breakpoint ***varName value***: Starts comparing the current value of the specified variable to the specified *value* following the execution of each subsequent script command. If the variable contains the specified *value*, the execution halts and a message is displayed. Otherwise, the execution continues normally. Useful for debugging purposes.

clear-breakpoints: Clears all the previously defined breakpoints.

builtInChipName method argument (s): Executes the specified method of the specified built-in chip-part using the supplied arguments. The designer of a built-in chip can provide methods that allow the user (or a test script) to manipulate the simulated chip. See figure A3.2.

Variables of built-in chips: Chips can be implemented either by HDL programs or by externally supplied executable modules. In the latter case, the chip is said to be *built-in*. built-in chips can facilitate access to the chip's state using the syntax *chipName*[*varName*], where *varName* is an implementation-specific variable that should be documented in the chip API. See figure A3.2.

For example, consider the script command set RAM16K[1017] 15. If RAM16K is the currently simulated chip, or a chip-part of the currently simulated chip, this command sets its memory location number 1017 to 15. And, since the built-in RAM16K chip happens to have GUI side effects, the new value will also be reflected in the chip's visual image.

If a built-in chip maintains a single-valued internal state, the current value of the state can be accessed through the notation *chipName*[]. If the internal state is a vector, the notation *chipName*[*i*] is used. For example, when simulating the built-in Register chip, one can write script commands like set Register[] 135. This command sets the internal state of the chip to 135; in the next time unit, the Register chip will commit to this value, and its output pin will start emitting it.

Methods of built-in chips: Built-in chips can also expose *methods* that can be used by scripting commands. For example, in the Hack computer, programs reside in an instruction memory unit implemented by the built-in chip ROM32K. Before running a machine language program on the Hack computer, the program must be loaded into this chip. To facilitate this service, the built-in implementation of ROM32K features a load method that enables loading a text file containing machine language instructions. This method can be accessed using a script command like ROM32K load *fileName*.hack.

Ending example: We end this section with a relatively complex test script designed to test the topmost Computer chip of the Hack computer.

One way to test the Computer chip is to load a machine language program into it and monitor selected values as the computer executes the program, one instruction at a time. For example, we wrote a machine language program that computes the maximum of RAM[0] and RAM[1] and writes the result in RAM[2]. The program is stored in a file named Max.hack.

Note that at the low level in which we are operating, if such a program does not run properly it may be either because the program is buggy or because the hardware is buggy (or, perhaps, the test script is buggy, or the hardware simulator is buggy). For simplicity, let us assume that everything is error-free, except for, possibly, the simulated Computer chip.

To test the Computer chip using the Max.hack program, we wrote a test script called ComputerMax.tst. This script loads Computer.hdl into the hardware simulator and then loads the Max.hack program into its ROM32K chip-part. A reasonable way to check whether the chip works properly is as follows: Put some values in RAM[0] and RAM[1], reset the computer, run the clock enough cycles, and inspect RAM[2]. This, in a nutshell, is what the script in figure A3.3 is designed to do.

```
/* ComputerMax.tst script.
Uses a Max.hack program that sets
RAM[2] to max(RAM[0], RAM[1]). */

// Loads Computer and sets up for the simulation:
load Computer.hdl,
output-file ComputerMax.out,
compare-to ComputerMax.cmp,
output-list RAM16K[0] RAM16K[1] RAM16K[2];

// Loads Max.hack into the ROM32K chip-part:
ROM32K load Max.hack,

// Sets the first 2 cells of the RAM16K chip-part
// to test values:
set RAM16K[0] 3,
set RAM16K[1] 5,
output;

// Runs enough clock cycles to complete the
// program's execution:
repeat 14 {
    tick, tock,
    output;
}

// (Script continues on the right)
```

```
// Sets up for another test, using other values.

// Resets the Computer: Done by setting
// reset to 1, and running the clock
// in order to commit the Program Counter
// (PC, a sequential chip) to the new reset value:
set reset 1,
tick,
tock,
output;

// Sets reset to 0, loads new test values, and
// runs enough clock cycles to complete the
// program's execution:
set reset 0,
set RAM16K[0] 23456,
set RAM16K[1] 12345,
output;
repeat 14 {
    tick, tock,
    output;
}
```

Figure A3.3 Testing the topmost Computer chip.

How can we tell that fourteen clock cycles are sufficient for executing this program? This can be found by trial and error, by starting with a large value and watching the computer's outputs stabilizing after a while, or by analyzing the run-time behavior of the loaded program.

Default test script: Each Nand to Tetris simulator features a default test script. If the user does not load a test script into the simulator, the default test script is used. The default test script of the hardware simulator is defined as follows:

```
// Default test script of the hardware simulator:
repeat {
    tick,
    tock;
}
```

A3.3 Testing Machine Language Programs on the CPU Emulator

Unlike the *hardware simulator*, which is a general-purpose program designed to support the construction of any hardware platform, the supplied

CPU emulator is a single-purpose tool, designed to simulate the execution of machine language programs on a specific platform: the Hack computer. The programs can be written either in the symbolic or in the binary Hack machine language described in chapter 4.

As usual, the simulation involves four files: the tested program (*Xxx*.asm or *Xxx*.hack), a test script (*Xxx*.tst), an optional output file (*Xxx*.out), and an optional compare file (*Xxx*.cmp). All these files reside in the same folder, normally named *Xxx*.

Example: Consider the multiplication program Mult.hack, designed to effect RAM[2]=RAM[0]*RAM[1]. Suppose we want to test this program in the CPU emulator. A reasonable way to do it is to put some values in RAM[0] and RAM[1], run the program, and inspect RAM[2]. This logic is carried out by the test script shown in figure A3.4.

```
// Loads the program and sets up for the simulation:
load Mult.hack,
output-file Mult.out,
compare-to Mult.cmp,
output-list RAM[2]%D2.6.2;

// Sets the first 2 RAM cells to test values:
set RAM[0] 2,
set RAM[1] 5;

// Runs enough clock cycles to complete the program's execution:
repeat 20 {
  ticktock;
}
output;

// Re-runs the program, with different test values:
set PC 0,
set RAM[0] 8,
set RAM[1] 7;

// Mult.hack is based on a naïve repetitive addition algorithm,
// so greater multiplicands require more clock cycles:
repeat 50 {
  ticktock;
}
output;
```

Figure A3.4 Testing a machine language program on the CPU emulator.

Variables: Scripting commands running on the CPU emulator can access the following elements of the Hack computer:

`A:`	Current value of the address register (unsigned 15-bit)
`D:`	Current value of the data register (16-bit)
`PC:`	Current value of the Program Counter (unsigned 15-bit)
`RAM[`*i*`]:`	Current value of RAM location *i* (16-bit)
`time:`	Number of *time units* (also called *clock cycles*, or *ticktocks*) that elapsed since the simulation started (a read-only system variable)

Commands: The CPU emulator supports all the commands described in section A3.2, except for the following changes:

load *progName*: Where *progName* is either *Xxx*.asm or *Xxx*.hack. This command loads a machine language program (to be tested) into the simulated instruction memory. If the program is written in assembly, the simulator translates it into binary, on the fly, as part of executing the load *programName* command.

eval: Not applicable in the CPU emulator.

builtInChipName method argument (*s*): Not applicable in the CPU emulator.

tickTock: This command is used instead of tick and tock. Each ticktock advances the clock one time unit (cycle).

Default Test Script

```
// Default test script of the CPU emulator:
repeat {
    ticktock;
}
```

A3.4 Testing VM Programs on the VM Emulator

The supplied *VM emulator* is a Java implementation of the virtual machine specified in chapters 7–8. It can be used for simulating the execution of VM programs, visualizing their operations, and displaying the states of the effected virtual memory segments.

A VM program consists of one or more .vm files. Thus, the simulation of a VM program involves the tested program (a single *Xxx*.vm file or an *Xxx* folder containing one or more .vm files) and, optionally, a test script (*Xxx*.tst), a compare file (*Xxx*.cmp), and an output file (*Xxx*.out). All these files reside in the same folder, normally named *Xxx*.

Virtual memory segments: The VM commands push and pop are designed to manipulate *virtual memory segments* (argument, local, and so on). These segments must be allocated to the host RAM—a task that the VM emulator carries out as a side effect of simulating the execution of the VM commands call, function, and return.

Startup code: When the VM translator translates a VM program, it generates machine language code that sets the stack pointer to 256 and then calls the Sys.init function, which then initializes the OS classes and calls Main.main. In a similar fashion, when the VM emulator is instructed to execute a VM program (a collection of one or more VM functions), it is programmed to start running the function Sys.init. If such a function is not found in the loaded VM code, the emulator is programmed to start executing the first command in the loaded VM code.

The latter convention was added to the VM emulator to support unit testing of the VM translator, which spans two book chapters and projects. In project 7, we build a basic VM translator that handles only push, pop, and arithmetic commands without handling function calling commands. If we want to execute such programs, we must somehow anchor the virtual memory segments in the host RAM—at least those segments mentioned in the simulated VM code. Conveniently, this initialization can be accomplished by script commands that manipulate the pointers controlling the base RAM addresses of the virtual segments. Using these script commands, we can anchor the virtual segments anywhere we want in the host RAM.

Example: The FibonacciSeries.vm file contains a sequence of VM commands that compute the first *n* elements of the Fibonacci series. The code is designed to operate on two arguments: *n* and the starting memory address in which the computed elements should be stored. The test script listed in figure A3.5 tests this program using the arguments 6 and 4000.

```
/* The FibonacciSeries.vm program computes the first n Fibonacci numbers.
In this test n = 6, and the numbers will be written to RAM addresses 4000 to 4005. */

load FibonacciSeries.vm,
output-file FibonacciSeries.out,
compare-to FibonacciSeries.cmp,
output-list RAM[4000]%D1.6.2 RAM[4001]%D1.6.2 RAM[4002]%D1.6.2
            RAM[4003]%D1.6.2 RAM[4004]%D1.6.2 RAM[4005]%D1.6.2;

// The program's code contains no function/call/return commands.
// Therefore, the script initializes the stack, local and argument segments explicitly:

set SP 256,
set local 300,
set argument 400;

// Sets the first argument to n = 6, the second argument to the address where the series
// will be written, and runs enough VM steps for completing the program's execution:

set argument[0] 6,
set argument[1] 4000;
repeat 140 {
  vmstep;
}
output;
```

Figure A3.5 Testing a VM program on the VM emulator.

Variables: Scripting commands running on the VM emulator can access the following elements of the virtual machine:

Contents of VM segments:

`local[`*i*`]`:	Value of the *i*-th element of the `local` segment
`argument[`*i*`]`:	Value of the *i*-th element of the `argument` segment
`this[`*i*`]`:	Value of the *i*-th element of the `this` segment
`that[`*i*`]`:	Value of the *i*-th element of the `that` segment
`temp[`*i*`]`:	Value of the *i*-th element of the `temp` segment

Pointers of VM segments:

`local`:	Base address of the `local` segment in the RAM
`argument`:	Base address of the `argument` segment in the RAM
`this`:	Base address of the `this` segment in the RAM
`that`:	Base address of the `that` segment in the RAM

Implementation-specific variables:

`RAM[i]`:	Value of the *i*-th location of the host RAM
`SP`:	Value of the stack pointer
`currentFunction`:	Name of the currently executing function (read-only)
`line`:	Contains a string of the form *currentFunctionName*`.`*lineIndexInFunction* (read-only). For example, when execution reaches the third line of the function `Sys.init`, the `line` variable contains the value `Sys.init.3`. Can be used for setting breakpoints in selected locations in the loaded VM program.

Commands: The VM emulator supports all the commands described in Section A3.2, except for the following changes:

load *source*: Where the optional *source* parameter is either *Xxx*.vm, a file containing VM code, or *Xxx*, the name of a folder containing one or more .vm files (in which case all of them are loaded, one after the other). If the .vm files are located in the current folder, the source argument can be omitted.

tick / tock: Not applicable.

vmstep: Simulates the execution of a single VM command and advances to the next command in the code.

Default Script

```
// Default script of the VM emulator:
repeat {
    vmStep;
}
```

Appendix 4: The Hack Chip Set

The chips are sorted alphabetically by name. In the online version of this document, available in www.nand2tetris.org, this API format comes in handy: To use a chip-part, copy-paste the chip signature into your HDL program, then fill in the missing bindings (also called *connections*).

```
Add16(a= ,b= ,out= ) /* Adds up two 16-bit two's complement values */
ALU(x= ,y= ,zx= ,nx= ,zy= ,ny= ,f= ,no= ,out= ,zr= ,ng= ) /* Hack ALU */
And(a= ,b= ,out= ) /* And gate */
And16(a= ,b= ,out= ) /* 16-bit And */
ARegister(in= ,load= ,out= ) /* Address register (built-in) */
Bit(in= ,load= ,out= ) /* 1-bit register */
CPU(inM= ,instruction= ,reset= ,outM= ,writeM= ,addressM= ,pc= ) /* Hack CPU */
DFF(in= ,out= ) /* Data flip-flop gate (built-in) */
DMux(in= ,sel= ,a= ,b= ) /* Routes the input to one out of two outputs */
DMux4Way(in= ,sel= ,a= ,b= ,c= ,d= ) /* Routes the input to one out of four outputs */
DMux8Way(in= ,sel= ,a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= ) /* Routes the input to one out of 8 outputs */
DRegister(in= ,load= ,out= ) /* Data register (built-in) */
HalfAdder(a= ,b= ,sum= , carry= ) /* Adds up two bits */
FullAdder(a= ,b= ,c= ,sum= ,carry= ) /* Adds up three bits */
Inc16(in= ,out= ) /* Sets out to in + 1 */
Keyboard(out= ) /* Keyboard memory map (built-in) */
Memory(in= ,load= ,address= ,out= ) /* Data memory of the Hack platform (RAM) */
Mux(a= ,b= ,sel= ,out= ) /* Selects between two inputs */
Mux16(a= ,b= ,sel= ,out= ) /* Selects between two 16-bit inputs */
Mux4Way16(a= ,b= ,c= ,d= ,sel= ,out= ) /* Selects between four 16-bit inputs */
Mux8Way16(a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= ,sel= ,out= ) /* Selects between eight 16-bit inputs */
Nand(a= ,b= ,out= ) /* Nand gate (built-in) */
Not(in= ,out= ) /* Not gate */
Not16(in= ,out= ) /* 16-bit Not */
Or(a= ,b= ,out= ) /* Or gate */
Or16(a= ,b= ,out= ) /* 16-bit Or */
Or8Way(in= ,out= ) /* 8-way Or */
PC(in= ,load= ,inc= ,reset= ,out= ) /* Program Counter */
RAM8(in= ,load= ,address= ,out= ) /* 8-word RAM */
RAM64(in= ,load= ,address= ,out= ) /* 64-word RAM */
RAM512(in= ,load= ,address= ,out= ) /* 512-word RAM */
RAM4K(in= ,load= ,address= ,out= ) /* 4K RAM */
RAM16K(in= ,load= ,address= ,out= ) /* 16K RAM */
Register(in= ,load= ,out= ) /* 16-bit register */
ROM32K(address= ,out= ) /* Instruction memory of the Hack platform (ROM, built-in) */
Screen(in= ,load= ,address= ,out= ) /* Screen memory map (built-in) */
Xor(a= ,b= ,out= ) /* Xor gate */
```

Appendix 5: The Hack Character Set

32:	space	56:	8	80:	P	104:	h	127:	DEL
33:	!	57:	9	81:	Q	105:	i	128:	newLine
34:	"	58:	:	82:	R	106:	j	129:	backSpace
35:	#	59:	;	83:	S	107:	k	130:	leftArrow
36:	$	60:	<	84:	T	108:	l	131:	upArrow
37:	%	61:	=	85:	U	109:	m	132:	rightArrow
38:	&	62:	>	86:	V	110:	n	133:	downArrow
39:	'	63:	?	87:	W	111:	o	134:	home
40:	(	64:	@	88:	X	112:	p	135:	end
41:	)	65:	A	89:	Y	113:	q	136:	pageUp
42:	*	66:	B	90:	Z	114:	r	137:	pageDown
43:	+	67:	C	91:	[	115:	s	138:	insert
44:	,	68:	D	92:	/	116:	t	139:	delete
45:	-	69:	E	93:	]	117:	u	140:	esc
46:	.	70:	F	94:	^	118:	v	141:	f1
47:	/	71:	G	95:	_	119:	w	142:	f2
48:	0	72:	H	96:	`	120:	x	143:	f3
49:	1	73:	I	97:	a	121:	y	144:	f4
50:	2	74:	J	98:	b	122:	z	145:	f5
51:	3	75:	K	99:	c	123:	{	146:	f6
52:	4	76:	L	100:	d	124:	|	147:	f7
53:	5	77:	M	101:	e	125:	}	148:	f8
54:	6	78:	N	102:	f	126:	~	149:	f9
55:	7	79:	O	103:	g			150:	f10
								151:	f11
								152:	f12

Appendix 6: The Jack OS API

The Jack language is supported by eight standard classes that provide basic OS services like memory allocation, mathematical functions, input capturing, and output rendering. This appendix documents the API of these classes.

Math

This class provides commonly needed mathematical functions.

`function int multiply(int x, int y)`: Returns the product of `x` and `y`. When a Jack compiler detects the multiplication operator `*` in the program's code, it handles it by invoking this function. Thus the Jack expressions `x * y` and the function call `Math.multiply(x,y)` return the same value.

`function int divide(int x, int y)`: Returns the integer part of `x / y`. When a Jack compiler detects the division operator `/` in the program's code, it handles it by invoking this function. Thus the Jack expressions `x / y` and the function call `Math.divide(x,y)` return the same value.

`function int min(int x, int y)`: Returns the minimum of `x` and `y`.

`function int max(int x, int y)`: Returns the maximum of `x` and `y`.

`function int sqrt(int x)`: Returns the integer part of the square root of `x`.

String

This class represents strings of char values and provides commonly needed string processing services.

constructor String new(int maxLength): Constructs a new empty string with a maximum length of maxLength and initial length of 0.

method void dispose(): Disposes this string.

method int length(): Returns the number of characters in this string.

method char charAt(int i): Returns the character at the i-th location of this string.

method void setCharAt(int i, char c): Sets the character at the i-th location of this string to c.

method String appendChar(char c): Appends c to this string's end and returns this string.

method void eraseLastChar(): Erases the last character from this string.

method int intValue(): Returns the integer value of this string until a non-digit character is detected.

method void setInt(int val): Sets this string to hold a representation of the given value.

function char backSpace(): Returns the backspace character.

function char doubleQuote(): Returns the double quote character.

function char newLine(): Returns the newline character.

Array

In the Jack language, arrays are implemented as instances of the OS class Array. Once declared, the array elements can be accessed using the syntax arr[i]. Jack arrays are not typed: each array element can hold a primitive data type or an object type, and different elements in the same array can have different types.

function Array new(int size): Constructs a new array of the given size.

method void dispose(): Disposes this array.

Output

This class provides functions for displaying characters. It assumes a character-oriented screen consisting of 23 rows (indexed 0…22, top to bottom) of 64 characters each (indexed 0…63, left to right). The top-left character location on the screen is indexed (0,0). Each character is displayed by rendering on the screen a rectangular image 11 pixels high and 8 pixels wide (which includes margins for character spacing and line spacing). If needed, the bitmap images ("font") of all the characters can be found by inspecting the given code of the `Output` class. A visible cursor, implemented as a small filled square, indicates where the next character will be displayed.

`function void moveCursor(int i, int j)`: Moves the cursor to the `j`-th column of the `i`-th row and overrides the character displayed there.

`function void printChar(char c)`: Displays the character at the cursor location and advances the cursor one column forward.

`function void printString(String s)`: Displays the string starting at the cursor location and advances the cursor appropriately.

`function void printInt(int i)`: Displays the integer starting at the cursor location and advances the cursor appropriately.

`function void println()`: Advances the cursor to the beginning of the next line.

`function void backSpace()`: Moves the cursor one column back.

Screen

This class provides functions for displaying graphical shapes on the screen. The Hack physical screen consists of 256 rows (indexed 0…255, top to bottom) of 512 pixels each (indexed 0…511, left to right). The top-left pixel on the screen is indexed (0,0).

`function void clearScreen()`: Erases the entire screen.

`function void setColor(boolean b)`: Sets the current color. This color will be used in all the subsequent `draw`*Xxx* function calls. Black is represented by `true`, white by false.

function void drawPixel(int x, int y): Draws the (x,y) pixel using the current color.

function void drawLine(int x1, int y1, int x2, int y2): Draws a line from pixel (x1,y1) to pixel (x2,y2) using the current color.

function void drawRectangle(int x1, int y1, int x2, int y2): Draws a filled rectangle whose top-left corner is (x1,y1) and bottom-right corner is (x2,y2) using the current color.

function void drawCircle(int x, int y, int r): Draws a filled circle of radius $r \leq 181$ around (x,y) using the current color.

Keyboard

This class provides functions for reading inputs from a standard keyboard.

function char keyPressed(): Returns the character of the currently pressed key on the keyboard; if no key is currently pressed, returns 0. Recognizes all the values in the Hack character set (see appendix 5). These include the characters newLine (128, return value of String.newLine()), backSpace (129, return value of String.backSpace ()), leftArrow (130), upArrow (131), rightArrow (132), downArrow (133), home (134), end (135), pageUp (136), pageDown (137), insert (138), delete (139), esc (140), and f1–f12 (141–152).

function char readChar(): Waits until a keyboard key is pressed and released, then displays the corresponding character on the screen and returns the character.

function String readLine(String message): Displays the message, reads from the keyboard the entered string of characters until a newLine character is detected, displays the string, and returns the string. Also handles user backspaces.

function int readInt(String message): Displays the message, reads from the keyboard the entered string of characters until a newLine character is detected, displays the string on the screen, and returns its integer value until the first non-digit character in the entered string is detected. Also handles user backspaces.

Memory

This class provides memory management services. The Hack RAM consists of 32,768 words, each holding a 16-bit binary number.

function int peek(int address): Returns the value of RAM[address].

function void poke(int address, int value): Sets RAM[address] to the given value.

function Array alloc(int size): Finds an available RAM block of the given size and returns its base address.

function void deAlloc(Array o): Deallocates the given object, which is cast as an array. In other words, makes the RAM block that starts in this address available for future memory allocations.

Sys

This class provides basic program execution services.

function void halt(): Halts the program execution.

function void error(int errorCode): Displays the error code, using the format ERR<errorCode>, and halts the program's execution.

function void wait(int duration): Waits approximately duration milliseconds and returns.

Index