# Contents

# Chapter 1

# Lexical Conventions

## 1.1 Comments

In Jtrix, `/*` opens a multi-line comment and `*/` closes the comment. `//` opens a single line comment that is ended by a newline. Comments are ignored and they do not nest. Comments cannot be contained in literals.

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

/* This is a multi-line comment,
   so I can end it down here. */

int x = 13; //Comments can follow lines of code
```

## 1.2 White Space

```
//white space is defined as follows:

' '      //space character
'\t'    //horizontal tab character
'\n' //newline character
```

White space is ignored, but can be utilized in cases of tokenization.

```
//white space is used to differentiate between the single token >= and the two tokens
    > and =

a >= b;  //this is a valid statement which evaluates whether a is greater than or
    equal to b, whereas
a > = b; //is not a valid statement
```

## 1.3 Tokens

Jtrix uses five different tokens:

*Identifiers*
*Keywords*
*Literals*
*Separators*
*Operator*

### 1.3.1 Identifiers

Jtrix Identifiers are any $n$-length set of characters beginning with an English letter that is either uppercase (`A - Z`) or lowercase (`a - z`) that do not form a Jtrix Keyword. Identifiers may not start with integers (`0 - 9`) or with underscores (_) but can include them in any following position.

```
//valid identifiers
thing
thing1
thingTwo
ThingTwo
Thing_Two

/* Note: identifiers are case sensitive, so thingTwo and ThingTwo are treated
   distinctly different */
```

### 1.3.2 Keywords

The following ASCII character sequences are reserved identifiers that may only be used as Jtrix Keywords: `break boolean char col continue else float for foreach if int matrix return row string void while`

### 1.3.3 Operators

Jtrix has both binary and unary operators. Operators are used in combination with variables or literals to create expressions. Binary operators are represented and evaluated in infix notation and unary operators are evaluated in prefix notation. Jtrix uses the following types of operators:

*Arithmetic Operators*
*Equivalence Operators*
*Logical Operators*

#### Arithmetic Operators

```
// addition applies to int, float, matrix
  +
```

```
// subtraction applies to int, float, matrix
   -
// multiplication applies to int, float, matrix
   *
// division applies to int, float
   /
// modulo applies to int
   %
```

### Equivalence Operators

```
// equality applies to int, boolean, float, matrix
   ==
// inequality applies to int, boolean, float, matrix
   !=
// greater than applies to int, float
   >
// greater than or equal to applies to int, float
   >=
// less than applies to int, float
   <
// less than or equal to applies to int, float
   <=
```

### Logical Operators

```
// all logic operators apply only to boolean

// AND
   &&
// OR
   ||
// NOT
   !
```

### Unary Operators

The operator not ( `!` ) is used strictly as a unary operator. The minus sign ( `-` ) can be used as both a unary and binary operator.

```
boolean a = true;
return !a; // using not as a unary operator

int pos1 = 1;
```

```
int neg1 = -1; // using the minus sign as a unary operator
return (neg1 - pos1); // using the minus as a binary operator
```

### Precedence of Operators

```
// Jtrix will evaluate operators in the following order:
 // 1. NOT, and arithmetic negation
!  -

// 2. multiplication, division, and modulo
*  /  %

// 3. addition, and subtraction
+  -

// 4. equal to, and not equal to
==  !=

// 5. less than, less than or equal to, greater than, and greater than or equal to
<  <=  >  >=

// 6. AND
&&

// 7. OR
||
```

## 1.3.4 Literals

Jtrix uses the following literals:

*Integer*
*Float*
*Boolean*
*Character*
*String*

### Integer Literals

Integer literals represent whole number decimal values using characters
`0 - 9` in a sequence

```
//examples of integer literals:
0
1
```

### Float Literals

Float literals represent whole or non-whole number values. Floats must have a decimal point followed by at least one number `1 - 9`. Floats may have `0 - 9` preceding the decimal. Floats may be exponents in which case the sequence is concluded with `+` or `-` and `e` or `E` and `0 - 9`.

```
//examples of float literals:
.3
0.3
3.6
3.6e+3
3.6E-3
```

### Boolean Literals

Boolean literals are `true` or `false`. The type `boolean` has two values that are represented by the boolean literals.

### Character Literals

```
//examples of character literals:
```

### String Literals

String literals consist of zero or more character sequences enclosed within double quotes, `"`. A backslash is used as an escape to represent special characters.

```
//examples of String literals:
"hello world"      // an 11 character string
""                 // an empty string

//special characters:
"\n"            // a string containing the newline character
"\""            // a string containing the double quote character
"\'"            // a string containing the single quote character
```

## 1.3.5 Separators

Jtrix has the following separators:

```
( )   { }   [ ]   ;   ,   .
```

# Chapter 2

# Data Types

## 2.1 Primitive Data Types

Jtrix has the following primitive data types:

### 2.1.1 Boolean

The boolean type stores either true or false in 1 bit.

```
bool x = true;
```

### 2.1.2 Char

The char type stores ASCII letters in 2 bytes.

```
char first = 'f';
```

### 2.1.3 Int

The int type stores integer values in 32 bits (from –2,147,483,648 to 2,417,483,647).

```
int x = 32;
```

### 2.1.4 Float

The float type stores floating point values in 32 bits (from -3.4E+38 to +3.4E+38).

```
float x = 32.5;
```

### 2.1.5 Void

The void type is an empty value that is returned when a function does not return anything. It is not a variable type.

## 2.2 Standard Library Types

### 2.2.1 Arrays

Jtrix supports the arrays, denoted *type[] arrayName*, that either takes an empty array of a certain size or an array with data in it. Once created, the array's length can no longer be changed, but the elements inside can be.

```
int[] intArr = [1, 2, 3]; // an array of integers
print(intArr[0]); // 1
```

#### Declaring and Initializing Arrays

```
int[] x= [1, 2, 3, 4, 5]; // Array x is of type int and of size 5
```

### 2.2.2 Strings

Jtrix also supports strings that are essentially sequences of characters. The strings must contain only ASCII characters.

```
String a = "foo";       // this is a string
```

### 2.2.3 Matrices

One of Jtrix's main features is the matrix. In order to create a matrix of size n $\times$ m, one must first specify the size in the initialization of the matrix. One can access element $a_{i,j}$, one would write the command as if the matrix were a 2 $\times$ 2 array.

```
matrix<int, 2, 2> mat = [1, 2; 3, 4]; // a 2 x 2 matrix
print(mat[1][1]); // 4
```

#### Declaring Matrices

They can be declared by specifying the types that are to be contained and the dimensions of the matrix.

```
matrix<int, 3, 5> mat; // Creates a new 3 x 5 matrix that accepts only integers
```

### Initializing matrices

Matrices can be initialized to contain either elements of type float or int.

```
matrix<float, 3, 5> mat = [1.1, 2.3 ; 4.5, 5.1]; // 2 x 2 matrix
```

## 2.3 Functions

Functions in Jtrix are first class objects (i.e. one can pass functions as values to other functions or return functions as values). They can take inputs that are specifically typed (i.e. a function can take *int a* but not just *a*). Each function has to return something, whether it be an integer, float, string, or void.

```
fun void helloWorld = () {
   print("Hello World!")
};

fun void doSomething = (fun x) {
   x();
};

doSomething(helloWorld()); // "Hello World!"
```

## 2.4 Mutability

All data types in Jtrix are immutable. The user can assign identifiers to each type but cannot overwrite the object that the identifier is assigned to. To effectively change a variable, the user reassigns the identifier to reference a different piece of data.

```
/* this assigns variable x to integer literal 3 and then reassigns it to integer
    literal 4 */
x = 3;
x = 4;
/* x is now referencing a different value, but the integer literal 3 did not change */

// string example:
strFoo = "Foo ";
strBar = "Bar";
strFoo = strFoo.concat(strBar);
/* strFoo was reassigned to the concatenation of the literal values "Foo " and "Bar"
    to result in strFoo to reference "Foo Bar" */
```

## 2.5 Casting

While Jtrix does not implicity cast variables, one can explicitly cast types. For primitive types, as long as one can properly convert from one type to another (i.e. the char '9' can be converted to int while the char 'a' cannot). One can only explicitly cast the following:

```
int(*float*); // converts float to int
float(*int*); // converts int to float
String(*int*); // converts int to str
String(*float*); // converts float to str
String(*char*); // converts char to str
```

# Chapter 3

# Standard Library

## 3.1 Matrices

For matrices, Jtrix will support basic operations such as returning a specific column or row, removing columns or rows, switching rows, getting the transpose, and returning the dimensions.

| Operation | Result |
|---|---|
| mat.col(x) | Returns an array of the (x + 1)-th column |
| mat.row(x) | Returns an array of the (x + 1)-th row |
| mat.spliceColumn(k) | Returns a n x (m - 1) matrix that contains the same information as the input without the (k + 1)-th column |
| mat.spliceRow(k) | Returns a (n - 1) x m matrix that contains the same information as the input without the (k + 1)-th row |
| mat.switchRows(x, y) | Returns a n x m matrix that swaps the (x + 1)-th row and the (y + 1)-th row |
| mat.transpose() | Returns the transpose of a matrix |
| mat.dim() | Returns the dimensions of the matrix in an array |

```
matrix<2,2> mat = [1, 2; 3, 4];
mat.col(1); // [2, 4]
mat.row(0); // [1, 2]
mat.spliceColumn(1); //[1; 3]
mat.spliceRow(0); // [3, 4]
mat.switchRows(0, 1); // [3, 4; 1, 2]
mat.transpose; // [1, 3; 2, 4]
mat.dim(); // [2, 2]
```

## 3.2 Strings

Jtrix will support the basic string operations: indexing, concatenation, changing the case of the string (uppercase or lowercase), and splitting. Since strings are immutable in Jtrix, each of these operations will return a new string instead of modifying the string input.

| Operation | Result |
|---|---|
| str[n] | Returns the n-th character of str as a string |
| str1.concat(str2) | Returns a new string concatenating str1 and str2 |
| str.toUpper() | Returns a string equal to str with all the letters in uppercase |
| str.toLower() | Returns a string equal to str with all the letters in lowercase |
| str.split(n) | Returns a substring of str from the first character to the (n - 1)-th character |

```
string hello = "Hello World!";
print(hello[2]); // "l"
string bye = "Good bye!";
print(hello.concat(bye)); // "Hello World!Good bye!"
print(hello.toUpper()); // "HELLO WORLD!"
print(hello.toLower()); // "hello world!"
print(hello.split(5)); // "Hello"
```

## 3.3 Print

*print(x)* will return a string version of what x is to the standard output. For primitive types, it would return the variable after being casted to a string. For arrays, the string representation would be in the form *[ elements ]*. For matrices, the string representation would be in the form *[row n]*, with each row on a different line. Print implicitly adds a new line character at the end of the converted input.

```
int x = 5;
print(x); // 5

string hello = "Hello";
print(hello); // Hello

float[] floatArr = [1.0, 2.0, 1.5, 3.0];
print(floatArr); // [1.0, 2.0, 1.5, 3.0]

char[] charArr = ['a', 'b', 'c', 'd', 'e'];
print(charArr); // ['a', 'b', 'c', 'd', 'e']

matrix<3,3> mat3 = [1, 2, 3; 4, 5, 6; 7, 8, 9];
print(mat3);
```

```
/*
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
*/
```

---

# Chapter 4

# Declarations and Statements

Jtrix uses *declarations* to introduce an item into the code. In Jtrix, all declarations are explicit with the aim of increasing the readability of the code to help solve runtime errors. Jtrix does not implicitly type declarations which reduces compiling errors that many implicitly typed languages experience. Jtrix offers a number of `control flow` statements that are useful for iterating over data structures, executing expressions multiple times, and executing specific operations on user defined conditions.

## 4.1 Functions

Functions are statements that can be executed in Jtrix. Functions may take parameters as inputs. Primitive function parameters are passed by value in Jtrix. For other data structures like matrices, performing operations on that matrix variable will be reflected in all variables that point to that specific matrix in memory.

## 4.2 Variables

Variables are names for objects that refers to a location in the memory where they are stored.

### 4.2.1 Declaration

Variables are of the type declared before the variable name.

```
int x;
float y;
char z;
bool t;
```

## 4.3 Control Flow

### 4.3.1 While Statements

while statements are used to iterate through a set of code based on a user defined boolean expression. The statement evaluates the expression and, if the expression is true, the code within the brackets will be executed. After executing the code within the brackets, the loop will restart by reevaluating the expression and repeating this process a long as the statement is true.

```
/* this code prints y and then concatenates y with foo until y is equal to x*/
string x="foo foo foo";
string y = " " ;
string foo = "foo";
while (x !=y){
   print(y);
   y=y.concat(foo);
}

// first loop prints blank
// second loop prints "foo "
// third loop prints "foo foo "
// fourth loop prints "foo foo foo " and terminates the loop
```

### 4.3.2 For Statements

for statements are similar to while statements, however they require the variable used in the boolean statement to be initialized within the loop. Therefore the scope of the variable is only within the for loop which can be useful for writing multiple loops. for loops define 3 arguments. The first initializes a variable, the second is the conditional statement, and the third is an expression to change the value of the variable.

```
for( int i = 0; i < 5 ; i = i + 1 ){
   //some code
   }
```

### 4.3.3 If-Else Statements

if-else statements consist of a condition and a serious of statements. The series of statements are evaluated only if the condition is True, otherwise, the program will continue unless an optional else clause is added.

```
if(x=6){
   print(x);
}
else{
```

```
   print(6);
}
```

# Chapter 5

# Memory

Jtrix memory allocation works similar to Java, in that memory allocation and deallocation is done automatically. Garbage collection works though a simple reference-counting mechanism that deallocated memory with no references. Thus, variable declarations automatically assign variables to a memory resource. When the variable goes out of scope, that memory resource is automatically freed. Variables are generally stored on the stack, with the exception of matrices. Because matrices contain subarrays (i.e. matrix.rows or matrix.cols), they are stored on the heap. This dynamic allocation allows for matrices to be passed along threads easily.