MicroOOP Language Manual

MOOP

Anthony Tranduc anthony.tranduc@tufts.edu
Isabelle Lai isabelle.lai@tufts.edu
Jacqueline Chin jacqueline.chin@tufts.edu
Maxwell Anavian maxwell.anavian@tufts.edu

Dedicated to Moopert
Moopert lives on
(ノ°□°)ノ ⌒__ ̆_

2/28/2022

Contents

1 Introduction

This manual describes the syntax and semantics for the MicroOOP language, providing users with a resource to write their own complete programs in MicroOOP. MicroOOP is an object oriented language based on the MicroC language that includes built-in support for Classes, Objects, Inheritance, Encapsulation, Strings, and cool, cute comments.

2 Reading Directions

The MicroOOP Language Reference Manual uses the following notation:

- prose - represented in plain font
- inline code - `represented in typewriter font`
- blocks of code - represented like the following block

```
class Main {
    string name = "Hello World";
}
```

- grammar rules - *represented in italicized font*

3 Lexical Conventions

3.1 Identifiers

Identifiers may contain letters, digits, and '_'. They must begin with an uppercase or lowercase letter.

3.2 Keywords

Reserved keywords are listed here:

```
if else for while return int bool float
string void true false class this super
print new null
```

3.3 Symbols

Reserved symbols are listed here:

```
( ) < > = <- .  { } + - / *   !    | ;
```

3.4 Whitespace

Newlines, tabs, and spaces are ignored.

3.5 Comments

Single line comments are denoted with a '//' at the beginning of the line.

The official way to denote multi-line comments is as follows:

```
/* p.s.  there may be another way to
   comment multiple lines ;) */
```

3.6 File Extensions

It is standard convention for MicroOOP files to end in '.moop'

4 Data Types

4.1 Primitive Data Types

Name	Formal Type	Example	Description
Integer	`int`	1, -42	Represented by numerals, optionally led by '-'.
Float	`float`	1.813	Uses '.' to denote decimal value
Boolean	`bool`	`true`/`false`	Only has two values and two keywords
String	`string`	"bruh"	Any sequence of characters encased in double parentheses
Void	`void`	n/a	Void has no value and reprensents a lack of a data type
Null	`null`	`null`	Null represents no value

4.2 Interchanging Types

Booleans have corresponding integer values. `true` has a value of `1`. `false` has a value of `0`.

Null is a special data type. Any data type can have a `null` value. For example, a variable can be of type float but have a null value.

5 Operators

5.1 Operators for Numeric Arithmetic

These operators can be used with integers and floats. When used with integers, they return integers. When used with floats, they return floats. All the operators are binary unless noted otherwise.

Operator	Description
+	Numeric addition
-	Numeric subtraction
-	Unary inversion
*	Numeric multiplication
/	Numeric division

5.2 Logical Operators

These operators can be used with booleans and return booleans.

Operator	Description
&&	Binary logical and
||	Binary logical or
!	Unary logical negation

5.3 Comparison Operators

These operators can be used with integers, floats, and booleans. When used, they return a boolean value. All the operators are binary unless noted otherwise.

Operator	Description
==	Equivalence
>	Less than
<	Greater than
>=	Greater than or Equal to
<=	Less than or Equal to

5.4 Other Operators

Operator	Description
~	Negates accessibility of instance variables or methods. When used with an instance variable, the scope of the variable becomes public. When used with a method, the scope of the method becomes private.
(Dot) .	Used to access a member of an object
<-	Used to extend parent classes

Example in Java

```
public int x
private string Animal () { return "MOO" }
```

Translates to in MOOP

```
~int x
~string Animal () { return "MOO" }
```

6 Expressions

6.1 Literals

Any literal is an expression.
Numeric literals are either integers or floats, and may consist of a sequence of digits, with or without decimals or the negative sign:

```
12345
123.45
-42
```

Boolean literals are represented by the keywords `true` or `false`.

String literals consist of a sequence of characters within a set of double quotes:

```
"Hello"
"Chicken_Nuggets32"
```

6.2 Operator Expressions

Expressions can be formed with unary operators (*unop*) or binary operators (*binop*) and other expressions (*expr*) as follows:

unop expr
expr binop expr

The type of *exprs* used in the operator expression and the type of *expr* returned by each operator expression will depend on the operator. See **Section 5 Operators** above for more details.

6.3 Assignment Expressions

The assignment expression will store the expression with a specific identifier. The expression is formed as follows:

identifier = *expr*

6.4 Method Calls

Since methods can only be defined within the scope of a class, you must call the method using the following form:

identifier-1.*identifier-2*(<*optional exprs*>)

Identifier-1 corresponds to the class or object name. Class names are used for static methods and object names are used for instance methods. *Identifier-2* corresponds to the method name.

6.5 Constructor Calls

A constructor call creates an instance of that class. The expression is formed using the "new" keyword followed by the class name identifier to denote that the constructor is being called. It is formed as follows:

`new` *identifier*(<*optional exprs*>)

6.5.1 Parent Constructor (super) Calls

In the instance in which a class is the child of another class i.e

```
class Child <- Parent {
    ...
}
```

The `super` keyword can be used in the constructor of the `Child` class to explicitly create an instance of the `Child` class using the `Parent` class constructor with any necessary parameters.

`super`(<*optional exprs*>);

7 Statements

7.1 Expressions as Statements

An expression may take the form of a statement if followed by a semicolon.

expr;

For example, assignments can be made as a standalone statement:

```
x = 10;
```

7.1.1 Declaring and assigning variables as statements

Variables can be declared and assigned with the format:

datatype identifier = *expr*;

For example

```
int    moop   = 10;
float  ert    = 10.0;
string lives  = "Moopert";
bool   on     = true;
```

7.2 Return Statements

Return statements exit the current method and denote what value will be returned to the caller of the method. They have the form:

`return` *optional_ expression*;

If no expression is given, the return type is `void`.

7.3 Conditionals

Conditional statements execute statements based on the value of the expression. They have two forms. The first form is:

`if (` *expr* `) {` *statement* `}`

In the first form, the statement is only executed if the expression evaluates to a non-zero value. The second form is:

`if (` *expr* `) {` *statement-1* `} else {` *statement-2* `}`

This form functions similar to the first form. However, if the expression evaluates to zero, *statement-2* is executed.

For example

```
int x = 10;
int y = 5;
if (x == y) {
    print("moopert");
} else {
    print("shmoopert");
}
```

7.4 While Loops

While loop statements use the `while` keyword. They have the form:

`while ( `*expr*` ) { `*statement*` }`

For example

```
int x = 1;
while (x < 10) {
    x = x + 1;
}
```

As long as the expression within the parentheses evaluates to a non-zero value, the sub-statement is repeatedly executed. The expression is always evaluated before the sub-statement and is re-evaluated every repeated iteration.

7.5 For Loops

For loop statements use the `for` keyword. They are similar to while loops, but specify three additional expressions to be used as a test to continue the loop. They have the form:

`for (` *expr-1*`;` *expr-2*`;` *expr-3*`) {` *statement* `}`

For example

```
int x = 0;
for (int i = 0; x < 10; i = i + 1) {
    x = x + 1;
}
```

Expr-1 is an initialization of a variable to be used in *expr-2*. *Expr-2* is evaluated before each iteration. *Expr-3* modifies whatever is initialized in *expr-1* after each execution of the statement.

This is equivalent to using the following while loop:

expr-1;
`while (` *expr-2* `) {`
    statement;
    expr-3;
`}`

Expr-1 and *expr-3* are optional. The loop will execute as expressed in the while loop expansion above without *expr-1* and *expr-3*.

8 Methods

Methods are defined within class declarations. They take the form:

datatype identifier `(` *<optional variables>* `) {`
    statement
`}`

Method declarations must start by providing the output type, followed by an identifier. The optional list of formal variables, separated by commas, should consist of the type and identifier for each formal variable. The body of the method consists of statements. Each body should conclude with a `return` statement that returns the appropriate type. The only exception to this rule is the `void` return type. In this case, the `return` statement can be omitted or simply `return;`.

9 Classes

Classes define the contents of objects. They are declared using the `class` keyword and consist of variables and method definitions.

9.1 Declaration

Class declarations have the following form:

`class` *identifier* `{`
    body
`}`

The class *identifier* must begin with an uppercase letter. The *body* can consist of variable and method declarations.

9.2 Constructors

All classes must have a constructor declaration. Constructors have the form:

identifier `(`*<optional variables>*`) {` *statements* `}`

The *identifier* must correspond to the class name.

The constructor for the Dog class is shown below:

```
1 class Dog {
2     string name;
3 
4     Dog (string name) {
5         this.name = name;
6     }
7 }
```

9.3 Inheritence

9.3.1 Extends

Inheritence is signified using the `<-` symbol.

The extends symbol is used in the format *identifier-1* `<-` *identifier-2*, where *identifier-1* is the child class and *identifier-2* is the parent class.

For example, the code block can be read as "Class Dog Extends Animal."

```
class Dog <- Animal {}
```

A class can only have one parent at most, although multiple classes can inherit from the same parent class.

9.3.2 Super

In order to access the constructor of a parent class, children have access to the `super` keyword. The parent constructor can be invoked like so:

```
class Animal {
    Animal () {
        print ("Animal Constructor");
    }
}

class Dog <- Animal {
    Dog () {
        super();
        print ("Dog Constructor");
    }
}
```

Output:

```
$ > Animal Constructor
$ > Dog Constructor
```

9.3.3 This

The `this` keyword can be used to access the current class' instance variables and methods. For example:

```
class Dog {
    string sound;
    Dog (string noise) {
        this.sound = noise;
    }
}
```

`this` cannot be used to access the constructor.

9.4 Scope

Member variables of a class have a private scope by default. This means that the variable can only be accessed within the class declaration it is defined in. Methods of a class have a public scope by default. This means that the method can be called from anywhere within the program.

Both scopes can be changed using the invert operator detailed in section **5.4 Other Operators**.

10 Programs

A program in MicroOOP is defined as a sequence of classes. Each program must have at least a `Main` class with a `main ()` method defined. These are the first parts of the program to be evaluated.

The `main` method must have the following method definition: `int main()`.

The following is a full example of a working program:

```
// This is a single line comment.
/* This is a multi
   line comment. */

/* This example code demonstrates how to create objects and how to
   inherit from other objects.

   Note that the main program is a class itself. */

class Main {
    int main () {
        print ("Hello World")
        Dog moopertTheDog = new Dog("Moopert", 5);

        moopertTheDog.animalNoise();
        moopertTheDog.getAge();
        return 0;
    }
}

class Animal {
    /* public field variables */
    ~ string name;
    ~ int age;

    Animal (string nameInput, int ageInput) {
        this.name = nameInput;
        this.age = ageInput;
    }

    int getAge() {
        return age;
    }

    void animalNoise() {
```

```
36          print("Generic animal noise");
37      }
38  }
39
40  class Dog <- Animal {
41      Dog (string nameInput, int ageInput) {
42          super(nameInput, ageInput);
43      }
44
45      // example of overriding the parent class animalNoise function
46      void animalNoise() {
47          print("MOO");
48      }
49  }
```

11 Built-in Functions

`void printi(int n)`
Prints an integer n to standard out.

`void printb(bool b)`
Prints a bool b to standard out.

`void printf(float f)`
Prints a float f to standard out.

`void print(string s)`
Prints a string s to standard out.

`void toString(int n)`
Converts int to string.

12 Standard Library

The standard library is written in MicroOOP, providing users with useful classes and methods to help them write their own MicroOOP programs. These standard library additions also provide examples to users for what code in MicroOOP looks like.

12.1 Linked List

The following is a basic implementation of a LinkedList. The Node class represents one element of the list, and the LinkedList class contains a LinkedList constructor, insert method, delete method, a method to get an element by index, a method to get the size of the LinkedList, and a method to print the data of the LinkedList.

```
class Node {
    int data;
    Node next;

    Node (int data) {
        this.data = data;
    }

    void setNext (int data) {
        this.next = new Node(data);
    }

}
```

```
class LinkedList {
    Node first;
    int size;

    LinkedList () {
        this.size = 0;
    }

    void insertElem (int elem) {
        if (this.size == 0) {
            this.first = new Node(elem);
        } else {
            Node curr = this.first;
            for (int i = 0; i < this.size; i = i + 1) {
                curr = curr.next;
            }
            curr.setNext(elem);
        }
        this.size = this.size + 1;
    }

    void deleteElem (int index) {
        if (index == 0) {
            this.first = first.next;
            return;
        }

        Node prev = this.first;
        Node curr = this.first.next;

        for (int i = 0; i < size; i = i + 1) {
            if (i == index) {
                prev.next = curr.next;
                this.size = this.size - 1;
                return;
            } else {
                prev = curr;
                curr = curr.next;
            }
        }

        print("Element does not exist.");
    }
```

```
44
45      int getSize () {
46          return this.size;
47      }
48
49      int getElem (int index) {
50          Node curr = this.first;
51          for (int i = 0; i < index; i = i + 1) {
52              curr = curr.next;
53          }
54          return curr.data;
55      }
56
57      void printLinkedList () {
58          Node curr = this.first;
59          for (int i = 0; i < index; i = i + 1) {
60              printi (this.data);
61              curr = curr.next;
62          }
63      }
64
65
66
67  }
```

12.2 Stack

The following is basic implementation of a Stack using a LinkedList as the underlying structure. It has the basic push and pop functionalities of a standard Stack.

```
1  class Stack {
2      LinkedList stack;
3
4      Stack () {
5          this.stack = new LinkedList();
6      }
7
8      void push (int elem) {
9          stack.insertElem(elem);
10     }
11
12     int pop () {
13         if (stack.getSize() > 0) {
14             int poppedElem = stack.getElem(stack.getSize() - 1);
15             stack.deleteElem(stack.getSize() - 1);
16             return poppedElem;
17         }
18         return null;
19
20     }
21 }
```

12.3 Queue

The following is a basic implementation of a Queue using a LinkedList as the underlying structure. It has the basic queue and dequeue functionalities of a standard Queue.

```
class Queue {
    LinkedList queue;

    Queue () {
        this.queue = new LinkedList();
    }

    void queue (int elem) {
        queue.insertElem(elem);
    }

    int dequeue () {
        if (queue.getSize() > 0) {
            int dequeuedElem = queue.getElem(0);
            queue.deleteElem(0);
            return dequeuedElem;
        }
        return null;
    }

}
```