

**An  
Informal Tutorial  
on  
Some Practical Aspects  
of the  
Yoix<sup>®</sup> Language  
and  
Its Interpreter**



Introduction.....	1
Getting Started.....	1
Security.....	2
Additional Remarks.....	3
Interactive Mode.....	3
Additional Remarks.....	6
Import.....	6
The Command Line.....	7
Additional Remarks.....	8
Global.....	8
Additional Remarks.....	9
VM Revisited.....	9
Additional Remarks.....	10
Comments.....	11
Names.....	11
True or False.....	12
Additional Remarks.....	12
Types.....	12
Numbers.....	13
Additional Remarks.....	14
Strings.....	14
Additional Remarks.....	16
Arrays.....	16
Additional Remarks.....	17
Dictionaries.....	18
Functions.....	20
Additional Remarks.....	23
This.....	24
Attributes.....	25
Growable Objects.....	27
Pointers.....	27
Indirection Operator.....	30
Address Operator.....	31
The new Operator.....	32
Equality Operators.....	35
Additional Remarks.....	37
Instanceof Operator.....	37
Additional Remarks.....	38
Declarations Revisited.....	38
The if Statement.....	40
The switch Statement.....	41
Additional Remarks.....	42
The for Loop.....	42
The defined() Built-In.....	44
The unroll() Built-In.....	45
Concluding Remarks.....	47



## Introduction

Although Yoix technology has been around for many years<sup>1</sup>, we have not yet gotten around to producing a good tutorial. It is on our long list of *things-to-do*, but in the mean time we offer this temporary, incomplete attempt as a placeholder for the promised tutorial. Our goal for this instance of a tutorial is rather modest: provide the background you will need if you decide to poke around in any or our example code. In particular, this tutorial was put together to provide background for anyone digging around in our *YChart*-based implementation of the periodic table (see `elements.yx`).

Incidentally, Yoix is a registered trademark of AT&T Intellectual Property and Java is a trademark of Sun Microsystems.

## Getting Started

Yoix software is distributed through our website at

<http://www.yoix.org/>

or, synonymously,

<http://www.research.att.com/sw/tools/yoix/>

where all the source code is available if you want to build the package on your own, but there is also a binary distribution that installs everything you need, including scripts that start the interpreter and run demos. We are going to assume you have our interpreter start-up scripts, that they seem to be working for you, and that they are in your **PATH**.

Two options, namely **-?** and **--info**, summarize the command line and describe the available interpreter options, so you can type

```
yoix -?
```

to get a short summary or

```
yoix --info
```

for a longer explanation. In both cases it goes to standard output so you can use a pager like *more* or *less* to scroll through the text. The command line syntax summary that you get using either option looks like the following:

```
yoix [options] [script [args...]]
```

---

<sup>1</sup> The first public version was released in late 2000.

where `[options]` control the Yoix interpreter and `[args...]` are handled by the script. As you might expect, brackets surround the optional parts of the command line. We will come back to the command line after we take a short detour to talk about security and introduce several techniques that will let you follow our discussion of the command line.

## Security

The Yoix interpreter has supported command line security options for quite a while now, and we will talk about them shortly, but new in release *2.2.0* is code that automatically runs a command line script that ends in the `.yxs` suffix under a security manager that tries to enforce the Java security policy that is currently installed on your system. We tend to refer to this as *applet* mode even though it has nothing to do with HTML or your browser, but instead refers to a Yoix script that runs with restricted capabilities that pretty much duplicate the restrictions imposed on a Java applet by a browser. For example, a Yoix script downloaded from a URL and run as in this applet security mode usually will not be able to access the local file system or establish network connections to different hosts than the one from which it came. Java security policies can be customized by system administrators and users can even make their own private adjustments, so what a Java or Yoix program running under the applet security policy can or cannot do is ultimately system and user dependent.

By default the Yoix interpreter does not impose security restrictions on scripts with names that do not end in `.yxs`, however the `--applet` command line option asks the interpreter to run the script as an applet. In other words, if the script is named `elements.yx`, then the following:

```
yoix --applet elements.yx
```

runs `elements.yx` in applet mode, while

```
yoix elements.yx
```

runs `elements.yx` as a so-called *completely trusted application*, which means it can, among other things, read and write local files, execute programs on your computer, and establish connections to sites on the internet. So, if you wish, move or copy the script to `elements.yxs` so that:

```
yoix elements.yxs
```

and

```
yoix --applet elements.yxs
```

both run as Yoix applets, i.e., in applet-level security mode.

In the case of `elements.yx`, all the data needed to build and display the tables is included in the script file, so most things work in applet mode. The main exceptions are

external web sites that you can connect to using the *Open* button that is located in the upper right corner of the screen. By default the interpreter will not let you open them unless you explicitly allow it before requesting applet mode, which means using specifically crafted **-S** options, which must precede the **--applet** option if you want to access external web sites. For example,

```
yoix -Sallow:connect:www.webelements.com:80 --applet elements.yx
```

runs the script as an applet, but still lets you read web pages on the *webelements* site, while

```
yoix -Sallow:connect:en.wikipedia.org:80 --applet elements.yx
```

lets you connect to the *wikipedia* site<sup>2</sup>. If you precede **--applet** by both of the **-S** options shown above, you will be able to read web pages on both the *wikipedia* and *webelements* sites.

The **-S** option also lets you ask to be prompted for an answer when there is a security check, so the following:

```
yoix -Sprompt:connect --applet elements.yx
```

will show a dialog with information about the connection whenever there is a connect security check. Prompting is interesting and works nicely with connect, but it is not appropriate for every security policy *category* because it occasionally causes deadlock that we suspect may be unavoidable.

It is always important to think about security, which is why we wanted to talk about it first, but now we are ready to move on to other topics that should help you learn more about Yoix technology.

### Additional Remarks

As you might expect, you can find more details about Yoix security using the **--info** option. The *SecurityManager* reference page in the Yoix documentation is the place to go for a brief description of the *SecurityManager* functions, and if that is not enough you can always grab the Yoix source code from our website and look through both `YoixSecurityOptions.java` and `YoixSecurityManager.java`, which are two important security related source files.

## Interactive Mode

Running the Yoix interpreter is easy, so we are going to show you how and describe ways you can get useful information out of the interpreter without knowing much about the language. Typing just the following:

---

<sup>2</sup> Look at the URLs mentioned in the `GetExternalSiteURL()` function, which is defined in `elements.yx`.

```
yoix
```

starts the Yoix interpreter and tells it to read lines that you type using your keyboard. This *modus operandi* is interactive mode and the interpreter will continue reading and executing the statements that you type until it is interrupted (e.g., via **CTRL-C**) or encounters end of file (e.g., via **CTRL-D** on Unix or **CTRL-Z** on Windows). Unless you already know what you are doing, interactive mode might be confusing because there is no prompt. Moreover, when invoked with no command line options, as in this example, the feedback you are most likely to get are error messages. However, adding the **-d1** option, namely the letter *d* followed by the digit *1*, to the command line as follows:

```
yoix -d1
```

provides more feedback and also turns out to be a useful way to explore. The option asks the interpreter to dump to the standard output stream the results from every statement that happens to be an expression. Thus, if you type the following, being sure to include the final semi-colon:

```
2 + 3;
```

the interpreter answers:

```
5
```

Similarly, if you type:

```
VM;
```

the contents of the dictionary named `VM` will be printed on your screen. The output is version dependent and has lots of uninteresting fields, at least as far as this tutorial is concerned, so we are only going to show you a small part of the dump:

```
Dictionary[28:0]
  Notice="Copyright 2000-2008 AT&T. All rights reserved."
  addtags=1
  applet=0
  create=1
  debug=1
  dumpdepth=1
  exitmodel=1
  screen=Dictionary[15:0]
  securityoptions=NULL:STRING
  >tmpdir=" /tmp"
```

Notice that there is a variable named `debug` in `VM` that is set to 1, which is where the 1 from our **-d1** option ended up. We will talk more about `VM` in another section, but for now, if the interpreter is still running, type the lines:

```
Color c;
c;
```

and you should then see the following:

```
Color[3:0]
  blue=0.0
  green=0.0
  >red=0.0
```

printed on the standard output stream. *Color* objects are short and simple, which is why we showed you the full dump, but the same technique works with any object. For example,

```
JButton b;
b;
```

creates a *JButton* with default values assigned to its fields and prints them on standard output. We are not going to explain the dump format here except to say that the fields in objects that associate names with values, e.g., a *Color* or *JButton*, are usually sorted in the dump, so the ordering you see in the *VM*, the *Color*, and the *JButton* dumps does not tell you how elements in those objects are arranged in memory.

As we showed, debug flags can be set on the command line using the **-d** option, but you can also change them while the interpreter is running. So, for example, after you type:

```
VM.debug = 0;
```

expressions will no longer be dumped. Now when you want to look inside an object you have to work harder and use the "%O" format specification with the `printf()` or `fprintf()` built-ins defined in *yoix.stdio*. Try typing:

```
yoix.stdio.printf("%O\n", VM);
```

and the contents of the *VM* dictionary will be printed on the standard output stream exactly the way they were earlier when we were using the **-d1** option. Alternatively, typing:

```
yoix.stdio.fprintf(stderr, "%O\n", VM);
```

will send the same output to the standard error stream. By default the %O output specification dumps only one level of an object, but you can change the *precision* of the specification as illustrated by the following:

```
yoix.stdio.printf("%.20\n", VM);
```

which dumps two levels of nested object structure, namely `VM` and the dictionaries defined in `VM`. We often use `printf()` or `fprintf()` when we are debugging a script or just trying to understand what is going on, and it is something you will also want to use if you start poking around in the example scripts.

### Additional Remarks

You can use `--info` to get a little more information about the debug flags, but if you want all the details, we once again recommend that you grab the source from our website , if you have not already done so, and this time look for the definition of **DEBUG\_EXPRESSION** in *YoixConstants.java*. That will position you very near to the definitions of the other debug flags. After that just *grep* for any of those constants in the rest of the source files and you will find the code that it controls.

Take a look at our reference pages for more about objects like *JButton* or *Color*.

## Import

Yoix scripts often start with the line:

```
import yoix.*.*;
```

because it saves some typing and usually means you do not have to remember where built-ins are defined, e.g., `fprintf()` is defined in *yoix.stdio*, but it does not have to be the first line in a script. If you are still running the interpreter as you read this sentence, type the `import` line and after that type:

```
printf("%O\n", VM);
```

which will work the same as before, when it was fully specified, but now we were able to omit the *yoix.stdio*. If you are curious about exactly what was dropped, type:

```
printf("%O\n", yoix.stdio);
```

and you will see it is a dictionary that includes `printf()`, `fprintf()`, and a bunch of other built-ins with names that C programmers most likely will recognize. At this point you probably can guess that *yoix* is also a dictionary, so when you type:

```
printf("%O\n", yoix);
```

you will see `stdio` and several dozen other dictionaries. These are the dictionaries that contain the built-ins, constants, and other objects that you will need in your scripts. Poke around some more if you want, but do not forget that both the `-d1` option and its equivalent:

```
VM.debug = 1;
```

are particularly easy ways to get dumps in interactive mode.

## The Command Line

Recall the command line syntax that we mentioned earlier, namely:

```
yoix [options] [script [args...]]
```

Now that you know how to get useful information from the interpreter we will show you where Yoix scripts can find the `[script [args...]]` portion of the command line. Start the interpreter, as before, using:

```
yoix -d1
```

then type:

```
argv;
```

and

```
Array[1:0]
>^-stdin-
```

will print on the standard output stream. It is not a particularly interesting dump because there was not anything to the command line, but you probably can guess that the first element in `argv` will always be the name of the script that the interpreter is executing or the string `-stdin-` when it is reading from the standard input stream.

To get a more interesting dump use the following command line:

```
yoix -d1 -- - -a12 -Dqwert dummy
```

to start the interpreter, then type:

```
argv;
```

and

```
Array[4:0]
>^-stdin-
^-a12
^-Dqwert
^-dummy"
```

prints on your screen. Working backwards through the `argv` array and the command line leads to the conclusion that the dash or hyphen (`-`) in our command line must stand for the standard input stream and that `-d1` and the double-dash (`--`) are part of the `[options]` that are handled by the Yoix interpreter. In fact, the double-dash is not really an option but instead is used to mark the end of the `[options]` because without

it the dash that is supposed to stand for the standard input stream would just look like another option<sup>3</sup>. The way we use double-dash and dash follows well established conventions, but do not worry if you are confused because you will rarely need either one. We wanted to let you use interactive mode to look at a non-trivial `argv` array and we needed double-dash and dash to help, but that meant we also had to take some time to explain them.

After the interpreter builds `argv`, it initializes a variable named `argc` with the number of elements in that array, so if you type:

```
argc;
```

you should see the number four (4) print on your screen. As we will see in the section on attributes, the Yoix language lets you find out how big any array is, so `argc` is not the only way to count arguments.

### Additional Remarks

A function named `Options()` defined in the `elements.yx` script, for example, processes the `[args...]` portion of the command line, so it is a good place to look if you just want an example. The *Option* reference page in the Yoix documentation is the place to go for a complete discussion of command line options.

## Global

Yoix scripts always have access to a growable dictionary named *global*. It is where `VM`, `argv`, and `argc` live and where variables are created when the script starts. *global* is a keyword, i.e., it is part of the language, so it cannot be hidden or redefined. You can dump the contents of *global* using any of the techniques outlined earlier. For example, if you start the interpreter in debug mode and type:

```
global;
```

then something like the following:

```
global=Dictionary[7:0]
  VM=Dictionary[28:0]
>argc=1
  argv=Array[1:0]
  envp=NULL:ARRAY
  errordict=Dictionary[7:0]
  importdict=Array[0:0]
  typedict=Dictionary[168:0]
```

---

<sup>3</sup> Actually, it would trigger an error.

should print on the standard output stream. Next define a variable and dump *global* again as we demonstrate here:

```
String xyz = "this is a test";
global;
```

and

```
Dictionary[8:0]
  VM=Dictionary[28:0]
  >argc=1
  argv=Array[1:0]
  envp=NULL:ARRAY
  errordict=Dictionary[7:0]
  importdict=Array[0:0]
  typedict=Dictionary[168:0]
  xyz="this is a test"
```

prints on your screen and if you look carefully you will see the definition of *xyz*. The Yoix language would not be much of a programming language if all definitions ended up in *global*, but it is usually the last stop when the interpreter is looking for a variable.

### Additional Remarks

In most cases applications only see one *global* scope, but a built-in named `execute()` can be used to run other Yoix scripts and an important part of the built-in's utility is to make sure the scripts execute using their own *global* dictionary. We do not use `execute()` in the `elements.yx` script, but we thought it deserved a brief mention. You will find more information in our reference pages and it is used extensively in our YWAIT package.

Functions, which we will discuss in a little while, define their own `argv` and `argc` variables, so inside a function you have to use `global.argv` and `global.argc` to access the ones we talked about here.

## VM Revisited

VM is a dictionary that contains information about the interpreter, your display, namely `VM.screen`, Java's current look and feel, namely `VM.screen.uimanager`, and a collection of variables that can be used to control the behavior of the interpreter. We have already discussed `VM.debug`, but you should know about several other variables. The first is:

```
VM.create
```

which is **TRUE**, i.e., non-zero, by default and means the interpreter will not complain when you assign a value to a variable that has not been declared, but instead

automatically creates the variable in *global* when it does not find an existing definition. It is particularly convenient in interactive mode and in small scripts because you can omit many declarations, but big scripts like `elements.yx` usually include the following line:

```
VM.create = FALSE;
```

which means variables must be declared before a value can be assigned to them.

The another VM variable of interest is:

```
VM.addtags
```

which is **TRUE**, i.e., non-zero, by default and means the interpreter will keep track of line number and source file information and include that information in all error messages. That is, of course, important information during development or when you are debugging problems, but it is relatively expensive to maintain and often slows things down by 5 to 10 percent, so most programs include the line:

```
VM.addtags = FALSE;
```

once they are ready for production.

GUI applications usually need information about your display and it can be found in the VM dictionary:

```
VM.screen
```

The values stored in:

```
VM.screen.width
```

and

```
VM.screen.height
```

represent the total size of your display in units of 72 per inch, while:

```
VM.screen.bounds
```

is a rectangle that tries to account for window manager decorations and it usually has a height that is slightly smaller than `VM.screen.height`. `VM.screen.bounds` is accurate on most platforms, but unfortunately the rectangle that we get back from Gnome, on Linux, always seems to match the total width and height.

### Additional Remarks

The numbers needed to convert the pixels that Java uses to the 72 units per inch used by Yoix scripts are automatically determined when the interpreter starts and in many cases the default values are perfectly acceptable. However, sometimes Java's notion of

screen resolution needs adjusting and that is where the **-D** command line option can be useful. It lets users specify the length, in inches, of their screen's diagonal and that number, in turn, means the interpreter can do a better job matching 72 units to one inch on your screen. The length of your screen's diagonal is always stored in:

```
VM.screen.diagonal
```

so when you use the **-D** option you can expect to see the number that you specified on the command line stored there.

`VM.create` only affects the behavior of the interpreter when it executes a statement. In other words, it does not trigger parse tree checking that will catch errors before a statement is executed.

The **-g** command line option overrides `VM.addtags` and guarantees that error messages include line number and source file information, no matter what value is assigned to `VM.addtags`.

## Comments

As a quick look at an example script will tell you, Yoix supports C++ and Java style single line comments introduced by a double-slash (`//`) and as you probably suspect multiline C style comments delimited by slash-star (`/*`) and star-slash (`*/`) also work. We prefer the double-slash for permanent comments and usually reserve the multiline style for when we want to temporarily remove, i.e., comment out, large blocks of code. That is our reasoning, but you can comment any way you want, of course.

## Names

Names in a Yoix script must begin with a letter, an underscore (`_`), or a dollar sign (`$`), and can be followed by any number of letters, digits, underscores, or dollar signs. We often adopt simple conventions in our scripts:

- Functions that we define begin with an uppercase letter, which makes them easy to spot because Yoix built-ins always begin with a lowercase letter.
- Special strings, known as tags, that can be assigned to components, like a *JButton*, always begin with a dollar-underscore (`$_`) prefix and only contain characters that are allowed in Yoix variable names.
- Names assigned to the variables that are local to a function, including the arguments, usually consist of lowercase letters, digits, and underscores.
- Names consisting of uppercase letters, digits, and underscores are usually reserved for constants or variables that only change during some very early initialization, e.g., when command line options are processed.

Remember, these are just guidelines that we usually try to follow them as good programming practice, but they are not enforced in any way. Still, knowing our conventions should help a little if you decide to poke around in our code.

## True or False

Being able to recognize whether an expression is true or false lets `if` statements pick the correct branch and looping statements decide when they are done. The Yoix model pretty much follows the C model and does it without a boolean type. The rules are:

- A numeric expression that has a non-zero value is true and one that evaluates to zero is false.
- Objects that are not numbers are true if they are not **NULL** and false if they are **NULL**.

As a convenience, the Yoix interpreter automatically translates the keywords **TRUE** and **true** to **1**, and **FALSE** and **false** to **0**, but it is important to realize they are not a substitute for a boolean type because every non-zero number is treated as true, but most do not equal the value that is assigned to the keywords **TRUE** or **true**.

### Additional Remarks

Another convention we follow is to use the keywords **TRUE**, **FALSE** and **NULL** rather than **true**, **false** and **null**, even though they are, respectively, synonyms, because we like the fact that the upper-case versions stand out and are easier to spot when reading over the code.

## Types

The Yoix types that we are going to focus on next are *int*, *double*, *String*, *Array*, *Dictionary*, and *Function*. However there are about 170 others, which for the most part correspond to Java classes, though they behave like Yoix dictionaries with special predefined fields that you read, write, or execute when you want access the underlying Java class. In addition, there are types, like *Number*, *Object*, *Pointer*, *Callable*, and *Stream*, that serve as containers for different kinds of objects, e.g., an *int* or *double* is also a *Number*.

A dictionary named `typedict` is the repository for the types supported by the Yoix interpreter. It is a read-only dictionary and the values stored in `typedict` also cannot be accessed, but despite those restrictions just being able to get list of the available type names can be useful. To see what we are talking about, start the interpreter in debug mode and type just the following:

```
typedict;
```

and about 170 lines will print on you screen. We are not going to include the entire dump here, but the first five lines or so would look something like:

```
Dictionary[168:0]
  ActionEvent=--unreadable--
  AdjustmentEvent=--unreadable--
  Array=--unreadable--
  AudioClip=--unreadable--
```

Even though we have not talked about *attributes* or *for-loop* statements yet (both are covered in later sections), we still want to show you now a few lines of code that just print the type names. First turn debugging off via:

```
VM.debug = 0;
```

or by restarting the interpreter and then type:

```
import yoix.*.*;

for (n = 0; n < typedict@sizeof; n++)
  printf("%s\n", typedict[n]@nameof);
```

and all the type names will print on your screen. The list is not sorted, but that would not be hard to remedy, e.g., pipe the output through the Unix *sort* command or use the Yoix *qsort()* built-in.

## Numbers

The number types currently supported by the Yoix language are *int* and *double*, which behave exactly like the corresponding Java types. In other words an *int* can store a 32 bit signed integer while a *double* uses 64 bits to store an IEEE 754 floating point number. There are no unsigned numbers or any smaller or larger versions of integers or doubles.

Integers can be written in decimal, octal, or hexadecimal notation, so the following:

```
65535          // decimal
0177777       // octal
0xFFFF        // hex
```

all represent the same number. A number that includes a decimal point or a signed exponent, as a power of 10, introduced by an *e* or *E* is a *double*, so the following:

```
1234.5
1.2345E3
12.345e2
123.45E+1
12345e-1
```

all represent the same number. Surrounding a Unicode character with single-quotes, as shown, for example, by the following:

'w'

creates an integer that contains the code that represents the character. The escape sequences that you can use inside single-quotes include:

Escape	Meaning
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	tab
\\	backslash
\'	single-quote
\"	double-quote
\DDD	octal encoding (each digit D is 0-7)
\xDDDD	hex encoding (each digit D is 0-9, A-F, or a-f)

*Table 1. Character escapes and their meanings.*

All of them also work in string literals, which are discussed in the next section. A backslash (\) is ignored if the character following it is not listed in the above table, which obviously means we could drop \ ' and \ " from the table if we really wanted.

#### *Additional Remarks*

You will find some number-related constants, like the maximum and minimum values for *int* or *double*, in *yoix.math*.

## **Strings**

Strings are used to store text. They can be thought of as arrays of 16 bit integers that are often initialized by characters enclosed in double-quotes, so a statement such as:

```
String s1 = "Now is the time";
```

creates a string named *s1* that contains the 15 characters between the double-quotes. The escape sequences described in the last section also work in strings, so the following:

```
String s2 = "\tNow is the time\n";
```

adds a leading tab and trailing newline to the string. Strings can also be created entirely from hexadecimal digits. In that case, all you have to do is use `0x"` as the opening delimiter instead of just a double-quote. For example,

```
String s3 = 0x"4E6F77206973207468652074696D65";
```

creates a string that is identical to `s1`. The opening and closing double-quote usually must be on the same line, however if the interpreter is looking for the closing double-quote but finds a backslash at the end of a line it will discard both the backslash and the newline and then continue collecting characters on the next line.

There are other delimiters that can be used to create strings. The most useful are the pair `@<` and `>@` and the pair `@<<` and `>>@`. Both let string literals span lines and include unescaped newlines in the string. The difference between them is that the `@<` and `>@` delimiters accept escape sequences, namely the ones we listed in Table 1, while `@<<` and `>>@` do not.

Yoix strings are usually mutable. Characters stored in a string can be read or written using familiar array notation. Thus, the following:

```
s1[0] = 'n';
```

is allowed and replaces the character `N` in string `s1`, referring to our earlier example, with character `n`.

You can explicitly specify the size of a string in its declaration, so the following:

```
String s4[20] = "Now is the time";
```

creates a string named `s4` that can store 20 characters. The first 15 come from the initializer while all the others start out as the **NULL** character, which has the value zero and can be represented by the `\0` escape sequence. You can assign new values to any of the 20 characters in `s4`, so

```
s4[15] = '.';
```

replaces the **NULL** character at position 15, namely the 16<sup>th</sup> character, with a period. So, if you try the following:

```
printf("%s\n", s4);
```

you should get:

```
Now is the time.
```

on the standard output stream. Notice the period at the end.

Adding strings together concatenates them, so the following:

```
String s5 = s1 + " " + "for all good men";
```

is allowed and works the way you would expect. As usual, if you are not certain what the answer is run, the interpreter in debug mode and see for yourself, but do not forget to define `s1`. If you are a Java programmer, be careful because concatenation using the addition operator only happens when both operands are strings. In particular, note that the Yoix language lets you add an integer to a string, but the result is not the concatenation of the string and a string representation of the integer, but instead closely resembles pointer arithmetic that C programmers will recognize. We will talk more about Yoix pointers in a later section, so do not worry if this all sounds mysterious.

A reserved built-in named `toString()` takes an object, e.g., an `int`, and returns a string representation of the object and it is often used when you want to concatenate strings. For example,

```
int    day = 131;
int    year = 2008;
String today = "Day " + toString(day) + " of " + toString(year);
```

is how you might use it to create a string using concatenation. Another built-in named `strfmt()`, which is defined in `yoix.string`, offers a more flexible approach that uses syntax borrowed from `printf()` and `fprintf()`. The following:

```
today = strfmt("Day %d of %d", day, year);
```

shows how to use `strfmt()` to create the same string.

### *Additional Remarks*

You will find lots of string related built-ins in `yoix.string`. Many will be familiar to C programmers, while others have names that Java programmers will recognize though the calling conventions of these latter are different because the Yoix versions of familiar Java `String` class methods require the target string as their first argument.

## **Arrays**

Arrays are heterogeneous and they are usually initialized by comma separated expressions enclosed in braces. For example,

```
Array a1 = {1, 2.5 + 3, "Now is the time", NULL};
```

creates an array named `a1` with four elements and a dump of that array using any of the techniques outlined earlier would look like:

```
Array[4:0]
  >1
    5.5
```

```
    ^"Now is the time"
    NULL:POINTER
```

The last expression in the initializer can be followed by a comma. This flexibility can be convenient when you add a new expression to an existing array or rearrange the existing expressions.

The first object stored at an index in an array behaves like a declaration and determines what happens when that object is replaced with a different one. For example, `a1[0]` is the integer 1, so then the following:

```
    a1[0] = 12.34;
```

is allowed, but behind the scenes the interpreter quietly converts 12.34 to an integer, thus 12, rather than 12.34, is stored in `a1[0]`. However, if you try something like:

```
    a1[2] = 12.34;
```

you get a *typecheck* error because `a1[2]` is a *String* and a *Number* cannot be directly assigned to a string variable. You also can explicitly specify the size of an array in its declaration as follows:

```
    Array a2[5] = {1, 2.5 + 3, "Now is the time", NULL};
```

The above creates an array named `a2` that can store five objects, but only four of them are initialized and a dump of `a2` would look like:

```
    Array[5:0]
    >1
    5.5
    ^"Now is the time"
    NULL:POINTER
    --uninitialized--
```

Right now nothing is in `a2[4]` and that makes it unusable until you store something there using an assignment statement. In case you are wondering, you can use the *defined()* built-in to test if an element is undefined, namely:

```
    defined(4, a2);
```

would return 0, meaning false, because the element at index 4 in `a2` is undefined.

### *Additional Remarks*

Arrays are very important in GUI applications because the arrangement of components in a screen or panel is controlled by the contents of an array named `layout` that is defined in that screen or panel. In other words, you need to understand the techniques

used to initialize arrays before you can use Yoix technology to build screens for a GUI application.

Although we will talk more about the *Object* type later, we should mention now that if you need an array element to accept any type of object, you can initialize it using an *Object*. For example, by declaring array `a3` as follows:

```
Array a3 = { 5, new Object, "text", };
```

element `a3[1]` can accept any type object so that the following:

```
a3[1] = "string";
a3[1] = 9;
a3[1] = 3.141;
```

will not generate any *typecheck* errors or try to coerce one type into another. We show below how the same is true in the case of a Dictionary.

## Dictionaries

Dictionaries associate names with values and they are usually initialized by one or more variable declarations enclosed in braces. The following:

```
Dictionary d1 = {
    String text = "Now is the time";
    int    count = 1;
    double value = 5.5;
    Object obj;
};
```

creates a dictionary named `d1` with four elements and a dump of `d1` would look like:

```
Dictionary[4:0]
  count=1
  >obj=NULL:OBJECT
  text="Now is the time"
  value=5.5
```

In this example declarations were explicit, so the fact that

```
d1.count = 12.34;
```

stores 12 in `d1.count` or that

```
d1.text = 12.34;
```

results in a *typecheck* error should not be surprising. However, there is a special Yoix type named *Object* that accepts any type value, and since we used it in the declaration of the `obj` field, the following:

```
d1.obj = "for all good men";
d1.obj = 1;
d1.obj = 12.34;
```

are all allowed and there is no silent behind-the-scenes conversion by the interpreter.

As with arrays you can explicitly specify the size of a dictionary in its declaration. Thus the following:

```
Dictionary d2[7] = {
    String text = "Now is the time";
    int    count = 1;
    double value = 5.5;
    Object obj;
};
```

creates a dictionary named `d2` that can store seven objects, but only four of them are initialized and a dump of `d2` would look like:

```
Dictionary[7:0]
  count=1
  obj=NULL:OBJECT
  text="Now is the time"
>value=5.5
  --uninitialized--
  --uninitialized--
  --uninitialized--
```

Right now there are three empty slots in `d2` that are unusable until we store something in them. One way to do so is to assign a value to a field that is not currently defined, say `qwert`, as follows:

```
d2.qwert = "poiuy";
```

The above works the way you would expect and would occupy one of the uninitialized slots. However, suppose the name you want to use does not qualify as a Yoix variable name, perhaps because it includes one or more spaces. In that case you can use the notation,

```
d2["speed of light"] = 186000;
```

to add a definition for an entry named *speed of light* to dictionary `d2` and you can use `d2["speed of light"]` whenever you want to access its value. You could also save the name, i.e., *speed of light*, in a String variable, as follows:

```
String sol = "speed of light";
```

and use

```
d2[sol];
```

when you want to access the value.

Dictionaries can also be initialized by comma separated expressions that are enclosed in braces. Expressions are processed as name/value pairs, so

```
Dictionary states[0, ...] = {  
    "Delaware", 1787,  
    "Pennsylvania", 1787,  
    "New Jersey", 1787,  
    "Georgia", 1788,  
    "Connecticut", 1788,  
};
```

creates a growable dictionary named `states` that maps several state names to the year each joined the union. If the first element in a pair is not a string it is automatically converted to one using the internal version of the `toString()` built-in. If there are an odd number of expressions the last name is ignored.

## Functions

Function definitions look something like the following:

```
Add(double num1, double num2) {  
    return(num1 + num2);  
}
```

which defines a function named `Add()` that takes two numbers as arguments and returns their sum. Yoix functions currently do not declare the type of the object that they return, if any, and the type names that precede each argument in our example are also optional and would be *Object* by default. You call `Add()` the way you would expect, which is as follows:

```
printf("The sum is %g\n", Add(12.3, 45));
```

prints

```
The sum is 57.3
```

on the standard output stream. There is another, shorthand way to define simple functions like *Add()* that just evaluate and return an expression, namely:

```
AddExpr(double num1, double num2) = num1 + num2;
```

The above creates a function named *AddExpr()* that behaves just like *Add()*, but it runs slightly faster, mostly because there is no return statement.

The body of a function has access to an integer variable named *argc* and an array variable named *argv*. The array starts with a string in *argv[0]* that is the name of the function and ends with the arguments in the same order that they appeared in the function call. The value of *argc* is the length of the *argv* array, which means it is one more than the number of arguments that appeared in the function call. The globally defined *argc* and *argv* are hidden by the function's local definitions, but they always can be accessed by using *global argc* and *global argv*.

A function can be called with a variable number of arguments when the ellipsis token (*...*), i.e., three dots, appears at the end of its parameter list. In that case, the *argv* array can be used to access the unnamed arguments. For example,

```
Average(double num1, ...) {
    double sum = 0;
    int     n;

    for (n = 1; n < argc; n++)
        sum += argv[n];
    return(sum/(argc - 1));
}
```

takes a variable number of arguments, which should all be numbers to avoid a *typecheck* error, and uses the function's *argc* and *argv* variables to compute the average of the arguments. Notice that there has to be at least one argument named *num1* to insure that *argc - 1* cannot be zero, but we never reference it by name. Also notice that our loop starts at *argv[1]* because *argv[0]* is the name of the function and not one of the numbers we are supposed to average. A more robust version of this function that only averages its numeric arguments follows:

```
Average(Number num1, ...) {
    double sum = 0;
    int     count = argc - 1;
    int     n;

    for (n = 1; n < argc; n++) {
        if (argv[n] instanceof Number)
            sum += argv[n];
        else count--;
    }
}
```

```
        return(sum/count);
    }
```

The above uses the *instanceof* operator, which we discuss in a later section, to make sure each argument is a *Number*, i.e., an *int* or a *double*, and let it skip the ones that are not.

Functions generally behave the way you would expect, but after they are created they can be used just like any other variable, e.g., they can be assigned to another variable or passed as an argument in a function call, and that introduces some subtle behavior that we need to discuss. Start the interpreter (debugging mode is not needed), then type:

```
import yoix.*.*;

Hello() {
    printf("greeting=%s\n", greeting);
}
```

and you have defined an absolutely trivial function, but when you invoke it:

```
Hello();
```

you will get an error message that looks something like:

```
Error: undefined; Name: greeting; Line: 4; Source: -stdin-
```

Upon reflection this result should not be a surprise because there is no definition of *greeting* in the function itself and there is certainly not one in *global*. However, type,

```
String greeting = "hi";
```

followed by

```
Hello();
```

and

```
greeting=hi
```

prints on the standard output stream. Once again the behavior is not surprising, but it does show that the variables referenced in a function are resolved at runtime rather than when the function is defined. It also confirms that the search for variables referenced in functions will use *global* if it is needed. Next create a dictionary that has room for two objects and uses *Hello()* to define a new function named *HelloWorld()* as shown:

```
Dictionary dict[2] = {
    Function HelloWorld = Hello;
};
```

and then call it:

```
dict.HelloWorld();
```

and once again the result is:

```
greeting=hi
```

on the standard output stream. That outcome is probably what you expected, but now add a definition of `greeting` to `dict` as indicated below:

```
dict.greeting = "hello, world";
```

and invoking:

```
dict.HelloWorld();
```

prints:

```
greeting=hello, world
```

on the standard output stream. This simple example illustrates how functions look for variables that are not defined locally, i.e., within the body of the function or in the parameter list. The interpreter first searches the object, e.g., a dictionary, where the function that it is executing was defined and then it searches the *global* dictionary. Actually that last part is not completely correct because the *global* dictionary that it searches is the one that was in place when the function was originally defined. However, it is good enough explanation for now.

### *Additional Remarks*

We will discuss pointers, the indirection operator and for-loops in later sections, but we can use all three now to rewrite our function for computing averages to make it more efficient and slightly more succinct as follows:

```
Average(Number num1, ...) {
    double sum = 0;
    int    count = argc - 1;

    for (ptr in &argv[1]) {
        if (*ptr instanceof Number)
            sum += *ptr;
        else count--;
    }
}
```

```
    return(sum/count);
}
```

## This

Now that we have introduced functions, it makes sense to mention the *this* keyword. The code in the body of a function can use the *this* keyword to get a reference to the object that contains the function that the interpreter is executing. It is evaluated at runtime, so the value of *this* depends on where the function that is being executed is defined. For example, if you enter the following and run function it defines, as shown:

```
import yoix.*.*;

String str = "defined in global";

Dictionary dict = {
    String str = "defined in dict (i.e., this)";

    Testing() {
        String str = "a local definition";

        printf("str is %s\n", str);
        printf("str is %s\n", this.str);
        printf("str is %s\n", global.str);
    }
};

dict.Testing();
```

then:

```
str is a local definition
str is defined in dict (i.e., this)
str is defined in global
```

will print on the standard output stream. However if you copy `dict.Testing` to *global* and execute the copy you just made as follows:

```
global.Testing = dict.Testing;
Testing();
```

then:

```
str is a local definition
str is defined in global
str is defined in global
```

prints on the standard output stream.

### Additional Remarks

There is no rule that says you have to be in a function to use *this*, but outside a function *this* just stands for the *global* dictionary, so it is not particularly useful.

## Attributes

All Yoix objects are endowed with properties called attributes that you can access using a special notation that includes the at-sign character (@). There currently are nine attributes, which are named:

- access
- growable
- length
- major
- minor
- nameof
- offset
- sizeof
- typename

We are only planning to discuss *length*, *offset*, and *sizeof* here. These three attributes are really only useful when they are applied to objects like arrays, dictionaries, or strings, namely objects that have a visible internal structure.

To see how attributes are used start the interpreter (debug mode is not needed) and then type:

```
import yoix.*.*;
String str[10] = "poiuy";

printf("length=%d, sizeof=%d, offset=%d\n",
       str@length, str@sizeof, str@offset);
```

and the following:

```
length=10, sizeof=10, offset=0
```

will print on the standard output stream. The results are hardly surprising considering how string *str* was created, but now type:

```
str += 3;
printf("length=%d, sizeof=%d, offset=%d\n",
       str@length, str@sizeof, str@offset);
```

and you will get:

```
length=10, sizeof=7, offset=3
```

on the standard output stream. This example gives a little taste of pointer arithmetic, which we mentioned earlier and will discuss again in a little while, but for now it should be clear that adding an integer to a string adjusted the *offset* and *sizeof* attributes associated with that string. It is tempting to assume that *sizeof* and *offset* always add up to *length*, but *offset* can take on any value while *sizeof* never goes negative or exceeds *length*. In fact, *sizeof* is zero when *offset* is negative or exceeds *length*.

Incidentally, *length*, *offset* and *sizeof* are all zero when the associated object is **NULL**.

At this point you should be curious about the state of string `str`, so type:

```
printf("str=%s\n", str);
```

and you will get:

```
str=uy
```

Adding 3 to `str` modified the string's offset which apparently means we no longer see three characters that were originally part of `str`, but they are really not missing because:

```
printf("%c%c%c%c%c\n",
      str[-3], str[-2], str[-1], str[0], str[1]);
```

yields:

```
poiuy
```

on the standard output stream. The same techniques apply to objects like arrays and dictionaries, so it is worth spending a little extra time to make sure you understand this example, then see what happens when you make an obvious mistake and access `str[-4]` or `str[7]`.

Here is another simple pointer arithmetic example:

```
String digits = "0123456789";

for (; digits@sizeof; digits++)
    printf("digit=%c\n", digits[0]);

digits -= digits@offset;
printf("digits=%s\n", digits);
```

This example prints the individual characters in a string using *sizeof* to decide when the loop should stop and then uses the *offset* attribute to restore, in essence, the original string.

## Growable Objects

So far our strings, arrays, and dictionaries have all been fixed size, but it is easy to create objects that grow. For example,

```
Dictionary dict[0, 5];
```

creates a dictionary that starts out empty, but eventually could contain up to 5 objects. As another example, consider:

```
Array arr[7, ...];
```

This example creates an array that starts with 7 empty slots, but can grow indefinitely, which is what the ellipsis (...) as an upper limit means. To create a String that is empty, but has no upper limit to its size, type:

```
String buf[0, ...];
```

The *sizeof* attribute is sometimes used to append a new value to a growable object, like `buf`, that has filled all of its allocated slots, if any. Thus, the following:

```
buf[buf@sizeof] = '.'
```

appends a period to characters that are currently stored in `buf`.

On the other hand, a statement like this one:

```
buf[1000] = 'X';
```

automatically grows `buf`, if necessary, and then stores the character `x` at index 1000.

Growable objects are convenient and we often use them, but they are not always a good choice. Growing objects can be expensive and will affect performance if it is overused. The bigger the object, the more expensive the operation and the more you do it the harder Java's memory manager will have to work. Judicious use of growable objects, particularly in big applications, like `example.yx`, is important.

## Pointers

Most of the types that the Yoix language supports can be used to create objects that are called pointers, which means their internal implementation consists of two separate parts. One part, which is called the *body*, serves as a kind of storage area for the pointer. The other part contains several flags, a reference to the body and an integer

called the *offset*, which should recall to you the *offset* attribute we just discussed. Yoix scripts end up working on this latter part. We will give you a few examples using arrays, which are pointers, and then end with a picture, which might make things clearer.

To begin the example, start the interpreter in debug mode and type:

```
Array a1 = {"zero", "one", "two", "three", "four", "five",};
a1;
```

The following will appear on the standard output stream:

```
Array[6:0]
>^"zero"
^"one"
^"two"
^"three"
^"four"
^"five"
```

The colon separated numbers enclosed in the square brackets in the dump are the length of the body of the array and the offset associated with `a1`. In addition the greater-than sign (`>`) that appears in the dump is attached to the element in the body at the current offset, assuming it is in range.

Incidentally, the caret (`^`) preceding each quoted string is the current position of the *String* pointer offset, but we are not going to discuss that here, though we encourage you to experiment.

If you increment `a1` by 2, as shown here:

```
a1 += 2;
```

The offset of `a1` will be changed and the following will be dumped to the standard output stream:

```
Array[6:2]
^"zero"
^"one"
>^"two"
^"three"
^"four"
^"five"
```

Hopefully you expected this result.

The body of a pointer can be referenced many times. For example, the declaration:

```
Array a2 = a1;
```

creates a new array that references the same body as a1 references. Incrementing a2 by 3, as shown here:

```
a2 += 3;
```

prints the following will show up on the standard output stream:

```
Array[6:5]
  ^"zero"
  ^"one"
  ^"two"
  ^"three"
  ^"four"
  >^"five"
```

The offset of a1, however, is unchanged. Figure 1 shows what is going on here.

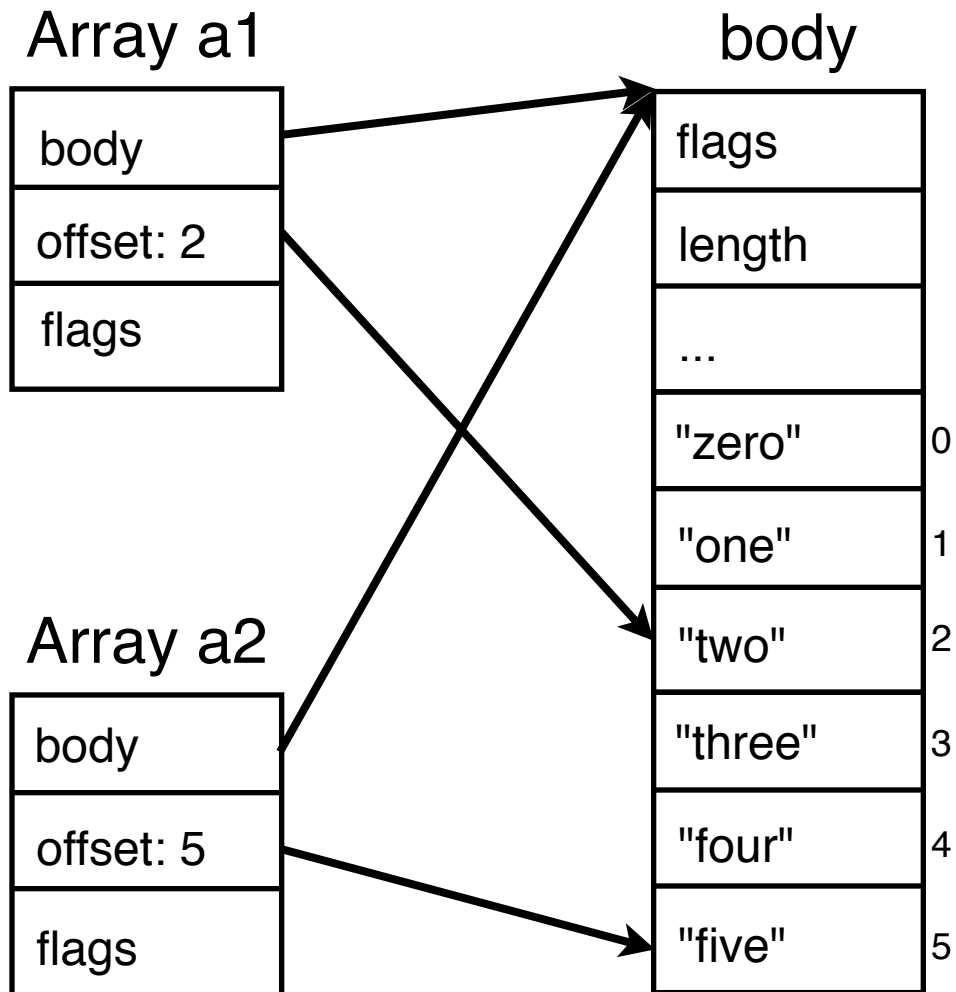


Figure 1: Two pointers referring to the same body at different offsets.

No matter what the offset of a pointer is, subtracting the current offset value from a pointer sets its offset back to zero, so the following:

```
a1 -= a1@offset;
```

yields:

```
Array[6:0]
>^"zero"
^"one"
^"two"
^"three"
^"four"
^"five"
```

on the standard output stream and, as you can see, the offset of a1 now is zero.

A pointer's offset is automatically used when the interpreter locates an element in the body referenced by that pointer. For example, type:

```
a1[0];
```

and you get:

```
^"zero"
```

However, do the same thing with a2:

```
a2[0];
```

and you get:

```
^"five"
```

It is easy explain what just happened: the interpreter quietly adds the pointer's offset to the integer that is enclosed in square brackets and then uses that number to index into the body.

## Indirection Operator

Yoix supports the indirection operator, i.e., the asterisk (\*), that C programmers will recognize, and, just like C, indirection and the array subscript notation that we have been using are interchangeable. For example, define the string:

```
String s1 = "Now is the time";
```

and you can use:

```
s1[0] = 'n';
```

or:

```
*s1 = 'n';
```

to replace the character *N* at the start of *s1* with the character *n*. Interior characters can also be accessed using indirection, so:

```
s1[7] = 'X';
```

and:

```
*(s1 + 7) = 'X';
```

make exactly the same change.

## Address Operator

Yoix scripts can create new pointers using the address operator, i.e., the ampersand (&), which can be applied to any object that can appear on the left side of an assignment statement and so for that reason is sometimes referred to as an *lvalue*.. For example, start the interpreter in debug mode using the command line shown below:

```
yoix -dl -- - -a12 -Dqwert dummy
```

then type:

```
ptr1 = &argv;
```

and something like the following:

```
Dictionary[8:5]
  VM=Dictionary[28:0]
  argc=4
  >argv=Array[4:0]
  envp=NULL:ARRAY
  errordict=Dictionary[7:0]
  importdict=Array[0:0]
  ptr=Dictionary[8:5]
  typedict=Dictionary[168:0]
```

prints on the standard output stream. In other words, when you take the address of an element in a dictionary, you end up with a dictionary that has its offset set to that element. It is a result that makes sense when you realize that:

```
ptr1[0];
```

or:

```
*ptr1;
```

print:

```
Array[4:0]
>^-stdin-
^-a12
^-Dqwert
^-dummy"
```

on the standard output stream, which is exactly what you get from:

```
argv;
```

The same thing happens if you take the address of an element in an array, string, or any other pointer: you get an object of that type that has its offset set to that element. Just to drive the point home, try typing the following:

```
Array ptr2 = &argv[2];
*ptr2;
```

and:

```
^-Dqwert"
```

will print on the standard output stream.

## The *new* Operator

The Yoix *new* operator looks something like a variable declaration, but it is an expression, e.g., you can use it in array initializers. Moreover, you get to omit the variable name. A variable declaration creates an object and a place to store that object, but *new* just creates an object that ends up as the value of the *new* expression, which means it disappears if you do not store it somewhere. If you understand the syntax used in Yoix variable declarations you will only need a few simple examples to understand the *new* operator., Start the interpreter in debug mode and type:

```
Array a1 = {1, 2, 3};
a1 = new Array {4, 5, 6, 7};
```

and you will get the following on the standard output stream:

```
Array[4:0]
>4
5
```

6  
7

Remember that debug mode prints the value of any statement that also happens to be an expression, so the dump is the value of the assignment statement, which should be the new value of `a1`. If you want to make sure just type:

```
a1;
```

and you will get the same dump. Compare the first two example lines in this sections and you will see how if you know how to form a declaration, you will know how to form a *new* expression. Namely, take the declaration, add the keyword *new*, drop the variable name, and, if there is an initializer, omit the equal sign. You can specify the size of the object the same way you do in array, dictionary, and string declarations. For example:

```
new Array[5] {8, 9, 10, 11};
```

prints:

```
Array[5:0]
>8
 9
10
11
--uninitialized--
```

on the standard output stream. The comma separated list of expressions used to initialize objects, like an array, can use *new* to create nested objects like an array of arrays or an array of dictionaries. Thus, it is not unusual to see code like:

```
Array a2 = {
  new Dictionary {
    String key = "poiuy";
    int    value = 100;
  },
  new Array {
    new Array {100, 200},
    300,
  },
  "hello, world",
};
```

in Yoix scripts.

Incidentally, if you wanted to dump the innermost array you could type the following:

```
a2[1][0]
```

and you would get the following on the standard output stream:

```
Array[2:0]
  >100
  200
```

Recall that we mentioned the `layout` array that GUI applications have to initialize in an earlier section. Now that we have introduced the `new` operator, we can show you what a trivial screen looks like using the `new` operator inside the `layout` array. Save the following text to a file:

```
import yoix.*.*;

JFrame f = {
  int visible = TRUE;

  Array layout = {
    new JTextArea {
      String text = "Here is some text";
      int    columns = 40;
    },
    CENTER,

    new JButton {
      String text = "Press Me";

      actionPerformed(e) {
        printf("I've been pressed\n");
      }
    },
    SOUTH,
  };
};
```

then invoke the interpreter on the file and a version of the screen shown in Figure 2 will appear on your screen. When you press the button labeled *Press Me*, the following



Figure 2: Screen generated from the above sample code.

string will appear on the standard output stream:

```
I've been pressed
```

Notice how we used the *new* operator to create the *JTextArea* and *JButton* objects that are displayed in the screen and how `CENTER` and `SOUTH`, which are integer constants defined in *yoix.swing*, followed each of those components to provide positioning information to the default *BorderLayout* layout manager.

## Equality Operators

The `==` and `!=` operators are used to test whether two objects are equal or not, and the answer, either a 1 for true or a 0 for false, is the value of the expression. Numbers are compared the way you would expect, but there is a double named **NaN**, which is defined in *yoix.math*, that behaves differently and deserves a brief mention. **NaN**, which stands for *Not-a-Number*, is a special bit pattern stored in a double that can be returned when the operands in an operation are invalid, e.g., `0/0` or `sqrt(-3)`. To see what we mean, start the interpreter (debug mode is not needed) and type:

```
import yoix.*.*;

printf("100 == 100: answer=%d\n", 100 == 100);
printf("100 == 200: answer=%d\n", 100 == 200);
printf("100 != 100: answer=%d\n", 100 != 100);
printf("100 != 200: answer=%d\n", 100 != 200);
```

and:

```
100 == 100: answer=1
100 == 200: answer=0
100 != 100: answer=0
100 != 200: answer=1
```

prints on standard output, exactly as you should expect. Next type:

```
printf("NaN == NaN: answer=%d\n", NaN == NaN);
printf("NaN != NaN: answer=%d\n", NaN != NaN);
```

and you get:

```
NaN == NaN: answer=0
NaN != NaN: answer=1
```

In other words, comparing **NaN** to itself using `==` or `!=` produces answers that you do not get from any other number.

Other objects, not just numbers, can be compared too. In fact, since the Yoix language is not strict like the Java language, you can compare objects even if there is no possibility they could be equal, e.g., one is an *int* and the other is a *String*.

As we have already mentioned, most of the types the Yoix language supports create objects called pointers, and pointers are considered to be equal, when compared using `==` or `!=`, only when they reference the same body and have the same offset. For example, the three strings:

```
String s1 = "hello, world.";
String s2 = s1;
String s3[] = s1;
```

contain exactly the same characters, e.g., the `strcmp()` built-in defined in `yoix.string` would consider them equal, but compare them using `==` as follows:

```
printf("s1 == s2: answer=%d\n", s1 == s2);
printf("s1 == s3: answer=%d\n", s1 == s3);
printf("s2 == s3: answer=%d\n", s2 == s3);
```

and you will see that `s3` equals neither `s1` nor `s2`:

```
s1 == s2: answer=1
s1 == s3: answer=0
s2 == s3: answer=0
```

The square brackets used in the declaration of `s3` means it gets its own copy of the characters, i.e., it has a different body, so it cannot equal `s1` or `s2`, which share a body and currently have matching offsets. We will have a bit more to say about the square bracket notation used in the the declaration of `s3` in another section, but for now increment `s2` and then compare it to `s1` as shown below:

```
s2 += 2;
printf("s1 == s2: answer=%d\n", s1 == s2);
```

and you get:

```
s1 == s2: answer=0
```

because `s1` and `s2` now have different offsets.

Obviously `==` and `!=` do not compare the individual characters in a string, but two other equality operators, namely `===` and `!==`, do. Try typing:

```
printf("s1 === s3: answer=%d\n", s1 === s3);
```

and you get:

```
s1 === s3: answer=1
```

because they contain the same characters. Both operators, i.e., `===` and `!==`, compare null terminated strings, which is exactly what the `strcmp()` built-in does, but they are a little faster because they can skip all the overhead involved in calling a built-in. A built-in named `compareTo()` can also be used to compare strings, but it does not stop at null characters so it is not exactly the same as either `===` or `strcmp()`.

For most other objects `===` and `!==` behave just like `==` and `!=`, but there are two more exceptions. If both operands are `Colors` the red, green, and blue fields in each operand are compared and the result is only true if all three fields are equal.

We will leave `NaN` comparison using `===` or `!==` as a simple exercise.

### Additional Remarks

You should not get the impression that the Yoix language requires null terminated strings like the C language. Null termination is not required, but it is available for those who, for reasons of their own, might wish to make use of it.

## Instanceof Operator

The Yoix language supports an *instanceof* operator, just like the Java language. You can use it to test whether an object is of a particular type, e.g., an *int* or *String*, or belongs to one of the so-called *container* types, e.g., *Pointer* or *Object*, that we mentioned earlier in this tutorial. It is a binary operator that expects an object as the left operand and a valid type name as the right operand. It returns one (1) if the left operand is an instance of that type and zero (0) if it is not. The *instanceof* operator always returns zero (0) when the left operand is `NULL`. To try some examples, start the interpreter (debug mode is not needed) and type:

```
import yoix.*.*;

Array a1 = {1, 2, 3};

printf("a1 is an instanceof Array: %d\n",
      a1 instanceof Array);
printf("a1 is an instanceof Pointer: %d\n",
      a1 instanceof Pointer);
printf("a1 is an instanceof Dictionary: %d\n",
      a1 instanceof Dictionary);
```

and, as you might expect, the result will be:

```
a1 is an instanceof Array: 1
a1 is an instanceof Pointer: 1
a1 is an instanceof Dictionary: 0
```

Next, assign **NULL** to `a1` and repeat the three tests as shown below:

```
a1 = NULL;

printf("a1 is an instanceof Array: %d\n",
      a1 instanceof Array);
printf("a1 is an instanceof Pointer: %d\n",
      a1 instanceof Pointer);
printf("a1 is an instanceof Dictionary: %d\n",
      a1 instanceof Dictionary);
```

and you get:

```
a1 is an instanceof Array: 0
a1 is an instanceof Pointer: 0
a1 is an instanceof Dictionary: 0
```

because, as we mentioned above, the *instanceof* operator always returns zero, i.e., **FALSE**, when the left operand is **NULL**.

### Additional Remarks

Early versions of the Yoix interpreter did not support *instanceof*, but instead provided a collection of built-ins in *yoix.type* that could be used to check whether an object was of a particular type. Those built-ins were important in early applications and they are still available, but their use is not recommended.

## Declarations Revisited

Declarations can include initializers, but when they do not the Yoix interpreter always supplies a default value. It is zero for numbers and **NULL** for strings, arrays, dictionaries, functions, and several other types that we have not discussed here. However most of the types that the Yoix language supports look like dictionaries with a predefined internal structure. For all of those types, the default value is an object of that type with the predefined fields initialized to values that are appropriate for the object. For example, start the interpreter in debug mode and type:

```
Dictionary dict;
dict;
```

and:

```
NULL:DICTIONARY
```

prints on standard output, but type:

```
Rectangle rect1;
```

```
rect1;
```

and you get:

```
Rectangle[4:0]
  height=0.0
  width=0.0
  >x=0.0
  y=0.0
```

because a rectangle is one of the types that has a predefined internal structure, i.e., `x`, `y`, `width`, and `height` fields. In this case all the fields are initially zero resulting in an empty rectangle. If instead you want a **NULL** rectangle, you must explicitly indicate that as follows:

```
Rectangle rect2 = NULL;
rect2;
```

The above results in the following on the standard output stream:

```
NULL:Rectangle
```

More than one variable can be created by a single declaration, so the following:

```
int w = 12, x, y = 100, z;
```

is allowed and does what you should expect. One variable per declaration seems more readable and is the style we prefer for the most part, even though grouping is slightly more efficient.

It is easy to use declarations or the *new* operator to make a quick copy of an object like an array, dictionary, or string. To see what we mean, make sure the interpreter is running in debug mode and then type,

```
String s1 = "Now is the time";
String s2 = s1;
String s3[] = s1;

s1[0] = 'X';
printf("s1=%s\n", s1);
printf("s2=%s\n", s2);
printf("s3=%s\n", s3);
```

and, as an earlier example should have led you to expect, the following:

```
s1=Xow is the time
s2=Xow is the time
s3=Now is the time
```

prints on the standard output stream. Notice how changing the first character in `s1` affected `s2`, but not `s3`. The empty square brackets in the declaration of `s3` asks the interpreter to create a new string that is the same size as the initializer, i.e., the string `s1`, and copy characters from `s1` into the new string. The square brackets do not have to be empty, though, so the following:

```
String s4[3] = s1;
String s5[3] = s1 + 7;

printf("s4=%s\n", s4);
printf("s5=%s\n", s5);
```

puts:

```
s4=Xow
s5=the
```

on the standard output stream. In both cases only three characters can fit in the new strings, even though the initializer is much longer, so that is all the interpreter copies. Unfortunately this technique does not currently work with objects that have a predefined internal structure. We hope to remedy the omission in the near future, but until we do you should use the `unroll()` built-in, described in an upcoming section, to create copies of them.

## The *if* Statement

C programmers should not have any trouble with the Yoix *if* statement, but Java programmers need to remember there is no boolean type. We have already discussed **TRUE** and **FALSE**, so all we will do here is give you a few simple examples. Start the interpreter (debug mode is not needed) and type:

```
import yoix.*.*;

int n = 12;

if (n)
    printf("n was true\n");
else printf("n was false\n");
```

and:

```
n was true
```

prints on the standard output stream. You use the *not* operator, i.e., exclamation-point character (!) to change the truth value of an expression. To give that a try, type the following:

```

if (!n)
    printf("n was false\n");
else printf("n was true\n");

```

which prints:

```
n was true
```

on the standard output stream. Finally, we should emphasize a point we made earlier, namely:

```

if (n == TRUE)
    printf("true branch\n");
else printf("false branch\n");

```

results in:

```
false branch
```

because `n` is 12 while **TRUE** is defined to be 1. In other words, if your intention was to compare the integers 12 and 1 then everything is fine, but if you really wanted to test whether the value stored in `n` was true or false you certainly did not get the right answer.

## The *switch* Statement

The Yoix *switch* statement looks pretty much like the ones you find in the C and Java languages except that the *case* keyword can be followed by an arbitrary unary expression rather than just an integer constant. The *case* expressions are evaluated once when the *switch* statement is first executed and the results are saved and used as the *case* labels. After that the *switch* statement evaluates the expression in parentheses and compares it to the saved *case* labels using an internal equivalent of the `===` equality operator, which means individual characters in strings and the fields in colors are compared.

A simple example should be sufficient, so start the interpreter (debug mode is not needed) and define a function named *SwitchTest()* by typing:

```

import yoix.*.*;

SwitchTest(Object arg) {
    switch (arg) {
        case "hello":
            printf("arg matched hello\n");
            break;

        case 0:
            printf("arg matched 0\n");

```

```
        break;

    case 1.2345:
        printf("arg matched 1.2345\n");
        break;

    case NaN:
        printf("arg matched NaN\n");
        break;

    default:
        printf("did not find a match\n");
        break;
}
}
```

then call it a few times. We tried the following:

```
SwitchTest("hello");
SwitchTest(0);
SwitchTest(12345E-4);
SwitchTest(NaN);
SwitchTest("Hello");
```

and we got the following:

```
arg matched hello
arg matched 0
arg matched 1.2345
arg matched NaN
did not find a match
```

on the standard output stream.

### **Additional Remarks**

Even though the grammar allows it, expressions with side effects, e.g., `x++`, are hard to justify as *case* expressions and we recommend they be avoided. In practice most *switch* statements use numbers, strings, and occasionally regular expressions, the latter of which are not covered in this tutorial.

## **The *for* Loop**

The Yoix language supports standard *for* loops that should be familiar to C and Java programmers, but it does not currently let you declare loop variables the way Java *for* loops do. The Yoix language also does not support the for-each loop that was somewhat recently added to the Java language, but it does have a *for* loop variation,

which we will also refer to as a for-each loop, that is convenient and often preferred when performance is a concern. We will demonstrate both loops in this section.

One of the first things we showed you in this tutorial was how to dump objects like `VM` in interactive mode, but suppose the dump is not exactly what you need and instead you just want the names of the fields defined in `VM`. You could use a familiar *for* loop as we show here:

```
import yoix.*.*;

int n;

for (n = 0; n < VM@sizeof; n++)
    printf("%s\n", VM[n]@nameof);
```

and the field names would be printed on the standard output stream. The list would not be sorted, but if you really wanted you could collect the names in an array and sort that array using the `qsort()` built-in. The `nameof` attribute used in the print statement was listed in the attributes section, but it was not discussed. We are sure, however, you can figure out what it does without an explanation.

The script, rewritten to use a for-each loop, looks like the following:

```
import yoix.*.*;

for (ptr in VM)
    printf("%s\n", ptr[0]@nameof);
```

and generates exactly the same list of names. In this kind of *for* loop a local variable is automatically created and set equal to a target, which must be an object, like an array, dictionary, or string, that has some internal structure, i.e., it has to be a *Pointer*. The local variable, which we called `ptr` in our example, is readonly so it can be dereferenced using array or pointer notation, e.g., `ptr[0]` or `*ptr`, but it cannot be changed. After each iteration, the offset of the local variable is incremented by one and the loop stops when the `sizeof` attribute of the variable, namely `ptr`, is zero.

If we were only interested in every other name we could use:

```
for (ptr in VM by 2)
    printf("%s\n", ptr[0]@nameof);
```

which increments the offset of `ptr` by two instead of one. Nested loops can use the same variable name, we often pick `ptr`, when statements in the inner-loop do not need access to the outer-loop variable as the eponymous inner-loop variable will hide access to the outer-loop variable.

A for-each loop runs faster than a standard *for* loop because the loop variable can be initialized, incremented, and tested entirely by Java code without requiring intermediate interpretation. The fact that it is a read-only variable also helps because the value does not require a look-up by the interpreter after each loop iteration. In most cases you will not be able to notice a difference in performance so pick the style with which you feel most comfortable.

## The *defined()* Built-In

The reserved built-in *defined()* is used to check whether an object exists or not. When *defined()* is called with a single string argument it asks the interpreter to look for a variable in the current scope that has that string as its name. When it is called with two arguments the first must be an integer or string. If the first argument is an integer, *defined()* considers that number to be an index value and checks if the element at that index is defined in the second argument. If the first argument is a string, *defined()* considers that string to be a name and checks whether the second argument contains an object defined with that name. In all these cases, the return value is one if the definition exists and zero otherwise.

For example,

```
printf("VM %s defined\n",
      defined("VM") ? "is" : "is not");
```

prints:

```
VM is defined
```

because the interpreter found a variable named VM when it went looking through the current scope. The following statements:

```
printf("VM.screen %s defined\n",
      defined("screen", VM) ? "is" : "is not");
printf("VM.qwert %s defined\n",
      defined("qwert", VM) ? "is" : "is not");
```

produce the result:

```
VM.screen is defined
VM.qwert is not defined
```

because there is a field named `screen` defined in `VM`, but no field named `qwert`.

## The *unroll()* Built-In

The reserved built-in *unroll()* copies elements from one object to another, but it is almost always used because it has special properties when it is invoked as an argument in a function call or as one of the comma separated expressions in an initializer. In each case, the interpreter notices the *unroll()* call and it decides that the individual elements in the object that *unroll()* returns, which is usually an array, rather than the object itself should be used as the function arguments or as the initializer expressions. This description is rather mysterious, but we hope an example or two should help to make it understandable. Start the interpreter (debug mode is not required) and type:

```
import yoix.*.*;

UnrollTest(String name, int value) {
    printf("name=%s, value=%d\n", name, value);
}

UnrollTest("counter", 1);
```

and:

```
name=counter, value=1
```

prints on the standard output stream, exactly as you should expect. Next try creating a simple array such as:

```
Array args = {"counter", 2};
```

and call *UnrollTest()* with that array as the argument:

```
UnrollTest(args);
```

and something like:

```
Error: badcall; Function: UnrollTest; Call:
UnrollTest; Line: 10; Source: -stdin-
```

will print on the standard error stream, though not wrapped as we were forced to do here to make the text fit. Do not worry about deciphering the error message, but the fact that the interpreter complained should not be at all surprising. *UnrollTest()* expects two arguments and we gave it one, namely a two-element array. However, since it is a two-element array, if we type:

```
UnrollTest(unroll(args));
```

we will see:

```
name=counter, value=2
```

on the standard output stream. In other words, using `unroll(args)` tricked `UnrollTest()` into believing we had actually invoked:

```
UnrollTest("counter", 2);
```

That is how `unroll()` works in function calls. To take a quick look at how it works with initializers, try creating an array such as:

```
Array a1 = {1, 2, 3, 4};
```

and then type:

```
Array a2 = {0, a1, 5};
```

```
printf("a2=%O\n", a2);
```

You will find that:

```
a2=Array[3:0]
>0
  Array[4:0]
  5
```

prints on the standard output stream, exactly as you should expect. However, when you now try creating a new array using `unroll()` as follows:

```
Array a3 = {0, unroll(a1), 5};
```

```
printf("a3=%O\n", a3);
```

you will get:

```
a3=Array[6:0]
>0
  1
  2
  3
  4
  5
```

on the standard output stream because, this time, using `unroll(a1)` functioned as if we had initialized `a3` by typing:

```
Array a3 = {0, 1, 2, 3, 4, 5};
```

As discussed in an earlier section, declarations or the *new* operator make it easy to duplicate an array, dictionary, or string through the use square brackets, which is a cue to the interpreter to create a new object that is the right size and copy the initializer into that object. Unfortunately, as we also mentioned, the mechanism does not currently work with objects with predefined fields such as `Point`, `Rectangle`, `Color`, but `unroll()` provides an alternative method. Thus, the following:

```
Point p1 = {
    double x = 100.234;
    double y = 12.5;
};

Point p2 = {
    double x = p1.x;
    double y = p1.y;
};

Point p3 = {unroll(p1)};
```

creates a point named `p1` and shows you two ways to duplicate that point. However, were you to try to use square brackets the way you would with a dictionary, namely:

```
Point p4[] = p1;
```

a message of the form:

```
Error: baddeclaration; Name: Point; Line: 18; Source: -stdin-
```

would print on the standard error stream. There are subtle internal reasons why the bracket syntax does not currently work, but we believe they can be resolved and it is something we hope to accomplish in the near future.

## Concluding Remarks

We will end this informal language tutorial here. Clearly there is a need for a usage tutorial that would cover topics such as GUI construction, I/O techniques and 2D drawing, to name just a few.

We hope, however, that this tutorial and the examples we have provided, such as `elements.yx`, will give you a good start. For the adventurous, the source to `YWAIT` and to the `Byzgraf` tools also provides a wealth of Yoix language techniques and usage examples.

