.3. Int

Intgers in ROOPL are 64 bit signed integers.

3.4. Float

Floating point numbers are 64 bits wide.

3.5. The Future of Primitives

Admitedly, there are few primatives supported. One might hope for double precision floats, or long integers, and so on. In order to keep the scope of this project smaller, I have opted to only include few primatives such that the language is still fully-featured.

A Appendix

A.1 EBNF

```
identifier  = letter , { letter | digit | "_" } ;

number      = digit , { digit } ;

string      = '"' , { letter - '"' } , '"' ;

letter      = "A" ... "Z" | "a" ... "z" ;
digit       = "0" ... "9" ;

program     = { import_stmt } , { class_decl } ;

import_stmt = "import" , import_path , ";" ;
import_path = identifier , { "." | ":" } , identifier ;

class_decl  = "class" , identifier , [ inheritance ] ,
              "{" , { class_member } , "}" ;

inheritance = "<" , identifier ;

class_member
            = property_decl
            | method_decl ;

property_decl
            = identifier , ":" , type ,
              [ "=" , expression ] , ";" ;

method_decl = identifier , "(" , [ parameters ] , ")" ,
              ":" , type ,
              block ;

parameters  = parameter , { "," , parameter } ;
parameter   = identifier , ":" , type ;

type        = "int"
            | "float"
            | "string"
            | "void"
            | identifier ;

statement   = block
            | var_decl
            | assignment
            | if_stmt
            | for_stmt
            | return_stmt
            | expr_stmt ;

block       = "{" , { statement } , "}" ;

var_decl    = type , identifier ,
              [ "=" , expression ] , ";" ;
```

```
assignment  = ( object_access | identifier ) ,
              "=" , expression , ";" ;

expr_stmt   = expression , ";" ;

return_stmt = "return" , [ expression ] , ";" ;

if_stmt     = "if" , "(" , expression , ")" , statement ,
              [ "else" , statement ] ;

for_stmt    = "for" , "(" ,
                [ var_decl | assignment | expr_stmt ] ,
                expression , ";" ,
                [ assignment | expr_stmt ] ,
              ")" ,
              statement ;

expression  = logical_or ;

logical_or  = logical_and , { "||" , logical_and } ;
logical_and = equality ,    { "&&" , equality    } ;

equality    = relational , { ( "==" | "!=" ) , relational } ;

relational  = additive , { ( "<" | ">" | "<=" | ">=" ) , additive } ;

additive    = multiplicative , { ( "+" | "-" ) , multiplicative } ;

multiplicative
            = unary , { ( "*" | "/" | "%" ) , unary } ;

unary       = [ "-" | "!" ] , primary ;

primary     = literal
            | identifier
            | object_access
            | method_call
            | "(" , expression , ")" ;

literal     = number | string ;

object_access
            = primary , "->" , identifier ;

method_call = ( primary , ( "->" | "." ) , identifier ,
                "(" , [ arguments ] , ")" )
            | ( identifier , "(" , [ arguments ] , ")" ) ;

arguments   = expression , { "," , expression } ;
```