Abed Tony BenBrahim
ba2305@columbia.edu

JTemplate

Language Reference Manual

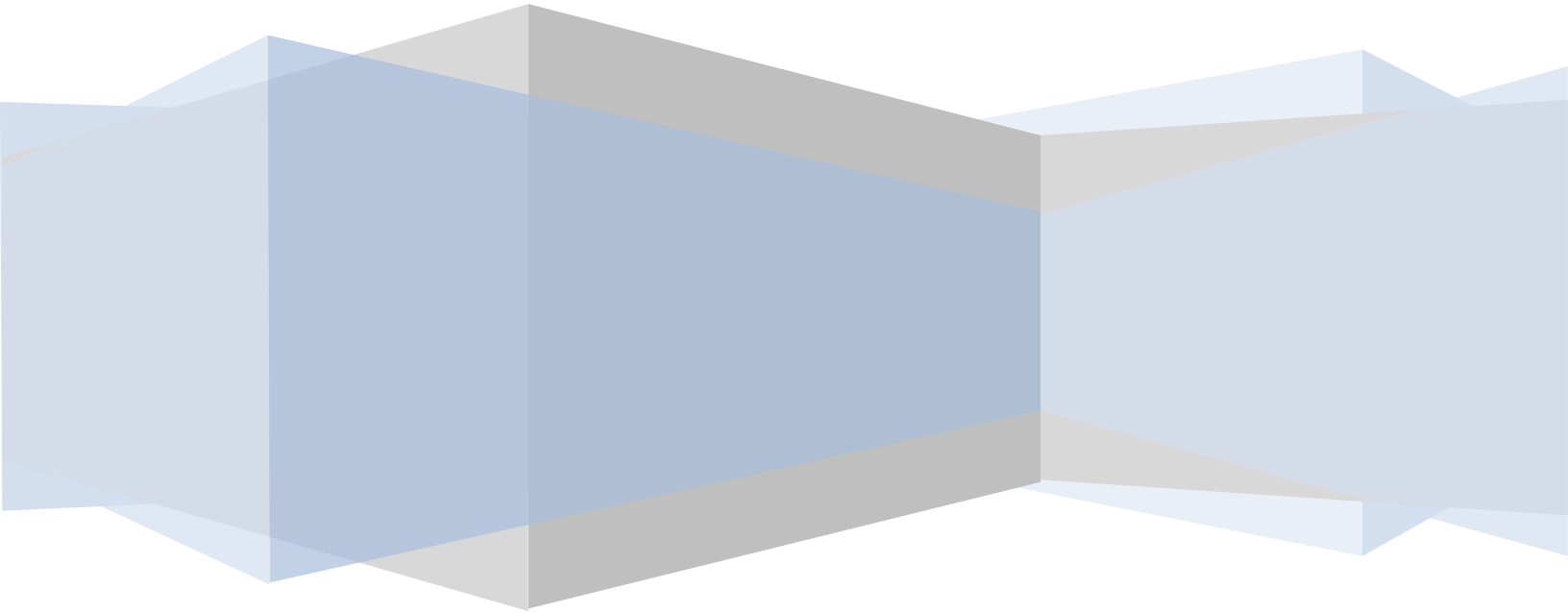

Table of Contents

1 Introduction

Jtemplate is a dynamically typed language meant to facilitate the generation of text from template definitions. Jtemplate's support for prototypal inheritance, functions as first class values and a basic library permits the development of robust applications. While Jtemplate bears a strong resemblance to ECMAScript, there are number of significant differences that should be noted. Stronger type checking (for example, addition of a function and an integer, valid in ECMAScript, is not valid in Jtemplate) , mandatory declaration of variables before they are used, the different implementation of prototypal inheritance and singular implementation of varargs and partial function application, as well as improvements in scope visibility make Jtemplate quite distinct from ECMAScript.

2 Lexical Convention

2.1 Character set and whitespace

Programs in Jtemplate are written using the ASCII character set. Whitespace characters serve to separate language elements, except within strings and comments, and consist of spaces, tabs, carriage returns and newlines.

2.2 Comments

Multiline comments begin with the first character sequence `/*` and end with the first character sequence */ that is encountered.

Single line comments begin with the character sequence `//` and end at the end of the line

Example:

```
/*
 * This is a multiline comment
 */
let i=1; // this is single line comment
```

2.3 Identifiers

Identifiers begin with an uppercase or lowercase letter, an underscore (_) or a dollar sign ($) symbol. Following the first character, identifiers may optionally contain any number of uppercase or lowercase letters, digits 0 through 9, underscores or dollar signs. The following reserved keywords may *not* be used as identifiers:

```
break case catch continue default else false finally for foreach
function import in instructions let NaN once return use switch
throw true try template var Void when while
```

Identifiers in Jtemplate are case sensitive. The identifiers foo, Foo and FOO represent three different identifiers

2.4 Values

A value in Jtemplate assumes one of the following types: integer, float, string, Boolean, function, array, map, NaN or Void.

2.4.1 Integer

Integers are composed one or more digits, to form a whole number. A single optional minus (-) sign may precede the integer to negate its value. Integers may be in the range of -1073741824 to 1073741823 inclusive

2.4.2 Floating Point Number

Jtemplate supports IEEE-754 like double precision floating point numbers. Floating point numbers consist of:

- An optional minus sign (-) that negates the value
- A significand consisting of either or the sequence of both of :
    - An integer
    - A decimal point (.) followed by an integer, representing the fractional part
- An optional exponent part consisting of the character e, followed by an optional + or – sign, followed by an integer

Either the exponent or fractional part of the significand (or both) must be specified to form a valid floating point number. The following are examples of valid floating point numbers:

```
0.123 1.23 148.23e-32 1.e+12 1e+12
```

Jtemplate differs from IEEE-754 in its treatment of NaN, infinity and –inifinity, which are all converted to the non float value NaN when they occur (Section 2.4.8).

2.4.3 String

Strings represent a sequence of ASCII characters. Strings start with either a single quote or double quote delimiter, and are terminated by the first non escaped matching delimiter. Strings may span several lines, and any newline spanned becomes part of the string. Non printable characters or string delimiters may be embedded in a string by using the following escape sequences:

\b	backspace
\n	Newline
\r	carriage return
\t	Tab
\’	single quote
\”	double quote
\\	Backslash

The following are examples of valid strings

```
'a string'
'a multiline
string'
"another string"
```

2.4.4 Boolean

Boolean values represent the logical values true and false. Boolean values consist of the two values `true` and `false`.

2.4.5 Function

A function represents a group of statements that can be invoked with an optional list of arguments and returns a value upon completion. Functions are defined with the `function` keyword, followed by a parenthesized list of zero or more identifiers, followed by a statement block:

`function (`*arglist*`)` *statement_block*

arglist is a comma separated list of zero or more identifiers.

statement_block begins with an opening brace (`{`), ends with a closing brace (`}`), and contains zero or more statements. Statements are fully described in section 4.1.

Example:

`function(){`*statements**`}` a function with no arguments

`function(x,y){`*statements**`}` a function with two arguments, x and y

The last identifier may optionally be followed by three periods, to indicate that the function accepts any number of values for the last argument, all of which will be placed in an array with the same name as the identifier.

Example:

`function println(items...){`*statement**`}` a function with any number of arguments (including none). When this function is invoked, any parameters will be passed in array `items`.

`function printat(x,y,strings...){`statement*`}` a function with at least two arguments x and y, and optionally any number of arguments that will be placed in an array named `strings`. For example, if this function is invoked with five arguments, the first will be bound to `x`, the second to `y`, the third, fourth and fifth will be added to an array bound to the identifier `strings`.

2.4.6 Array

An array represents an ordered list of values. Array values begin a left bracket (`[`), contain a comma delimited list of zero or more expressions (described in section 3), and end with a right bracket (`]`).

Example:

`[1,2,3]`	an array with 3 integer values
`[]`	an empty array
`[1,1.2,'abc',function(x,y){return x+y;}]`	an array with an integer, float, string and function value

2.4.7 Map

A map represents a container where a collection of values can each be associated with a key. Map values begin with an opening brace (`{`), contain a list of zero or more properties, and end with a closing brace (`}`). A property consist of an identifier, followed by a colon (`:`), followed by a value.

Example:

{}	An empty map
`{x:10,y:120,name:'test'}`	A map where `x` is associated to the integer 10, `y` to the integer 120 and `name` to the string 'test'
`{add:function(x,y){return x+y;}, subtract:function(x,y){return x-y;},x:10}`	A map with two functions, one mapped to the key `add` and one mapped to the key `subtract`, and an integer 10 mapped to the key `x`

Note than unlike other values which can be used anywhere where an expression can be used, the value for an empty map (`{}`) can only be used to the right of an assignment or declaration statement (Section 3.5), as function call arguments (Section 3.8), as property values in maps and as array elements.

2.4.8 NaN

`NaN` is the type of values that indicate that a value is not a number. There is a single eponymous value for this type, `NaN`. An example use of NaN is in a function that converts a float to an integer, to indicate that the passed in value could not be converted to a float. NaN is also used in floating point operations to indicate that the result is invalid, such as when dividing by zero.

2.4.9 Void

Void is the type of values that indicate the absence of a value. There is a single eponymous value for this type, `Void`. Void is the value returned from functions that do not explicitly return a value, and may also be used in other contexts to indicate the absence of a value.

3 Expressions

Values can be combined with operators and other expressions to form expressions.

3.1 Values

Values are expressions, and may be used interchangeably with expressions where expressions are indicated in the rest of this manual, noting the exception for empty maps described in section 2.8.

3.2 Variables and 'left hand side' expressions

Left hand expressions are locations where values can be stored. Left hand expressions can be variables identified by an identifier, members of maps (Section 3.8) or array members (Section 3.7). The following are examples of left hand side expressions:

`foo`	A variable named `foo`
`foo.x`	member x of map `foo`
`bar[10]`	The eleventh element of array `bar`

3.3 Arithmetic expressions

3.3.1 Binary arithmetic expressions

Arithmetic expressions operate on two expressions with an operator to create a new value. The format of an operation is *expression operator expression* where operator is one of

+	addition
−	subtraction
*	multiplication
/	division
%	modulo

The resulting value of evaluating an arithmetic expression depends on the type of the expressions and the operator, as shown in the table below (without regard to the ordering of the expressions):

Value 1 type	Value 2 type	Operator	Result
integer	Integer	+ − * / %	integer result of operation
float	float	+ − * /	float result of operation
float	integer		
string	any type	+ only	concatenation of first value to second value, with non string values converted to strings

Any combination of value types and operators not listed above, such as adding two Booleans, results in a runtime error.

3.3.2 Unary arithmetic expressions

The minus (−) operator, when prefixing an expression, serves to negate the expression it precedes. The minus operator can only be applied to expressions that evaluate to integer or float values.

Example: `-1` `-a`

3.4 Comparison expressions

3.4.1 Binary comparison expressions

Binary comparison expressions compare two expressions with a comparison operator, and evaluate to a Boolean value indicating whether the comparison is true or false. The format of a comparison expression is:

expression operator expression

where *operator* is one of:

<	Less than
<=	Less than or equal
==	Equal
!=	Not equal
>	Greater than
>=	Greater than or equal

Allowable comparison expression types, operator and their result are as follows

Value 1 type	Value 2 type	Operator	Result
Integer	Integer	Any	Comparison of integer values
Float	Float	Any	Comparison of float values
Float	Integer		
String	any type	`==` and `!=` only	comparison of first value to second value, with non string values converted to strings
Both types are Booleans, maps, arrays, functions, NaN or void		`==` and `!=` only	comparison of first value to second value, observing the semantics of equality described below.
Different types not listed above		`==` and `!=` only	Always returns false

There are special semantics of equality for maps, arrays and functions:

- Arrays a and b are equal if a and b have the same number of elements, and if for each index i in 0<=i<length, a[i]=b[i]
- Maps a and b are equal if a and b have the same keys, and if for each key k, a[k]=b[k]
- Functions a and b are equal if a and b have the same formal arguments, with the same name at the same position in the argument list, and have the same list of statements.

3.4.2 Ternary comparison expressions

Ternary expressions consist of an expression which when evaluated yields a Boolean value, causing one of two expressions to be evaluated:

Boolean-expression `?` *expression_if_true* `:` *expression_if_false*

Example

```
a<10 ? 'small' : 'large' //evaluates to the string 'small' if a<10
                         // or 'large' if a>=10
```

3.5 Logical expressions

Logical expressions operate on two Boolean expressions with an operator, or on a single expression with a unary operator, to produce a single Boolean value. The format of a logical expression is:

expression `and` *expression*	`true` if both expressions evaluate to true, `false` otherwise
expression `or` *expression*	`true` if either expression evaluates to true, `false` otherwise
`not` *expression*	`true` if *expression* evaluates to false, `false` otherwise

Attempting to apply logical operators to expressions that are not of Boolean type results in a runtime error.

3.6 Declaration and assignment expressions

3.6.1 Declarations

Declarations bind a value *and its type* to a left hand side expression. Declarations consist of the keyword let (or its synonym var, a tribute to JTemplate's ECMAScript legacy), followed by a left hand side exception, followed by the equals sign, followed by an expression:

`let` lhs_expression = value

or

`var` lhs_expression = value

Example:

```
let x=1;        // binds identifier x to 1
let a=[1,2,3]; //binds identifier a to the array value [1,2,3]
let a[2]='abc'; // rebinds the third element of a to string 'abc'
```

Declaration expressions evaluate to the value of the expression on the right hand side of the equals sign, so that one can chain declarations.

Example:

```
let x=let y=let z=0;  // declare x y and z as integers
                      // initialized to 0
```

3.6.2 Assignment

Assignment changes the value of a previous declared left hand side expression. Once a left hand side expression has been declared, it is bound to a type and can only be assigned a value of the same type, unless it is re-declared with another value of a different type.

Example:

```
let a=[1,2,3]; //binds a to the array value [1,2,3]
a[2]=10 + a[2]; // assigns 12 to the third element of a
```

Attempting to assign a value to a left hand side expression that has not been previously declared, or attempting to assign a value of a type different than the type of the left hand side expression, results in runtime error.

Assignment expressions evaluate to the value of the expression on the right hand side of the equals sign, so that one can chain assignments.

Example:

```
x=y=z=0;  //set x y and z to 0
```

3.6.3 Combined arithmetic operation and assignment expressions

Combined arithmetic operation and assignment expressions perform an operation then assign a result to the left hand side expressions. These expressions exist in both binary and unary form.

3.6.3.1 Binary form

The format of a binary combined arithmetic operation and assignment expression is

lhs_expression operator = expression

where *lhs_expression* is valid left hand side expression. The *lhs_expression* and the *expression* are evaluated with the *operator* using the same semantics described in section 3.3, with the result assigned to the *lhs_exception*. The expression evaluates to the result of the assignment.

Example:

```
a+=2  // equivalent to a=a+2
```

3.6.3.2 Unary form

Unary combined arithmetic operation and assignment expressions increment or decrement a left hand side expression, assigning the result to the left hand side expression. These expressions exist in both postfix and prefix forms, which affect the value evaluated by the expression:

Syntax	Operation	Evaluates to
`++ lhs_expression`	Increment `lhs_expression`	`lhs_expression` after it has been incremented
`lhs_expression ++`	Increment `lhs_expression`	`lhs_expression` before it has been incremented
`-- lhs_expression`	Decrement `lhs_expression`	`lhs_expression` after it has been decremented
`lhs_expression --`	Decrement `lhs_expression`	`lhs_expression` before it has been incremented

Example:

```
let a=0
let b = ++a   // both a and b are 1
let b = a++   // a is 2, b is 1, a's value before it was incremented
a++           // a is 3
++a           // a is 4
```

Note that the prefix form is more efficient and should be used in favor of the postfix form when there is no semantic difference, as in the last two examples above.

3.7 Index expressions

Index expressions are used to access a member of an array or a map value, or an expression that evaluates to an array or a map. Index expressions begin with an expression, followed by an opening bracket (`[`), contain an expression and end with a closing bracket (`]`):

expression `[`*expression*`]`

When applied to a map expression, the index can evaluate to a string or integer. When applied to an array, the index must evaluate to an integer. In both cases, the index must exist in the map or array when the index expression is used on the right hand side of an expression or when used on the left side of an assignment.

Example:

```
let arr=[1,2,'abc']; //define array arr
let b=arr[0];        // declare b and initialize
                     // with the first element of arr
let c=1;
b=arr[c];                //assign the second element of arr to b

let m={a:124,point:{x:10,y: 120}};  // define map m
let p=m['a'];                       // initialize p with member a of m
let name='x';
let q=m['point'][name];             // initialize q with member x of
                                    // map point in map m
```

3.8 Member expressions

Member expressions are used to access a member of a map value, or an expression that evaluates to a map value . Member expressions begin with an expression, followed by a period (.) followed by an identifier

expression.expression

Example:

```
let m={a:124,point:{x:10,y: 120}};  // define map m
let p=m.a;                          // initialize p with member a of m
let q=m.point.x;                    // initialize q with member x of
                                    // map point in map m
```

3.9 Function Calls

3.9.1 Function Invocation

Function calls cause the statements in a previously defined function value to be executed, first initializing the function value's formal parameters with the arguments passed in to the function call. The format of a function call is an expression that evaluates to a function value, followed by an opening parenthesis, a comma separated list of zero or more expressions, followed by a closing parenthesis:

expression (*expression_list*)

Example:

```
let sum=function(a,b){return a+b; }; // declare a function
let a=sum(1,2);                    // assign 1 to a, 2 to b then
                                   // evaluate the statements in sum
println('Hello',' ','World');      //  calls function assigned to println
                                   //   defined as function(items...){}
function(a,b){return a*b;}(10,20)// calling a function definition
```

Calling a function with fewer or more arguments than are required results in a runtime error.

Function calls evaluate to the value returned by a `return` statement (Section 4.3.4.3), or to `Void` if a `return` statement is not executed before leaving the function body.

3.9.2 Partial application

Partial application, also known as currying, allows a new function to be defined from an existing function, binding some of the curried function's parameters to existing values and leaving others unbound. Partial application in JTemplate is invoked by calling the function to be curried, and indicating which parameters should remain unbound by preceding them with an at symbol (`@`). The result is a new function with the unbound parameters as the new parameters. For example, consider the function declaration of `sum` below:

```
let sum=function(a,b){return a+b; };
```

We create a new function `increment` from `sum` by leaving one the variables unbound and binding the other one to 1. We can then invoke increment with one parameter.

```
let increment = sum(@value,1);
let b=increment(10);
```

Partial application can also operate on varagrs. Consider the function println defined as

`let println=function(values...){` *library_code* `};`

We create a new function print2 that assigns the first element of values to the string `'>'`

```
let print2=println('>',@values...);
```

The print2 function has the signature `function(values...)` and when invoked, the first element of values will contain `'>'` and the succeeding elements will contain any additional parameters passed in the values `vararg`. As a result, any line that is printed by the `print2` function will be prefixed with the '>' character.

3.10 Grouping

Expressions can be grouped with parentheses to control the order of evaluation.

Example:

```
(2+3)*10     // evaluates to 50
2+(3*10)     // evaluates to 32
```

3.11 Operator precedence

When expressing are not grouped as shown in section 3.10, Jtemplate uses operator precedence rules to determine the order in which operations are performed. For example, in the expression `2+3*10`, the interpreter needs a rule to decide whether to evaluate the multiplication or addition first.

Expressions are evaluated using the operator precedence rules listed below, with the highest precedence listed at the top:

Expression	Operators	Section
Member expressions, index expressions	`.  []`	3.7, 3.8
Function calls	`()  ()[]  ().`	3.9
Postfix expression	`++  --`	3.6.3.2
Prefix expression	`++  --`	3.6.3.2
Negation	`-`	3.3.2
Logical Not	`!`	3.5
Multiplication/Division/Modulo	`*  /  %`	3.3.1
Addition/Subtraction	`+  -`	3.3.1
Comparison	`<  <=  >  >=  ==  !=`	3.4.1
Logical operator	`&&  ||`	3.5
Ternary comparison	`?:`	3.4.2
Assignment/Declaration	`=`	3.6
Arithmetic assignment	`*=  /=  %=`	3.6.3
Arithmetic assignment	`+=  -=`	3.6.3

When operators with the same precedence are encountered, they are evaluated from left to right, except for negation, logical not , arithmetic assignment, assignment and declaration which evaluates from right to left.

Example:

```
println(7-2-2);//associates to the left, equivalent to (7-2)-2
let a=let b=1; //associates to the right, equivalent to let a=(let b=1)
println(a-=b-=2); //associates to the right, equivalent to a-=(b-=2)
```

4 Statements

A program is composed of zero or more statements, all of which are described in this section.

4.1 Statement Blocks

A statement block is a group of statements with an inner scope (Scope is discussed in section 5). A statement block starts with an opening brace ({) , contain zero or more statements, and end with a closing brace(}).

Example

```
{                           // a statement block
      let a=1;
      let b=2;
}
{}                          // an empty statement block
```

4.2 Expressions

Expressions can be used as statements, when they are suffixed with a semicolon:

expression ;

Example:

```
let a=1;      //this expression assigns 1 to a
1+2;          //this expression has no side effect
;             //an empty expression, again with no side effect
```

Any expression can be used as a statement, except for the empty map (`{}`), as noted in section 2.3.7. Expressions which do not cause a value to be assigned or evaluate a value that is used by a subsequent statement, such as second and third expression in the example above have no side effect and serve no purpose.

4.3 Iteration statements

An iteration statement causes a statement or statement block to be executed repeatedly until a condition has been met.

4.3.1 `for` loops

For loops consist of an optional initializer expression, an optional conditional expression, and an optional counting expression, arranged as follows:

`for` (*initializer_expression?* ; *conditional_expression?*; *counting_expression?*)
statement_or_statement_block

Execution of a `for` statement proceeds as follows:

1. The initializer expression is evaluated, if is present.
2. The conditional expression is evaluated if it is present. If it is not present, the missing expression is substituted with `true`. If the expression evaluates to false, the for loop statement is completed and execution proceeds to the next statement.
3. If the expression evaluates to true, the statement or statement block is executed one time.
4. The counting expression is evaluated, if it is present
5. Execution proceeds to step 2.

This control flow can be interrupted by a break, continue and return statements, as described in section 4.3.4, and by exceptions, as described in section 4.5.

Example:

```
for(var i=0; i<10; ++i){
      println('Hello');
      println('World!');
}
```

4.3.2 `while` loops

While statements consist of a conditional expression and a statement or statement block arranged as follows:

`while` (*conditional_expression*)
    statement_or_statement_block

Execution of a `while` statement proceeds as follows:

1. The conditional expression is evaluated. If it evaluates to false, the `while` loop statement is completed and execution proceeds to the next statement.
2. If the expression evaluates to true, the statement or statement block is executed one time.
3. Execution proceeds to step 1.

This control flow can be interrupted by a break, continue and return statements, as described in section 4.3.4, and by exceptions, as described in section 4.5.

Example:

```
let i=0;
while(i<10){
      println('Hello');
      println('World!');
      ++i;
}
```

4.3.3 `foreach` loops

`foreach` statements execute a statement or statement block once for each element in an expression that evaluates to a map or an array. The syntax of a `foreach` statement is:

`foreach` (*element* `in` *collection*) *statement_or_statement_block*

where element is a left hand side expression, collection is an expression that evaluates to a map or array type.

If the collection expression evaluates to an array, the element expression is assigned with successive elements of the array, starting with the first element and proceeding in sequence, before each execution of the associated statement or statement block. Execution of the statement ends after execution of the statement or statement block for the last element of the array.

If the collection expression evaluates to a map, the element expression is assigned with successive property values of the map, presented in no particular order, before each execution of the associated statement or statement block. Execution of the statement ends after execution of the statement or statement block for the last value in the map.

Attempting to execute a `foreach` statement with an expression that cannot be assigned or with a type that is not a collection results in a runtime error. The control flow for a `foreach` statement can be interrupted by a break, continue and return statements, as described in section 4.3.4, and by exceptions, as described in section 4.5.

Example:

```
let a=[1,'abc',2];
foreach(el in a) println(el);
let b={a:1,b:'xyz'};
foreach (el in b) println(el);
```

Output:

```
1
abc
2
xyz
1
```

4.3.4 Altering loop statements control flow

The iteration in the loop statements described earlier in this section can be interrupted or modified by the following instructions.

4.3.4.1 `break` statement

The `break` statement causes the loop statement to terminate immediately. Execution resumes at the statement following the loop statement.

Example: This “inifinite” loop exists after 10 iterations.

```
let i=0;
for (;;){
      ++i;
      if (i==10) break;
}
```

Using a `break` statement outside of a loop statement or `switch` statement (Section 4.4.2) results in a runtime error.

4.3.4.2 `continue` statement

The continue statement causes execution to be skipped for all statements in a statement block following the invocation of continue, and causes the loop statement to proceed to the next iteration.

Example: Only even numbers are printed

```
for(var i=0;i<10;++i){
      if (i%2==1) continue;
      println(i);
}
```

Using a `continue` statement outside of a loop statement results in a runtime error.

4.3.4.3 `return` statements

The `return` statement, used exclusively inside a function definition's function body, causes the execution of a function call to stop and immediately evaluate to the expression specified by the return statement. The syntax of a return statement is

`return` *expression?* `;`

where the expression is optional. If the expression is not specified, `return` returns `Void`.

Example:

```
let sum=function(x,y){return x+y;}

let foo=function(){
      let x=10;
      for (var i=0;i<10;++i){
            x+=i;
            if (x>10) return i;
      }
}
```

Invoking `return` outside of a function body results in a runtime error.

4.4 Conditional statements

A conditional statement conditionally executes a statement or statement block based on the evaluation of an expression.

4.4.1 `if` statement

The `if` statement executes one of two statements or statement blocks based on the evaluation of a conditional expression. An `if` statement starts with the `if` keyword followed by a parenthesized conditional expression, followed by a statement or statement block, and optionally followed by the `else` keyword and another statement or statement block:

`if (`*cond-expression*`)` *statement_or statement_block*

`if (`*cond-expression*`)` *statement_or statement_block* `else` *statement_or statement_block*

An `if` statement's conditional expression is evaluated, and if it evaluates to true, the succeeding statement or statement block is executed. If it evaluates to false, and an `else` clause is present, the statement or statement block following the else keyword is executed. If the expression does not evaluate to a Boolean expression, a runtime error occurs.

Example: prints `a is equal to b`

```
let a=let b=10;
if (a>b)
      println('a is greater than b');
else
      if (a<b)
            println('a is smaller than b');
      else
            println('a is equal to b');
```

4.4.2 `switch` statement

The switch statement executes statements following a case label containing an expression that is equal to an expression being compared. A `switch` statement consists of the `switch` keyword, followed by a parenthesized expression, followed by a statement block. The statement block consists of the statements described in this section, and special `case` statements, consisting of the `case` keyword followed by an expression followed by a colon, or the `default` keyword followed by a colon:

```
switch(expression){
      case case_expression:
            statements
      case case_expression:
            statements
      default:
            statements
}
```

When a `switch` statement is executed, the expression is first evaluated. Then the statement block is executed, by looking for the first `case` statement where the evaluation of the associated `case` expression is equal to the `switch` value, using the rules of equality defined in section 3.4.1. The special `default` case statement, if encountered, matches any value. If a matching case statement is encountered, all statements following the case statement are executed, until the end of the statement block (any case statement encountered is not evaluated), or until a `break` or `return` instruction interrupts the flow of control.

Example:

```
let arith=function(x,op,y){
      switch(op){
      case '+':
            println(x,op,y,' is ',x+y);
            break;
      case '*':
            println(x,op,y,' is ',x*y);
            break;
      default:
            println('Only addition and multiplication are supported');
      }
};
```

Note that using a case statement outside of a switch statement results in a parsing error.

4.5 exception handling & recovery statements

Exception handling and recovery statements provide a structured way to deal with errors.

4.5.1 `throw` statement

The `throw` statement interrupts the normal flow of control by raising an error. Following the execution of a `throw` statement, execution of the program terminates, or if the `throw` statement is executed inside a `try/catch` block (Section 4.5.2), execution proceeds to the catch clause of the `try/catch` block. A `throw` statement consists of the `throw` keyword, followed by an expression, followed by a semicolon:

`throw` *expression*`;`

Example:

```
let safeDivide=function (x,y){
      if (y==0)
            throw 'Division by 0';
      return x/y;
};
```

4.5.2 `try/catch` block

A `try/catch` block allows a statement to catch exceptions thrown by a `throw` statement, or internally by the Jtemplate interpreter. A `try/catch` block starts with the `try` keyword, followed by a statement block (the try block), followed by the `catch` keyword, followed by a parenthesized identifier, followed by another statement block (the catch block):

```
try{
      statements
}catch(identifier){
      statements
}
```

A `try/catch` block executes by executing the statements in the try block. If an error does not occur, the catch block is never executed. If an exception is thrown, the exception is assigned to the catch block identifier and the statements in the catch block are executed.

Example:

Without a `try/catch` block, executing the following statements result in a runtime error, because a Boolean cannot be added to a float. This causes the program to terminate.

```
let safeAdd=function(x,y){
      return x+y;
};
println('the result is ',safeAdd(true,1.2));
```

With a `try/catch` block, the error is intercepted inside the function and an alternate value (`Void`) is returned:

```
let safeAdd=function(x,y){
      try{
            return x+y;
      }catch(e){
            return Void;
      }
};
println('the result is ',safeAdd(true,1.2));
```

4.5.3 try/finally block

A `try/finally` block ensures that a block of code always execute, even in the presence of an exception that would normally immediately terminate program execution. A `try/finally` block starts with the `try` keyword, followed by a statement block (the try block), followed by the `finally` keyword, followed by another statement block (the finally block):

```
try{
      statements
}finally{
      statements
}
```

When a `try/finally` block is encountered, all statements in the try block are executed. Then the statements in the finally block execute, even if the execution of the try block caused an exception to be thrown. If an exception was thrown by the try block, it is handled after the finally block has executed.

Example:

```
try{
      println('entering the try block');
      let a=1.2+true;
}finally{
      println('entering the finally block');
}
```

Output:

```
entering the try block
entering the finally block
At line 5 in file lrm_samples.jtp: uncaught exception
Interpreter.Interpreter.EIncompatibleTypes("float", "boolean")
```

Note that even though an exception is thrown in the try block, the code in the finally block executes before execution terminates. `try/finally` blocks are frequently nested inside of `try/catch` blocks, to ensure that a specific action is performed prior to handling the error.

Example:

```
try{
      try{
            println('entering the try block');
            let a=1.2+true;
      }finally{
            println('entering the finally block');
      }
}catch(e){
      println('Something went wrong: ',e);
}
```

Output:

```
entering the try block
entering the finally block
Something went wrong:
Interpreter.Interpreter.EIncompatibleTypes("float", "boolean")
```

4.6 Importing definitions

As a program grows larger, if may be useful to separate the program's code into separate modules. It may also be useful to develop modules of functionality than can be reused in other projects. The `import` keyword enables this, by allowing declarations to be imported from another program file. The `import` statement consists of the `import` keyword, followed by a string specifying the file's location, followed by a semicolon:

`import '`*path/to/file*`';`

The path that is specified can either be relative or absolute. When an `import` statement is encountered, the path is normalized to an absolute path. Jtemplate then determines if the file has previously been imported, and if it has already been loaded, execution of the statement terminates and execution proceeds to the next statement. If the file has not previously been imported, the file is loaded and executed, executing *only* declaration expression statements and import statements, ignoring all other statement types.

Since imported files are only loaded one time, circular references are allowed. For example, foo.jtp can import bar.jtp, and bar.jtp can import foo.jtp, since foo.jtp will not be imported again.

Example:

myfile.jtp contains:

```
let multiplicationSign=function(a,b){
      if (a==0 || b==0)
            return 0;
      else
            if ((a>0 && b>0) || (a<0 && b<0))
                  return 1;
            else if ((a>0 && b<0)||(a<0 && b>0))
                  return -1;
};

println('This statement will not be executed when the file is
imported');
```

sample.jtp contains:

```
import 'myfile.jtp';

println('The sign is ',multiplicationSign(-19,-20));
```

myfile.jtp declares the function `multiplicationSign`. Any file that imports myfile.jtp can use the function as if it had been included in the same file.

4.7 Template statements

Template statements allow the generation of strings from a template definition and associated instructions.

4.7.1 template statement

A template defines a labeled block of text than can later be manipulated by processing instructions (Section 4.7.2). A template statement consists of the `template` keyword, followed by an identifier specifying the template name, followed by an opening bracket (`{`) , then zero or more line specifications, and closed by a closing bracket (`}`). A line specification consists of an optional identifier or integer that serves as a label for the processing instructions, followed by a hash sign (#) indicating the start of line, followed by text. A line specification ends when the end of line is reached:

`template` *template_name* `{`
*line_specification**
`}`

where line_specification is *label?* #text

Example:

```
template htmlTable{
           #<table>
           #<tr>
header     #<th>columnLabel</th>
           #</tr>
row        #<tr>
cell       #<td>cellData</td>
row        #</tr>
           #</table>
}
```

Note that nesting can be defined by repeating the labels. In the example above, the line labeled `cell` is nested between two lines labeled `row`. Specifying an illegal nesting structure, such as A B A B, where B is nested inside A, and A is nested inside B, results in a runtime error.

4.7.2 instructions statement

4.7.2.1 Instruction definition

An instruction statement defines how a template definition is turned into a string. An instruction statement starts with the `instructions` keyword, followed by the `for` keyword, followed by an identifier referencing the name of a template definition, followed by a parenthesized list of arguments using the same convention as for a function definition, followed by an opening brace (`{`), a list of zero or more processing instructions terminated with a semicolon, followed by a closing brace (`}`):

`instructions for` *template_name* `(`*arglist*`){`
processing_instructions **;**
`}`

A processing instruction consists of a label matching a label in the template definition, followed by a processing condition, followed by a colon (**:**), followed by a comma separated list of one more replacement specification.

label processing_condition **:** *replacement_specifications*

A processing condition takes one of the following forms:

Processing Condition	Action
`always`	always perform the replacements specified in the replacement specifications
`when (`*condition_expression*`)`	Perform the replacements specified in the replacement specification only if the *condition_expression* evaluates to true
`foreach (`*element* `in` *collection*`)`	Perform the replacements specified in the replacement specification, looping through each element of the map or array *collection*.
`foreach (`*element* `in` *collection*`)` `when (`*condition_expression*`)`	Perform the replacements specified in the replacement specification, looping through each element of the map or array *collection*, only if the *condition_expression* evaluates to true for the given iteration of the loop

Finally a replacement condition consists of an identifier followed by an equals sign (=) followed by an expression. The identifier is treated as a string. The expression is evaluated, then any text in the labeled template line matching the identifier is string is replaced with the value of the expression.

Example: an `instructions` statement for the template defined in 4.7.1

```
instructions for htmlTable(dataMap){
header foreach(label in dataMap.labels): columnLabel=label;
row foreach(dataArray in dataMap.data): ;
cell foreach(element in dataArray): cellData=element;
}
```

Note that any variable defined in a replacement condition is also available to nested definitions. In the example above, `dataArray` is introduced in the `row` definition, then used in the nested `cell` definition.

4.7.2.2 Invoking template instructions

Template instructions are invoked in the same manner as a function call, using the same semantics. For example, if `people` is defined as follows

```
let people={labels: ['Name','Age'], data: [['John',42], ['Mary',38]] };
```

invoking the instructions for `htmlTable` with `people` as an argument

```
let text=htmlTable(people);
```

results in the `text` variable being assigned the string

```
<table>
<tr>
<th>Name</th>
```

```
<th>Age</th>
</tr>
<tr>
<td>John</td>
<td>42</td>
</tr>
<tr>
<td>Mary</td>
<td>38</td>
</tr>
</table>
```

4.7.3 Replacement Methodology

It is useful to think of the replacements as occurring in parallel rather than serially. An example will better serve to illustrate this point.

Consider the input string '`foo bar`' given in template definition, and the replacement specification '`foo=x, bar=y`' given in a template instruction. If x has the value 'bar' and y has the value 'foo', performing the replacement serially would yield the string 'bar bar' after the first replacement and 'foo foo' after the second replacement. When the replacement is performed 'in parallel', the first foo is replaced with bar by the first replacement and bar is replace with foo by the second replacement, yielding the final string 'bar foo'.

To accomplish this, the offset of all replacements for each replacement condition in calculated before any replacements are performed. If any overlapping regions are detected, a runtime error occurs. As each replacement occurs in order, the offsets of the subsequent replacements are adjusted based on whether characters where added or replaced by the previous replacement.

Example: For the input string 'foo bar foo bar', and the replacement 'foo=x, bar=y', the following offsets are calculated, then sorted:

- Replace starting at 0 of length 3 with value of x
- Replace starting at 4 of length 3 with value of y
- Replace starting at 8 of length 3 with value of x
- Replace starting at 12 of length 3 with value of y

If x evaluates to 'a', after the first replacement, the subsequent offsets will be adjusted by 2 positions leftward, since the input string is now 'a bar foo bar'

- Replace starting at 2 of length 3 with value of y
- Replace starting at 6 of length 3 with value of x
- Replace starting at 10 of length 3 with value of y

5 Scope

Scope defines which variables are visible to the statement being executed. Scope is hierarchical, with declarations in an inner scope not visible to statements in an outer scope. Variables declared in an inner scope disappear as soon as the inner scope is exited.

5.1 Program level scope

When a program starts, all declarations occur in the top level scope. Any function declared in a scope can see functions declared in the same scope, even if the definition of the second function occurs after the definition of the first definition.

Example:

```
let odd=function(n){
      return n==0? false: even(n-1);
};

let even=function(n){
      return n==0? true: odd(n-1);
};
```

In the mutual recursion example above, the `odd` function body accesses the `even` function, even though it is declared after the `odd` function.

5.2 Statement block scope

A statement block, whether alone, or following a statement such as if or foreach, starts a new scope. Variables declared in the statement block are not visible to statements outside the statement block. Variables declared in a statement block with the same name as a variable declared outside the statement block is a separate variable, even if the name is the same.

Example:

```
let a=1;
print(a,' ');
{
      print(a,' ');
      let a=2;
      print(a,' ');
}
println(a,' ');
```

outputs 1 1 2 1

Note that the declaration of variable `a` inside the statement block does not affect the originally defined variable `a`, which maintains its value of 1.

5.3 Function scope

JTemplate is a lexically scoped language. As such, any variable reference inside a function statement that does reference an argument name or a variable declared inside the function body resolves to a variable in the same scope as the function definition.

Example:

```
let printX=function(){println(x);};

let x=0;
{
      let x=1;
      printX();
}
```

outputs 0, because the `printX` function sees the variable `x` defined in the same scope, not the value of `x` in the inner scope when `printX` was invoked.

6 Object Oriented Constructs

Jtemplate supports calling methods defined in map, in a manner reminiscent of an object oriented language. For example, the following map contains a function `test`, which can be invoked using a member expression:

```
let x={a:1, test: function(){println('hello world');}};
x.test();
```

The member notation has been extended to support calling member functions for non map types, and implementing flexible function dispatch for map types. For example, the runtime library exposes many operations on string types that are invoked as if they were a member function of the string itself. For example, to calculate the length of a string `myString`, the following statements would be invoked:

```
let myString='hello world';
let len=myString.length();
```

Note that a member function is invoked on a string.

6.1 Prototypes

6.1.1 Semantics for non map types

When a member function is invoked on an expression, the type of the expression is first determined. If the expression is not a map, the function is looked up in the type's prototype, which is simply a map containing functions for that type. Each type has a prototype, with zero or more functions, as shown below:

Type	Prototype
Integer	Integer.prototype
Float	Float.prototype
String	String.prototype
Boolean	Boolean.prototype
Array	Array.prototype
Map	Map.prototype
Void	Void.prototype
NaN	NaN.prototype
Function	Function.prototype

If the function is found in the prototype, it is invoked, passing the caller as an argument named `this`. Note that `this` is not part of a function's formal arguments. A type's prototype can be extended at run time, providing methods that can be invoked for all expressions of that type.

Example 1: extending the array type with a join method that concatenates all the elements as a string, and in this example, outputs `12abc1.2`.

```
let Array.prototype.join=function(){
      let result='';
      foreach(el in this) result+=el;
      return result;
};

let a=[1,2,'abc',1.2];
println(a.join());
```

Example 2: extending the array type with a map function, which takes a function as an argument and returns a new array with the function applied to each element of the array, and a clone function, which uses the map function to return a copy of the array:

```
let Array.prototype.map=function(f){
      let newArray=[];
      foreach(element in this)
            newArray.push(f(element));
      return newArray;
};
var Array.prototype.clone=function(){
      return this.map(function(x){return x;});
};
```

6.1.2 Semantics for map types

Since the map type has the ability to store member functions, or declare its own member map named `prototype`, resolving member functions for maps is more complex, and proceeds in the following order:

1. Try to find the definition for the function in the map. If it is found, it is invoked as a normal function call, in particular, `this` is assigned to Void. (`map.func()`, essentially a static invocation)
2. If the map has a map member named `prototype`, try to find the function definition in the map and invoke it, passing the caller in a variable named `this`. (`map.prototype.func()`)
3. If the map has a map member named `prototype`, and it contains a map with a prototype member, try to find the function definition in the map and invoke it, passing the caller in a variable named `this`. (`map.prototype.prototype.func()`)
4. Lookup the function is the `Map.prototype` map. If it is found, invoke it, passing the caller in a variable named `this`. (`Map.prototype.func()`)

Example:

```
let m={
          foo:         function(){print('hello');},
          prototype: {bar: function(){this.foo();println(' again');}}
       };
m.foo();println(); //using case 1
m.bar();           //using case 2
let a=m.keys();    //using case 4
```

The interesting use cases comes with case 3 above, which allows for a single level of inheritance.

Example:

```
let Foo={prototype: {print: function(){println('your value is
',this.value); }}};
let m={value:10, prototype: Foo};
m.print();              //using case 3
```

Here the declaration of `Foo` creates a new type that exposes function `print`. Any map with a prototype assigned to `Foo` can invoke functions of Foo's prototype as if it were a member function of the map.

A more involved example: The built in library exposes a Date map with a single(static) function, now(), which returns a map mimicking a tm structure, and with its prototype set to Date. This allows Date to be extended to add a toString() member function.

```
var Date.days =
['Sunday','Monday','Tuesday','Wednesday','Thursday','Friday','Saturday'
];

var Date.prototype={};

var Date.prototype.toString=function(){
    var offset='';
    if (this.gmtOffset>0){
        offset='+'+this.gmtOffset;
    }else{
```

```
        offset=this.gmtOffset+'';
    }
    return Date.days[this.dayOfWeek]+'
'+this.month+'/'+this.dayOfMonth+'/'+
        this.year+'
'+this.hour+':'+(this.minute+'').padLeft(2,'0')+':'+
        (this.second+'').padLeft(2,'0')+' (GMT'+ offset+')';
};
```

where `padLeft` is defined as

```
let String.prototype.padLeft=function(len,pad){
    var result=this+''; //cast to string
    while (result.length()<len){
        result=pad+result;
    }
    return result;
};
```

The expression `Date.now().toString()` evaluates to a string such as 'Sunday 6/21/2009 10:27:14 (GMT-5)'

6.2 Supporting multiple levels of inheritance

As noted in the previous section, prototypal inheritance only supports one level of inheritance. However, adding support for multiple levels can easily be accomplished. We begin by defining an object as a map and its prototype map and add a prototype function extend. Object will serve as the base class for all objects:

```
let Object={
      prototype: {
            extend: function(){
                  let obj={prototype:{}};
                  foreach(key in this.prototype.keys())
                        let obj.prototype[key]=this.prototype[key];
                  return obj;
            }
      }
};
```

As an example we then create a class Foo that extends from object, and introduces a new method foo():

```
let Foo=Object.extend();
let Foo.prototype.foo=function(){
      println('foo!');
};
```

We then define a new class Bar that extends Foo, and a new class Fun that extends Bar overriding foo in both classes and introducing a new method bar in the Bar class. We implement the calling of the inherited method by using the library function `Function.prototype.apply`, which lets us call an arbitrary map member method while passing an arbitrary `this` object

```
let Bar=Foo.extend();
let Bar.prototype.foo=function(){
      print('bar ');
      Foo.prototype.foo.apply(this);
};

let Bar.prototype.bar=function(){
      println('bar!');
};

let Fun=Bar.extend();
let Fun.prototype.foo=function(){
      print('fun ');
      Bar.prototype.foo.apply(this);
};
```

Finally, we create objects of each type, and invoke their methods. The method of constructing new objects is a little contrived, a matter that will be dealt with in the next section:

```
let foo={prototype: Foo};
let bar={prototype: Bar};
let fun={prototype: Fun};
print('foo.foo(): ');foo.foo();
print('bar.bar(): ');bar.bar();
print('bar.foo(): ');bar.foo();
print('fun.bar(): ');fun.bar();
print('fun.foo(): ');fun.foo();
```

which outputs:

```
foo.foo(): foo!
bar.bar(): bar!
bar.foo(): bar foo!
fun.bar(): bar!
fun.foo(): fun bar foo!
```

6.3 Implementing constructors

Jtemplate does not natively support a `new` constructor. However, the Object definition above can trivially be extended to provided a new() method:

```
let Object={
      prototype: {
            extend: function(){
                  let obj={prototype:{}};
                  foreach(key in this.prototype.keys())
                        let obj.prototype[key]=this.prototype[key];
                  return obj;
            },
            new: function(){
                  return {prototype: this };
            }
      }
```

```
};
```

Now the foo and bar objects in the previous example can be constructed using this new method

```
let fun=Fun.new();
fun.bar();
fun.foo();
```

which outputs:

```
bar!
fun bar foo!
```

In the same way that methods were overridden, constructors can be overridden. Overriding constructors lets us add parameters to the constructor and as importantly, define fields for our object.

Example:

```
let Point=Object.extend();
let Point.prototype.new=function(x,y){
      let point= Object.prototype.new.apply(this);
      let point.x=x;
      let point.y=y;
      return point;
};
let Point.prototype.print=function(){
      println('x: ',this.x,', y: ',this.y);
};

let p=Point.new(42,10);
print('p.print(): ');p.print();

let ThreeDPoint=Point.extend();
let ThreeDPoint.prototype.new=function(x,y,z){
      let point= Point.prototype.new.apply(this,x,y);
      let point.z=z;
      return point;
};
let ThreeDPoint.prototype.print=function(){
      print('z: ',this.z,', ');
      Point.prototype.print.apply(this);
};

let p3=ThreeDPoint.new(5,42,10);
print('p3.print(): ');p3.print();
```

Output:

```
p.print(): x: 42, y: 10
p3.print(): z: 10, x: 5, y: 42
```

Note how the ThreeDPoint class inherited members x and y, by calling the overridden Point constructor in its constructor and passing x and y.

7 Built in Library

7.1 Built in variables

7.1.1 Command line arguments

When a program is run, the program name and any arguments after the program name are placed in an array named `args`. `args`[0] will contain the program name, `args`[1] the first argument if present and so on.

7.1.2 Environment variables

When a program is run, the environment variable names and values are placed in a map named `env`, with environment variable names as keys and environment variable values as key values.

7.2 System Library

The system library contains functions to deal with Jtemplate native types, as well as functions to deal with the operating system environment.

Signature	Description
`Array.prototype.push(value)`	Adds `value` to the end of the caller array, returns `Void`
`Array.prototype.pop()`	Removes the last element from the caller array and returns the element that was removed
`Array.prototype.length()`	Returns the length of the caller array as an integer
`Map.prototype.remove(key)`	Removes `key` and its associated value from the caller map, returns `Void`.
`Map.prototype.contains(key)`	Returns true if `key` exists in the caller map, false otherwise.
`Map.prototype.keys()`	Returns an array with the caller's keys
`Integer.random(upperBound)`	Returns a pseudo random number between 0 and `upperBound-1` inclusive
`Float.prototype.round()`	Returns an integer with the float caller rounded up if the fractional part >0.5, rounded down otherwise
`Date.now()`	Returns a map representing today's date, with the following keys and values: `gmtOffset`: offset from GMT (integer) `second`: number of seconds in the time 0-59 (integer) `minute`: number of minutes in the time 0-59 (integer) `hour`: number of hours in the time 0-23 (integer) `dayofMonth`: number of day in the month 1-31 (integer) `month`: number of month in year 1-12 (integer) `year`: today's year (integer) `dayOfWeek`: today's day index relative to the week, starting at 0 for Sunday 0-6 (integer) `dayOfYear`: today's day index relative to the start of the year, starting at 0 for the first day of the year 0-366

	(integer) `dst`: true if daylight savings time is in effect, false otherwise (Boolean)
`Function.prototype.apply(this,args...)`	Calls caller map member function, passing `this` as parameter this, and passing any additional arguments specified in `args`
`typeof(value)`	Returns a string representing the type 's value, one of `string, integer, boolean, float, function, map, array, NaN, Void`
`System.command(command)`	Executes the external command specified by `command`, waits for execution to complete and returns an integer representing the exit code of the command.
`exit(exitcode)`	Causes the program to exit with exit code `exitcode`, which must be in the range -127 to 127 inclusive.
`Debug.dumpSymbolTable(incl)`	Dumps the symbol table to stdout, including library functions if `incl` is true
`Debug.dumpStackTrace()`	Dumps the current stack trace to stdout

7.3 String Library

The String library contains functions to perform string manipulation.

Signature	Description
`String.prototype.toUppercase()`	Returns a new string with every character in the string caller uppercased
`String.prototype.toLowercase()`	Returns a new string with every character in the string caller lowercased
`String.prototype.toFirstUpper()`	Returns a new string with the string caller's first letter uppercased
`String.prototype.toFirstLower()`	Returns a new string with the string caller's first letter lowercased
`String.prototype.length()`	Returns the length of the string caller as an integer
`String.prototype.charAt(index)`	Returns a new string with the character at the string caller's position indicated by `index`, with 0 indicating the first character. Throws a runtime error if the index is less than 0 or greater or equal to the string's length
`String.prototype.indexOf(substr)`	Returns an index representing the leftmost position of substring `substr` in the string caller, or -1 if the substring is not found.
`String.prototype.substr(st,len)`	Returns the substring in the string caller starting at position `st` of length `len`. Throw
`String.prototype.startsWith(substr)`	Returns true if the string caller starts with the substring `substr`, false otherwise
`String.prototype.endsWith(substr)`	Returns true if the string caller ends with the substring `substr`, false otherwise
`String.prototype.replaceAll(`	Returns a new string with every occurrence of

`substring, replacement)`	`substring` in the string caller replaced with `replacement`.
`String.prototype.split(sep)`	Returns an array containing the substrings in the string caller that are delimited by `sep`
`String.prototype.parseInt()`	Returns an integer parsed from the string caller, or Void if the string does not represent a valid integer
`String.prototype.parseFloat()`	Returns a float parsed from the string caller, or Void if the string does not represent a valid float

7.4 I/O Library

The I/O Library contains functions to deal with input and output to the console and file system, as well as functions to manipulate the file system.

Signature	Description
`print(value...)`	Prints the arguments to stdout
`println(value...)`	Prints the arguments to stdout, followed by a newline after the last argument
`readln()`	Returns a string with a line read from stdin
`File.openForWriting(handle, filename)`	Opens file `filename` for writing and associates the file handle with string `handle`.
`File.openForReading(handle, filename`	Opens file `filename` for reading and associates the file handle with string `handle`.
`File.close(handle)`	Closes a previously opened file, using the string `handle` associated with the file handle.
`File.write(handle,value...)`	Writes the `value` arguments (automatically cast to a string) to a file associated with the string `handle`
`File.writeln(handle,value...)`	Writes the `value` arguments (automatically cast to a string) to a file associated with the string `handle`, then writes a newline after the last argument is written.
`File.readln(handle)`	Returns a string read from a file associated with the string `handle`, previously opened for reading
`File.eof(handle)`	Returns true if the file previously opened for reading associated with the string `handle` has reached end of file.
`File.exists(filename)`	Returns true if `filename` exists, false otherwise
`File.delete(filename)`	Deletes `filename`, returns true if the file was successfully deleted, false otherwise
`File.rename(oldname,newname)`	Renames the file named `oldname` to name `newname`, returns true if the file was renamed, false otherwise.
`Directory.exists(dirname)`	Returns true if the directory named `dirname` exists, false otherwise
`Directory.delete(dirname)`	Deletes the directory named `dirname`, returns true if the directory was successfully deleted, false

	otherwise
`Directory.list(dirname)`	Returns an array containing every file and directory contained in the directory `dirname`
`Directory.create(dirname)`	Creates directory `dirname`, returns true if the directory was successfully created, false otherwise

Example:

This program takes two arguments, a source text file and a destination text file, and copies the source text file to the destination text file.

```
if (args.length() != 3) {
      println('Usage: ', args[0], ' source_file destination_file');
      exit(-1);
}
if (!File.exists(args[1])) {
      println('File ', args[1], ' does not exist.');
      exit(-1);
}
File.openForReading('in', args[1]);
try {
      File.openForWriting('out', args[2]);
      try {
            let lines = 0;
            while (!File.eof('in')) {
                  let s = File.readln('in');
                  File.writeln('out', s);
                  ++lines;
            }
            println(lines, ' lines copied');
      } finally {
            File.close('out');
      }
} finally {
      File.close('in');
}
```