

# Programming with Tcl & jTcl

## 1.1. Scope of the document

This document is part of the JTCL project set of documentation, it aims at describing how to write application programs with the Tcl language and its object-oriented extension: jTcl. No previous knowledge of either jTcl or Tcl is required in order to understand what is explained in this document, but we make the assumption that the reader has a background in programming techniques in general and object-oriented programming in particular.

The jTcl language is the base language used for the JTCL tool-kit. This tool-kit consists of a set of classes useful for building client/server applications, as described in [1].

This document has the following structure:

- Chapter 2 – **Installation**: explains how to install the Tcl, jTcl and all JTCL packages on the different target platforms.
- Chapter 3 – **Fundamentals of the Tcl language**: aims to give the basic notions of programming under that language for newcomers to Tcl.
- Chapter 4 – **The jTcl extension**: explains how to use classes in Tcl thanks to the jTcl object-oriented extension.
- Chapter 5 – **Study of an example in jTcl**: gives an example of program implementing a few toy classes in order to illustrate the concepts explained in the previous chapter.

## Table of contents

<b>Programming with Tcl &amp; jTcl.....</b>	<b>1</b>
1.1. Scope of the document.....	1
Table of contents.....	2
1.2. Glossary.....	4
1.3. Bibliographic references.....	4
1.4. Typographic conventions.....	4
1.5. Trademarks and copyrights.....	5
 <b>2. Installation.....</b>	 <b>6</b>
2.1. Installation of Tcl and Tk.....	6
2.2. Installation of jTcl and JTCL packages.....	7
 <b>3. Fundamentals of the Tcl language.....</b>	 <b>8</b>
3.1. Overview of the language.....	8
3.1.1. Commands.....	8
3.1.2. Evaluation cycle of a command.....	8
3.1.3. Grouping.....	9
3.1.4. Character substitution.....	9
3.1.5. Variable substitution.....	10
3.1.6. Command substitution.....	10
3.2. Data structures.....	11
3.2.1. Character strings.....	11
3.2.2. Arrays.....	11
3.2.3. Lists.....	12
3.3. Control structures.....	13
3.3.1. The if condition.....	13
3.3.2. The switch condition.....	14
3.3.3. The while loop.....	14
3.3.4. The foreach loop.....	15
3.3.5. The for loop.....	15
3.3.6. The break and continue commands.....	16
3.3.7. Comments.....	16
3.4. Procedure definition.....	16
3.4.1. The proc command.....	16
3.4.2. Optional arguments.....	17
3.4.3. Variable argument list.....	17
3.4.4. Global variables.....	18
3.4.5. The upvar command.....	18
3.5. Examples.....	18
3.5.1. Factorial in recursive mode.....	18
3.5.2. Factorial in loop mode.....	19
3.5.3. Search for a palindrome.....	19

<b>4. The jTcl extension.....</b>	<b>21</b>
4.1. What is jTcl.....	21
4.2. New keywords.....	21
4.3. How to use jTcl.....	23
4.4. jTcl syntax.....	25
4.4.1. Comments.....	25
4.4.2. Class declaration.....	25
4.4.3. Instance variables.....	26
4.4.4. Class variables.....	26
4.4.5. Methods.....	27
4.4.6. Constructors.....	27
4.4.7. Destructors.....	27
4.5. The Object class.....	28
 <b>5. Study of an example in jTcl.....</b>	 <b>29</b>
5.2. The Gum class.....	29
5.3. The FlavorGum class.....	31
5.4. The MachineGum class.....	33

## 1.2. Glossary

The following terms and abbreviations are used throughout this document.

<b>jTcl</b>	<u>J</u> ava-like <u>T</u> ool <u>C</u> ommand <u>L</u> anguage, an object-oriented extension to the Tcl language. Note that, although it bears the name “Java-like”, its syntax is still Tcl-ish and has nothing to do with Java.
<b>Tcl</b>	<u>T</u> ool <u>C</u> ommand <u>L</u> anguage, a free platform-independent scripting language designed in the late 1980's by Prof. John Ousterhout of the University of California, Berkeley.
<b>Tk</b>	<u>T</u> ool <u>K</u> it, a graphical tool-set (also free and platform-independent) associated with Tcl.

## 1.3. Bibliographic references

The following references are used throughout this document.

- [2] Brent B. Welch  
Practical Programming in Tcl and Tk, 2<sup>nd</sup> edition  
Prentice Hall, 1997  
ISBN 0-13-616830-2
- [3] <ftp://ftp.sunlabs.com/pub/tcl/>
- [4] <http://www.sunscript.com/>
- [8] <ftp://ftp.atd.ucar.edu/pub/vxworks/vxtcl8.0-diff>

## 1.4. Typographic conventions

The following typographic conventions apply throughout this document.

- Names of Tcl keywords are printed with a typewriter-like font, for example: “the `e l s e` part of an `i f` command is optional”.
- In extract of Tcl code, the characters typed at the shell prompt are shown preceded by the percent character (which is the default prompt character for the Tcl shell). If a single command is spawn onto several lines, each continuing line is preceded by the “>” character. Output from the shell is shown preceded by the “⇒” character.

## 1.5. Trademarks and copyrights

Windows, Windows 3.x, Windows 95, Windows NT are trademarks from Microsoft Corporation.

Macintosh, MacOS are trademarks from Apple Computers.

Tcl/Tk is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., and other parties.

jTcl is copyrighted by FRIDU, a free software company, South Brittany University and other parties.

All other marks are trademarks of their respective owners.

## 2. Installation

The purpose of this chapter is to explain how to install the Tcl and jTcl distribution packages. All the files needed for the installation are provided with the distribution CD of the JTCL software. In addition to being available on the CD, Tcl and Tk software can also be freely downloaded by ftp from [3] or by http from [4].

### 2.1. Installation of Tcl and Tk

Pre-compiled releases are available for the following Windows and Macintosh platforms.

- Windows 3.1, Windows 95 and Windows NT self-extracting installer (tcl80p2.exe) (about 1,7 Mbytes). The file is a self-extracting executable. It will install the Tcl and Tk libraries, the wish and tclsh programs, and documentation. This file works on Windows 3.1 if you have the Win32s subsystem installed. If you use OS/2, you need an older version of Win32s.
- Macintosh 68K and PowerPC self-extracting installer (mactk8.0p2.sea.hqx) (about 3,1 Mbytes). The file is in binhex format, which is understood by Fetch, StuffIt, and many other Mac utilities. The unpacked file is a self-installing executable: double-click on it and it will create a folder containing all that you need to run Tcl and Tk.

If you're running on a platform other than the ones listed above, or if you want to make modifications to Tcl and Tk, you'll need to retrieve the source releases. These are available separately for Tcl and Tk in several different forms.

- Windows source releases:
  - Tcl sources for Windows (tcl80p2.zip): zip file (about 1,8 Mbytes). This is identical to the Tcl file listed under UNIX sources, but just in ZIP format instead of compressed tar.
  - Tk sources for windows (tk80p2.zip): zip file (about 2,4 Mbytes). This is identical to the Tk file listed under UNIX sources, but just in ZIP format instead of compressed tar.
- Macintosh source releases (These files are in binhex format, which is understood by Fetch, StuffIt, and many other Mac utilities. The unpacked file is a self-installing executable: double-click on it and it will create a folder containing sources files used to build Tcl and Tk):
  - Macintosh Tcl sources (mactcl-source-8.0p2.sea.hqx): contains the source for the Tcl language.
  - Macintosh Tk sources (mactk-source-8.0p2.sea.hqx): contains the source for the Tk toolkit.
  - Full Macintosh Source and Libraries (mactcltk-full-8.0p2.sea.hqx): contains the source for both the Tcl language and the Tk toolkit as well as the binaries and the MoreFiles package which is needed to compile the Tcl language.

- UNIX Source Releases: You'll want both Tcl and Tk sources. Choose between compressed tar and gzipped tar format. The ZIP files listed under the Windows sources contain the same information, too.
  - Tcl sources (tcl8.0p2.tar.Z): compressed tar file (about 2,4 Mbytes).
  - Tk sources (tk8.0p2.tar.Z): compressed tar file (about 3,3 Mbytes).
  - Tcl sources (tcl8.0p2.tar.gz): gzip'ed tar file (about 1,5 Mbytes).
  - Tk sources (tk8.0p2.tar.gz): gzip'ed tar file (about 2,1 Mbytes).

When you retrieve one of these files, you'll get a compressed tar file with a name like tcl8.0p2.tar.gz or tcl8.0p2.tar.Z. The files are identical except for the technique used to compress them (.gz files are generally smaller than .Z files). To unpack the distribution, invoke shell commands like the following, depending on which version of the release you retrieved:

- `gunzip -c tcl8.0p2.tar.gz | tar xf -`
- `zcat tcl8.0p2.tar.Z | tar xf -`
- `unzip tcl80p2.zip`

Each of these commands will create a directory named tcl8.0, which includes the sources for all platforms, documentation, and the script library for Tcl 8.0. If you already have a Tcl 8.0 distribution and you're upgrading to a patch release, you should rename the old source directory so that it doesn't get overwritten by the patch release. To compile and install the distribution, follow the instructions in the README file in the distribution directory. Be sure to compile Tcl before Tk, since Tk depends on information in Tcl.

Tcl and Tk should compile with little or no effort on any platform that runs a UNIX-like operating system and the X Window System. This includes workstations from Sun, HP, IBM, SGI, and DEC, PCs running a number of UNIX operating systems such as Solaris, Linux, SCO UNIX, and FreeBSD, plus many other platforms such as Cray and NEC supercomputers. These releases should also compile with little or no effort on Windows and Macintosh platforms.

In addition, third parties have created ports of Tcl/Tk to other operating systems such as VxWorks, VMS, OS/2.

## 2.2. Installation of jTcl and JTCL packages

Since both the jTcl language and the JTCL packages are entirely written in the Tcl/Tk language, they include no binaries in their distribution and their installation is therefore platform independent. The whole distribution tree (found under the jTcl root directory of the distribution CD) has to be copied anywhere on the hard disk.

Details of the distribution files are given in [6].

## 3. Fundamentals of the Tcl language

The purpose of this chapter is to provide the reader with a first taste of the programming with the Tcl language. Information given in this chapter are necessary to understand the next chapters. Nonetheless this chapter is not intended as a complete reference for the Tcl language nor as a full tutorial. A good tutorial can be found, for example, in [2]. On-line reference manuals are provided with the official distribution, see [3] or [4]. Readers who are already aware of the fundamental concepts of the language may skip the present chapter and start reading the next chapter.

### 3.1. Overview of the language

#### 3.1.1. Commands

Tcl being a scripting language, a program written in Tcl is in fact a sequence of command lines. A Tcl command line has the following syntax:

*command-name argument1 argument2 argument3 ...*

The different items are separated by one or more blanks (a blank is either a space or a tabulation character) and the command is terminated by a new-line or a semi-colon character.

```
% puts stdout Hello!  
⇒ Hello!
```

**Example 3-1:** a simple command

In the example above, puts is the command name. The puts command accepts two arguments and display the second argument to the output designed by the first one. The first argument is optional and is stdout (the standard output display) by default.

#### 3.1.2. Evaluation cycle of a command

A command is always evaluated according to the three following steps (the order matters):

- ① grouping
- ② substitution
- ③ execution

In the previous example, since neither grouping nor substitution instructions were issued, the command was executed as is.



### 3.1.3. Grouping

The grouping of several blank-separated items in a command is performed by enclosing them either by double quotes, either by curly brackets.

```
% puts stdout Hello, world!  
⇒ bad argument "world!": should be "nonewline"  
% puts stdout {Hello, world!}  
⇒ Hello, world!  
% puts stdout "Hello, world!"  
⇒ Hello, world!
```

**Example 3-2:** grouping of several blank-separated items

The difference between the two solutions is that substitutions are performed inside double quotes but not between curly braces. This feature will be illustrated in the discussion about substitution which follows.

### 3.1.4. Character substitution

This is the simplest substitution. A group of characters is replaced by a single character. This enables the use of special characters reserved by the language such as the grouping characters for example. These substitutions sequences are introduced by the backslash character followed by the character we want to print.

```
% puts stdout {  
>  
% puts stdout \  
⇒ {
```

**Example 3-3:** character substitution

Character substitution is also used when a command is so long that it has to be split on several lines. In that case a backslash character followed by a new-line indicates that the command continues on the next line.

```
% puts stdout {this is a very long command, it has to be \  
> split on several lines}  
⇒ this is a very long command, it has to be split on several lines
```

**Example 3-4:** command split on several lines

### 3.1.5. Variable substitution

In Tcl there is no need to declare variables. variables are automatically allocated on the stack as soon as they are referenced for the first time. The dollar sign enables the substitution of a variable name by its value. The set command is used most often to create a new variable. The command accepts two arguments, the first one is a variable name and the second one is the associated value.

```
% set foo 42
⇒ 42
% puts stdout foo
⇒ foo
% puts stdout $foo
⇒ 42
```

**Example 3-5:** variable substitution

### 3.1.6. Command substitution

It is possible that a command argument is itself a command. In such a case the embedded command shall be enclosed by square brackets.

```
% puts stdout [expr 2+1]
⇒ 3
```

**Example 3-6:** command substitution

In the example above, we used the expr command which takes a variable number of arguments and arithmetically compute them. Expressions are formed the usual way using operators such as +, -, \*, etc.

It is of course possible to embed several commands, the innermost one is evaluated first.

```
% puts stdout [expr [expr 1+2] + 10]
⇒ 13
```

**Example 3-7:** embedding several commands

Substitutions are not performed inside curly braces, but they are when enclosed inside double quotes.

```
% set foo 42
⇒ 42
% puts stdout {[expr 1+1] $foo}
⇒ [expr 1+1] $foo
% puts "[expr 1+1] $foo"
⇒ 2 42
```

**Example 3-8:** difference between curly braces and double quotes

## 3.2. Data structures

### 3.2.1. Character strings

Character strings are the base type in Tcl. By default, each variable is of type string.

Tcl offers several facilities for string manipulation. The most often used command is the `string` command. Its first argument is a keyword indicating the type of operation we want to perform on strings, the remaining arguments are the arguments necessary for the specified operation to complete. Possible operations are: comparison (`compare`, `match`), character retrieval (`index`), sub-string extraction (`first`, `last`, `range`), string modification (`tolower`, `toupper`, `trimleft`, `trimright`), etc.

```
% set s {Woody Woodpecker}
⇒ Woody Woodpecker
% string index $s 4
⇒ y
% string toupper $s
⇒ WOODY WOODPECKER
% string match "W???? W*pecker" $s
⇒ 1
```

**Example 3-9:** string manipulation functions

### 3.2.2. Arrays

The Tcl language uses associative arrays (implemented as hash-tables) instead of indexed arrays. As a result, the index (which in fact is a key) is not necessarily an integer, it is a character string (remember that character strings are the base type of the language) and may take any value, as illustrated in the following example where the string `foo` is used as an index. The index of an array is surrounded by brackets.

```
% set a(0) 10
⇒ 10
% set a(foo) bar
⇒ bar
% puts stdout $a
⇒ can't read "a": variable is array
% puts stdout $a(foo)
⇒ bar
```

**Example 3-10:** index in an array

As for the character strings there is a special command, called `array` this time, which enables to perform several kinds of operations on an array. This command takes an operation name as its first argument and then arguments relevant to the specified operation.

```
% set a(0) 10; set a(1) 11; set a(2) 12
⇒ 12
% array name a
⇒ 0 1 2
% array size a
⇒ 3
```

**Example 3-11:** the array command

### 3.2.3. Lists

A list has exactly the same structure as a Tcl command line. In fact lists are only character strings with a special format (items separated by blanks), they are parsed in order to retrieve the items. The drawback of this method is that the performances are very poor in the case of lists with many items. In that latter case, it is more efficient to use arrays rather than lists.

The `list` command is used to built new lists.

```
% set x {1 2 3}
⇒ 1 2 3
% set y foo
⇒ foo
% set l [list $x "a b c" $y]
⇒ {1 2 3} {a b c} foo
```

**Example 3-12:** declaration of a list

Unlike the `array` command for arrays and the `string` command for the character strings, the `list` command only creates lists and performs no operation on lists. To perform operations on lists, we need to use some other commands. These commands are easily recognizable since most of them begin with the letter "l".

```
% llength {a b {c d} "e f g" h}
⇒ 5
% set x {1 2 3}
⇒ 1 2 3
% lindex $x 1
⇒ 2
% lrange {a b c d {e f} g h} 3 5
⇒ d {e f} g
% set l {users/foo/project}
⇒ users/foo/project
% split $l /
⇒ users foo project
% lindex [split $l /] 1
⇒ foo
% lsort -ascii -decreasing {a Z n2 z n100}
⇒ z n2 n100 a Z
% set y [list a {b c} e d f]
⇒ a {b c} e d f
% lreplace $y 1 3 b c d e
⇒ a b c d e f
% puts $y
⇒ a {b c} e d f
% set z [lreplace $y 1 3 b c d e]
⇒ a b c d e f
% puts $z
⇒ a b c d e f
```

**Example 3-13:** some operations on lists

## 3.3. Control structures

### 3.3.1. The if condition

This command has the following syntax:

```
if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?
```

Blocks embedded by question marks are optional (this is the usual notation in Tcl-related documentation). In the condition blocks, it is useless to call the `expr` command since this is performed automatically by the `if` command.

```
if {$x==0} {
    puts stderr "Divide by zero!"
} else {
    set q [expr $y/$x]
}
```

**Example 3-14:** a simple if condition

Note the rather odd-looking placement of the curly braces in the above example. This is due to the fact that a command is terminated by a new-line, except if something (such as brackets, quotes, etc.) is open, in that case the command last till the matching something is found. The Tcl implementers proposed a standard programming style, see [5].

### 3.3.2. The switch condition

The switch command has the following syntax:

**switch** *?options? string pattern body ?pattern body ...?*

or:

**switch** *?options? string {pattern body ?pattern body ...?}*

*options* can take one or more of the following values: `-exact`, `-glob`, `-regexp`, `--`. This parameter enables to perform pattern matching operations between the *string* and the *pattern*. With `-exact` an exact matching (character by character comparison) is performed, with `-glob` an glob-like pattern matching is performed and with `-regexp` a grep-like pattern-matching is performed. The last option: `--`, is useful in case the *string* begins with an `"-"` character in order not to confuse it with an actual option. It is a good programming practice to always use `--` as the last option.

One of the *patterns* can be the default keyword. In that case, the *body* following default is always executed if there is no other match. If the *body* equals the single `"-"` character, then the following *body* is executed, this is the fall-through operator.

```
switch -exact -- $x {
    hello      -
    "good morning" {puts stdout "How do you do?"}
    "good bye"   {puts stdout "See you soon!"}
    default     {puts stdout "Don't know what to say to you!"}
}
```

**Example 3-15:** the switch command

### 3.3.3. The while loop

This command is similar to the corresponding instruction in the C language. Its syntax is the following:

**while** *test body*

```
set i 0
while {$i<10} {
    incr i
}
```

**Example 3-16:** the while command

### 3.3.4. The foreach loop

The foreach command has the following syntax:

**foreach** *varname list body*

or

**foreach** *varlist1 list1 ?varlist2 list2 ...? body*

When executing *body*, *varname* takes all values given in *list*.

```
foreach v {1 3 5 7 11 13} {
    puts stdout $v
}
```

**Example 3-17:** a foreach loop with a single variable

It is also possible to use as many variables as one wants.

```
foreach {color value} {red 10 green 11 blue 12 yellow} {
    puts stdout "$color / $value"
}
```

displays the following result:

```
red / 10
green / 11
blue / 12
yellow /
```

**Example 3-18:** a foreach loop with two variables

### 3.3.5. The for loop

This command is similar to the for loop in the C language. Its syntax is the following:

**for** *start test next body*

```
for {set i 0} {$i<10} {incr i +2} {  
    puts $i  
}
```

**Example 3-19:** the for loop

### 3.3.6. The break and continue commands

As for the C language, the break command enables to exit out of a loop and the continue command enables to jump to the processing of the next loop item.

```
for {set i 0} {$i<10} {incr i +1} {  
    if {$i>5} {  
        break  
    }  
    if {$i==3} {  
        continue  
    }  
    puts -nonewline stdout "$i "  
}  
puts stdout " "
```

displays the following result:

```
0 1 2 4 5
```

**Example 3-20:** use of break and continue commands in a for loop

### 3.3.7. Comments

Although comments are not commands, we have to deal with this feature somewhere. A comment in Tcl is introduced by the “#” character and is terminated by the new-line character. The “#” character can be used only where a command name can appear. To put a comment on the same line of a command we have to use the “;#” character sequence instead.

## 3.4. Procedure definition

### 3.4.1. The proc command

A procedure in Tcl is defined using the proc keyword and has the following syntax:

```
proc name args body
```



Once defined, a procedure becomes a command which can be used as any other command. A procedure can return a value by using the `return` command, but this is not mandatory. By default a procedure returns the result of the evaluation of the last command executed inside the procedure. Nonetheless it is considered as good programming practice to always explicitly use a call to `return`.

```
proc power {x n} {  
    set result 1  
    for {set i 0} {$i<$n} {incr i +1} {  
        set result [expr $result * $x]  
    }  
    return $result  
}
```

Example of use:

```
% power 3 2  
⇒ 9  
% power 4 0  
⇒ 1  
% power [expr sqrt(2)] 2  
⇒ 2.0000000000000004
```

**Example 3-21:** declaration and use of a procedure

### 3.4.2. Optional arguments

In an argument list of a procedure, it is possible to indicate that some arguments are optional. Optional arguments must have a default value and be the last ones in the argument list.

```
proc p {a {b 1} {c 2}} {  
    expr $a + $b + $c  
}
```

Example of use:

```
% p 100 10 1  
⇒ 111  
% p 100 10  
⇒ 112  
% p 100  
⇒ 103  
% p  
⇒ no value given for parameter "a" to "p"
```

**Example 3-22:** procedure with optional arguments

### 3.4.3. Variable argument list

In an argument list of a procedure, it is also possible to have a number of arguments not fixed in advance. For that the keyword `args` shall be the last argument. During the procedure call, it will be replaced by a list containing all the arguments (possibly zero).

### 3.4.4. Global variables

Any variable defined outside any procedure body is a global variable. When in a procedure body, it is only possible to access to the variables declared inside the procedure body. In order to access (or to declare) a global variable inside a procedure body, the `global` keyword shall be used. Its syntax is the following:

**global** *varname* *?varname ...?*

For the duration of the current procedure (and only while executing in the current procedure), any reference to any of the *varnames* will refer to the global variable by the same name.

### 3.4.5. The upvar command

The `upvar` command is a generalization of the `global` command. Its syntax is the following:

**upvar** *?level?* *otherVar myVar ?otherVar myVar ...?*

This command arranges for one or more local variables in the current procedure to refer to variables in an enclosing procedure call or to global variables. If *level* is an integer then it gives a distance (up the procedure calling stack) to move before executing the command. If *level* consists of the “#” character followed by a number then the number gives an absolute level number (“#0” represents the global context, so “`upvar #0`” is the same as “`global`”). If *level* is omitted then it defaults to 1.

## 3.5. Examples

### 3.5.1. Factorial in recursive mode

The following example computes the factorial of an integer using a recursive algorithm. Note: the procedure is not safe since no test are performed on the input value (such as illegal value, exceeds capacity, etc.)

```
proc fact {n} {
    if {$n==0} {
        return 1
    } else {
        return [expr [fact [expr $n - 1]] * $n]
    }
}
```

**Example 3-23:** factorial (recursive mode)

### 3.5.2. Factorial in loop mode

The following example also performs a computation of a factorial, except that, this time, the algorithm is iterative rather than recursive. In addition, an auxiliary procedure has been defined: the `decr` procedure which decrements a given variable (this is the counterpart of the Tcl built-in procedure, except that `decr` first test the existence of the variable, illustrating the use of the `upvar` command).

```
proc decr {n {dec 1}} {
    upvar $n p
    if {[info exist p]} {
        # if variable exists, then decrement it
        set p [expr $p - $dec]
    } else {
        # if variable does not exist, then set it to -dec
        set p [expr 0 - $dec]
    }
}

proc n! {n} {
    set prod 1
    while {[decr n]} {
        set prod [expr $prod * ($n + 1)]
    }
    return $prod
}
```

**Example 3-24:** factorial (iterative mode)

### 3.5.3. Search for a palindrome

The following example shows how to determine that a given string is a palindrome (i.e. a string which can be read either from left to right or right to left, such as “Laval” or “Bob”).

```
proc palindrome {s} {  
    # remove the punctuation  
    set l [split $s " \t.,:;!?'\""]  
    set p ""  
    foreach i $l {  
        set p [append p $i]  
    }  
  
    # convert to uppercase  
    set p [string toupper $p]  
  
    # compare characters 2 by 2 between head and tail  
    set begin 0  
    set end [string length $p]  
    while {($end - $begin) > 1} {  
        set cbegin [string index $p $begin]  
        set cend [string index $p [expr $end - 1]]  
        if {[string compare $cbegin $cend] != 0} {  
            return "\"$s\" is not a palindrome"  
        }  
        incr begin; incr end -1  
    }  
    return "\"$s\" is a palindrome"  
}
```

**Example 3-25:** palindrome search

## 4. The jTcl extension

### 4.1. What is jTcl

jTcl is an extension to the Tcl language. This extension provides an object-oriented layer. It is entirely written in Tcl and requires no external modules in any language whatsoever. jTcl enables the following mechanisms in Tcl:

- definition of a class,
- definition of class and instance variables for a given class,
- definition of private or public methods for a given class,
- implementation of inheritance between classes (multiple inheritance is allowed),
- definition of a constructor or a destructor for a given class.

jTcl adds new commands (under the form of procedures) to the standard Tcl. The new jTcl commands are parsed during the loading of the script file containing them. So, except during this loading operation, no overhead of time is induced by the use of jTcl. jTcl and the JTCL packages require Tcl/Tk version 8.0 or above. They are currently based upon the version 8.0 patch level 2 of Tcl/Tk.

### 4.2. New keywords

jTcl adds the following keywords (implemented as Tcl procedures) to the Tcl language:

**class**      definition of a jTcl class. Description of this command is given further (see: 4.4.2 Class declaration).

**new**        creation of a new object, this procedure returns a handle to the object. The syntax of the command is the following:

**new** *class-name param-list*

where *class-name* shall be a class already declared using the **class** keyword (see: 4.4.2 Class declaration) and *param-list* is the list of parameters (possibly zero) needed by the constructor (see: 4.4.6 Constructors).

Of course, since most of the time, we want to be able to use the newly created object, the returned value of the **new** command should be stored in a variable:

set *object-handle* [**new** *class-name param-list*]

The handle returned by the new command has a **\_O\_***class-namex* format, where x is

a counter incremented with each new object creation (for example: `_0_FooBar3` for the third creation of an object of class `FooBar`).

**free** destruction of an object, given its handle. The syntax of the command is the following:

**free** *object-handle*

where *object-handle* is the value returned by the **new** command.

**get** access to the value of an instance variable. The syntax of the command is the following:

*object-handle* **get** *var-name*

where *object-handle* is the value returned by the **new** command and *var-name* the name of the instance variable we want to access. The call to that procedure returns the current value of the variable. Note: it is not possible to specify that an instance variable is not public.

**set** change the value of an instance variable. The syntax of the command is the following:

*object-handle* **set** *var-name* *var-value*

where *object-handle* is the value returned by the **new** command, *var-name* the name of the variable and *var-value* the new value we want to set to the variable. Note 1: it is not possible to specify that an instance variable is not public. Note 2: since this keyword can be used only after an object handle, there is no risk of overlapping with the Tcl `set` built-in command.

In addition, the following keywords can be used inside a class definition, they are recognized and processed by the parser of the **class** procedure:

**extends** defines the super-classes of the class.

**set** defines an instance variable. Since this keyword can be used only at a specified position in a class definition, there is no risk of overlapping with the Tcl `set` built-in command.

**static** defines a class variable.

**public** defines a public method.

**private** defines a private method.

**free** default name for the destructor.

Inside a class definition, the following variables can be used:

**\$CLASS** represents the current class (note: this is not the class name but an internal class descriptor), this variable is used to access class variables and public methods from within a class body.

**\$SUPER** represents the superclass of the current class. It is useful for calling a method of the superclass and is often used inside constructors. In case of multiple inheritance, it

represents a list of all the superclasses and then should be accessed using the `lindex Tcl` built-in command.

**\$MY** represents the current object instance name, it is sometimes used to call a method from within the class body. It is useful any time we need to give the handle of the current object to another object. For example: `method-name $MY args`.

**\$THIS** represents the current object instance array, it is used to access instance variables from within a class body. For example: `set THIS(my-instance-var) value`.

### 4.3. How to use jTcl

In order to have access to all of the keywords explained in the above paragraph, the `Package.jTcl` file shall be imported in the Tcl interpreter using the `source Tcl` built-in command. After that, the `jClassImport` command is available to source all the packages required to program with jTcl, see [7] for a complete description of all the packages.

In order to run properly, the `Package.jTcl` needs the following variables to be defined:

**PKG\_STATE** can have the value “prod” for use of the production tree or “dev” for use of the development tree.

**PKG\_DIR** base directory of the jTcl tree.

The development tree shall have the following content:

- `$PKG_DIR/lang/Tcl` contains the files for the `lang` packages,
- `$PKG_DIR/tcp/Tcl` contains the files for the `tcp` packages,
- `$PKG_DIR/ic/Tcl` contains the files for the `ic` packages.

The production tree shall have the following content:

- `$PKG_DIR/Tcl` contains the files for all packages.

The following lines gives an example of a typical jTcl program header:

```

# indicate production tree
set PKG_STATE prod

# if JTCL_HOME not defined, try to find one
if {[info exists env(JTCL_HOME)]} {
    set DIR [file dirname $argv0]
    cd $DIR/../../..
    set env(JTCL_HOME) [pwd]
    puts "Warning: JTCL_HOME defaulted to $env(JTCL_HOME)"
}

if {[info exist JTCL.PACKAGE.LOADED]} {
    # If No JTCL Path, create a default one
    if {[info exists env(JTCL_PATH)]} {
        # Add JTCL file in version reverse order
        if [catch {lsort -decreasing
            [glob $env(JTCL_HOME)/*/lib]} env(JTCL_PATH)] {
            if [catch {lsort -decreasing
                [glob $env(HOME)/jTcl/*/lib]} env(JTCL_PATH)] {
                if [catch {lsort -decreasing
                    [glob /usr/local/jTcl/*/lib]} env(JTCL_PATH)] {
                    puts "ERROR: Can't find JTCL in default location"
                    exit
                }
            }
        }
    }
    puts "Warning: JTCL_PATH defaulted to $env(JTCL_PATH)"
}

# Now loop on JTCL_PATH for Dvb Etc/package files
foreach DIR $env(JTCL_PATH) {
    if {[info exist JTCL.PACKAGE.LOADED]} {
        if [file exists $DIR/jTcl/Etc/Package.jTcl] {
            set PKG_DIR [file join $DIR jTcl]
            if [info exists env(JTCL_DEBUG)] {
                puts "sourcing $PKG_DIR/Etc/Package.jTcl"
            }
            uplevel #0 source \"$PKG_DIR/Etc/Package.jTcl\"
            break
        }
    }
} ;# end foreach JTCL_PATH
}

# Import full package search fonctionnality
# -----
jClassImport {lang.debug.*}
jClassImport {lang.search.*}

```

**Figure 4-1:** Introduction code before using jTcl in any Tcl program.



## 4.4. jTcl syntax

### 4.4.1. Comments

There is a slight difference between comments in Tcl and comments in jTcl: text of the comment shall be enclosed between quotes. Comments in jTcl have the following syntax:

```
# "this is the text of the comment"
```

for a comment at the beginning of a line

and:

```
command ;# "this is the text of the comment"
```

for a comment at the end of a line after a command.

The “#” character alone is not allowed.

Misuse of the above rules stops the jTcl parsing.

### 4.4.2. Class declaration

A class is declared using the **class** keyword. The class declaration syntax is the following:

```
class class-name {extends superclasse-name ...} body
```

The *body* part of the class declaration follows the following syntax:

```
{
    # "class variables declaration"
    static var-name var-value
    ...

    # "instance variable declaration"
    set var-name var-value
    ...

    # "constructor declaration"
    class-name {param-list} {
        ...
    }

    # "destructor declaration"
    free {} {
        ...
    }

    # "private methods declaration"
    private method-name {param-list} {
```

```
        ...
    }
    ...

    # "public methods declaration"
    public method-name {param-list} {
        ...
    }
    ...
}
```

Of course, any of these parts are optional and can be listed in any order. All of these parts are explained in the following paragraphs.

#### 4.4.3. Instance variables

Instance variables shall be declared inside the *body* part of a class using the **set** keyword. The syntax of the declaration is the following:

```
set var-name var-value
```

It is not necessary to declare all instance variables in the class body when declaring the class. Instance variables can be declared “on the fly” at any time, exactly as it is the case for ordinary Tcl variables.

To create a new instance variable (or to set a new value to an already existing instance variable) inside the body of a method, the syntax is the following:

```
set THIS(var-name) var-value
```

To create a new instance variable (or to set a new value to an already existing instance variable) outside the body of a method, the syntax is the following (*\$object-handle* is the name of the instance, as returned by the new command):

```
$object-handle set var-name var-value
```

Nonetheless, even if allowed by the language, it is considered as a very poor programming practice to allow for creation of instance variable outside the class body. Good programming practice should be to declare explicitly all instance and class variables in the body of the class using the **set** and **static** keywords and define get and set methods for each variable, as shown in the examples of the next chapter.

#### 4.4.4. Class variables

Class variables shall be declared inside the body of a class using the **static** keyword. The syntax of the declaration is the following:

```
static var-name var-value
```

To create a new class variable (or to set a new value to an already existing class variable) inside the body of a method, the syntax is the following:

**set CLASS**(*var-name*) *var-value*

The same remarks as for the instance variables also apply for the class variables (see: 4.4.3. Instance variables).

#### 4.4.5. Methods

Methods can be either private or public. Nonetheless there is no protection mechanism implemented and therefore no means to restrict from calling a private method of an object anywhere.

Public methods are declared inside a class body using the **public** keyword. The syntax of the declaration is the following:

**public** *method-name arg-list body*

Private methods are declared inside a class body using the **private** keyword. The syntax of the declaration is the following:

**private** *method-name arg-list body*

To call a public method inside a class body, the syntax is the following:

**CLASS**(*method-name*) **\$MY** *arg-list*

To call a private method inside a class body, the syntax is the following:

**THIS**(*method-name*) **\$MY** *arg-list*

To call any method outside a class body, the syntax is the following:

*object-handle method-name arg-list*

#### 4.4.6. Constructors

Constructors are special public methods which are automatically called when an object is created. They are useful to perform non-trivial initialization of instance variables. Note that trivial initialization of instance variables can be performed by the **set** part of the class declaration. Constructors are not mandatory and they shall be declared **public** when used. The name of a constructor is the name of the class. A constructor can have any number of parameters.

#### 4.4.7. Destructors

Destructors are special public methods which are automatically called when an object is destroyed. They are useful to perform release of allocated resources, such as closing of files, sockets, etc. Destructors are not mandatory and they shall be declared **public** when used. The name of a destructor is the keyword **free**. A destructor accepts no parameters.

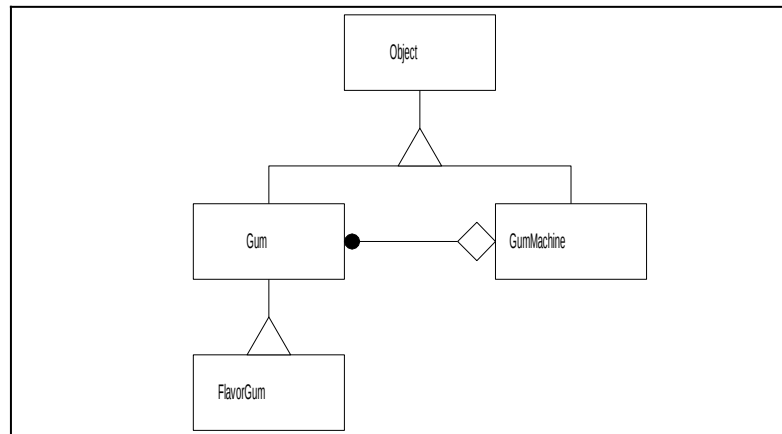
## 4.5. The Object class

The `Object` class is defined in the `lang::object::*` package. It serves as a root class, providing some basic functionality. It is useful (although not mandatory) to inherit from this class when creating a new set of classes.

A complete description of this class is given in [7].

## 5. Study of an example in jTcl

This chapter is intended as a tutorial to the jTcl language. It will present the in-depth study of a small example. This example consists of three classes (inherited from the jTcl Object class) interacting together. The following OMT diagram indicates the relationships between the classes.



**Figure 5-1:** OMT diagram of the example classes

### 5.2. The Gum class

The first class (Gum) defines a bubble-gum. This class enables to create a bubble-gum of a given color and to retrieve its color.

This class illustrates the following mechanisms:

- Declaration of a simple class
- Use of constructor with default value
- Setting values to instance variables
- Getting values of instance variables

The code is given below:

```
# "Gum class -- a simple (tasteless) bubble-gum"
# "-----"
class Gum {extends Object} {

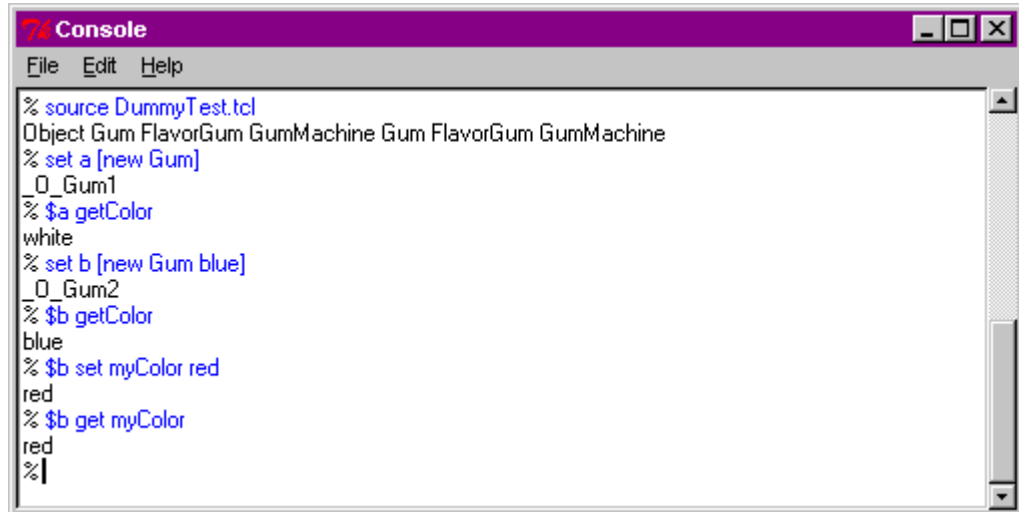
    # "instance variables"
    # "-----"
    set myColor white      ;# "default color for a gum"

    # "constructor"
    # "-----"
    private Gum {{aColor {}}} {
        if {$aColor != {}} {
            set THIS(myColor) $aColor
        }
    }

    # "public methods"
    # "-----"
    public getColor {} {
        return $THIS(myColor)
    }
}
```

**Figure 5-2:** jTcl code for the Gum class

An example of use of the Gum class is given below:



**Figure 5-3:** example of use of the Gum class

In the example above, we can notice that it is possible to access (and to modify) any instance variable, even if no get or set methods are associated. The jTcl language provides no security mechanism. Nonetheless it is considered as bad programming practice to use these capabilities of the language. Implementing get and set methods to access the instance and class variables should be undertaken.

### 5.3. The FlavorGum class

The second class (FlavorGum) defines a tasty bubble-gum. This class is a specialization of the previous one. in addition of the Gum class characteristics, we have the flavor property and the possibility to read that value.

This class illustrates the following mechanisms:

- Declaration of an inherited class
- Call of the super-class constructor inside the constructor
- Getting values of a class variable

The code is given below:

```

# "FlavorGum class -- a tasty bubble-gum"
# "-----"
class FlavorGum {extends Gum} {

    # "class variables"
    # "-----"
    static ourKnownFlavors {mint green strawberry pink cherry \
        red vanilla yellow apricot orange}

    # "instance variables"
    # "-----"
    set myFlavor none ;# "default flavor for a flavored gum"

    # "constructor"
    # "-----"
    private FlavorGum {{aFlavor {}} {aColor {}}} {
        if {$aFlavor != {}} {
            if {$aColor != {}} {
                # "if color is specified then use it"
                set THIS(myFlavor) $aFlavor
                $SUPER $MY $aColor
            } else {
                # "try to deduce color from known flavors,"
                # "if failed then use default color"
                foreach {oFlavor oColor} $CLASS(ourKnownFlavors) {
                    if {[string compare $oFlavor $aFlavor] == 0} {
                        # "flavor is found in the list"
                        set THIS(myFlavor) $oFlavor
                        $SUPER $MY $oColor
                        return
                    }
                } ;# "end foreach"
                set THIS(myFlavor) $aFlavor
            } ;# "end if"
        } ;# "end if"
    }

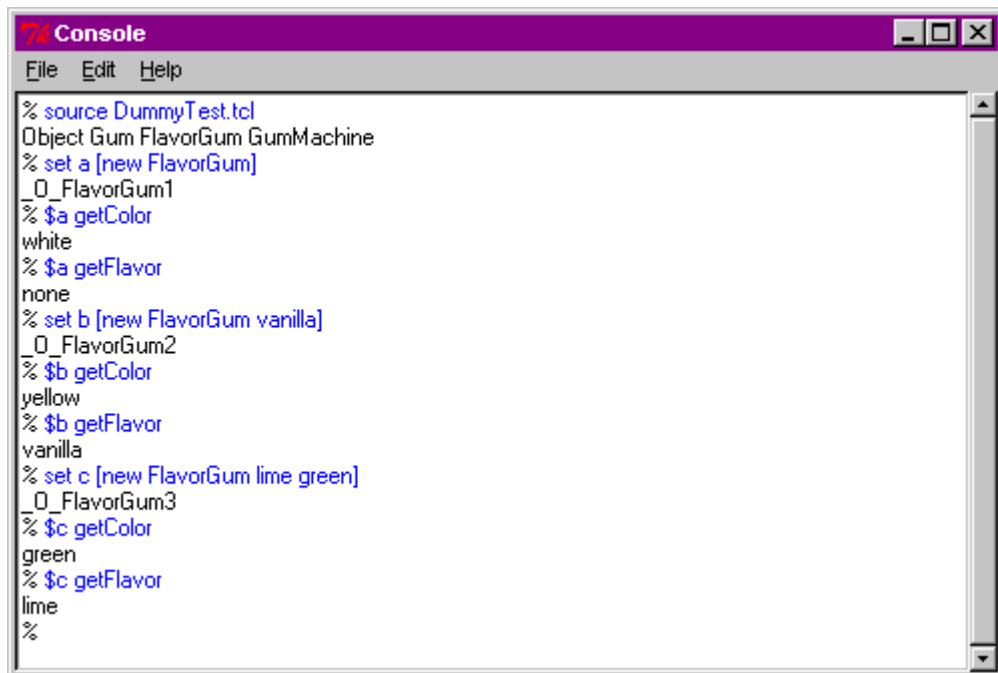
    # "public methods"
    # "-----"
    public getFlavor {} {
        return $THIS(myFlavor)
    }
}

```

**Figure 5-4:** jTcl code for the FlavorGum class

An example of use of the FlavorGum class is given below:





```
% source DummyTest.tcl
Object Gum FlavorGum GumMachine
% set a [new FlavorGum]
_O_FlavorGum1
% $a getColor
white
% $a getFlavor
none
% set b [new FlavorGum vanilla]
_O_FlavorGum2
% $b getColor
yellow
% $b getFlavor
vanilla
% set c [new FlavorGum lime green]
_O_FlavorGum3
% $c getColor
green
% $c getFlavor
lime
%
```

**Figure 5-5:** example of use of the FlavorGum class

In the example above, we notice that it is always possible for a given class to call all the methods defined in the super-classes, even if they are not redefined.

## 5.4. The MachineGum class

The third class (MachineGum) defines a bubble-gum container. This class enables to add and retrieve gums (flavored or not) to the container.

This class illustrates the following mechanisms:

- Declaration of a simple class
- Use of constructor with default value
- Setting values to instance variables
- Getting values of instance variables

The code is given below:

```

# "GumMachine class -- a free gum distribution machine"
# "-----"
class GumMachine {extends Object} {

    set myContent {}      ;# "the gums"
    set myCapacity 25      ;# "max number of gums in the machine"

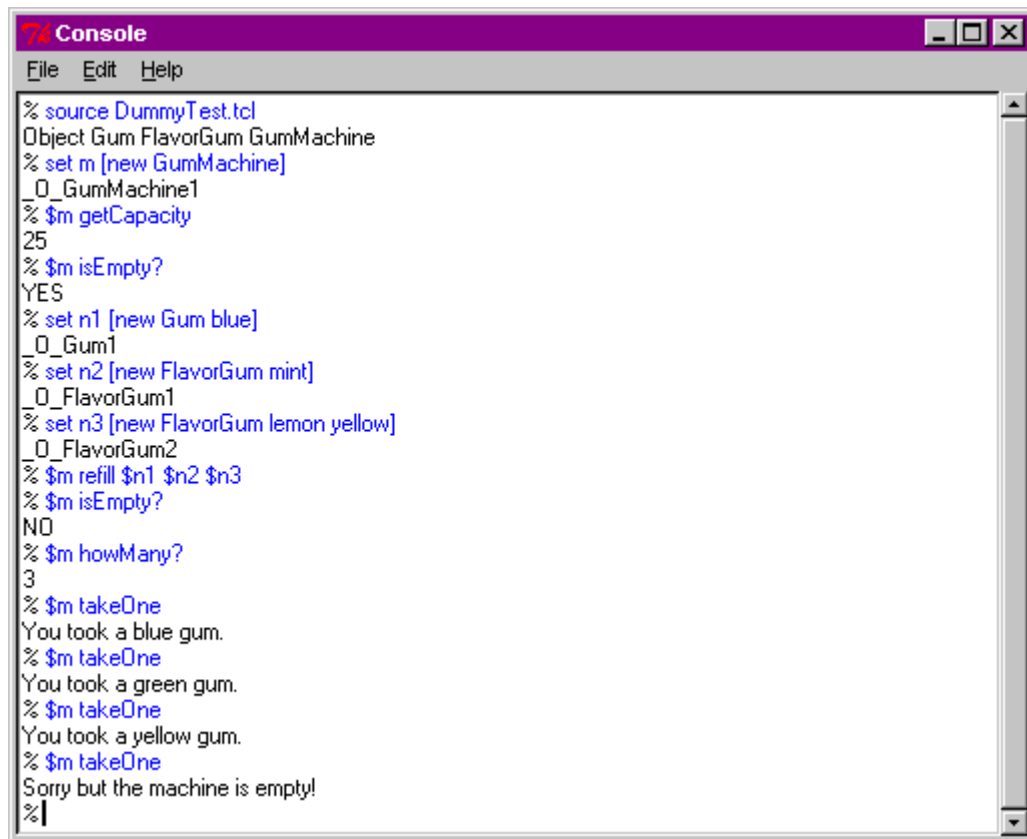
    private GumMachine {{aCapacity {}}} { ;# "constructor"
        if {$aCapacity != {}} {
            set THIS(myCapacity) $aCapacity
        }
    }

    # "public methods"
    public getCapacity {} {
        return $THIS(myCapacity)
    }
    public isEmpty? {} {
        if {[MY howMany?] == 0} {
            return YES
        } else {
            return NO
        }
    }
    public reset {} {      ;# "remove all gums in the machine"
        set THIS(myContent) {}
    }
    public howMany? {} { ;# "return the # of gums in the machine"
        return [llength $THIS(myContent)]
    }
    public takeOne {} { ;# "remove a gum from the machine"
        if {[string compare [MY isEmpty?] YES] == 0} {
            puts stdout "Sorry but the machine is empty!"
        } else {
            set oGum [lindex $THIS(myContent) 0]
            set THIS(myContent) [lrange $THIS(myContent) 1 end]
            # "test if not gum to be done..."
            set oColor [$oGum getColor]
            puts stdout "You took a $oColor gum."
        }
    }
} ;# "end takeOne"
public refill {args} ;# "refill the machine with some gums"
{
    set oDiff [expr [llength $args] + [MY howMany?]]
    if {$oDiff <= 0} {
        puts stdout "Sorry, not enough room!"
        return
    }
    foreach oGum $args {
        lappend THIS(myContent) $oGum
    }
} ;# "end refill"
}

```

**Figure 5-6:** jTcl code for the MachineGum class

An example of use of the Gum class is given below:



```
% source DummyTest.tcl
Object Gum FlavorGum GumMachine
% set m [new GumMachine]
_O_GumMachine1
% $m getCapacity
25
% $m isEmpty?
YES
% set n1 [new Gum blue]
_O_Gum1
% set n2 [new FlavorGum mint]
_O_FlavorGum1
% set n3 [new FlavorGum lemon yellow]
_O_FlavorGum2
% $m refill $n1 $n2 $n3
% $m isEmpty?
NO
% $m howMany?
3
% $m takeOne
You took a blue gum.
% $m takeOne
You took a green gum.
% $m takeOne
You took a yellow gum.
% $m takeOne
Sorry but the machine is empty!
%|
```

**Figure 5-7:** example of use of the GumMachine class

