

The Java Interface for MuPAD

by
Jörn Maas

Contents

1	Collection of Examples	1
1.1	Example 1: A Basic Invocation	1
1.2	Example 2: Changing the Classpath	1
1.3	Example 3: Creating Objects I	2
1.4	Example 4: Creating Objects II	3
1.5	Example 5: MatrixEditor	4
1.6	Example 6: Applets	7
1.7	Conclusion	8
2	A Complete User Reference	9
2.1	Package Reference	11
2.1.1	Some words about the Java user package	11
2.1.2	closeAppletFrame	11
2.1.3	console	12
2.1.4	getMatrix	13
2.1.5	getPath	14
2.1.6	getStatus	15
2.1.7	getWorkingDirectory	15
2.1.8	invoke	16
2.1.9	list	19
2.1.10	match	20
2.1.11	matrixEditor	21
2.1.12	runApplet	22
2.1.13	setWorkingDirectory	23
2.1.14	showJavaInfo	24
2.1.15	showMatrix	25
2.1.16	start	26
2.2	Java Reference	29
2.2.1	Some words about the Java user classes	29
2.2.2	DebugConsole	29
2.2.3	Lister	29

3	A Complete Programmers Reference	30
3.1	Module Reference	32
3.1.1	Some words about the Java module	32
3.1.2	callHO_clearList	33
3.1.3	callHO_getClassname	33
3.1.4	callHO_getList	33
3.1.5	callHO_getSignature	34
3.1.6	callHO_removeObject	34
3.1.7	console	34
3.1.8	debug	35
3.1.9	finalize	35
3.1.10	garbageCollect	35
3.1.11	getStatus	35
3.1.12	initJVM	36
3.1.13	invoke	37
3.1.14	list	38
3.1.15	Java_calcMuPAD	38
3.1.16	setShowErrors	39
3.2	Package Reference	40
3.2.1	Some words about the Java programmer package	40
3.2.2	callHO_clearList	40
3.2.3	callHO_getClassname	40
3.2.4	callHO_getList	41
3.2.5	callHO_getSignature	42
3.2.6	callHO_removeObject	42
3.2.7	checkForMatches	43
3.2.8	checkNumber	44
3.2.9	createInterface	45
3.2.10	debug	46
3.2.11	finalize	46
3.2.12	garbageCollect	47
3.2.13	setJavaKey	47
3.3	Java Reference	48
3.3.1	Some words about the Java programmer classes	48
3.3.2	AppletStarter	48
3.3.3	HandleObjects	50
3.3.4	MuPADSecurityManager	50
3.3.5	MuPADHelper	52

Collection of Examples

1.1 Example 1: A Basic Invocation

The following example invokes the method `returnInt` in the class `Prog` that is located somewhere in the classpath. It returns the number the user used as parameter for the Java method.

```
MuPAD
--> package("java"):
    java::Prog::returnInt(42)

Output

42
```

Now we have to start Java manually out of MuPAD, because of the modified classpath (notice the double backslash, which has to be used, since the backslash introduces special

characters and the backslash itself is one such.):

<div style="border-bottom: 1px solid black; margin-bottom: 5px;"> MuPAD </div> <div style="margin-bottom: 5px;"> <pre>>> package("java"): java::start(ClassPath = "C:\\Projects\\java\\"): java::Prog::returnInt(42)</pre> </div> <div style="border-bottom: 1px solid black; margin-bottom: 5px;"> Output </div> <div> <pre>42</pre> </div>
--

1.3 Example 3: Creating Objects I

The two examples already presented cover most of the situations a normal user comes in touch with. Nevertheless, there exist situations (if for example graphical applications come into play, or two different Java methods need to exchange data) in which it is unavoidable to create Java objects and reuse them later again in Java methods as parameters.

So now we assume, a class **First** exists, containing a method **returnUnknownObject**. This method returns a Java object, which is unknown to MuPAD but nonetheless can be used in Java. Furthermore assume, there exist a second class called **Second**, containing a method **useUnknownObject**, which needs an Instance of such an object as parameter and returns something, say 42, if the object was such an instance.

For a better understanding the two classes and their important methods are presented:

```
public class First {
    public static UnknownObject returnUnknownObject() {
        ...
    }
    ...
}

public class Second {
    public static int useUnknownObject(UnknownObject uo) {
        ...
        return 42;
    }
    ...
}
```

Again we assume that the classes are located in the directory specified in example 2. To first create the unknown object, we have to state:

```
MuPAD —————  
>> package("java"): java::start(ClassPath = "C:\\Projects\\java\\"):  
      unknownObject := java::First::returnUnknownObject()  
————— Output —————  
  
"@MuPAD_JNI_00000000"
```

The returned String defines the Java object representative for the stored Java object `UnknownObject`. Now we can use this representative to invoke the Java method of the second class:

```
MuPAD -----  
>> result := java::Second::useUnknownObject(unknownObject)  
----- Output -----
```

42

1.4 Example 4: Creating Objects II

If now the methods we need to use are not static but rely on a previously instantiated object we have to use this object and invoke the method on it.

In order to keep it simple, we modify the example from 1.3 as follows:

```
public class First {
    public First() {
        ...
    }

    public UnknownObject returnUnknownObject() {
        ...
    }

    public int useUnknownObject(UnknownObject uo) {
        ...
        return 42;
    }
    ...
}
```

Both Java methods reside now in one class and are no more static. In order to invoke `returnUnknownObject`, an instance of the class `First` has to be created that can be used for further invocations:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> package("java"):
    java::start(ClassPath = "C:\\Projects\\java\\"):
    class := java::First::First()
    ──── Output ─────────────────────────────────────────────────────────────────────────────────┘

"@MuPAD_JNI_00000000"
└────────────────────────────────────────────────────────────────────────────────────────────────┘

```

Now we can use this object and invoke its methods:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> unknownObject := java::First::returnUnknownObject(UseObject=class)
    ──── Output ─────────────────────────────────────────────────────────────────────────────────┘

"@MuPAD_JNI_00000001"
└────────────────────────────────────────────────────────────────────────────────────────────────┘

```

Finally we get the result after a second use of the instantiated object `class`

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> result := java::First::useUnknownObject(unknownObject, UseObject=class)
    ──── Output ─────────────────────────────────────────────────────────────────────────────────┘

42
└────────────────────────────────────────────────────────────────────────────────────────────────┘

```

1.5 Example 5: MatrixEditor

The following is a rather complex example and demonstrates a complete interaction in practice. The `MatrixEditor` was integrated into the Java package and can easily be invoked via `java::matrixEditor`¹. Instead of using `java::matrixEditor` (which would be fairly simple) it will be explained how the original project *Taberion* is invoked.

It is assumed, that the original project *Taberion* and another necessary file (*kunststoff.jar*) are located in `C:\Projects\Taberion\classes\`.

¹For more details about invoking the `MatrixEditor` have a look at [2.1.11](#) on page 21

```

MuPAD
--> package("java"):
    CPath := "C:\\Projects\\Taberion\\classes\\";
           "C:\\Projects\\Taberion\\classes\\kunststoff.jar;";
    DPath := "%MUPAD_JRE_PATHjre\\bin\\client\\":
    java::start(ClassPath = CPath, DLLPath = DPath)

----- Output -----

true

```

First the package is loaded and then the classpath is defined. The project needs access to a special graphical environment, which is inside the `kunststoff.jar` file. A jar file can be stated within a classpath and then used as if the directory itself was stated. Thus the mere mentioning in the classpath at startup is sufficient.

The path defined by a user will be appended *before* the default path, so in spite of being a part of the Java package, the matrix editor will be loaded from the **Projects** directory since it will be found there first. This way it is assured that changes can be tested without interfering with any classes predefined by MuPAD.

The location of the library was changed for demonstration purposes and now uses the `client` JVM instead of the predefined `server` version. Now that the preparations are made, the main class can be used to make a call to the constructor and create a `Taberion` object:

```

MuPAD
--
>> a := [[1, 2, 4],[2, 3, 5],[3, 4, 7]]:
    b := java::taberion::Taberion::Taberion(a, TRUE)
--
Output
--
The creation of the object did work. It is stored as @MuPAD_JNI_00000000
@MuPAD_JNI_00000000

```

The Java object is now stored and can henceforth be referred to from MuPAD by using `b`, or by directly using the internal representative `@MuPAD_JNI_00000000`.

Concurrently a Java window opens displaying the program Taberion (as presented in Figure 1.1).

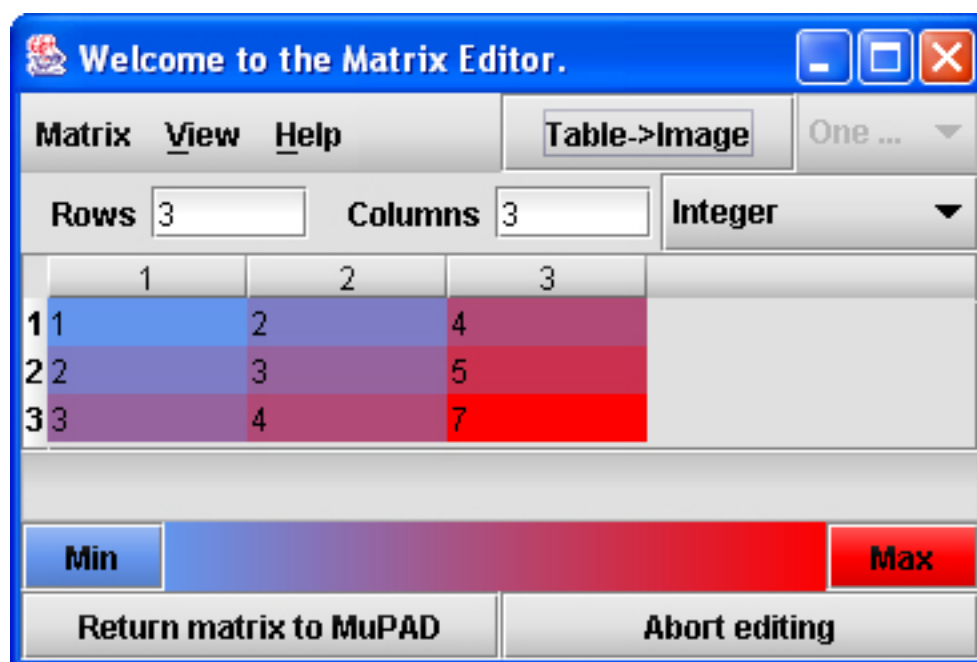


Figure 1.1: A Matrix Editor for MuPAD (Edit Mode).

The assignment to variable b was only done to save some typing.

Remember: A variable *unknown* can hold an object with the internal name @MuPAD_JNI_00000000. It is then of no importance if the user enters
 java::class::method(UseObject=unknown) or
 java::class::method(UseObject="@MuPAD_JNI_00000000").
 The result will be the same, since the representative is stored as a string.

The creation used the non modal form of the MatrixEditor and therefore MuPAD is not waiting for any user actions inside the Java application.

A list representation of the actual matrix in the MatrixEditor can now be retrieved by invoking the respective method of the object:

```

MuPAD
--
>> c := java::taberion::Taberion::getIntegerMatrix(UseObject=b)
--
Output
--
Using the object @MuPAD_JNI_00000000
[[1, 2, 4],[2, 3, 5],[3, 4, 7]]

```

Variable c holds now a list of lists containing the data of the matrix in Taberion. This way

the user is able to manipulate matrices and get them from the Java program on the fly.

It is not difficult at all to invoke any method in any Java program. The user should be careful nevertheless, because although it is possible for example to call a private declared method, it is not advisable to do so.²

Now that an object is created, which can be used by MuPAD directly, a seamless integration is achieved. The result could for example be used to create a Graph³ and run a shortest path search on it.

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> G := Graph::createGraphFromMatrix(matrix(c)):
    Graph::shortestPathAllPairs(G)
    ─────────── Output ───────────────────────────────────────────────────────────────────────────┘

-- table(                                     --
|   (3, 3) = 0,                               |
|   (3, 2) = 5,   table(                     |
|   (3, 1) = 4,   (3, 2) = 3,                 |
|   (2, 3) = 4,   (3, 1) = 3,                 |
|   (2, 2) = 0,,   (2, 3) = 2,                 |
|   (2, 1) = 2,   (2, 1) = 2,                 |
|   (1, 3) = 3,   (1, 3) = 1,                 |
|   (1, 2) = 2,   (1, 2) = 1                 |
|   (1, 1) = 0   )                             |
-- )                                     --

```

1.6 Example 6: Applets

Applets can not be invoked directly, so a package procedure must be used instead.

For this example we assume, that an applet called Trisentis exists, which is also located in the directory used in the examples [1.2](#), [1.3](#) and [1.4](#).

In the example we see, that two objects are created. The first (**applet**) holds the actual applet to be used. The second (**program**) is needed later.

The following example is enough to make an applet run in MuPAD:

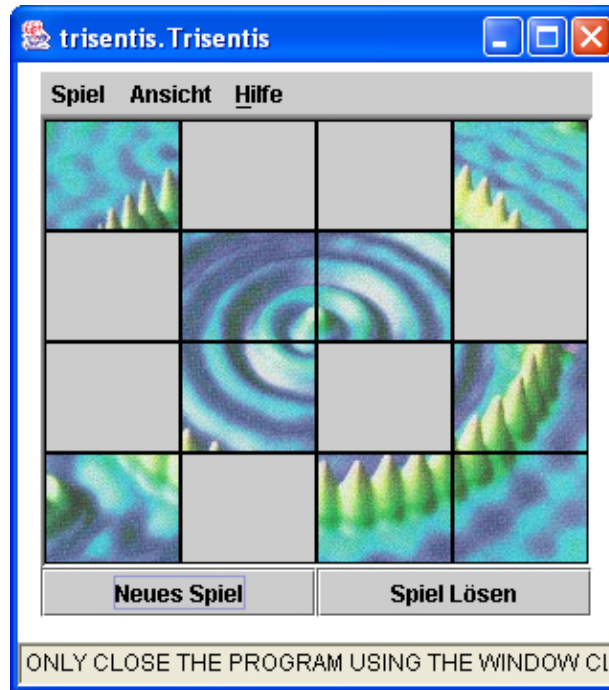
²Most often programmers know exactly what they do when they use *access specifiers* to limit access. Only methods declared **public** are meant to be invoked by methods outside the actual package.

³How to create and handle Graphs in MuPAD can be looked up in the online help of MuPAD

MuPAD

```
>> java::start(ClassPath = "C:\\Projects\\java\\"):
java::setWorkingDirectory("C:\\Projects\\java\\"):
applet := java::Trisentis::Trisentis():
program := java::runApplet(applet)
```

Graphics



The variable `program` was used since sometimes applets refuse to shutdown, or the frame containing the applet will not close. In such a situation, the following command closes the frame for good:

MuPAD

```
>> java::closeAppletFrame(program):
```

1.7 Conclusion

There is no limit as to how Java and MuPAD can work together. Arbitrary Java results can be used to gain results for MuPAD-internal routines and afterwards handed back to a Java method.

Chapter 2

A Complete User Reference

Altogether there exist two different parts, which are taken into account, so this user reference is divided into two parts.

The first part contains all the information needed to use the Java package in MuPAD.

The second part, which starts on page [30](#) contains information about modules, packages and classes that the normal MuPAD-user has no need of and usually does not come in contact with.

Nevertheless they might prove helpful for people who are developing with the aid of this implementation or want to extend it.

MuPAD Code Documentation (Packages) [See section [2.1](#) at page [11](#)]

Procedure	Description	Page
overview	Some words about the Java user package	11
console	opens a Java console	12
getMatrix	receives a matrix of a matrixEditor object	13
getPath	operating system dependent path transformation	14
getStatus	returns the state of the JVM	15
getWorkingDirectory	gets the Java user directory	15
invoke	calls a Java method	16
list	lists the methods in a Java class	19
match	sets/gets the default matching	20
matrixEditor	calls the matrixEditor	21
closeAppletFrame	closes the JFrame containing the applet	11
runApplet	executes a specified applet	22
setWorkingDirectory	sets the Java user directory	23
showJavaInfo	prints details about a JVM setting	24
showMatrix	reopens the window of a matrixEditor object	25
start	starts the Java virtual machine	26

Table 2.1: Short Reference for the Java package

Java Code Documentation (Classes) [See section [2.2](#) at page [29](#)]

Class	Description	Page
overview	Some words about the Java classes	29
DebugConsole	The console that shows the Java output	29
Lister	Retrieves information about another class	29

Table 2.2: Short Reference for the Java Classes

2.1 Package Reference

2.1.1 Some words about the Java user package

Some predefined variables can be used throughout the use of the package. These variables will be replaced according to the operating system used on. If an option or its value is surrounded by angle brackets, it means that this option or the bracketed value is the default.

%MUPAD_PATH	defines the root path of the MuPAD installation. for example :
	Operating system Result
	Windows C:\Programme\SciFace\mupad Pro 3.0\
	Unix/Linux /usr/local/bin/mupad/
%MUPAD_JAVA_PATH	adds packages\java\ to %MUPAD_PATH for example :
	C:\Programme\SciFace\mupad Pro 3.0\packages\java\
	The directory separator is chosen automatically depending on the operating system that is used.
	Also a file separator is automatically added at the end.
%MUPAD_JRE_PATH	adds javart\ to %MUPAD_JAVA_PATH for example :
	C:\Programme\SciFace\mupad Pro 3.0\packages\java\javart\
	The directory separator is chosen automatically depending on the operating system that is used.
	Also a file separator is automatically added at the end.
@NULL	(String) can be used to hand a NULL object in invocations as a parameter. WARNING: A String must be passed and therefore the quotation marks have to be used.
	for example :
	java::TestClass::testMethod("@NULL", 5, Match=Best)

2.1.2 closeAppletFrame

This procedure closes the JFrame that contains the applet. Sometimes an applet can not be closed directly but after invoking this procedure it will be forced to do so.

Call:

runApplet(jframe)

Parameters:

jframe — The JFrame that contains the applet.

Returns: null.

The following example does not provide any graphical output since it would just consume space.

Example:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> applet1 := java::trisentis::Trisentis::Trisentis():
    program := java::runApplet(applet1):
└────────────────────────────────────────────────────────────────────────────────────────────────┘

```

Now let us assume, the applet can not be closed by itself. All the user has to do is to use the following command:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> java::closeAppletFrame(program)
└────────────────────────────────────────────────────────────────────────────────────────────────┘

```

2.1.3 console

brings up a console that displays all the output, which was written to `System.out` and `System.err` during the session by any Java program called from within MuPAD. The window can be closed and reopened anytime without losing the content inside. Inside the text field the user is able to edit the text completely. Text can therefore be added or deleted. The button in the south of the frame deletes the whole content of the text field, leaving it blank.

Call:

`console()`

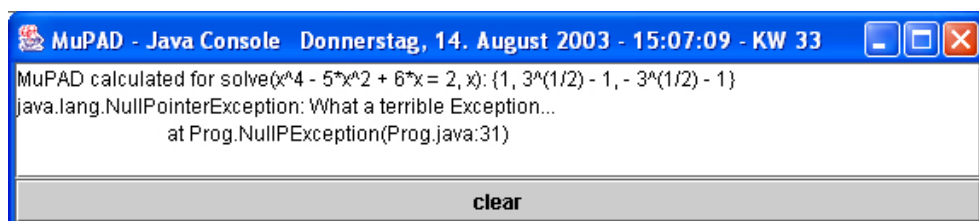
Returns: null.

Example:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> java::console()
    ┌── Graphics ─────────────────────────────────────────────────────────────────────────────────┘

```



2.1.4 getMatrix

retrieves the matrix stored in a former Java matrix object. If the stated object is not of the correct type, MuPAD will raise an error message.

Calls:

```
getMatrix(sign)
getMatrix(sign, [Integer | <FloatingPoint> | String ])
getMatrix(sign, [<Matrix> | List])
```

Parameters:

sign — Has to be a stored Java matrix created with the Matrix Editor [2.1.11].

Options:

Integer	—	The values of each cell will be an integer number.
FloatingPoint	—	The values of each cell will be a floating point number. (Default)
String	—	The values of each cell will be a String.
List	—	The result will be returned as a List.
Matrix	—	The result will be returned as a matrix (Default).

Returns: Either a matrix or a list containing the columns of the matrix represented as a list.

Example 1:

```
┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> a1 := java::matrixEditor(3, NonModal):
    java::getMatrix(a1, Integer)
    ─────────── Output ───────────────────────────────────────────────────────────────────────────
    [Depending on what the user did to the matrix! Default would be:]
    +-      +-
    |  0, 0, 0  |
    |           |
    |  0, 0, 0  |
    |           |
    |  0, 0, 0  |
    +-      +-
└────────────────────────────────────────────────────────────────────────────────────────────────┘
```

Example 2:

```
┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> a2 := java::matrixEditor(3, NonModal):
    java::getMatrix(a2, String, List)
└────────────────────────────────────────────────────────────────────────────────────────────────┘
```

 Output

```
[Depending on what the user did to the matrix! Default would be:]
[["0", "0", "0"], ["0", "0", "0"], ["0", "0", "0"]]
```

2.1.5 getPath

returns the path as it could be used with the current operating system. Additionally it converts used predefined variables (see 2.1.1 on page 11) into proper directory names.

Call:

getPath(path)

Parameters:

path — a path to be converted depending on the operating system.

Returns: The platform-dependent transformed path.

Example 1:

 MuPAD

```
>> java::getPath("/user/my/")
```

 Output

```
Windows      : \user\my\
Unix/Linux   : /user/my/
```

Also the directory variables %MUPAD_PATH, %MUPAD_JAVA_PATH and %MUPAD_JRE_PATH can be included in the path.

Example 2:

 MuPAD

```
>> java::getPath("%MUPAD_JAVA_PATHadditional")
```

 Output

```
Windows      : C:\Programme\SciFace\MuPAD Pro 3.0\packages\java\additional\
Unix/Linux   : /usr/local/bin/mupad/packages/java/additional/
```

Missing file separators are only added after the directory variables. If they are not explicitly set, they will not be added at the end of the newly created string. Also it does not matter

if \ or / is used for separating. Depending on the operating system the correct separator will be inserted. (2nd example)

2.1.6 getStatus

returns the current status of the Java Virtual Machine. The result can be either one of:

- 1 The JVM was terminated abnormally.
- 0 The JVM is up and running.
- 1 The JVM is not initialized.

If -1 is returned, the user should either reconnect the notebook (under Windows) or exit MuPAD and restart it (under Unix/Linux). This state can only be reached if some severe error had happened inside the JVM. In most cases the programmer of the Class did forget to catch exceptions, which lead to killing his application and then the whole JVM.

Call:

```
getStatus()
```

Returns: -1, 0, 1 depending on the state of the JVM.

Example:

```
MuPAD
>> java::getStatus()

Output

0
```

2.1.7 getWorkingDirectory

returns the current working directory of the user in the JVM.

Call:

```
getWorkingDirectory()
```

Returns: The users working directory as set in the JVM.

Example:

```
MuPAD
--> java::getWorkingDirectory()
-- Output --
```

C:\Programme\SciFace\mupad Pro 3.0\packages\java

2.1.8 invoke

The main procedure that handles all invocations. The call is analyzed and mapped (according to the parameters) to the corresponding module function. Under normal circumstances the user does not need to use this procedure, but instead uses the more intuitive way described below. *This procedure is not accessible via tab completion. It is intended for internal use or in special cases only.*

Calls:

`invoke(dom)`

`invoke(dom, p_1 [, p_2 , ..., p_n])`

`invoke(dom, p_1 [, p_2 , ..., p_n], Match = [First | \langle Max \rangle | Best])`

`invoke(dom, p_1 [, p_2 , ..., p_n], Match = Native, Signature = sign)`

`invoke(dom, p_1 [, p_2 , ..., p_n], ShowMethod = [TRUE | FALSE])`

Parameters:

`dom` — a complete Java domainname.

`$p_1 \dots p_n$` — parameter(s) required by the Java method.

Options:

`Match` — **First**
The first possible (according to the parameters) method is used for invocation.

— **Max**
default: A ranking system is used to decide, which method to use for invocation. (The return value is never included).

Type	Subtype (Java Type)	Points
Integer	BigInteger	5
	Long	4
	Int	3
	Short	2
	Byte	1
Floating Point	BigDecimal	3
	Double	2
	Float	1
Text	String	2
	Char	1

(continued on next page.)

Type	Subtype (Java Type)	Points
Other	Boolean	1
	Array	100
	object	1
	unknown	1

The ranking system is only useful if the method to be invoked appears with different signatures. For example (Java methods):

1) public int signatureTest(int integerNumber1, int integerNumber2)
 2) public float signatureTest(float floatNumber1, int integerNumber2)
 3) public BigDecimal signatureTest(BigDecimal bdNumber1, int integerNumber2)

Method	1	2	3
Points	$3 + 3 = 6$	$1 + 3 = 4$	$5 + 3 = 8$

If the user would call the method via

```
java::Classname::signatureTest(10, 10, Match=Max)
```

then method 2 and 3 would not be taken into consideration, since 10 is no floating point number. If the parameters would be (10.2, 10) then method 1 would not be used for ranking.

— Best

The ranking is the same as in Max matching but this time the smallest deviation is taken into account (a Min matching).

If the user would call the method via

```
java::Classname::signatureTest(12345.67, 10, Match=Best)
```

then method 2 would be chosen, since 12345.67 is within the range of a float.

— Native

The method with the given native signature is used. If a native matching was explicitly demanded, the signature has to be stated, too!

Signature

— (S)R

Valid arguments for S and R are:

Signature	Z	C	B	S	I	J	F
Java type	Boolean	Char	Byte	Short	Int	Long	Float
Signature	D	L	V				
Java type	Double	Other	Void				

whereas other can be any valid Java class-signature.

It has to be used like `L + Classname + ;`

`Classname` has to be stated in UNIX form.

For example : `Ljava/lang/Object;` would use an Object.

The signatures for the three methods mentioned above are:

Method	1	2	3
Signature	(II)I	(FI)F	(Ljava/math/BigDecimal;I)LJava/math/BigDecimal;

ShowMethod

— TRUE or FALSE

Before the method is invoked, the complete Java signature is printed. For example:
 Using static method 'public static java.math.BigDecimal Prog.numberTest(java.math.BigDecimal,boolean)'
 UseObject — @MuPAD_JNI_xxxxxxx
 If an unknown Java object (no primitive type) was returned by the JVM, it is stored and a literal representation will be returned. It always begins with MuPAD_JNI_ followed by an eight digit number. The number starts with 0 and increases with every object returned by the JVM. Since the number is always eight digits long, preceding spaces will be filled with 0's. The instantiated object behind the representative will be used to invoke the given method. This is useful if for example a class is used to store and display arbitrary data.

Returns: The result of the invocation.

A complete Java domainname always begins with the keyword `java`, followed by the class-name and the methodname of the method to be invoked.

For example: $\underbrace{java}_{\text{keyword}} :: \underbrace{java :: lang :: Object}_{\text{classname}} :: \underbrace{toString()}_{\text{methodname}}$

In this example `java::lang::Object` is the class to be used and `toString` the method to be used.

Assume, the result was stored in a variable `a` (one could also state the constructed identifier (`@MuPAD_JNI_00000000`) directly) and we want to invoke the method `toString` on it. This can now easily be done by typing:

$\underbrace{java}_{\text{keyword}} :: \underbrace{java :: lang :: Object}_{\text{the operational object}} :: \underbrace{toString(UseObject=a)}_{\text{methodname and used object}}$

Be careful to use the correct class for invocation! It is possible to do the following:

```
>> java::mypackage::graphics::Triangle::toString(UseObject=a)
```

It would use the method `toString` from a completely unknown `Object` and invoke it on the stored object. This is only possible if the objects derive from one another (like for example `Class` derives from `Object`)

So if `UseObject` is stated and an arbitrary `Object` stored as `@MuPAD_JNI_00000000` handed, the name of the already created `Object` would be returned.

WARNING! Since the matching is just checking if the type is unknown (something different from primitive types, Strings, BigIntegers and BigDecimals) it is possible to hand for example an applet where a `JPanel` would be expected. To do so could lead to a crash of the kernel!

As a parameter the keyword "`@NULL`" can be passed to define a `NULL-Object`. Remember

that it is a String, which is passed and therefore needs the quotation marks.

2.1.9 list

returns a list containing three additional lists. The first contains the constructors, the second the methods and the third the fields of the class. If the parameter **ShowInJava** is stated, a Java window will be opened that shows all available constructors, methods and fields of a given class and its superclass(es) depending on the access specifier stated.

Calls:

```
list(classname)
```

```
list(classname, AccessSpecifier = [Private | Protected | <Public> ])
```

```
list(classname, ShowInJava)
```

```
list(classname, ShowInMuPAD)
```

```
list(classname, ReturnList)
```

Parameters:

classname — the complete classname to be listed.
It has to use / (slash) as a separator.
For example : Java/lang/System

Options:

AccessSpecifier	—	Private displays <i>every</i> constructor, method and field in the class and its superclasses.
	—	Protected displays all constructors and all methods and fields except the private ones.
	—	Public <i>default</i> : displays all constructors and only the public methods and fields.
ShowInJava	—	displays a Java popup containing all the information about the constructor, the methods and the fields in the class (and its superclasses) and returns a null.
ShowInMuPAD	—	displays the information as formatted output in MuPAD and returns a null.
ReturnList	—	forces the return of the list.

Returns: A list containing the constructors, methods and fields according to the AccessSpecifier. If either ShowInJava or ShowInMuPAD was set and ReturnList is not explicitly set, no list will be returned.

Example:

— MuPAD —

```
>> java::list("HandleObjects", ShowInMupad)
```

```
----- Output -----
```

```
Number of constructors : 1
Number of methods      : 20
Number of fields       : 0
```

```
public HandleObjects()
```

```
public java.lang.String[] HandleObjects.getList(boolean)
```

```
...
```

2.1.10 match

It sets the default match to be used for future Java methods invocations and/or returns the last default match setting.

Calls:

```
match()
```

```
match(Match = [First | <Max> | Best])
```

Options:

```
Match  —  First | <Max> | Best
```

Returns: the last setting for the default matching.

The **native** matching was excluded, since every call with a native match needs a signature that can not be set permanently.

Example 1:

```
└─ MuPAD
```

```
>> java::match(Match = Max)
```

```
----- Output -----
```

```
2
```

Example 2:

```
└─ MuPAD
```

```
>> java::match()
```

```
----- Output -----
```

2.1.11 matrixEditor

starts the Java program *Taberion* that can be used in a modal state (default) or non-modal. The program was created to modify Matrices of arbitrary dimension.

Calls:

```
matrixEditor(input)
matrixEditor(input, [List | <Matrix>])
matrixEditor(input, [<Modal> | NonModal])
```

Parameters:

input	— Has to be either a <code>DOM_LIST</code> , a <code>Dom::Matrix</code> or a <code>DOM_INT</code>
-------	---

Options:

Modal	— displays the Java program Taberion in a modal state. Additionally two buttons are added to the mainframe. These are called Return matrix to MuPAD and Abort editing . The control will only be returned to MuPAD after the user exited the Java program. No object is stored and the matrix must be handed again to alter it.
NonModal	— An object is created and stored on the MuPAD side. While the main Java program is running, the user can still use MuPAD to calculate. The window does not contain the two additional buttons. If the user needs a matrix it has to be retrieved with a call to the Method <code>getMatrix</code> [2.1.4]. <i>If NonModal was stated, the options List and Matrix are superfluous.</i>
List	— The result will be returned as a list containing the columns of the matrix represented as a list.
Matrix	— The result will be returned as a matrix.

Returns: Either a matrix or a list containing the columns of the matrix represented as a list.

Example 1:

```

MuPAD
--
>> java::matrixEditor(3, Matrix, Modal)
--
Output
--
[Depending on what the user did to the matrix! Default would be:]
+-      +-

```

```

| 0, 0, 0 |
| 0, 0, 0 |
| 0, 0, 0 |
+-      +-

```

Example 2:

```

MuPAD
--
>> java::matrixEditor(3, List)
--
Output
--
[Depending on what the user did to the matrix! Default would be:]
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]

```

2.1.12 runApplet

This procedure executes the stated applet with the given arguments and resizes the applet according to the definitions.

Calls:

```

runApplet(applet)
runApplet(applet, arguments)
runApplet(applet, Width=width, Height=height)
runApplet(applet, arguments, Width=width, Height=height)

```

Parameters:

applet	—	The preloaded applet that is to be executed
arguments	—	The arguments stated in a list. Each argument must be a string so that the applet can parse it. Default = an empty list

Options:

Width	—	The width of the frame in pixels. (Default = 640)
Height	—	The height of the frame in pixels. (Default = 480)

Returns: a frame object (stored as object representative).

The following examples do not provide any graphical output since they would just consume space.

Example 1:

```
MuPAD -----
>> applet1 := java::trisentis::Trisentis::Trisentis():
      java::runApplet(applet1)
```

Example 2:

```
MuPAD
>> applet1 := java::trisentis::Trisentis::Trisentis():
      java::runApplet(applet1, Width=400, Height=200)
```

Example 3:

```
MuPAD
>> applet1 := java::trisentis::Trisentis::Trisentis():
      java::runApplet(applet1, ["Green","5"], Width=400, Height=200)
```

2.1.13 setWorkingDirectory

sets the current working directory of the JVM and returns the previous one. The predefined package variables can be used to set the path (see 2.1.1 on page 11).

Call:

```
setWorkingDirectory(directory)
```

Parameters:

`directory` — the new working directory for the JVM.

Returns: The last working directory of the JVM.

The path is converted depending on the operating system. (see 2.1.5 on page 14)

Example 1:

```
MuPAD -----  
>> java::getPath("%MUPAD_JAVA_PATHadditional")  
----- Output -----
```

C:\Programme\SciFace\mupad Pro 3.0\packages\Java\additional\

Example 2:

```

MuPAD
>> java::setWorkingDirectory("/Myprojects/Trisentis")
      Output
C:\Myprojects\Trisentis

```

2.1.14 showJavaInfo

retrieves information from the currently running JVM. If no parameter is stated, a list of possible invocations is presented. The numbers that can be used are listed together with their associated values in Table 2.4.

Call:

`showJavaInfo(key)`

Parameters:

key — the number defining the associated value.

Returns: The value of the key.

The key can be chosen out of the following list:

Nr.	Description of Associated Value
1	Java Runtime Environment version
2	Java Runtime Environment vendor
3	Java vendor URL
4	Java installation directory
5	Java Virtual Machine specification version
6	Java Virtual Machine specification vendor
7	Java Virtual Machine specification name
8	Java Virtual Machine implementation version
9	Java Virtual Machine implementation vendor
10	Java Virtual Machine implementation name
11	Java Runtime Environment specification version
12	Java Runtime Environment specification vendor
13	Java Runtime Environment specification name
14	Java class format version number
15	Java class path
16	List of paths to search when loading libraries
17	Default temp file path

Table 2.4: Keys for the JVM and their associated values (continued on next page page)

Nr.	Description of Associated Value
18	Name of JIT compiler to use
19	Path of extension directory or directories
20	Operating system name
21	Operating system architecture
22	Operating system version
23	File separator ("/" on UNIX)
24	Path separator (":" on UNIX)
25	Line separator ("\n" on UNIX)
26	User's account name
27	User's home directory
28	User's current working directory

Table 2.4: Keys for the JVM and their associated values

Example 1:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> java::showJavaInfo(10)
      ─────────── Output ───────────────────────────────────────────────────────────────────────────┐
      Java HotSpot(TM) Server VM
└──────────────────────────────────────────────────────────────────────────────────────────────────┘

```

Example 2:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> java::showJavaInfo(4)
      ─────────── Output ───────────────────────────────────────────────────────────────────────────┐
      C:\PROGRA~1\SciFace\MUPADP~1.0\packages\java\javart\jre
└──────────────────────────────────────────────────────────────────────────────────────────────────┘

```

2.1.15 showMatrix

Makes a formerly created and closed Matrix Editor visible again. If the stated object is not of the correct type, MuPAD will raise an error message.

Call:

```
showMatrix(sign)
```

Parameters:

- sign — Has to be a stored Java matrix created with the Matrix Editor [2.1.11].

Returns: null.

Example:

```
>> a1 := java::matrixEditor(3, NonModal):  
      java::showMatrix(a1)
```

2.1.16 start

It loads a Java virtual machine into memory. If no parameters are stated, default values will be used for start up. The predefined package variables can be used to set the path (see 2.1.1 on page 11). Either *true* (the JVM is up and running) or *false* (the JVM could not be initialized) will be returned. A defined path is appended *before* the default class- or library-path.

Calls:

```
start(ClassPath = cp)
start(LibraryPath = lp)
start(DLLPath = dll)
start(UseSecurityManager = [<TRUE>|FALSE])
start(UseLogFile = [TRUE|<FALSE>])
start(UseOptionList = [See enumeration below])
```

Options:

cp	—	The classpath to be used for the JVM.
lp	—	The librarypath to be used for the JVM.
dllp	—	The path where the file <code>jvm.dll</code> (Windows) or <code>libjvm.so</code> (Unix/Linux) is located.
UseSecurityManager	—	If set to <code>FALSE</code> , the security manager is not used when the JVM is initialized.
UseLogFile	—	If set to <code>TRUE</code> , all <code>System.out</code> and <code>System.err</code> messages will be written to the file <code>MuPADout.log</code> and <code>MuPADerr.log</code> . They are located in the user directory of the JVM. The user directory can be retrieved with a call to <code>getWorkingDirectory</code> [2.1.7]
UseOptionList	—	Allows to hand additional parameters to the JVM. Below is a list of valid commands for the JVM 1.4

Returns:

- TRUE — The JVM is up and running.
- FALSE — Something went wrong while initializing the JVM.

Default classpath is set to

Windows: %MUPAD_JAVA_PATHclasses\;%MUPAD_JRE_PATHjre\lib\rt.jar;
 Unix/Linux: %MUPAD_JAVA_PATHclasses/:%MUPAD_JRE_PATHjre/lib/rt.jar:

Default library path is set to

Windows: %MUPAD_JRE_PATHjre\lib\;%MUPAD_JAVA_PATHclasses\
 Unix/Linux: %MUPAD_JRE_PATHjre/lib/i386/:%MUPAD_JAVA_PATHclasses/:

Default JVM path is set to

Windows: %MUPAD_JRE_PATHjre\bin\server\
 Unix/Linux: %MUPAD_JRE_PATHjre/lib/i386/server/

Before handing the parameters to the Java the following changes are made:

classpath gets the prefix `-Djava.class.path=`

librarypath gets the prefix `-Djava.library.path=`

dllpath gets the suffix

Windows: `jvm.dll`
 Unix/Linux: `libjvm.so`

Additional parameters, which can be stated are:

```
-D<name>=<value>
    set a system property
-verbose[:class|gc|jni]
    enable verbose output
-ea[:<packagename>...|:<classname>]
-enableassertions[:<packagename>...|:<classname>]
    enable assertions
-da[:<packagename>...|:<classname>]
-disableassertions[:<packagename>...|:<classname>]
    disable assertions
-esa | -enablesystemassertions
    enable system assertions
-dsa | -disablesystemassertions
    disable system assertions
```

Also some parameters can be stated that might prove interesting in special cases. The *X* parameters are subject to change without prior notice in any Java version.

```

-Xmixed          mixed mode execution (default)
-Xint            interpreted mode execution only
-Xbootclasspath:<directories and zip/jar files separated by ;>
                 set search path for bootstrap classes and resources
-Xbootclasspath/a:<directories and zip/jar files separated by ;>
                 append to end of bootstrap class path
-Xbootclasspath/p:<directories and zip/jar files separated by ;>
                 prepend in front of bootstrap class path
-Xnoclassgc      disable class garbage collection
-Xincgc          enable incremental garbage collection
-Xloggc:<file>    log GC status to a file with time stamps
-Xbatch          disable background compilation
-Xms<size>       set initial Java heap size
-Xmx<size>       set maximum Java heap size
-Xss<size>       set java thread stack size
-Xprof           output cpu profiling data
-Xrunhprof[:help] |[:<option>=<value>, ...]
                 perform JVMPI heap, cpu, or monitor profiling
-Xdebug          enable remote debugging
-Xfuture         enable strictest checks, anticipating future default
-Xrs             reduce use of OS signals by Java/VM (see documentation)
-Xcheck:jni      perform additional checks for JNI functions

```

Example 1:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> java::start(ClassPath = "%MUPAD_JAVA_PATHmyJavaPath/classes/",
                LibraryPath = "%MUPAD_JAVA_PATHmyJavaPath/libraries/")
    ──── Output ───────────────────────────────────────────────────────────────────────────┐
    TRUE
└──────────────────────────────────────────────────────────────────────────────────────────┘

```

Example 2:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> java::start(UseLogFile=TRUE, UseOptionList=["-Xms16M", "-Xmx128M"])
    ──── Output ───────────────────────────────────────────────────────────────────────────┐
    TRUE
└──────────────────────────────────────────────────────────────────────────────────────────┘

```

2.2 Java Reference

2.2.1 Some words about the Java user classes

Since it would be useless to enumerate all the Java methods that were created for the thesis it will be restricted to the classes and the possibilities they offer. Also only the classes, which are directly linked to MuPAD are enumerated. Classes like `SimpleCalc` or `Taberion` are not explicitly mentioned.

2.2.2 DebugConsole

This class was created to redirect output to `System.out` and `System.err`. The output will be presented in a `JTextField` so additional input from a user is possible (if some statements are to be inserted before copying or between several runs it proved to be without problems). Only the constructor is available because the interaction between the user and the console is limited and only the constructor (and an inner class redirecting all output) is needed.

- `DebugConsole()`
Creates a Console that redirects the output into a `JTextField`. As default the frame is not shown and has to be opened by the user explicitly (see [2.1.3](#) on page 12 for more details).

2.2.3 Lister

This class provides the user with a nice GUI that shows the constructor(s), method(s) and field(s), which were handed to it. See [2.1.9](#) on page 19 for a detailed overview of the possible invocations from MuPAD.

- `Lister(String[] constructors, String[] methods, String[] fields)`
Creates a new Lister object and adds the constructors, methods and fields to the according tabs and shows them.
- `static int listClasses(String[] constructors, String[] methods, String[] fields)`
Shows the handed constructors, methods and fields in a frame with tabs for each selection. Until now the return value *always* returns 0.

Chapter 3

A Complete Programmers Reference

Altogether there are three different aspects taken into account and therefore this reference is divided into three parts. The module reference is new and the Package and Class Reference contains only the procedures and Classes not mentioned before:

Native Code Documentation (Modules) [See section 3.1 at page 32]

Function	Description	Page
overview	Some words about the Java module	32
callHO_clearList	removes all stored Java objects	33
callHO_getClassname	gets the classname of a Java object	33
callHO_getList	gets/shows all stored Java objects	33
callHO_getSignature	gets the signature of a Java object	34
callHO_removeObject	removes a specified Java object	34
console	opens a Java console	34
debug	sets/gets the debugging level	35
finalize	tries to finalizes objects in the JVM	35
garbageCollect	tries to garbage collect objects in the JVM	35
getStatus	returns the state of the JVM	35
initJVM	starts a JVM	36
invoke	calls a Java method	37
Java_calcMuPAD	allows a callback to MuPAD from Java	38
setShowErrors	defines if module errors are printed	39

Table 3.1: Short Reference for the Java Module

MuPAD Code Documentation (Packages) [See section 3.2 at page 40]

Procedure	Description	Page
overview	Some words about the Java programmer package	40

Table 3.2: Short Reference for the Java package (continued on next page page)

Procedure	Description	Page
callHO_clearList	removes all stored Java objects	40
callHO_getClassname	gets the classname of a Java object	40
callHO_getList	gets/shows all stored Java objects	41
callHO_getSignature	gets the signature of a Java object	42
callHO_removeObject	removes a specified Java object	42
checkForMatches	assigns possible data type matches	43
checkNumber	checks if a number is in a boundary	44
createInterface	sets the interface of a domain	45
debug	sets/gets the debugging level	46
finalize	tries to finalizes objects in the JVM	46
garbageCollect	tries to garbage collect objects in the JVM	47
list	retrieves information about a class.	19
setJavaKey	changes a System property of the current JVM	47

Table 3.2: Short Reference for the Java package

Java Code Documentation (Classes) [See section [3.3](#) at page [48](#)]

Class	Description	Page
overview	Some words about the programmer Java classes	48
HandleObjects	The class that handles the unknown objects	50
DebugConsole	The console that shows the Java output	29
MuPADHelper	The class containing helper methods	29

Table 3.3: Short Reference for the Java Classes

3.1 Module Reference

3.1.1 Some words about the Java module

The module itself is system dependent due to the fact that it is written entirely in C/C++. Unlike the Java code it has to be compiled anew under every platform and the resulting module must be transferred to its proper location in the Java tree. Some predefined variables can be used throughout the use of the module. Most of them can be adjusted by the Java package, too.

<code>generalMatching</code>	defines the matching to be used if none is stated with the call of <code>invoke</code> . It can be either one of	
Name	value	Description
<code>MATCH_FIRST</code>	1	chooses the first possible match.
<code>MATCH_MAX</code>	2	chooses the match with maximum values.
<code>MATCH_BEST</code>	3	chooses the match with least deviation.
<code>MATCH_NATIVE</code>	4	chooses according to the signature of the method.
<code>killed</code>	defines whether the JVM was terminated unexpectedly. If killed is set to true, every time a method is to be invoked, a warning message comes up. A new kernel and a new JVM have to be started before any further invocations on the Java side (stored objects in the old JVM are unrecoverably gone).	
<code>_debug</code>	sets the debug level for the module.	
	value	Description
	2	print debug messages in <code>reflect</code> and <code>useJava</code> .
	1	print debug messages in <code>useJava</code> .
	0	print no debug messages at all.

Then there are also some constants defined used throughout all procedures to check for certain object types:

Name	value	Description
<code>JAVA_BOOLEAN</code>	1	the object is of type boolean.
<code>JAVA_CHAR</code>	2	the object is of type char.
<code>JAVA_BYTE</code>	3	the object is of type byte.
<code>JAVA_SHORT</code>	4	the object is of type short.
<code>JAVA_INT</code>	5	the object is of type integer.
<code>JAVA_LONG</code>	6	the object is of type long (Java).
<code>JAVA_FLOAT</code>	7	the object is of type float.
<code>JAVA_DOUBLE</code>	8	the object is of type double.
<code>JAVA_VOID</code>	9	the return value of the method is void.

Table 3.4: Java constants for the Java module (continued on next page page)

Name	value	Description
JAVA_BIGINTEGER	11	the object is of type BigInteger (Java).
JAVA_BIGDECIMAL	12	the object is of type BigDecimal (Java).
JAVA_STRING	13	the object is of type String (Java).
JAVA_UNKNOWN	21	the object is another unspecified Java object.
JAVA_OBJECT	22	object is either "@NULL or a formerly stored one (MuPAD_JNI_XXXXXXXXXX).
JAVA_ARRAY	100	the object is any kind of array.

Table 3.4: Java constants for the Java module

3.1.2 callHO_clearList

All objects stored inside the JVM for this session are deleted. The counter is not reseted by this action.

Call:

callHO_clearList()

Returns: MVnull.

3.1.3 callHO_getClassname

returns the Java classname of the object whose MuPAD signature is given as paramter.

Call:

callHO_getClassname(sign)

Parameter:

sign DOM_STRING the MuPAD-signature for the object. see [3.2.1](#) for details about signatures.

Returns: The classname of the respective Java object.

3.1.4 callHO_getList

Depending on the optional parameters, a list of strings representing the stored objects in Java is returned, shown in Java (and/or) shown in MuPAD.

Calls:

callHO_getList()

callHO_getList(showInJava)

Options:

showInJava DOM_BOOL If true, a Java PopUp will show the list containing the stored objects.

Returns: A list containing the string representation of all stored Java objects.

3.1.5 callHO_getSignature

returns the Java signature of the object whose MuPAD signature is given as paramter.

Call:

callHO_getSignature(sign)

Parameter:

sign DOM_STRING the MuPAD-signature of the stored object.

Returns: The signature of the respective Java object.

3.1.6 callHO_removeObject

removes the Java object that is stored with the given MuPAD-signature.

Call:

callHO_removeObject(sign)

Parameter:

sign DOM_STRING the MuPAD-signature of the stored object.

Returns: MVnull.

3.1.7 console

brings up a console that displays all the output, which was written to **System.out** and **System.err** during the session by any Java program called from within MuPAD. The window can be closed and reopened anytime without losing the content inside. Inside the text field the user is able to edit the text completely. Text can be added or deleted. The button in the south of the frame deletes the whole content of the text field, leaving it blank.

Call:

console()

Returns: MVnull.

3.1.8 debug

Sets the debug level within the module.

Calls:

debug()
debug(level)

Options:

level	⟨0⟩	show no debug messages at all.
	1	show only debug messages in useJava_xxx.
	2	show debug messages in useJava_xxx and reflect.

Returns: the current debug level.

3.1.9 finalize

Runs the method `runFinalization` in the current JVM.
(Invoked within the `java/lang/System` class).

Call:

`finalize()`

Returns: `MVnull()`.

3.1.10 garbageCollect

Runs the method `gc` in the current JVM (invoked within the `java/lang/System` class).

Call:

`garbageCollect()`

Returns: `MVnull()`.

3.1.11 getStatus

returns the current status of the Java Virtual Machine. The result can be either one of:

-1 The JVM was terminated abnormally.

0 The JVM is up and running.

1 The JVM is not initialized.

If -1 is returned, the user should either reconnect the notebook (under Windows) or exit MuPAD and restart it (under Unix/Linux). This state can only be reached if some severe

error had happened inside the JVM. In most cases the programmer of the Class did forget to catch exceptions, which lead to killing his application and then the whole JVM.

Call:

`getStatus()`

Returns: -1, 0, 1 depending on the state of the JVM.

3.1.12 `initJVM`

Takes the following steps in setting up a Java Virtual Machine:

1. Initialize the JVM with predefined arguments
2. wrap a security manager around (prevents from `System.exit()` calls
3. Load the exception class to catch any upcoming exception.

Call:

`initJVM(cp, lp, dllp, useSecurityManager, useFile, mupParam)`

Parameters:

<code>cp</code>	<code>char*</code>	the classpath that the JVM should use (same as <code>Java -cp</code>).
<code>lp</code>	<code>char*</code>	the librarypath that the JVM should use (same as <code>Java -DJava.library.path</code>).
<code>dllp</code>	<code>char*</code>	the path where the <code>jvm.dll</code> or <code>libjvm.so</code> is located.
<code>useSecurityManager</code>	<code>jboolean</code>	defines whether to use a predefined security manager (default) or not.
<code>useFile</code>	<code>jboolean</code>	defines whether to write <code>System.out</code> and <code>System.err</code> statements into the two files <code>MuPADout.log</code> and <code>MuPADerr.log</code> in the user directory of the JVM. The user directory can be retrieved with <code>getWorkingDirectory</code> [2.1.7] .
<code>mupParam</code>	<code>MFlist</code>	contains additional parameters for the JVM. Have a look at 2.1.16 for a list of options.

Example:

```
initJVM("C:\\;C:\\Java\\classes\\", "C:\\Java\\libraries\\",
        "C:\\Java\\include\\win32\\jvm.dll", JNI_FALSE,
        JNI_FALSE, mupParam)
```

initializes the JVM `C:\Java\include\win32\jvm.dll` and sets its classpath to `C:\;C:\Java\classes\` and its librarypath to `C:\Java\libraries\`. Furthermore no security manager is to be used (`System.exit` crashed will not be prevented) and the output will be redirected into the MuPAD-Console. *mupParam* is of no importance for this example.

3.1.13 invoke

The main procedure that handles all invocations.

Calls:

```
invoke(class, method, argList, showMethod, isConstructor,
        useObject, objectName, matchList)
invoke(class, method, argList, showMethod, isConstructor,
        useObject, objectName, matchList, matching)
invoke(class, method, argList, showMethod, isConstructor,
        useObject, objectName, matchList, matching, methSign)
```

Parameters:

class	DOM.STRING	The class that contains the method.
method	DOM.STRING	The method to be called.
argList	DOM.LIST	A list containing the parameters.
showMethod	DOM.BOOL	If TRUE the method that was chosen will be displayed, if FALSE, not.
isConstructor	DOM.BOOL	If TRUE the method is a constructor. If FALSE its a method.
useObject	DOM.STRING	if a predefined object is to be used instead of a new "clean" object.
objectName	DOM.STRING	The internal name for the stored object (for example <code>MuPAD_JNI.000000000</code>).
matchList	DOM.LIST	A list containing the possible data types for the matching.

Options:

matching	DOM.BOOL	The matching that is to be chosen.
methSign	DOM.STRING	The signature of the method. This parameter is only necessary if matching was set to 4 (native match)

Example:

```
Invoke(Prog, numberTest, [10.2,FALSE], FALSE, FALSE,
        FALSE, "", [ [7,8,12],[1] ])
```

uses the class `Prog` to call its method `numberTest`. If the method is non static, a default constructor is invoked before the method is executed. The parameters are the floating point number 10.2 (which could be either one of the data types *float*, *double* or *BigDecimal*) and

FALSE (which can only be a *boolean*). The chosen method will not be printed and since the name of the class and the method differ, it can not be a constructor. The method is not executed on a formerly created object and so the name is of no importance. The lists define -for each parameter a list is at the respective position- numerical representatives for the just mentioned data types, so the function can sort out the false methods easy and according to the defined match take the appropriate correct one.

3.1.14 list

Retrieves all constructors, methods and fields of a class and its superclasses. The methods and fields of a class include all private and protected ones, whereas the superclasses lists only contain the public ones.

Call:

`list(className)`

Parameters:

`className` DOM_STRING The name of the class that is to be examined.

Returns: A list containing five other lists. The organization is as follows:

```
[ [constructors], [Class methods], [Class fields],
  [Superclass methods],[Superclass fields] ]
```

3.1.15 Java_calcMuPAD

A method with callback functionality that is added to every class which is invoked manually by the user and contains the necessary line.

Call:

`Java_calcMuPAD(env, thisClass, query)`

Parameters:

<code>env</code>	JNIEnv*	The ubiquitous pointer to the JNI.
<code>thisClass</code>	jclass	The class from where the callback request came.
<code>query</code>	jstring	The callback request.

With the aid of this method a callback can be made at runtime and an arbitrary Java method can communicate with the MuPAD-core through it. To do so, a single line in the source code of the class that wants to make callback(s) is sufficient:

Listing 3.1: The Line to be Added for Callback Support

```
public native static String calcMuPAD(String query);
```

After this line has been added to the code the underlying native method can be used like any other Java method.

Example (in Java):

Listing 3.2: A Simple Callback Example

```
public class TestCallBack
{
    /* to include the callback functionality */
    public native static String calcMuPAD(String query);

    public static String callBack()
    {
        return calcMuPAD("solve(x^4 - 5*x^2 + 6*x = 2, x)");
    }

    public static void main(String[] args)
    {
        String result = callBack();
        System.out.println("MuPAD calculated for "
            + "solve(x^4 - 5*x^2 + 6*x = 2, x): " + result);
    }
}
```

Now just start this program out of MuPAD (assuming its located in the classpath) via `java::testCallBack::main([])`.

3.1.16 setShowErrors

Defines if severe errors during a function call in the module are printed or not. It was implemented because of the tab completion of classes. Under normal circumstances errors would be printed if for example a class can not be found.

Call:

`setShowErrors(state)`

Parameters:

state	DOM_BOOL	Defines if errors are to be printed (TRUE) or not (FALSE).
-------	----------	--

Returns: MVnull

3.2 Package Reference

3.2.1 Some words about the Java programmer package

Object representatives are stored in Strings of the form `MuPAD_JNI_xxxxxxx`. An object representative always begins with `MuPAD_JNI_` followed by an eight digit number. The counter starts with 0 and increases with every object returned by the JVM. Since the number is always eight digits long, preceding spaces will be filled with 0's. The fourth stored object would as a result be assigned to the representative `MuPAD_JNI_00000003`. Throughout the chapter the term *signature* rather than *object representative* will be used.

3.2.2 `callHO_clearList`

All objects stored inside the JVM for this session are deleted. The counter is not reseted by this action. *This procedure is not accessible via tab completion and not intended for official use by any user of MuPAD. It is intended for internal use only.*

Call:

`callHO_clearList()`

Returns: null.

Example:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> java::callHO_clearList()
└────────────────────────────────────────────────────────────────────────────────────────────────┘

```

3.2.3 `callHO_getClassname`

Returns the Java classname of the object whose MuPAD signature is given as paramter. *This procedure is not accessible via tab completion and not intended for official use by any user of MuPAD. It is intended for internal use only.*

Call:

`callHO_getClassname(sign)`

Parameter:

sign — the MuPAD-signature for the object. see [3.2.1](#) for details about signatures.

Returns: The classname of the respective Java object.

Example:

```

MuPAD
--> frame1 := java::java::awt::Frame::Frame("A Frame"):
    java::callHO_getClassname(frame1)
-- Output --
java/awt/Frame

```

3.2.4 callHO_getList

Depending on the optional parameters, a list of strings representing the stored objects in Java is returned, shown in Java (and/or) shown in MuPAD. *This procedure is not accessible via tab completion and not intended for official use by any user of MuPAD. It is intended for internal use only.*

Calls:

```
callHO_getList()
callHO_getList(ShowInMuPAD)
callHO_getList(ShowInJava)
callHO_getList(NoList)
```

Options:

ShowInJava	—	displays a Java popup containing the information about the objects.
ShowInMuPAD	—	displays the information as formatted output in MuPAD.
NoList	—	suppresses the return of the list (null will be returned instead).

Returns: A list containing the string representation of all stored Java objects if not redefined by NoList.

Example:

```

MuPAD
>> java::callHO_getList()

Output

Entry [0] (@MuPAD_JNI_00000000) = muplets
Entry [1] (@MuPAD_JNI_00000001) = muplets
Entry [2] (@MuPAD_JNI_00000002) = java.awt.Frame
Entry [3] (@MuPAD_JNI_00000003) = java.awt.Button

```

3.2.5 callHO_getSignature

Returns the Java signature of the object whose MuPAD signature is given as parameter. *This procedure is not accessible via tab completion and not intended for official use by any user of MuPAD. It is intended for internal use only.*

Call:

callHO_getSignature(sign)

Parameter:

sign — the MuPAD-signature of the stored object.

Returns: The signature of the respective Java object.

Example:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> frame1 := java::java::awt::Frame::Frame("A Frame"):
    java::callHO_getSignature(frame1)
    ─────────── Output ───────────────────────────────────────────────────────────────────────────┘

    Ljava/awt/Frame;
└────────────────────────────────────────────────────────────────────────────────────────────────┘

```

3.2.6 callHO_removeObject

Removes the Java object that is stored with the given MuPAD-signature. *This procedure is not accessible via tab completion and not intended for official use by any user of MuPAD. It is intended for internal use only.*

Call:

callHO_removeObject(sign)

Parameter:

sign — the MuPAD-signature of the stored object.

Returns: null.

Example:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> frame1 := java::java::awt::Frame::Frame("A Frame"):
    java::callHO_removeObject(frame1)
└────────────────────────────────────────────────────────────────────────────────────────────────┘

```

 Output

[[11]]

Example2:

 MuPAD

>> java::checkForMatches([[TRUE, FALSE], [FALSE]], 127, "MuPAD is great")

 Output

[[201], [4,3,5,6,11],[13]]

3.2.8 checkNumber

Checks if the stated number is within predefined boundaries. This procedure is invoked via callback from a Java module function. *This procedure is not accessible via tab completion and not intended for official use by any user of MuPAD. It is intended for internal use only.*

Call:

checkNumber(number, mode)

Parameters:

- number — The number to be checked.
 mode — The identifier for the range between where the number has to be.

Mode	Type
0	Byte
1	Short
2	Integer
3	Long
4	Float
5	Double

Returns:

- TRUE — The number is within the range.
 FALSE — The number is not within the defined range.

The ranges were predefined with the following values:

Mode	Type	lowest value	highest value
0	Byte	-128	127
1	Short	-32768	32767
2	Integer	-2147483648	2147483647
3	Long	-9223372036854775808	9223372036854775807
4	Float	-1.4e45	3.4028235e38
5	Double	-4.9e324	1.7976931348623157e308

Example:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> java::checkNumber(126, 0)
      ──── Output ───────────────────────────────────────────────────────────────────────────┘
      TRUE
└────────────────────────────────────────────────────────────────────────────────────────────────┘

```

3.2.9 createInterface

This procedure is needed in the creation phase of a domain and sets the **interface** of the newly created domain. So far, the *public* methods of a class are enlisted, if the domain proved to be a complete classname. Internally it uses the **list** function of the module (Cf. 3.1.14 on page 38). *This procedure is not accessible via tab completion and not intended for official use by any user of MuPAD. It is intended for internal use only.*

Call:

createInterface(dom)

Parameters:

dom — a DOM_DOMAIN.

Returns: either `nil` or a set of method names available in the stated class.

Example:

```

┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> java::createInterface(java::SimpleCalculator)
      ──── Output ───────────────────────────────────────────────────────────────────────────┘
      {actionPerformed, calcMuPAD, center, main, SimpleCalculator}
└────────────────────────────────────────────────────────────────────────────────────────────────┘

```

3.2.10 debug

Sets the debug level within the module. *This procedure is not accessible via tab completion and not intended for official use by any user of MuPAD. It is intended for internal use only.*

Calls:

```
debug()
```

```
debug([<0>, 1, 2])
```

Options:

-
- | | | |
|---|---|---|
| 0 | — | show no debug messages at all. |
| 1 | — | show only debug messages in useJava.xxx. |
| 2 | — | show debug messages in useJava.xxx and reflect. |

Returns: the current debug level.

Example:

```
MuPAD
--> java::debug(2)
--> Output
2
```

3.2.11 finalize

Runs the method `runFinalization` in the current JVM. Calling this method suggests that the Java Virtual Machine expend effort toward running the finalize methods of objects that have been found to be discarded but whose finalize methods have not yet been run. When control returns from the method call, the Java Virtual Machine has made a best effort to complete all outstanding finalizations. *This procedure is not accessible via tab completion and not intended for official use by any user of MuPAD. It is intended for internal use only.*

Call:

```
finalize()
```

Returns: null.

Example:

```
>> java::finalize()
```

3.2.12 garbageCollect

Runs the method `gc` in the current JVM. Calling the `gc` method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects. *This procedure is not accessible via tab completion and not intended for official use by any user of MuPAD. It is intended for internal use only.*

Call:

`garbageCollect()`

Returns: `null`.

Example:

```
┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> java::garbageCollect()
```

3.2.13 setJavaKey

changes properties of the currently running JVM. Not all properties can be changed at runtime. The change of the classpath has no effect on the current classpath. The numbers that can be used as keys are listed together with their associated values in Table 2.4 on page 25. *This procedure is not accessible via tab completion and not intended for official use by any user of MuPAD. It is intended for internal use only.*

Call:

`setJavaKey(key, value)`

Parameters:

- `key` — the number defining the associated value.
- `value` — a String holding the new value for the System property.

Returns: `null`.

Example 1:

```
┌── MuPAD ───────────────────────────────────────────────────────────────────────────────────┐
>> java::setJavaKey(3, "http://www.mupad.de")
```

3.3 Java Reference

3.3.1 Some words about the Java programmer classes

Important for Java programmers: If graphics or icons are to be used in Java programs be aware that the location has to be explicitly named for the loader to find them.

Example: `ImageIcon icon1 = new ImageIcon("SciFace.gif");`

might find the graphic, if called in the correct directory, whereas

```
ImageIcon icon1 = new ImageIcon(System.getProperty("user.dir")
    + System.getProperty("file.separator") + "SciFace.gif");
```

is guaranteed to find the image if it is placed in the directory where the Java program is located.

3.3.2 AppletStarter

An applet has to be handled different compared to a normal Java application. The startup phase is divided into two methods (**init** and **start**) as is the shutdown phase (**stop** and **destroy**). If further advancements are made they will be surely made either in one of the `get...` methods or in the subclass `DummyURLConnection`.

Important constructors:

- `AppletStarter(Applet applet, String args[])`
Creates a basic framework to work with the stated applet. The arguments are passed in `args[]`.
- `AppletStarter(Applet applet, String args[], int default_width, int default_height)`
Creates a basic framework to work with the stated applet. The arguments are passed in `args[]`. Additionally The width and height of the applet can be adjusted.

The following methods are needed internally:

- `void init(Applet applet, String args[], int default_width, int default_height, int startidx)`
Takes care of the initialization of the actual applet.
- `void parseArgs(String args[], int startidx)`
Parses the arguments and preprocesses them for the applet.
- `final URL getDocumentBase()`
Returns a predefined document base (right now the user directory).
- `final URL getCodeBase()`
Returns a predefined code base (right now the user directory).

- `final AppletContext getAppletContext()`
Returns the applet context.
- `void appletResize(int width, int height)`
Resizes the applet.
- `final AudioClip getAudioClip(URL url)`
So far it returns null.
- `final Image getImage(URL url)`
Tries to load the image defined in the URL.
- `String filenameFromURL(URL url)`
Tries to create a filename out of the URL.
- `final Applet getApplet(String name)`
So far only returns null.
- `final Enumeration getApplets()`
Returns the applet.
- `void showDocument(URL url)`
A dummy function that displays some text in the status line of the frame.
- `void showDocument(URL url, String target)`
The same as above, but this time with the text defined in `target` added.
- `void showStatus(String text)`
Displays `text` in the status line.
- `URLConnectionHandler createURLConnectionHandler(String protocol)`
Creates a new stream handler (so far it is ignored by this implementation.)
- `void setStream(String key, InputStream stream)`
Does nothing.
- `InputStream getStream(String key)`
Returns only null and does nothing besides that.
- `Iterator getStreamKeys()`
Returns only null and does nothing besides that.

The subclass `DummyURLConnectionHandler` contains the following methods:

- `DummyURLConnection(URL url)`
Sets the internal URL so that the following methods can access a valid file.

- `connect()`
Tries to connect to a local predefined file.
- `InputStream getInputStream()`
Opens -if necessary- a stream to the specified local file.

3.3.3 HandleObjects

This class takes care of handling the Java objects that are created and handled during a MuPAD-session. This class is one of the basic classes that *must* be available or the correct behavior of the Java package can not be guaranteed.

- `void clear()`
clears the complete hashtable, so that it contains no more elements.
- `String getClassname(String key)`
Returns the Java classname of object represented by the MuPAD-signature *name*.
- `String[] getList(boolean showInJava)`
Returns a list with descriptions of the stored objects. If `showInJava` is *true* then also a Swing-PopUp-Window will be opened displaying the list.
- `Object getObject(String key)`
Returns the object represented by the MuPAD-signature *name*.
- `String getSignature(String key)`
Returns the Java signature of object represented by the MuPAD-signature *name*.
- `void put(String key, Object object)`
Adds the given object with the given key to the hashtable for further usage.
- `void remove(String key)`
removes the entry with the handed MuPAD-signature from the hashtable.
- `int size()`
returns the number of elements stored in the hashtable.

3.3.4 MuPADSecurityManager

This class was created to use a security manager that prevents other classes from crashing the JVM. Calls to `System.exit` for example are prevented. This class is one of the basic classes that *must* be available or the correct behavior of the Java package can not be guaranteed.

- `static void setSecurityManager()`
Creates a new `SecurityManager` object that decides if actions are allowed to take place or not.

The following methods were overwritten so that no security exception will be thrown if any of these actions take place. The more methods are removed, the less the user is allowed to do:

- `void checkAccept(String host, int port)`
- `void checkAccess(Thread g)`
- `void checkAccess(ThreadGroup g)`
- `void checkAwtEventQueueAccess()`
- `void checkConnect(String host, int port)`
- `void checkConnect(String host, int port, Object context)`
- `void checkCreateClassLoader()`
- `void checkDelete(String file)`
- `void checkExit(int status)`
- `void checkExec(String cmd)`
- `void checkLink(String lib)`
- `void checkListen(int port)`
- `void checkMemberAccess(Class clazz, int which)`
- `void checkMulticast(InetAddress maddr)`
- `void checkPackageAccess(String pkg)`
- `void checkPackageDefinition(String pkg)`
- `void checkPermission(Permission perm)`
- `void checkPermission(Permission perm, Object context)`
- `void checkPrintJobAccess()`
- `void checkPropertiesAccess()`
- `void checkPropertyAccess(String key)`
- `void checkPropertyAccess(String key, String def)`
- `void checkRead(FileDescriptor fd)`

- `void checkRead(String file)`
- `void checkRead(String file, Object context)`
- `void checkSecurityAccess(String provider)`
- `void checkSetFactory()`
- `void checkSystemClipboardAccess()`
- `boolean checkTopLevelWindow(Object window) return true;`
- `void checkWrite(FileDescriptor fd)`
- `void checkWrite(String file)`

3.3.5 MuPADHelper

In this class some methods are defined, which are used internally to communicate with the JVM. If additional methods are to be written, they can be inserted into this class.

- `static String getSystemInformation(int number)`
Retrieves the System property information of the number stated. The numbers are explained in Table 2.4 on page 25.
- `static String setSystemInformation(int number, String value)`
Sets System properties according to the number stated. Afterwards `value` is the new content.

Index

@MuPAD_JNI_XXXXXXXXXX, [5](#), [41](#)

@NULL, [33](#)

access specifier, [7](#)

Graph, [7](#)

jar, [5](#)

Java classes

Programmer classes

AppletStarter, [48](#)

HandleObjects, [50](#)

MuPADHelper, [52](#)

MuPADSecurityManager, [50](#)

User classes

DebugConsole, [29](#)

Lister, [29](#)

Module functions

callHO_clearList, [33](#)

callHO_getClassname, [33](#)

callHO_getList, [33](#)

callHO_getSignature, [34](#)

callHO_removeObject, [34](#)

console, [34](#)

debug, [35](#)

finalize, [35](#)

garbageCollect, [35](#)

getStatus, [35](#)

initJVM, [36](#)

invoke, [37](#)

Java_calcMuPAD, [38](#)

list, [38](#)

setShowErrors, [39](#)

object representative, [40](#)

Package procedures

Programmer procedures

callHO_clearList, [40](#)

callHO_getClassname, [40](#)

callHO_getList, [41](#)

callHO_getSignature, [42](#)

callHO_removeObject, [42](#)

checkForMatches, [43](#)

checkNumber, [44](#)

createInterface, [45](#)

debug, [46](#)

finalize, [46](#)

garbageCollect, [47](#)

setJavaKey, [47](#)

User procedures

closeAppletFrame, [11](#)

console, [12](#)

getMatrix, [13](#)

getPath, [14](#)

getStatus, [15](#)

getWorkingDirectory, [15](#)

invoke, [16](#)

list, [19](#)

match, [20](#)

matrixEditor, [21](#)

runApplet, [22](#)

setWorkingDirectory, [23](#)

showJavaInfo, [24](#)

showMatrix, [25](#)

start, [26](#)

signature, [40](#)