MAZE Reference Manual

Alexander Brown (aab2212), Alexander Freemantle (asf2161),
Michelle Navarro (mn2614), Lindsay Schiminske (ls3245)

Introduction

MAZE is a Java-like object-oriented language with game design features that allow programmers to create text-based games. The core building blocks of MAZE are custom classes. The language supports inheritance to allow for flexibility. It is a strongly-typed language.

The language reference manual is organized as follows:

[1] Logo concept by Filip Lichtneker https://dribbble.com/shots/2641206-Maze-Media-logo-concept

1. Lexical Conventions

1.1 Comments

Comments begin with (* and end with *)

1.2 Identifiers

An identifier is any sequence of characters that can be used as a name for a variable, method, or new data type. Valid identifier characters include ASCII letters, decimal digits, and underscores. The first character of an identifier cannot be a digit. Identifiers cannot be the same sequence of characters as keywords.

1.3 Keywords

The following identifiers are reserved and cannot be used otherwise. They are case sensitive:

int	*while*	*null*
char	*bool*	*true*
return	*print*	*false*
if	*else*	*string*
void	*extends*	*class*

1.4 Literals

MAZE literals can be integers (see 2.1.1), booleans (2.1.2), floats (2.1.3), characters(2.1.4), and strings (2.1.5).

2. Data Types

2.1 Primitive Types

2.1.1 Integers

int

An integer is a whole value between -2^{31} and $2^{31} - 1$. The default value is 0.

2.1.2 Boolean

bool

A single byte that can have the value true or false. The default value is false.

2.1.3 Float

float

A float is an integer followed by a decimal and a fractional part. The default value is 0.0 .

2.1.4 Void

void

Use the void type to signify a function that has no return value.

2.1.5 Character

char

A character is a single byte. It holds the value of a character in the set of allowed characters. The default value is an empty char '' .

The following are escape-sequence characters

- '\\' - backslash
- '\"' - double-quote
- '\'' - single-quote
- '\n' - newline
- '\r' - carriage return
- '\t' - tab character

2.1.6 String

string

Strings are a sequence of zero or more characters, digits, spaces or other ASCII characters. The default value is the empty string “” .

ex. “This is a string”

2.2 Non-primitive Types

2.2.1 Arrays

A container that holds a fixed number of values of a single type. The size of the array must be specified at creation. However the default value at each index will be 0.

2.2.2 Objects

An instance of a class. See section 7 for more information of classes and how objects work within them. The default instance of an object is the one resulting from the invocation of the default constructor.

3. Expressions and Operators

3.1 Expressions

An expression is composed of:

- A literal value
- A variable identifier
- A reference
- An arithmetic expression
  An arithmetic expression consists of one or more operands and zero or more operators. The arithmetic expression may include parentheses for grouping.
  Arithmetic expression examples:
    Operand, 0 operators: 100;
    Two operands, 1 operator: 100 + 101;
    Use of Parentheses: (100*(102-101));
- A call to a method that returns some value or reference
- A call to a constructor to create an object

3.2 Operators

3.2.1 Assignment Operators

The assignment operators assign values from the right hand operand to the left side operand.

```
ex. int x = 4;
    int y = 3 + 7;
    int z = null;
```

3.2.2 Arithmetic Operators

The arithmetic operators include + (addition), - (subtraction), * (multiplication), / (division) and negation.

ex.

Addition:

```
int x = 6 + 2;
```

Subtraction:

```
int x = 6 - 2;
```

Multiplication:

```
int x = 6 * 2;
```

Division:

```
int x = 6 / 2;
```

Negation:

```
int x = -6;
```

3.2.3 Relational Operators

expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*

The operators are < (less than), >(greater than), <= (less than or equal to) and >=(greater than or equal to). The relational operators group left to right.

3.2.3 Equality Operators

expression == *expression*
expression != *expression*

The == (equal to) and != (not equal to) operators evaluate the expression to determine if the two expressions are equal or not equal.

3.2.4 Logical Operators

expression && *expression*
expression || *expression*

The && (logical AND) returns true if both expressions are met and false otherwise. The ||

(logical OR) returns true if at least one expression is true and false if no expressions are met.

4. Statements

4.1 Expression Statements

Expression statements are in the form:

expression ;

Usually expression statements are assignments or function calls.

ex. int value;
int value = 14;

4.2 if Statement

The two forms of conditional statements are:

if (*expression*) *statement*

AND

if (*expression*) *statement1* else *statement2*

The expression is evaluated in both cases and if it is true then the first statement is executed, if it evaluates to false statement2 is executed in the second case.

4.3 The while Statement

The while statement has the form:

while *(expression) statement*

The statement is executed repeatedly as long as the Expression evaluates to true.

4.4 Return Statement

The return statement has the form:

return ;

OR

return (*expression*) ;

In the first case nothing is returned to the caller of

the function, in the second case the expression is returned.

5. Methods

5.1 Method basics

A method is a collection of statements that are grouped together to perform an operation. A method can return a value or nothing when called. The void keyword is used to create a method that returns no value. If the void keyword is not used when creating a method then the method must return a value.

```
(* Method with return value *)
     int myMethod (int a, int b){
          (* body *)

          return (*int*)
     }

(* Method with no return value *)
     void myMethod (int a) {
          (* body *)
     }
```

The name of the method can not be the same as the name of the class it is defined in. (See Ch.7 for Classes)

5.2 Overloading Methods

A method can be overloaded. A class with two or more methods with the same name but with a different number or type of arguments is said to be overloaded.
This helps to increase the readability of the program.

A method can be overloaded by either changing the number of parameters or by changing their type.

```
(* Same method name with different types *)
     int myMethod (int a, int b){
          (* body *)
          return (*int*)
     }

     float myMethod (float a, float b){
          (* body *)
          return (*float*)
     }

(* Same method with different number of parameters *)
     int myMethod(int a, int b){
          (*body*)
          return (*int*)
     }

     int myMethod(int a, int b, int c){
          (*body*)
          return (*int*)
     }
```

5.3 The Main Method

The main method must appear once and only once in the program. The main method is the first method to be called, and in turn calls all of the other methods required to run the program.

Sample main method:

```
class example {
     void main(char[][] args){
          (* do something *)
     }
}
```

6. Scope

Scope refers to the lifetime and accessibility of a variable. The scope of the variable depends on where it is declared.

6.1 Local Variables

The scope of variables is assumed to be local when variables are defined as follows:

```
void funcname {
     int x = 20; (* x is a local variable *)
     print x;
}
```

Local variables can only be used within the method they are defined in. The variable is created when the method is entered and destroyed once the method is exited.

6.2 Global Variables

The scope of variables is assumed to be global when variables are defined as follows:

```
int a = 5; (* a is a global variable *)

if (expression){
     a = a + 3;
}
```

Global variables are declared outside of all functions, and are available throughout the entire program.

7. Classes

7.1 Class Definitions

A class is a template for creating different objects which defines its properties and behaviors. It can contain fields and methods to describe the behavior of the object.

```
class MyClass {
     (*field, constructor and method declarations*)
}
```

The class body contains all the code for the objects created from the class.

7.2 Class Fields

Fields contain data belonging to objects created of that class type.

7.3 Constructors

Constructors are used to initialize new objects.

7.3.1 Default constructors

Default constructors are invoked when no custom constructor is defined. The values of the object's fields are set to the default values of their types.

7.3.2 Custom constructors

A custom constructor creates an object with fields set to specified passed in values.

7.3.3 Standard defining of a class

```
class myClass {

     myClass (int myVar1, int myVar2){
          myVar1 = …;
          myVar2 = …;
     }
}
```

7.4 Inheritance

Inheritance is the process where one class acquires the properties (methods and fields) of another class. The subclass is the class which inherits the properties of another and the superclass is the class whose properties are inherited. Inheritance uses the 'extends' keyword to define the subclass with respect to the superclass.

```
class Car {

void carConstructor(string make, string model){
          make = "Tesla";
          model = "Model 3";
     }

     void setPrice(){
          ...
     }
}
class SportsCar extends Car {
     (*SportsCar inherits Car class methods and vars*)
     void setHp(){
         ...
     }
}
```