

# Instructor's Manual for the Teaching Machine 2

Updated May 2013

1	User Interface .....	2
1.1	Starting the program.....	2
1.2	Execution commands.....	3
1.3	Display Options .....	3
1.4	Window manipulation.....	3
1.5	Configuration .....	3
1.6	Exiting or hiding the teaching machine.....	3
1.7	Security restrictions .....	4
1.8	Trouble Shooting .....	4
1.9	The Languages Accepted by the Teaching Machine .....	4
1.10	Suffixes.....	4
1.11	Bugs and wishes .....	4
2	Applet interface .....	4
2.1	TMTinyApplet .....	4
2.2	The TMBigApplet.....	6
2.3	External Command Interface.....	6
2.3.1	Loading source files .....	6
2.3.2	Setting the configuration .....	7
2.3.3	Selection .....	7
2.3.4	Input characters .....	7
2.3.5	Program arguments .....	7
2.3.6	Hiding.....	7
2.3.7	Execution .....	7
2.3.8	Back up.....	7
3	Pedagogical Markup .....	7
3.1	General Syntax.....	8
3.2	Selections.....	8
4	C++ .....	10
4.1	General.....	10
4.2	Lexical conventions .....	10
4.3	Basic Concepts .....	11
4.4	Standard Conversions .....	11
4.5	Expressions .....	11
4.6	Statements .....	12
4.7	Declarations .....	12
4.8	Declarators and Function Definitions .....	13
4.9	Classes.....	13
4.10	Derived classes .....	14
4.11	Access Control .....	14
4.12	Special Member Functions.....	14
4.13	Overloading .....	14
4.14	Templates.....	14
4.15	Exception Handling .....	15

4.16	Preprocessing Directives.....	15
4.17	Library .....	15
4.17.1	Library support in Phase 1 .....	15
4.17.2	Library support in Phase 2.....	16
4.18	Known Bugs and Deficiencies.....	16
5	Java .....	16
5.1	Library .....	16
5.1.1	java.io .....	16
5.2	java.lang.....	16
5.3	java.util.....	16
6	Scripting and Plug-ins .....	16

## 1 User Interface

### 1.1 Starting the program

**Applet.** See section 2.

**Application.**

**On Windows**, if the TM is installed with its installer, then there should be a start menu item for it. This links to a .bat file that executes the following command line.

```
<p>\java.exe -cp "<id>\tm.jar;%CLASSPATH%" tm.TMMainFrame -ic "<id>\initial.tmcfg" -id "<id>"
```

where <p> is the path to your java installation and <id> is the installation director. On windows, <id> is typically **C:\Program Files (x86)\MUN\The Teaching Machine**

**On Linux, Unix and Mac**, the following command line should start the application, assuming the java executable is on your path.

```
java -cp <id>/tm.jar tm.TMMainFrame -id <id> -ic ~/initial.tmcfg
```

On Windows, Linux, Unix, and Mac you can add a file name to the end of the command. This would be the name of a C++ or Java file to be loaded.

The following options can follow tm.TMMainFrame on the command line.








- -id <directory> The installation directory. This is used, for example to locate the help files.
- -ic <file> The initial configuration file. This file configures the TM. If this option is not used, the TM uses a default configuration that is stored in its .jar file and can not be changed.

The options that precede tm.TMMainFrame on the command line are options to the Java Virtual Machine. In particular, the system property “debug” controls the level of debugging output. Usually the property is set to “high” or not set at all. It can also be set to “no”, “low”, or “medium”. In Sun’s JVM the command line parameter is “-Ddebug=high”.

**Loading programs.**




- **Applet.** The usual way of changing programs is to load the web page that describes the program. See section 2.
- **Application.** The menu command File/Load File may be used. Files must end with a suffix that identifies the language. See section 1.10.

## 1.2 Execution commands.

-  (Menu: Go/Into Expression) Execute the next expression in this subroutine.
-  (Menu: Go/Into Subroutine) Execute the next expression.
-  (Menu: Go/Over) Execute to the end of this subroutine.
-  Execute until the line the cursor is on or until the program terminates.
-  (Menu: Go/Forward) Execute until a subexpression is selected..
- (Menu: Go/Microstep) Execute one step. This is usually only useful for debugging the interpreter.
-  (Menu: Go/Backward) Undo last command. You may undo back to the start of executing the subject program.
-  Restart the interpretation of the program.

## 1.3 Display Options

Stack, Static, Heap, and Scratch Subwindows

-  Display one line per variable.
-  Display one line per byte.
-  Display values in binary.

## 1.4 Window manipulation.

All subwindows can be moved, resized, or brought to the front, though it sometimes takes a few tries to get the mouse in just the right spot.

All subwindows can be maximized, minimized, or closed. Once closed, they are gone until a file is opened.

## 1.5 Configuration

The layout of the subwindows and other aspects of the Teaching Machine's display can be saved later loaded. The configuration also records the current set of plug-ins.

- Menu: File/Save Configuration File. Write the configuration file to disk.
- Menu: File/Read Configuration File. Retrieve a previous configuration.

By manually editing the configuration files, you can change font styles, colours subwindow titles, and other aspects of the display.

## 1.6 Exiting or hiding the teaching machine

Selecting the menu command File/Exit will exit the TM, if it is running as an application. If it is running as an applet, this will merely hide the TM's window. The window will be redisplayed if another file is loaded into it. In Windows, clicking on the x in the upper-right corner is equivalent to selecting File/Exit.

Applets are terminated by the browser when they are no longer needed..

## 1.7 Security restrictions

If you ask the applet to do something that, because of a security restriction, it can not do, a window will pop up to tell you so. Typical restrictions involve reading or writing local files.

## 1.8 Trouble Shooting

For the Applet make sure that your browser is showing the Java console. Check the console for clues to the problem. Check your browser settings as regards applets.

## 1.9 The Languages Accepted by the Teaching Machine

Currently (2010), the TM supports C++ and Java. See sections 4 and 5.

## 1.10 Suffixes

File names and URLs must end with one of the following suffixes in order that the correct language be selected: .cpp, .cxx, .c++, .java, .jav (case insensitive).

## 1.11 Bugs and wishes

Send reports of bugs and suggested improvements to [theo@mun.ca](mailto:theo@mun.ca).

# 2 **Applet interface**

N.B. WebWriter++ automates much of the work of putting Teaching Machine content onto a web page. The rest of this section is for those not using WebWriter++.

The Teaching Machine can run as a stand-alone application or as an Applet on a web-page. As an Applet, there are two choices. `tm.TMTinyApplet` is an applet that creates a Window for the Teaching Machine to run in. This applet draws nothing on the web page. `tm.TMBigApplet` displays on the web page rather than opening a separate Window.

The TM should run in any modern browser that supports Sun's Java plugin.

## 2.1 TMTinyApplet

Suppose we want to start the TM up with a particular example C++ program loaded into it.

The following listing shows how to do this.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html> <head>
  <script language="JavaScript" type="text/javascript">
  <!--
α    function onLoadHandler()
      {
β      document.tm_applet.loadRemoteFile('example1.cpp') ;
γ      document.tm_applet.readRemoteConfiguration('default.cfg') ;
      }
  <!-->
  </script>
</head>
δ <body onload="onLoadHandler();">
```

```

ε    <APPLET NAME="tm_applet"
ζ      CODE="tm/TMTinyApplet.class"
η      WIDTH="1" HEIGHT="1" ALIGN="BOTTOM"
θ      ARCHIVE="tm.jar">
    </APPLET>

</body>
</html>

```

The APPLET tag creates an instance of the `tm.TMTinyApplet` class (ζ), giving it a name of `tm_applet` (ε). Since the Applet draws nothing on the space it is accorded on the web-page, we give it only a 1 by 1 pixel area (η). In this case the code of the TM is contained in a .jar (Java Archive) file (θ) located in the same directory as the page. Once the page has loaded, and the Applet has started, the “onLoad” event of the page’s Body element is triggered. We have established an event handler (δ) which first loads a C++ file into the instance of the applet (β) and then loads a configuration for the Teaching Machine (γ). Both the C++ file and the configuration file are stored on the same server as the web pages and the .jar files. See Section 2.3.1 for more on the arguments to the `loadRemoteFile` method.

The technique shown above is effective, but inefficient, since it requires the creation of a new Applet instance for each example. Worse, unless the user’s browser caches the .jar files, they will have to down-load again. Instead, we generally create one instance of the applet and load each example into it as the user requires. Next we look at a technique for doing that.

We generally accord the TMTinyApplet its own frame within the browser. For example, here is how one of my sites works. The main HTML file has three frames called `applet_frame`, `toc_frame`, and `content_frame`. Initially we load a welcome message into the `content_frame` and a “please wait” message into the “`toc_frame`”. Into the `applet_frame`, we load the following file:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML> <HEAD> </HEAD>
<BODY onLoad="top.toc_frame.location.href = 'tocframecont.html' " >
  <APPLET NAME="tm_applet"
    CODE="tm/TMTinyApplet.class"
    WIDTH="1" HEIGHT="1"
    ALIGN="BOTTOM"
    ARCHIVE="tm.jar">
  </APPLET>
  <A HREF="help/help.html" target="help">HELP</A>
</BODY> </HTML>

```

As can be seen, once the applet is loaded, the “please wait” message in the `toc_frame` is replaced by a file called `tocframecont.html`. This contains links that the user can click. In the head of `tocframecont.html` we have the line

```
<base target="content_frame">
```

so that clicking on these links causes them to go to the `content_frame`. A typical link in the `tocframecont.html` file is

```

<a HREF="data-structs-ex/03_make_list.html">
  Make and traverse an linked list.</a>

```

Each file like `data-structs-ex/03_make_list.html` describes a particular example and loads it into the teaching machine. Here is the start of `data-structs-ex/03_make_list.html`.

```
<html>
<head>
  <script LANGUAGE="JavaScript">
    function onLoad() {
      top.applet_frame.document.tm_applet.loadRemoteFile(
        'data-structs-ex/03_make_list.cpp') ;
      top.applet_frame.document.tm_applet.readRemoteConfiguration(
        'data-structs-ex/boxes.tmcfg') ; }
  </script>
</head>
<body ONLOAD="onLoad()">
```

Of course this is just one way to do it. The important thing is to prevent the user from being able to send any messages to the Applet until it has fully loaded and started. In this case we ensure that the page that contains the links that will cause C++ files to be loaded into the Applet is not loaded into its frame until the page the Applet is on is fully loaded. The technique above works in both IE and in Firefox on Windows. (I have some doubts about Firefox on the Mac.)

## 2.2 The TMBigApplet

tm.TMBigApplet displays its subwindows directly within the Browser's window. It does not have menus. The user can still interact with the Teaching Machine via the buttons on the subwindows. This Applet requires about 600 (width) by 375 (height) pixels, depending on the configuration.

## 2.3 External Command Interface

Both the tm.TMBigApplet and the tm.TMTinyApplet implement an interface called the ExternalCommandInterface. Any method in this interface can be invoked on an applet instance using JavaScript (provided the Browser supports JavaScript to Java method calls). The full interface is documented elsewhere. The following methods are useful for JavaScripting:

### 2.3.1 Loading source files

```
public void loadRemoteFile( String relativeURL);
```

The relativeURL is relative to the Applet's "document base"; there is no way to use an absolute URL. The relativeURL must end with a suffix that indicates the language (C++ or Java); see section 1.10 .

```
public void loadRemoteFile( String rootURL, String relativeURL );
```

The first URL is relative to the Applet's "document base"; there is no way to use an absolute URL. It gives the directory that should be used as a "root" for resolving include files and imported files. The relativeURL must end with a suffix that indicates the language (C++ or Java); see section 1.10 . The relativeURL is relative to the root.

```
public void loadString( String fileName, String programSource);
```

The fileName must end with a suffix that indicates the language (C++ or Java); see section 1.10 . This routine can be useful when the source is read off an HTML form.

```
public void reStart();
```

**reStart** reloads the current file, if any.

### 2.3.2 Setting the configuration

**public void readRemoteConfiguration( String URL ) ;**

The URL is relative to the Applet's "document base"; there is no way to use an absolute URL. As the configuration does not revert with each load of a source file, it is recommended to call readRemoteConfiguration after each load.

### 2.3.3 Selection

**public void setSelectionString( String expression ) ;**

See section 3.

**public String getSelectionString( ) ;**

See section 3.

### 2.3.4 Input characters

**public void addInputString( String input ) ;**

addInputString is used to stuff characters into cin (or System.in). It should be called after the program is loaded (or reStarted).

### 2.3.5 Program arguments

**public void addProgramArgument( String argument ) ;**

addProgramArgument is used add new strings as program argument (arguments to main). It should be called after the program is loaded (or reStarted).

### 2.3.6 Hiding

**public void quit();**

For the tm.TMTinyApplet, quit hides its Window. For the tm.TMBigApplet, nothing is done.

### 2.3.7 Execution

The following advance the state of the machine

**public void goForward();**

**public void microStep();**

**public void overAll();**

**public void intoExp();**

**public void intoSub();**

**public void toCursor( String fileName, int cursor ) ;**

In the case of toCursor, the parameter is the line number, where the first line is line 1.

### 2.3.8 Back up.

The goBack method returns the state to its next earliest point.

**public void goBack();**

## 3 Pedagogical Markup

The teaching machine recognizes certain comments as "pedagogical markup". It shares this syntax with WebWriter++.

### 3.1 General Syntax

Comments that start with “/\*#” are considered pedagogical markup and are not displayed to the user regardless of whether the Teaching Machine recognizes them or not. In general newlines should not be used in these comments; the /\*#I tag is an exception.

### 3.2 Hiding code with tags and selections

The TM only shows only selected tokens from the source files. This works as follows: Each source token is associated with a set of *tags*. There is a *current selection* Boolean expression. Tokens are only shown if their set of tags satisfies the expression.

Source files may be marked as to the start and end of tags as follows /\*#T *tag* \*/ adds the string “*tag*” to the current “tag set” and /\*#/T *tag* \*/ removes “*tag*” from the current tag set. The tag value does not need to be “*tag*”, of course; it can be any string of letters and numerals. Tags do not need to be properly nested. Spaces are allowed before or after the tag, but not within the tag. Tags are not case sensitive. For example, in

```
int /*#T A*/ x /*#T b*/ = /*#/Ta */ 13 /*#T/B*/ ;
```

we have tag sets associated with the tokens as follows

Token	Tag set
int	∅
x	{“a”}
=	{“a”, “b”}
13	{“b”}
;	∅

An alternative mark up method is to use the mark-up comment /\*#I *code* \*/ This is essentially equivalent to /\*#T *S*\*/*code*/\*#/T *S*\*/. Newlines are allowed within this kind of mark up comment. This is useful when you want to hide code not only from users of the Teaching Machine, but also from other Java compilers. Only // comments are allowed in invisible code.

The current selection expression can be set using either the menu or via the External Command Interface. The selection expression should be set *after* the source file has been loaded. The selection expression is a Boolean expression treating the tags as propositional variables. For example “a&b” selects only tokens with both “a” and “b” in their tag set, whereas “!a|b” selects only tokens with tag sets that either lack “a” or that have “b”. The syntax for selection expressions is as follows.

- Tags are strings of (ascii) letters and numerals. Tags are case insensitive.
- The key words “true” and “false” are reserved and mean what you would expect. Thus these strings should not be used for tags. Upper case letters may be used, if you like.
- Negation has the highest precedence and is represented by either prefix “!” or prefix “~”.
- Conjunction has the next highest precedence and is represented by either infix “&” or infix “.”.
- Disjunction has the lowest precedence and is represented by either infix “|” or infix “+”.
- Parentheses (“ and “)”) can override precedence.



- Spaces are permitted except within tags.

For example “( groucho | harpo ) & ! karl “ is a valid selection expression, as is “( groucho + harpo ) . ~ karl”, and they mean the same thing.

By convention, “S” is used to tag script calls and “L” is used to tag library code. The default selection is “!L & !S”. If you want to see script calls and library code, use “TRUE”.

### 3.2.1 Hidden code and suppressed code

Code that is hidden in the current selection will not be visible in the TM's code window. However, the TM may still stop at it when executing. For example, suppose you have

```
x = 1 ; /*#I y = 1 ; */
```

The TM will not display the `y=1;` statement in the code window, but it will stop on it, if being single stepped and the user will see it in the Expression Engine window. If a line is entirely hidden in the current selection (i.e., every single character is hidden, including spaces), then it will not only be hidden, but it will also be *suppressed*. That means the TM will not, normally, stop at it when stepping through code. For example, if I write

```
x = 1 ; /*#I
y = 1 ; */
```

then the second line will be suppressed. In the code window, suppressed lines are indicated by “...” where the line number would normally be displayed. If line numbers are not displayed, then there is no indication that the suppressed lines exist.

Here is an example. The code (with · indicating spaces) is

```
····public·static·void·main(String·args·)·{
········System.out.println(0)·;
/*#I········System.out.println(1)·;
*/········System.out.println(2)·;
/*#I········System.out.println(3)·;·*/
········System.out.println(4)·;·/*#I
········System.out.println(5)·;·*/
········System.out.println(6)·;
········System.out.println(7)·;·/*#I
········System.out.println(8)·;
*/····}
```

With the default selection (!S & !L) and line numbers displayed, we get

```
12:      public static void main(String args ) {
13:          System.out.println(0) ;
...
15:          System.out.println(2) ;
...
17:          System.out.println(4) ;
...
19:          System.out.println(6) ;
20:          System.out.println(7) ;
...
22:      }
```

Without line numbers, it looks like this

```
public static void main(String args ) {
    System.out.println(0) ;
    System.out.println(2) ;
    System.out.println(4) ;
    System.out.println(6) ;
    System.out.println(7) ;
}
```

If we call the four styles of suppressing lines illustrated by this example BB, BE, EE, and EB (respectively), the preferred style is EE, and if that can't be used, then BB. The reason is that EE and BB are most compatible with WebWriter++. EE has the advantage over BB of not messing up the indentation.

## 4 C++

C++ support is being delivered in phases. Currently all phase 1 items should be supported. Some phase 2 items are supported. *I will use italics to call attention to Phase 2 features that are not yet supported.*

The Teaching Machine is not really intended to be used on code with errors. Therefore, I will not go into the many kinds of errors it does not report. It will diagnose many compile and run-time errors, but not all.

The following subsections follow the ISO standard. The terminology used is that of the standard.

### 4.1 General

We do not claim to comply with the ISO standard. We do not diagnose a great many diagnosable errors in programs.

### 4.2 Lexical conventions

Phases of translation:

1. Mapping of physical to source characters. We assume ascii encoding. Trigraphs are not supported.
2. *Line splicing*. Not in Phase 1. Planned for Phase 2.
3. Decomposition into pptokens. Partially supported.
4. *Preprocessing*. Not fully supported in Phase 1. Include files are implemented. Full preprocessing is planned for phase 2.
5. Mapping of character literals to execution character set. The execution character set is ASCII, so this shouldn't pose a problem.
6. *Catenation of adjacent string literals*. Not in Phase 1. Planned for Phase 2.
7. Syntactic and semantic analysis. Yes, with limitations described below.
8. *Combining translation units*. Phase 1 only supports one translation unit. This will be done in Phase 2.
9. Resolution of externals. Yes.

Supported

- All keywords and most constants. All operators and punctuation.
- All escape sequences in character and string literals.

Not Supported

- Trigraphs and Digraphs are not recognized.
- Wide characters and wide strings are not supported.

Note

- Tabs stops are set at every 4 spaces.

### 4.3 **Basic Concepts**

Supported

- The following types: bool, char, signed char, unsigned char, double, float, long double, int, short, long, unsigned short, unsigned int, unsigned long, pointer, reference, array, struct, class
- const and volatile qualification
- *Wide characters (Phase 2).*
- data members (static and nonstatic)
- function members (static and nonstatic)
- *virtual functions (in Phase 2)*
- auto variables, nonlocal variables, heap variables
- *multiple translation units (in Phase 2)*
- qualified names
- *namespaces (in Phase 2, but in the mean time "using directives" are ignored, so you can use them if you want.)*

Not Supported

- union types, enum types, pointers to functions, pointers to members.

### 4.4 **Standard Conversions**

All standard conversions are supported. Except function to pointer conversions and "pointer to member " conversions.

### 4.5 **Expressions**

Supported:

- id-expressions
- parentheses
- literals (*except wide characters and wide strings in phase 1*)
- this
- Indexing of arrays and pointers with integers.
- function calls.
- Casts using int(f) notation or C(a,b,c) notation.
- postfix ++ and --
- class member selection with . or -> including selection of member functions.
- Casts using new operators static\_cast, reinterpret\_cast, const\_cast.
- *Casts using dynamic\_cast (in Phase 2)*
- sizeof
- prefix ++ --
- new (placement operands are allowed but ignored) You can not define your own new operator.
- delete and delete[]. You can not define your own delete operator.
- unary \*, &, +, -, !, ~
- Casts using old notation: (int)f.

## The Teaching Machine User's Manual

- \*, /, %
- +, - (including on pointers)
- >>, <<
- <, >, <=, >=, ==, != (including pointers)
- ^, &, |
- &&, ||
- ? : (*Phase 2*)
- =, \*=, /=, -=, >>=, <<=, &=, ^=, |=.
- ,

Not supported:

- anything involving the keywords “template”, “typeid”, or “typename”
- pointers to member selection: .\* and ->\*
- psuedo destructor calls

Note

- Precedence and associativity are fully respected. When the order of evaluation of operands is not specified by the standard, the TM (TNG) tends to go left-to-right at the moment.

### 4.6 Statements

Supported

- Expression statements
- Compound statements
- if & if-else statements
- switch, case, & default statements (but you can't use switch to jump over the declaration of any variables)
- while, do-while, for
- break, continue
- return
- declaration statements
- declarations within conditions (in “if”, “while”, & “for”)
- *try-catch, throw (Phase 2)*

Unsupported

- Labels, and goto (*May be supported to a limited extent in Phase 2*).

### 4.7 Declarations

Supported

- Simple-declarations (i.e. variable declarations and function declarations)
- Declarations of structs and classes.
- typedef
- const and volatile annotations.

Ignored

- using directive.
- inline, auto, register, extern, static,

Not Supported

- friend

- mutable (?)
- virtual, explicit
- union
- enum
- namespace definitions
- using declarations
- linkage specifications.
- asm

#### **4.8 Declarators and Function Definitions**

##### Supported

- \*, &, [] (with or without an expression)
- Parameter lists
- Initializers other than aggregate initializers (ISO 8.5.1).
- ctor-initializers

##### Not Supported

- pointer to member
- const or volatile pointers (no excuse for this; for consistency, these qualifiers should be ignored.
- Var-args (i.e. ... in parameter lists)
- Default arguments.
- function-try-block
- Aggregate initializers (ISO 8.5.1).

##### Deviations

#### **4.9 Classes**

##### Supported

- struct and class
- static data members
- nonstatic data members
- Nonvirtual nonstatic function members
- static function members
- *In-place definition of function members. (Coming in Phase 2)*

##### Ignored

- Access specifiers public, private, protected

##### Not Supported

- Unions
- Bit-fields
- Nested classes
- Local classes
- friend
- Constant initializers for const static data members

##### Deviation

-

#### **4.10 Derived classes**

Supported

- Base classes
- Multiple base classes
- *Virtual functions (Coming in phase 2)*
- 

Ignored

- access specifiers on base classes (public, private, protected)

Not Supported

- Virtual inheritance

Deviations

- 

#### **4.11 Access Control**

Access control is not supported. The friend specifier is ignored.

#### **4.12 Special Member Functions**

Supported

- Constructors
- *Destructors (Phase 2. In Phase 1 Destructors are compiled, but never called.*
- Implicit declaration and definition of the default constructor, a copy constructor, a destructor, and an assignment operator.
- Temporary objects are correctly constructed
- Constructor initializations (ctor-initializer)

Ignored

- 

Not Supported

- Conversion functions
- Programmer defined allocation operators (new, delete, new[], delete[]).

Deviations

- Implicitly declared member functions will also be implicitly defined, even if they are never called.
- Trivial constructors and destructors are implicitly declared, defined, and called. This may cause the TM to appear to call trivial routines that a real compiler would not bother with.

#### **4.13 Overloading**

Overloading is supported. There are certain constructs that we do not support and so the overloading rules that apply to them are irrelevant; these include ellipses in functions, default arguments, user defined conversion functions.

#### **4.14 Templates**

Not supported at all.

## 4.15 Exception Handling

Not supported at all in phase 1.

Phase 2 should support exception handling.

## 4.16 Preprocessing Directives

Only include is supported.

- For `#include <name>`: Only the TM's jar file is searched. You can add your own include files by adding them to the jar file in directory `cpp/include/`. The name of the include file should be `name.inc`, if `#include <name>` is to work.
- For `#include "name"`: First `name` is searched for relative to the same "directory" as the top-level file (typically the `.cpp` file). For files loaded over the net, the "directory" is the "document base" for the TM applet. If the file is not found that way, then the file is searched for as if it were included with `<name>`.

Fairly good support for preprocessing is expected in phase 2.

## 4.17 Library

### 4.17.1 Library support in Phase 1

A few library functions and objects are supported. You need to include the appropriate header files.

Note that namespaces are not supported so new library declarations are in namespaces.

Supported or partially supported

- `<iostream>`
  - `cin`, `cout`
  - "`cin >> x`" is supported where `x` is an lvalue of type `char`, another integral type, `float`, or `double`.
  - "`cout << x`" is supported where `x` is of type `char*`, `char`, another integral type, `float`, or `double`.
  - `cin.get( x )` is supported where `x` is an lvalue of type `char`
  - `cout.put( x )` is supported where `x` is an integral type
  - `cin` will NOT convert to a `bool`. Use `cin.fail()` instead.
  - `endl`
- `<math.h>`
  - The following functions with double arguments and double results: `abs` `acos` `asin` `atan` `atan2` `ceil` `cos` `exp` `fabs` `floor` `log` `log10` `pow` `sin` `sqrt` `tan`
- `<cmath>`
  - Everything in `math.h` plus `float` and `long double` versions of `abs` `acos` `asin` `atan` `atan2` `ceil` `cos` `exp` `fabs` `floor` `log` `log10` `pow` `sin` `sqrt` `tan`
- `<string.h>`, `<cstring>`
  - `strcpy`, `strcat`, `strstr`, and `strcmp`
- `<string>`
  - A partial implementation. Not stable enough to document. Look at the include file for details.
- `<new>`

- nothrow

#### **4.17.2 Library support in Phase 2**

In Phase 2 library support will be much improved by support for multiple translation units. Thus any library that can be coded in the supported C++ subset can be handled and users can add their own libraries. However since templates will not be supported, much of the C++ library will remain out of bounds. Other areas (e.g. file I/O) are limited by security restrictions for Java Applets.

#### **4.18 Known Bugs and Deficiencies**

- [TBD]

### **5 Java**

The whole of Java 1.2 is supported, except for concurrency. Consequently there is no support for generics.

Many errors in Java are not diagnosed. You should check your code.

Multiple source files are supported for Java. If you load the file that contains your main function all other files needed will be compiled too. All files to be loaded should be under the current "root" directory (or be supported library files).

#### **5.1 Library**

Library support for Java is very spotty.

The following library classes are at least partially supported. It is best to consult the source code to see the extent of support.

##### **5.1.1 java.io**

java.io.InputStream  
java.io.IOException  
java.io.PrintStream  
java.lang.CharSequence

##### **5.2 java.lang**

java.lang.CharSequence  
java.lang.Integer  
java.lang.Math  
java.lang.Number  
java.lang.Object  
java.lang.String  
java.lang.System

##### **5.3 java.util**

java.util.Iterator  
java.util.Scanner

### **6 Scripting and Plug-ins**

The TM can be dynamically extended via plug-ins written in Java.



## The Teaching Machine User's Manual

The TM supports scripting in the sense that calls in the interpreted source code can be translated into method calls to components or plug-ins of the TM.

Anyone interested in building new plug-ins or using scripting should contact the authors of the TM.