TEA USER MANUAL

AUGUST 27, 2001

PLEASE SEND ALL COMMENTS TO: OPENSOURCE@DIG.COM

PREFACE

This manual describes the basic functionality of Tea for Java developers and template authors.

There are a number of other documents complementary to this manual that cover topics associated with Tea and Tea-related tools. For additional Tea-related documentation, please visit http://opensource.go.com.

DOCUMENT REVISION AND EVOLUTION

Primary Author(s)	Description of Version	Date
Randall Rader	Initial public release.	06-30-2001
Michael Rathjen	Minor update.	07-12-2000
Michael Rathjen	Minor corrections.	08-31-2000
Michael Rathjen	Minor corrections.	10-05-2000
Michael Rathjen	Updated loop information for Tea 3.2.0.	08-27-2001

CONTENTS

1. INTRODUCTION 4

2. TEMPLATES 5

2.1 CODE REGION 5
2.2 TEMPLATE DECLARATION 5
2.3 PARAMETER LIST 5
2.4 TEXT REGION 5
2.5 COMMENTS 5

3. DATA TYPES 7

3.1 NUMBERS 7
3.1.1 Long Integers 7
3.1.2 Leading Digits 7
3.2 STRINGS 7
3.2.1 Escape Characters 8
3.3 BOOLEANS 8
3.4 OBJECTS 8

4. DATA OPERATIONS 9

4.1 VARIABLES 9
4.2 ACCESSING PROPERTIES AND ARRAYS 9
4.2.1 Non-indexed 9
4.2.2 Indexed 9
4.3 DEFINING CUSTOM ARRAYS 10
4.4 STRING CONCATENATION 11
4.5 ARITHMETIC 11
4.6 CALLING FUNCTIONS 12

5. PAGE OUTPUT 13

5.1 EXPRESSIONS 13
5.2 INVOKING TEMPLATES 13

6. FLOW CONTROL 15

6.1 LOOPING (FOREACH) 15
6.2 IF/ELSE 16
6.3 RELATIONAL TESTS 16
6.3.1 Equality Tests: 17
6.3.2 Comparison Tests: 17

6.3.3 isa Tests 17
6.4 LOGICAL OPERATORS 17

7. SAMPLE TEA TEMPLATES 18

7.1 SAMPLE 1 - DEPARTMENT TEMPLATE 18
7.2 SAMPLE 2 - EMPLOYEE TEMPLATE 18

1. INTRODUCTION

Tea is a template language with a simple syntax, which supports the notion of having code inside an ordinary html page, like an ASP.

Tea templates are compiled into Java bytecode, and are loaded into the Java Virtual Machine as if they were ordinary Java classes.

This manual is intended as a language reference only, and does not give instruction on template writing techniques.

2. TEMPLATES

Tea templates are made up of **code regions**, which contain simple programming instructions, and **text regions**, which include no programming and are used to output raw text.

2.1 Code Region

Code regions are delimited by the symbols **<%** and **%>**. As the name implies, code regions are for calling functions and performing assorted logical operations, so line entries are expected to be variables, logical operators, or functions. Text and html formatting are included in code regions by enclosing the strings in quotation marks.

Every template must begin with a template declaration, which is defined in a code region. Therefore, every template must begin with a code region.

2.2 Template Declaration

The template declaration defines the name of the template and any parameters that must be passed to the template. Tea templates must be saved in a file whose name is the same in the declaration and have the extension ".tea".

```
<% template TemplateName %>
```

2.3 Parameter List

The template's formal parameter list follows the template name in the declaration and must be delimited by parentheses. A comma separates each parameter in the list. Formal parameters require a class name followed by an identifier, which names it. This is the only place in a Tea template where a class name is used.

```
<% template TemplateName (ClassName referenceIdentifier) %>
```

2.4 Text Region

Any text not in the code region is output by the template as-is with only one minor conversion. All line break separator codes in text regions are converted to linefeeds `'\n'` (ASCII 10). Use html `<BR>` or `<P>` tags to add line breaks manually. Text and html in text regions do not need to be enclosed within quotation marks, and reserved characters do not need to be escaped.

2.5 Comments

Single line comments in Tea templates follow a double slash:

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

Multi-line comments are wrapped in slashes and asterisks as follows:

```
/*
* This is a multi-line comment.
* Multi-line comments are often used to
* provide basic header information,
```

```
* such as the purpose of the template,
* and the template designer's name.
*/
```

3. DATA TYPES

Tea is designed to work within a Java-based hosting system. All data that is accessible by Tea, provided by those systems, is in a form understood by Java.

Java has two kinds of data representations, objects and primitives, which are also understood by Tea. In Tea, however, all data should be treated as objects.

Tea data types include **numbers**, **strings**, **booleans**, and **objects**.

3.1 Numbers

Tea uses the Java standards for numbers (see Java Primitive Types and Values), with the following exceptions:

3.1.1 Long Integers

Long integers are not required to append the letter "L" after the number, as in Java code.

Example:

```
922337203685477580L
```

This may also be written in Tea as

```
922337203685477580
```

3.1.2 Leading Digits

Numbers with decimals (floating-point types) must be written with leading digits before the decimal point.

Example:

```
.8
```

This is not a valid number in Tea. The valid number is written as

```
0.8
```

3.2 Strings

Templates are usually written to perform textual output, so they often operate on strings, which are simply strings of characters. Tea can automatically convert all data into string representations when an operation is applied that requires it to be a string. These operations include concatenation, printing, and relational tests.

In Tea, strings must be enclosed within quotation marks, even when used as parameters in function calls. Either single quotation marks or double quotation marks may be used, so long as the open quote style matches the close quote style.

3.2.1 Escape Characters

In some situations, you may need to use characters where they don't fit with the syntax of the template. In this case, you need to escape the characters with a backslash.

\0	null character
\b	backspace
\t	tab
\n	newline
\f	formfeed
\r	carriage return
\'	single quote
\"	double quote
\uXXXX	unicode escape, where XXXX is a hexadecimal number
\\	backslash

Table 1 – Escape Characters

Example:

```
"<a href=\"index.html\">"
'The decade of the 1990\'s!'
```

In the above examples, double quotes are escaped in the first example, a single quote is escaped in the second example. The results of these examples would be:

```
<a href="index.html"> The decade of the 1990's!
```

3.3 Booleans

Booleans have values of *true* or *false*, and are used to perform logical tests. For more information on performing logical tests, see Relational Tests and Logical Operators.

3.4 Objects

Objects are passed to or used in templates through JavaBeans. Objects contain references to other data, including object properties and other objects.

For more information on working with objects and object properties, see Accessing Properties and Arrays.

4. DATA OPERATIONS

4.1 Variables

With the exception of passed-in parameters, Tea variables are not declared. Rather, their type is inferred based on assignment. A variable's type is dynamic, and can change after another assignment. Passed-in parameters behave like ordinary local variables in that they can be re-assigned, and they can change type.

Assignments can only be made to variables. Array elements cannot be assigned values, like they can in Java. Assignments of this form are not allowed: `a[i] = x`. This restriction makes it possible to pass collections to templates without fear of modification.

Examples:

Assign the string literal "Hello" to *message*:

```
message = "Hello"
```

Assign the integer literal 50 to *result*:

```
result = 50
```

Assign the calculated sum of *result* and 20:

```
amount = result + 20
```

Re-assign *message* with the integer *amount*:

```
message = amount
```

4.2 Accessing Properties and Arrays

Objects are passed to templates through JavaBeans. JavaBeans defines two types of object properties, **non-indexed** and **indexed**. Most property accesses are non-indexed.

4.2.1 Non-indexed

Tea can access all non-indexed properties on a JavaBean.

4.2.2 Indexed

JavaBeans defines several different forms for indexed properties, but Tea only understands one kind: unnamed indexed properties.

A property access is called a **lookup**, and Tea has different syntax for its two supported kinds of lookups. A lookup can be applied to any expression, but it's usually applied to variables or chained to other lookups. Non-indexed properties are looked up from an expression using a dot (.) followed by the property name. Indexed properties are looked up using an expression bounded by square brackets.

Examples:

Access the *name* object from *user*:

```
name = user.name
```

Access the *first* name:

```
f = name.first
```

Access the *last* name by chaining:

```
last = user.name.last
```

Access an element from the *users* array, and get the *age*:

```
users[10].age
```

Lookup a team object by code:

```
team = allTeams["SEA"]
```

Get the last character from a string:

```
str[str.length - 1]
```

4.3 Defining Custom Arrays

Tea templates can define read-only, pre-initialized arrays. Arrays created by this expression can be indexed by numerical position (zero based) or by an object key. An array that uses an object key is called an **associative array**. The array values can have any data type, and they can be composed of either constants or any expression. Associative arrays can have any type used for their keys, but usually strings or numbers will be used.

Based on the type of elements defined in the array, a common type is inferred. For any object retrieved from the array, the only properties that are accessible are those defined for the common type.

Arrays defined in templates can be used in the same way as an array passed to a template. This means that elements are accessed using array property syntax. (i.e., `x[i]`) In the case of indexed arrays, they have a length property and can be used in a foreach statement.

The array creation expression is prefixed with a single hash (`#`) to denote an indexed array, and a double hash (`##`) to denote an associative array. The list of elements is comma-delimited and is contained inside parentheses. Associative arrays must have an even number of elements in the list, and the list pattern is key, value, key, value, etc.

Examples:

An indexed array:

```
vowels = #("a", "e", "i", "o", "u")
```

Loop through an array:

```
foreach (word in #("The", "quick", "brown", "fox", "jumped")) {
    "Word: " & word & "<br>"
}
```

Loop over a set of numbers:

```
areaCodes = #(408, 404, 425, 312, 310, 212, 415, 860)
foreach (code in areaCodes) {
```

```
    "Area code is: " & code
}
```

An associative array - map area codes to locations:

```
map = ##(408, "Sunnyvale",
    404, "Atlanta",
    425, "Seattle",
    312, "Chicago",
    310, "Los Angeles",
    212, "New York",
    415, "San Francisco",
    860, "Bristol")
"<title>" & map[location.code] & "</title>"
```

4.4 String Concatenation

String concatenation combines two or more strings together to form a new one. String concatenation in Tea uses the ampersand operator.

Examples:

Equivalent to text = "Hello World":

```
text = "Hello" & " " & 'World'
```

Combining strings with arithmetic:

```
y = 5
'498234 divided by "y" is ' & 498234 / y
Output: 498234 divided by "y" is 99646
```

4.5 Arithmetic

Tea can perform arithmetic on both integers and floating-point numbers. Supported operations include:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (division remainder)
-	Negate

Table 2 – Arithmetic Operations

In complex arithmetic expressions, negate is performed first, left-to-right. Multiplication, division, and modulus are performed next, followed by addition and subtraction.

Examples:

Simple addition:

```
x = a + b
```

Increment a count:

```
count = count + 1
```

Complex example:

```
result = -((value - 10.0/y) * 50) % 2
```

4.6 Calling Functions

Built-in Java functions are called as follows.

Syntax:

```
<name> (<parameter list>) <block>
```

The parameter list contains 0 or more comma-delimited expressions. The block part of the call statement is optional, and is used to pass a block to a built-in Java function, but only if the function supports it.

Functions are defined in a template's runtime context class. The template has no control over what context it receives. It is the responsibility of the hosting system to provide one. A default context, which includes a set of "common" functions, is defined for use with the TeaServlet. See the *TeaServlet User's Manual* for a list of these functions. If you are not using the TeaServlet, see the *Tea Template Language* document for a list of basic functions.

Example:

Convert to lowercase:

```
message = toLowerCase(message)
```

5. PAGE OUTPUT

As stated in the Templates section, any text not in the code region is output by the template as-is, with only one minor conversion. All line break separator codes in text regions are converted to linefeeds `'\n'` (ASCII 10).

In Tea, using semi-colons to separate statements is optional, and is only necessary in rare cases to clarify statement separation. The following two statements

```
a
-b
```

would not be interpreted as printing the value of variable "a" followed by the negative value of variable "b". Instead it would be

```
a - b
```

which means print the value of "a" minus "b". If the "-" token was not overloaded to mean both minus and negate, this would not be a problem. Here is the disambiguated version:

```
a;
-b
```

5.1 Expressions

Expressions that are not passed as parameters or assigned to variables are output to the page. Therefore

```
var = "hello"
```

produces no output, but

```
var
```

or

```
"hello"
```

by themselves, would both output to the page as:

```
hello
```

In Tea, expressions are evaluated in a specific order, and parentheses can be used to override that ordering. A sub-expression bounded by parentheses is evaluated first within a sub-expression. Otherwise, the order of operations is the same as that for Java, where applicable.

5.2 Invoking Templates

Other templates are invoked using the **call** statement.

Syntax:

```
call <name> (<parameter list>) <block>
```

The optional block part of the call statement is used to pass a **substitution block** to another template.

A template may require that it receive one substitution block parameter. This parameter is denoted by the ellipsis symbol (**...**). With this parameter, any template invoking this template must pass in a block of code. A block of code begins with a left curly brace and ends with a right curly brace. The template that accepts the substitution parameter uses the substitution statement to insert the substitution.

A template can be called by its short name or its full name. A template's short name is the same as defined in its declaration. Its full name is based on the root directory from which it is stored. Essentially, templates can be packaged, but templates don't have package declarations. The full name of a template declared as "header" in the "common" directory is named "common.header". When a template invokes another template within the same package, the full name is not required.

Examples:

Invoke a template named "header" in the same package as this one:

```
call header("Header text")
```

Invoke a template using its full name:

```
call common.header("Header text")
```

Invoke a template and pass a substitution block:

```
name = "John"
call bigtext() {
if (name = = "John") {
    "Unauthorized!"
    }
else {
    "Welcome, " & name & "."
    }
}
```

The block declaration and substitution statements in the *bigtext* template are as follows:

```
<% template bigtext() {...} %>
<% ... %>
```

6. FLOW CONTROL

Flow control in Tea is accomplished through looping, using the **foreach** command, and through logical tests, using **if/else**.

6.1 Looping (foreach)

Syntax:

```
foreach ( <Variable in Expression> ) {<Do this block>}

foreach ( <Variable in Expression> reverse ) {<Do this block>}

foreach ( <Variable in Range Expression .. Expression> ) {<Do this block>}

foreach ( <Variable in Range Expression .. Expression> reverse ) {<Do this
block>}
```

Description:

The **foreach** statement is the only loop flow control statement in Tea. There is no way to modify the loop count variable from within the loop, but the **break** statement can be used to exit the loop. Foreach statements iterate over the values in an array, or a **range** of values, indicated by "**..**". When the optional **reverse** keyword is specified, the values are iterated in reverse order.

Parameters:

- variable
- first expression
- second expression
- reverse

Examples:

Loop through and print every item in the page's list:

```
foreach (item in page.list) { item }
```

Loop through in reverse:

```
foreach (item in page.list reverse) { item }
```

Loop from 1 to 10, inclusive:

```
foreach (count in 1..10) { "Count is " & count }
```

Print all the characters of a string:

Note: If the message length is zero, the range is 0..-1. Whenever the second value in the range is less than the first value, the foreach will loop zero times.

```
message = "hello"
foreach (index in 0..message.length - 1) {
    "Letter at index " & index & " is: " & message[index] & "\n"
}
```

Loop from 10 to 0, inclusive:

```
foreach (count in 0..10 reverse) { ... }
```

6.2 If/Else

Syntax:

```
if ( <Expression> ) {<Do this block>}

if ( <Expression> ) {<Do this block>} else {<Do this block>}

if ( <Expression> ) {<Do this block>} else if ( <Expression> ) {<Do this
block>}
```

Description:

Tea's if statement is just like Java's except that braces are always required for Tea's enclosed statements.

Parameters:

- expression
- block

Examples:

```
if (fileName == null) {
    fileName = "default"
}

if (param.length == 5) {
    if (param == "World") {
        isWorld = true
    }
else {
     isWorld = false
     }
}
```

6.3 Relational Tests

Tea supports expressions for performing relational tests. Relational tests are used primarily with if statement conditions, although their result can be assigned to a variable. The result of a relational test is always Booleans.

Six operators are supported, falling into two categories: equality tests and comparison tests.

6.3.1 Equality Tests:

`= =`	equals
`!=`	is not equal to

6.3.2 Comparison Tests:

`<`	is less than
`>`	is greater than
`<=`	is less than or equal to
`>=`	is greater than or equal to

6.3.3 isa Tests

The "isa" (read "is a") operator is a special operator used to identify specific content types. Identifying content types allows for referencing the unique properties of different content types from within a single collection. Designed to be combined with an **if** statement, where any operations on the left variable in the **then** part are automatically cast to the given type.

Examples:

```
if (content isa app.pkg.News) {
    "Story is: " & content.story
}

if (item isa app.pkg.Feature and content isa app.pkg.News) {
    "Info is: " & item.info
    "Story is: " & content.story
}
```

6.4 Logical Operators

Logical operations are often used in conjunction with relational tests because they only operate on booleans. Tea's logical operators are:

```
and
or
not
```

7. SAMPLE TEA TEMPLATES

The following are two simple browser-friendly examples of Tea templates designed to illustrate template declaration and basic flow control logic.

7.1 Sample 1 - Department Template

The Department template prints the department title property, then loops through an array of employee names and IDs to create a list of individual links to employee pages.

```
<% template DepartmentPage(hr.Department dept)

dept.title "<P>"

foreach (employee in dept.employees) {
        '<a href="Request.dll?EMP&ID=' & employee.ID & '">'
        employee.firstName & " " & employee.lastName
        '</a>'
        '<br>'
}

%>
```

Here's what the output looks like:

Corporate Relations

Jane Employee 1
Joe Employee 2

7.2 Sample 2 - Employee Template

The Employee template prints a list of employee properties. Note the usage for the dateFormat function.

```
<% template EmployeePage(hr.Employee emp)

emp.firstName & " " & emp.lastName " " & "-" & " "
    & emp.title "<P>"
"Phone Number:" & " " emp.phoneNumber "<p>"
"Department:" & " " emp.department.title "<p>"

dateFormat("MMMMM d, yyyy")

"Hire Date:" & " " emp.hireDate "<p>"
"Salary:" & " " emp.salary "<p>"

%>
```

Here's what the output looks like:

Jane Director – Director
Phone Number: 555-xxxx
Department: Corporate Relations
Hire Date: January 1, 2000
Salary: TBD