9 High-Level Language

High thoughts need a high language.

—Aristophanes (427–386 B.C.)

The assembly and VM languages presented so far in this book are low-level, meaning that they are intended for controlling machines, not for developing applications. In this chapter we present a high-level language, called Jack, designed to enable programmers to write high-level programs. Jack is a simple object-based language. It has the basic features and flavor of mainstream languages like Java and C++, with a simpler syntax and no support for inheritance. Despite this simplicity, Jack is a general-purpose language that can be used to create numerous applications. In particular, it lends itself nicely to interactive games like Tetris, Snake, Pong, Space Invaders, and similar classics.

The introduction of Jack marks the beginning of the end of our journey. In chapters 10 and 11 we will write a compiler that translates Jack programs into VM code, and in chapter 12 we will develop a simple operating system for the Jack/Hack platform. This will complete the computer's construction. With that in mind, it's important to say at the outset that the goal of this chapter is not to turn you into a Jack programmer. Neither do we claim that Jack is an important language outside the Nand to Tetris context. Rather, we view Jack as a necessary scaffold for chapters 10–12, in which we will build a compiler and an operating system that make Jack possible.

If you have any experience with a modern object-oriented programming language, you will immediately feel at home with Jack. Therefore, we begin the chapter with a few representative examples of Jack programs. All these programs can be compiled by the Jack compiler supplied in nand2tetris/tools. The VM code produced by the compiler can then be executed as is on any VM implementation, including the supplied VM emulator. Alternatively, you can translate the compiled VM code further into machine language, using the VM translator built in chapters 7–8. The resulting assembly code can then be executed on the supplied CPU emulator or translated further into binary code and executed on the hardware platform built in chapters 1–5.

Jack is a simple language, and this simplicity has a purpose. First, you can learn (and unlearn) Jack in about one hour. Second, the Jack language was carefully designed to lend itself nicely to common compilation techniques. As a result, you can write an elegant *Jack*

compiler with relative ease, as we will do in chapters 10 and 11. In other words, the deliberately simple structure of Jack is designed to help uncover the software infrastructure of modern languages like Java and C#. Rather than taking the compilers and run-time environments of these languages apart, we find it more instructive to build a compiler and a run-time environment ourselves, focusing on the most important ideas underlying their construction. This will be done later, in the last three chapters of the book. Presently, let's take Jack out of the box.

9.1 Examples

Jack is mostly self-explanatory. Therefore, we defer the language specification to the next section and start with examples. The first example is the inevitable *Hello World* program. The second example illustrates procedural programming and array processing. The third example illustrates how abstract data types can be implemented in the Jack language. The fourth example illustrates a linked list implementation using the language's object-handling capabilities.

Throughout the examples, we discuss briefly various object-oriented idioms and commonly used data structures. We assume that the reader has a basic familiarity with these subjects. If not, read on—you'll manage.

Example 1: Hello World: The program shown in figure 9.1 illustrates several basic Jack features. By convention, when we execute a compiled Jack program, execution always starts with the Main.main function. Thus, each Jack program must include at least one class, named Main, and this class must include at least one function, named Main.main. This convention is illustrated in figure 9.1.

```
/** Prints "Hello World". File name: Main.jack */
class Main {
   function void main() {
      do Output.printString("Hello World");
      do Output.println();   // New line
      return;                // The return statement is mandatory
   }
}
```

Figure 9.1 *Hello World*, written in the Jack language.

Jack comes with a *standard class library* whose complete API is given in appendix 6. This software library, also known as the *Jack OS*, extends the basic language with various abstractions and services such as mathematical functions, string processing, memory management, graphics, and input/output functions. Two such OS functions are invoked by

the Hello World program, affecting the program's output. The program also illustrates the comment formats supported by Jack.

Example 2: Procedural programming and array handling: Jack features typical statements for handling assignment and iteration. The program shown in figure 9.2 illustrates these capabilities in the context of array processing.

```
/** Inputs a sequence of integers, and computes their average. */
class Main {
   function void main() {
      var Array a;      // Jack arrays are not typed
      var int length;
      var int i, sum;
      let i = 0;
      let sum = 0;
      let length = Keyboard.readInt("How many numbers? ");
      let a = Array.new(length); // Constructs the array
      while (i < length) {
         let a[i] = Keyboard.readInt("Enter a number: ");
         let sum = sum + a[i];
         let i = i + 1;
      }
      do Output.printString("The average is: ");
      do Output.printInt(sum / length);
      do Output.println();
      return;
    }
}
```

Figure 9.2 Typical procedural programming and simple array handling. Uses the services of the OS classes Array, Keyboard, and Output.

Most high-level programming languages feature array declaration as part of the basic syntax of the language. In Jack, we have opted for treating arrays as instances of an Array class, which is part of the OS that extends the basic language. This was done for pragmatic reasons, as it simplifies the construction of Jack compilers.

Example 3: Abstract data types: Every programming language features a fixed set of primitive data types, of which Jack has three: int, char, and boolean. In object-based languages, programmers can introduce new types by creating classes that represent abstract data types as needed. For example, suppose we wish to endow Jack with the ability to handle rational numbers like 2/3 and 314159/100000 without loss of precision. This can be done by developing a standalone Jack class designed to create and manipulate fraction objects of the form *x/y*, where *x* and *y* are integers. This class can then provide a fraction abstraction to any Jack program that needs to represent and manipulate rational numbers. We now turn to describe how a Fraction class can be used and developed. This example illustrates typical

multi-class, object-based programming in Jack.

Using classes: Figure 9.3a lists a class skeleton (a set of method signatures) that specifies some of the services that one can reasonably expect to get from a fraction abstraction. Such a specification is often called an *Application Program Interface*. The client code at the bottom of the figure illustrates how this API can be used for creating and manipulating fraction objects.

```
/** Represents the Fraction type and related operations (class skeleton) */
class Fraction {

    /** Constructs a (reduced) fraction from x and y */
    constructor Fraction new(int x, int y)

    /** Returns the numerator of this fraction */
    method int getNumerator()

    /** Returns the denominator of this fraction */
    method int getDenominator()

    /** Returns the sum of this fraction and the other one */
    method Fraction plus(Fraction other)

    /** Prints this fraction in the format x/y */
    method void print()

    /** Disposes this fraction */
    method void dispose() {

    // More fraction-related methods:
    // minus, times, div, invert, etc.
}
```

```
// Computes and prints the sum of 2/3 and 1/5
class Main {
   function void main() {
      // Creates 3 fraction variables (pointers to Fraction objects)
      var Fraction a, b, c;
      let a = Fraction.new(4,6); // a = 2/3
      let b = Fraction.new(1,5); // b = 1/5
      // Adds up the two fractions, and prints the result
      let c = a.plus(b);    // c = a + b
      do c.print();         // Should print "13/15"
      return;
   }
}
```

Figure 9.3a Fraction API (top) and sample Jack class that uses it for creating and manipulating Fraction objects.

Figure 9.3a illustrates an important software engineering principle: users of an abstraction (like Fraction) don't have to know anything about its implementation. All they need is the class *interface*, also known as API. The API informs what functionality the class offers and how to use this functionality. That's all the client needs to know.

Implementing classes: So far, we have seen only the client perspective of the Fraction class—the view from which Fraction is used as a black box abstraction. Figure 9.3b lists one possible implementation of this abstraction.

```
/** Represents the Fraction type and related operations. */
class Fraction {
    // Each Fraction object has a numerator and a denominator
    field int numerator, denominator;
    /** Constructs a (reduced) fraction from x and y */
    constructor Fraction new(int x, int y) {
       let numerator = x;
       let denominator = y;
       do reduce();    // Reduces this fraction
       return this;    // Returns a reference to the new object
    }
    // Reduces this fraction
    method void reduce() {
       var int g;
       let g = Fraction.gcd(numerator, denominator);
       if (g > 1) {
          let numerator = numerator / g;
          let denominator = denominator / g;
       }
       return;
    }
    // Computes the greatest common divisor of two given integers
    function int gcd(int a, int b) {
       // Applies Euclid's algorithm
       var int r;
       while (~(b = 0)) {
          let r = a - (b * (a / b));   // r = remainder
          let a = b;
          let b = r;
       }
       return a;
    }
    // The Fraction class declaration continues on the top right
```

```
    /** Accessors */
    method int getNumerator() {
       return numerator;
    }
    method int getDenominator() {
       return denominator;
    }
    /** Returns the sum of this fraction and the other one */
    method Fraction plus(Fraction other) {
       var int sum;
       let sum = (numerator * other.getDenominator())+
                 (other.getNumerator() * denominator);
       return Fraction.new(sum, denominator *
                           other.getDenominator());
    }
    /** Prints this fraction in the format x/y */
    method void print() {
       do Output.printInt(numerator);
       do Output.printString("/");
       do Output.printInt(denominator);
       return;
    }
    /** Disposes this fraction */
    method void dispose() {
       // Frees the memory held by this object
       do Memory.deAlloc(this);
       return;
    }
    // More fraction-related methods can come here:
    // minus, times, div, invert, etc.
}   // End of the Fraction class declaration.
```

Figure 9.3b A Jack implementation of the Fraction abstraction.

The Fraction class illustrates several key features of object-based programming in Jack. *Fields* specify object properties (also called *member variables*). *Constructors* are subroutines that create new objects, and *methods* are subroutines that operate on the current object (referred to using the keyword this). *Functions* are class-level subroutines (also called *static methods*) that operate on no particular object. The Fraction class also demonstrates all the statement types available in the Jack language: let, do, if, while, and return. The Fraction class is of course just one example of the unlimited number of classes that can be created in Jack to support any conceivable programming objective.

Example 4: Linked list implementation: The data structure *list* is defined recursively as a value, followed by a list. The value null—the definition's base case—is also considered a list. Figure 9.4 shows a possible Jack implementation of a list of integers. This example illustrates how Jack can be used for realizing a major data structure, widely used in computer science.

```
/** Represents a list of integers. */
class List {
   field int data;      // A list consists of an int value,
   field List next;     // followed by a List
   /* Creates a list whose head is car and whose tail is cdr */
   // These identifiers are used in memory of the Lisp programming language
   constructor List new(int car, List cdr) {
      let data = car;
      let next = cdr;
      return this;
   }
   /* Accessors */
   method int getData() {return data;}
   method List getNext() {return next;}
   /* Prints the elements of this list */
   method void print() {
      // Initializes a pointer to the first element of this list
      var List current;
      let current = this;
      // Iterates through the list
      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);  // Prints a space
         let current = current.getNext();
      }
      return;
   }
   // The List class declaration continues on the top right
```

```
   /* Disposes this List */
   method void dispose() {
      // Disposes the tail of this list, recursively
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to free the memory
      //  held by this object.
      do Memory.deAlloc(this);
      return;
   }
   // More list-related methods can come here
} // End of the List class declaration.
```

```
// Client code example:
// Creates, prints, and disposes the list (2,3,5),
// which is shorthand for the list (2,(3,(5,null))).
// (This code can appear in any Jack class):
...
var List v;
let v = List.new(5,null);
let v = List.new(2,List.new(3,v));
do v.print();          // Prints 2 3 5
do v.dispose();        // Disposes the list
...
```

Figure 9.4 Linked list implementation in Jack (left and top right) and sample usage (bottom right).

The operating system: Jack programs make extensive use of the Jack operating system, which will be discussed and developed in chapter 12. For now, suffice it to say that Jack programs use the OS services abstractly, without paying attention to their underlying implementation. Jack programs can use the OS services directly—there is no need to include or import any external code.

The OS consists of eight classes, summarized in figure 9.5 and documented in appendix 6.

OS class	*Services*
`Math`	Common mathematical operations: `max(int,int)`, `sqrt(int)`, …
`String`	Represents strings and related operations: `length()`, `charAt(int)` , …
`Array`	Represents arrays and related operations: `new(int)`, `dispose()`
`Output`	Facilitates text output to the screen: `printString(String)`, `printInt(int)`, `println()`, …
`Screen`	Facilitates graphics output to the screen : `setColor(boolean)`, `drawPixel(int,int)`, `drawLine(int,int,int,int)` , …
`Keyboard`	Facilitates input from the keyboard: `readLine(String)`, `readInt(String)` , …
`Memory`	Facilitates access to the host RAM: `peek(int)`, `poke(int,int)`, `alloc(int)`, `deAlloc(Array)`
`Sys`	Facilitates execution-related services: `halt()`, `wait(int)`, …

Figure 9.5 Operating system services (summary). The complete OS API is given in appendix 6.

9.2 The Jack Language Specification

This section can be read once and then used as a technical reference, to be consulted as needed.

9.2.1 Syntactic Elements

A Jack program is a sequence of tokens, separated by an arbitrary amount of white space and comments. Tokens can be symbols, reserved words, constants, and identifiers, as listed in figure 9.6.

White space and comments	Space characters, newline characters, and comments are ignored.
	The following comment formats are supported:
	`//` Comment to end of line
	`/*` Comment until closing `*/`
	`/**` Aimed at software tools that extract API documentation. `*/`

Symbols	`()` Used for grouping arithmetic expressions and for enclosing argument-lists (in subroutine calls) and parameter-lists (in subroutine declarations) `[]` Used for array indexing `{}` Used for grouping program units and statements `,` Variable-list separator `;` Statement terminator `=` Assignment and comparison operator `.` Class membership `+ - * / & | ~ < >` Operators
Reserved words	`class`, `constructor`, `method`, `function` — Program components `int`, `boolean`, `char`, `void` — Primitive types `var`, `static`, `field` — Variable declarations `let`, `do`, `if`, `else`, `while`, `return` — Statements `true`, `false`, `null` — Constant values `this` — Object reference
Constants	• *Integer constants* are values in the range 0 to 32767. Negative integers are not constants but rather expressions consisting of a unary minus operator applied to an integer constant. The resulting valid range of values is –32768 to 32767 (the former can be obtained using the expression –32767–1). • *String constants* are enclosed within double quote (`"`) characters and may contain any character except newline or double quote. These characters are supplied by the OS functions `String.newLine()` and `String.doubleQuote()`. • *Boolean constants* are `true` and `false`. • The `null` constant signifies a null reference.
Identifiers	Identifiers are composed from arbitrarily long sequences of letters (`A-Z`, `a-z`), digits (`0-9`), and `"_"`. The first character must be a letter or `"_"`. The language is case sensitive: `x` and `X` are treated as different identifiers.

Figure 9.6 The syntax elements of the Jack language.

9.2.2 Program Structure

A Jack program is a collection of one or more classes stored in the same folder. One class must be named Main, and this class must have a function named main. The execution of a compiled Jack program always starts with the Main.main function.

The basic programming unit in Jack is a *class*. Each class *Xxx* resides in a separate file named *Xxx*.jack and is compiled separately. By convention, class names begin with an uppercase letter. The file name must be identical to the class name, including capitalization. The class declaration has the following structure:

```
class name {
    field variable declarations     // Must precede the subroutine declarations
    static variable declarations    // Must precede the subroutine declarations
    subroutine declarations         // Constructor, method and function declarations, in any order
}
```

Each class declaration specifies a name through which the class services can be globally accessed. Next comes a sequence of zero or more field declarations and zero or more static variable declarations. Next comes a sequence of one or more subroutine declarations, each defining a *method*, a *function*, or a *constructor*.

Methods are designed to operate on the current object. *Functions* are class-level *static methods* that are not associated with any particular object. *Constructors* create and return new objects of the class type. A subroutine declaration has the following structure:

```
subroutine type name (parameter-list) {
    local variable declarations
    statements
}
```

where *subroutine* is either constructor, method, or function. Each subroutine has a *name* through which it can be accessed and a *type* specifying the data type of the value returned by the subroutine. If the subroutine is designed to return no value, its type is declared void. The *parameter-list* is a comma-separated list of <*type identifier*> pairs, for example, (int x, boolean sign, Fraction g).

If the subroutine is a *method* or a *function*, its return type can be any of the primitive data types supported by the language (int, char, or boolean), any of the class types supplied by the standard class library (String or Array), or any of the types realized by other classes in the program (e.g., Fraction or List). If the subroutine is a *constructor*, it may have an arbitrary name, but its type must be the name of the class to which it belongs. A class can have 0, 1, or more constructors. By convention, one of the constructors is named new.

Following its interface specification, the subroutine declaration contains a sequence of zero or more local variable declarations (var statements) and then a sequence of one or more statements. Each subroutine must end with the statement return *expression*. In the case of a void subroutine, when there is nothing to return, the subroutine must end with the statement return (which can be viewed as a shorthand of return void, where void is a constant representing "nothing"). Constructors must terminate with the statement return this. This action returns the memory address of the newly constructed object, denoted this (Java constructors do the same, implicitly).

9.2.3 Data Types

The data type of a variable is either *primitive* (int, char, or boolean), or *ClassName*, where *ClassName* is either String, Array, or the name of a class residing in the program folder.

Primitive types: Jack features three primitive data types:

int: two's-complement 16-bit integer

char: nonnegative, 16-bit integer

boolean: true or false

Each one of the three primitive data types int, char, and boolean is represented internally as a 16-bit value. The language is weakly typed: a value of any type can be assigned to a variable of any type without casting.

Arrays: Arrays are declared using the OS class Array. Array elements are accessed using the typical arr[*i*] notation, where the index of the first element is 0. A multidimensional array may be obtained by creating an array of arrays. The array elements are not typed, and different elements in the same array may have different types. The declaration of an array creates a reference, while the array proper is constructed by executing the constructor call Array.new (*arrayLength*). For an example of working with arrays, see figure 9.2.

Object types: A Jack class that contains at least one method defines an object type. As typical in object-oriented programming, object creation is a two-step affair. Here is an example:

```
// This client code example uses the Car and Employee classes, whose code is not shown here.
// The Car class has two fields: model (a String) and licensePlate (a String).
// The Employee class has two fields: name (a String) and car (a Car).
...
// Declares a Car object and two Employee objects (three pointer variables):
var Car c;
var Employee emp1, emp2;
...
// Constructs a new car:
let c = Car.new("Aston Martin","007"); // Sets c to the base address of a memory block
                                       // containing the new car's data.
// Constructs a new employee, and assigns a car to it:
let emp1 = Employee.new("Bond",c);
...
// Creates an alias of Bond:
let emp2 = emp1; // Only the reference (address) is copied, no new object is constructed.
// We now have two Employee pointers referring to the same object.
```

Strings: Strings are instances of the OS class String, which implements strings as arrays of char values. The Jack compiler recognizes the syntax "foo" and treats it as a String object. The contents of a String object can be accessed using charAt(*index*) and other methods documented in the String class API (see appendix 6). Here is an example:

```
var String s;   // An object variable
var char c;     // A primitive variable
…
let s = "Hello World"; // Sets s to the String object "Hello World"
let c = s.charAt(6);   // Sets c to 87, the integer character code of 'W'
```

The statement `let s = "Hello World"` is equivalent to the statement `let s = String.new(11)`, followed by the eleven method calls `do s.appendChar(72)`, …, `do s.appendChar(100)`, where the argument of `appendChar` is the character's integer code. In fact, that's exactly how the compiler handles the translation of `let s = "Hello World"`. Note that the single character idiom, for example, `'H'`, is not supported by the Jack language. The only way to represent a character is to use its integer character code or a `charAt` method call. The Hack character set is documented in appendix 5.

Type conversions: Jack is weakly typed: the language specification does not define what happens when a value of one type is assigned to a variable of a different type. The decisions of whether to allow such casting operations, and how to handle them, are left to specific Jack compilers. This under-specification is intentional, allowing the construction of minimal compilers that ignore typing issues. Having said that, all Jack compilers are expected to support, and automatically perform, the following assignments.

- A character value can be assigned to an integer variable, and vice versa, according to the Jack character set specification (appendix 5). Example:

```
var char c;
let c = 33; // 'A'
// Equivalently:
var String s;
let s = "A";
let c = s.charAt(0);
```

- An integer can be assigned to a reference variable (of any object type), in which case it is interpreted as a memory address. Example:

```
var Array arr;       // Creates a pointer variable
let arr = 5000;      // Sets arr to 5000
let arr[100] = 17; // Sets the contents of memory address 5100 to 17
```

- An object variable may be assigned to an `Array` variable, and vice versa. This allows accessing the object fields as array elements, and vice versa. Example:

```
// Creates the array [2,5]:
var Array arr;
let arr = Array.new(2);
let arr[0] = 2;
let arr[1] = 5;

// Creates the Fraction 2/5:
var Fraction x;
let x = arr;   // sets x to the base address of the memory block
               // representing the array [2,5]

do Output.printInt(x.getNumerator())   // prints "2"
do x.print()                           // prints "2/5"
```

9.2.4 Variables

Jack features four kinds of variables. *Static variables* are defined at the class level and can be accessed by all the class subroutines. *Field variables*, also defined at the class level, are used to represent the properties of individual objects and can be accessed by all the class constructors and methods. *Local variables* are used by subroutines for local computations, and *parameter variables* represent the arguments that were passed to the subroutine by the caller. Local and parameter values are created just before the subroutine starts executing and are recycled when the subroutine returns. Figure 9.7 gives the details. The *scope* of a variable is the region in the program in which the variable is recognized.

Kind	Description	Declared in	Scope
Class variables	`static` *type varName*1, *varName*2, ... `;` One copy of each `static` variable exists, and this copy is shared by all the class subroutines (like *private static variables* in Java)	Class declaration	The class in which they are declared
Field variables	`field` *type varName*1, *varName*2, ... `;` Every object (instance of the class) has a private copy of the field variables (like *member variables* in Java)	Class declaration	The class in which they are declared
Local variables	`var` *type varName*1, *varName*2, ... `;` Created when the subroutine starts running and disposed when the subroutine returns.	Subroutine declaration	The subroutine in which they are declared
Parameter variables	Represent the arguments passed to the subroutine. Treated like local variables whose values are initialized by the subroutine caller.	Subroutine declaration	The subroutine in which they are declared

Figure 9.7 Variable kinds in the Jack language. Throughout the table, *subroutine* refers to either a *function*, a *method*, or a *constructor*.

Variable initialization: Static variables are not initialized, and it is up to the programmer to write code that initializes them before using them. Field variables are not initialized; it is expected that they will be initialized by the class constructor, or constructors. Local variables are not initialized, and it is up to the programmer to initialize them. Parameter variables are initialized to the values of the arguments passed by the caller.

Variable visibility: Static and field variables cannot be accessed directly from outside the class in which they are defined. They can be accessed only through accessor and mutator methods, as facilitated by the class designer.

9.2.5 Statements

The Jack language features five statements, as seen in figure 9.8.

Statement	*Syntax*	*Description*
`let`	`let` *varName* `=` *expression*`;` or: `let` *varName*`[`*expression*1`] =` *expression*2`;`	An assignment operation. The variable kind may be *static*, *local*, *field*, or *parameter*.

`if`	`if (`*expression*`) {`
	*statements*1`;`
	`} else {`
	*statements*2`;`
	`}`
		Typical *if* statement, with an optional *else* clause.
		The curly brackets are mandatory, even if *statements* is a single statement.

`while`	`while (`*expression*`) {`
	statements`;`
	`}`
		Typical *while* statement.
		The curly brackets are mandatory, even if *statements* is a single statement.

`do`	`do` *function-or-method-call*`;`	Used to call a function or a method for its effect, ignoring the returned value, if any.
`return`	`return` *expression*`;` or `return;`	Used to return a value from a subroutine. The second form must be used by `void` subroutines. Constructors must return the value `this`.

Figure 9.8 Statements in the Jack language.

9.2.6 Expressions

A Jack expression is one of the following:

- A *constant*
- A *variable name* in scope. The variable may be *static*, *field*, *local*, or *parameter*
- The this keyword, denoting the current object (cannot be used in functions)
- An *array element* using the syntax *arr*[*expression*], where *arr* is a variable name of type Array in scope
- A *subroutine call* that returns a non-void type
- An expression prefixed by one of the unary operators - or ~:
  - *expression*: arithmetic negation
  ~ *expression*: Boolean negation (bitwise for integers)
- An expression of the form *expression op expression* where *op* is one of the following binary

operators:

+ - * /: integer arithmetic operators

& |: Boolean And and Boolean Or (bitwise for integers) operators

< > =: comparison operators

- (*expression*): an expression in parentheses

Operator priority and order of evaluation: Operator priority is *not* defined by the language, except that expressions in parentheses are evaluated first. Thus the value of the expression `2 + 3 * 4` is unpredictable, whereas `2 + (3 * 4)` is guaranteed to evaluate to 14. The Jack compiler supplied in Nand to Tetris (as well as the compiler that we'll develop in chapters 10–11) evaluates expressions left to right, so the expression `2 + 3 * 4` evaluates to 20. Once again, if you wish to get the algebraically correct result, use `2 + (3 * 4)`.

The need to use parentheses for enforcing operator priority makes Jack expressions a bit cumbersome. This lack of formal operator priority is intentional, though, as it simplifies the implementation of Jack compilers. Different Jack compilers are welcome to specify an operator priority and add it to the language documentation, if so desired.

9.2.7 Subroutine Calls

A subroutine call invokes a function, a constructor, or a method for its effect, using the general syntax *subroutineName* (*exp*$_1$, *exp*$_2$, …, *exp*$_n$), where each argument *exp* is an expression. The number and type of the arguments must match those of the subroutine's parameters, as specified in the subroutine's declaration. The parentheses must appear even if the argument list is empty.

Subroutines can be called from the class in which they are defined, or from other classes, according to the following syntax rules:

Function calls / Constructor calls:

- *className*.*functionName* (*exp*$_1$, *exp*$_2$, … , *exp*$_n$)
- *className*.*constructorName* (*exp*$_1$, *exp*$_2$, … , *exp*$_n$)

The *className* must always be specified, even if the function/constructor is in the same class as the caller.

Method calls:

- *varName*.*methodName* (*exp*$_1$, *exp*$_2$, … , *exp*$_n$)

  Applies the method to the object referred to by *varName*.
- *methodName* (*exp*$_1$, *exp*$_2$, … , *exp*$_n$)

  Applies the method to the *current object*. Same as this.*methodName* (*exp*$_1$, *exp*$_2$, … , *exp*$_n$).

Here are subroutine call examples:

```
class Foo {
   …
   method void f() {
      var Bar b;          // Declares a local variable of class type Bar
      var int i;          // Declares a local variable of primitive type int
      …
      do Foo.g()          // Calls function g of the current class
      do Bar.h()          // Calls function h of class Bar
      do m()              // Calls method m of the current class, on the this object
      do b.q()            // Calls method q of class Bar, on object b
      let i = w(b.s(), Foo.t())  // Calls method w on the this object,
                                 // Calls method s of class Bar on object b,
                                 // Calls function or constructor t of class Foo.
    }
}
```

9.2.8 Object Construction and Disposal

Object construction is done in two stages. First, a reference variable (pointer to an object) is declared. To complete the object's construction (if so desired), the program must call a constructor from the object's class. Thus, a class that implements a type (e.g., Fraction) must feature at least one constructor. Jack constructors may have arbitrary names; by convention, one of them is named new.

Objects are constructed and assigned to variables using the idiom let *varName* = *className.constructorName* (*exp*$_1$, *exp*$_2$, … , *exp*$_n$), for example, let c = Circle.new (x,y,50). Constructors typically include code that initializes the fields of the new object to the argument values passed by the caller.

When an object is no longer needed, it can be disposed, to free the memory that it occupies. For example, suppose that the object that c points at is no longer needed. The object can be deallocated from memory by calling the OS function Memory.deAlloc (c). Since Jack has no garbage collection, the best-practice advice is that every class that represents an object must feature a dispose() method that properly encapsulates this de-allocation. Figures 9.3 and 9.4 give examples. To avoid memory leaks, Jack programmers are advised to dispose objects when they are no longer needed.

9.3 Writing Jack Applications

Jack is a general-purpose language that can be implemented over different hardware platforms. In Nand to Tetris we develop a *Jack compiler over the Hack platform*, and thus it

is natural to discuss Jack applications in the Hack context.

Examples: Figure 9.9 shows screenshots of four sample Jack programs. Generally speaking, the Jack/Hack platform lends itself nicely to simple interactive games like Pong, Snake, Tetris, and similar classics. Your projects/09/Square folder includes the full Jack code of a simple interactive program that allows the user to move a square image on the screen using the four keyboard arrow keys.

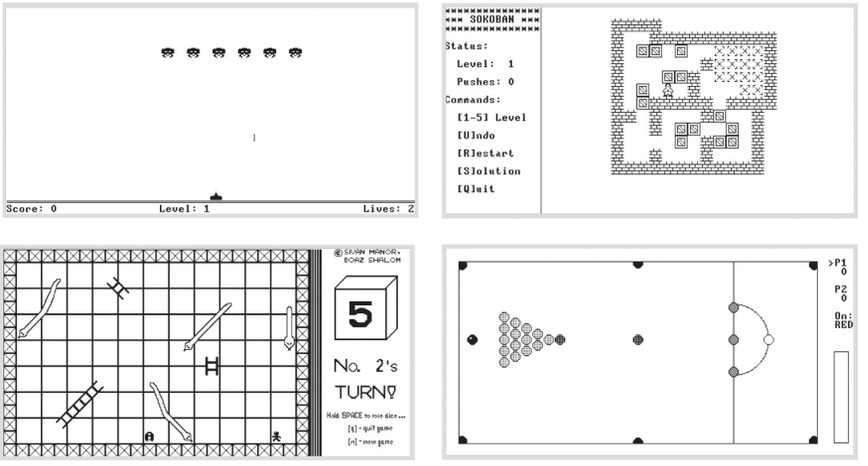

Figure 9.9 Screenshots of Jack applications running on the Hack computer.

Executing this program while reviewing its Jack source code is a good way for learning how to use Jack to write interactive graphical applications. Later in the chapter we describe how to compile and execute Jack programs using the supplied tools.

Application design and implementation: Software development should always rest on careful planning, especially when done over a spartan hardware platform like the Hack computer. First, the program designer must consider the physical limitations of the hardware and plan accordingly. To start with, the dimensions of the computer's screen limit the size of the graphical images that the program can handle. Likewise, one must consider the language's range of input/output commands and the platform's execution speed to gain a realistic expectation of what can and cannot be done.

The design process normally starts with a conceptual description of the desired program's behavior. In the case of graphical and interactive programs, this may take the form of handwritten drawings of typical screens. Next, one normally designs an object-based architecture of the program. This entails the identification of *classes*, *fields*, and *subroutines*. For example, if the program is supposed to allow the user to create square objects and move them around the screen using the keyboard's arrow keys, it will make sense to design a Square class that encapsulates these operations using methods like moveRight, moveLeft, moveUp, and moveDown, as well as a constructor subroutine for creating squares and a disposer subroutine for disposing them. In addition, it will make sense to create a SquareGame class that carries out the user interaction and a Main class that gets things started. Once the APIs of these classes

are carefully specified, one can proceed to implement, compile, and test them.

Compiling and executing Jack programs: All the .jack files comprising the program must reside in the same folder. When you apply the Jack compiler to the program folder, each source .jack file will be translated into a corresponding .vm file, stored in the same program folder.

The simplest way to execute or debug a compiled Jack program is to load the program folder into the VM emulator. The emulator will load all the VM functions in all the .vm files in the folder, one after the other. The result will be a (possibly long) stream of VM functions, listed in the VM emulator's code pane using their full *fileName.functionName* names. When you instruct the emulator to execute the program, the emulator will start executing the OS Sys.init function, which will then call the Main.main function in your Jack program.

Alternatively, you can use a VM translator (like the one built in projects 7–8) for translating the compiled VM code, as well as the eight supplied tools/OS/*.vm OS files, into a single .asm file written in the Hack machine language. The assembly code can then be executed on the supplied CPU emulator. Or, you can use an assembler (like the one built in project 6) for translating the .asm file further into a binary code .hack file. Next, you can load a Hack computer chip (like the one built in projects 1–5) into the hardware simulator or use the built-in Computer chip, load the binary code into the ROM chip, and execute it.

The operating system: Jack programs make extensive use of the language's *standard class library*, which we also refer to as the *Operating System*. In project 12 you will develop the OS class library in Jack (like Unix is written in C) and compile it using a Jack compiler. The compilation will yield eight .vm files, comprising the OS implementation. If you put these eight .vm files in your program folder, all the OS functions will become accessible to the compiled VM code, since they belong to the same code base (by virtue of belonging to the same folder).

Presently, though, there is no need to worry about the OS implementation. The supplied VM emulator, which is a Java program, features a built-in Java implementation of the Jack OS. When the VM code loaded into the emulator calls an OS function, say Math.sqrt, one of two things happens. If the OS function is found in the loaded code base, the VM emulator executes it, just like executing any other VM function. If the OS function is not found in the loaded code base, the emulator executes its built-in implementation.

9.4 Project

Unlike the other projects in this book, this one does not require building a hardware of software module. Rather, you have to pick some application of your choice and build it in Jack over the Hack platform.

Objective: The "hidden agenda" of this project is to get acquainted with the Jack language,

for two purposes: writing the Jack compiler in projects 10 and 11, and writing the Jack operating system in project 12.

Contract: Adopt or invent an application idea like a simple computer game or some interactive program. Then design and build the application.

Resources: You will need the supplied tools/JackCompiler for translating your program into a set of .vm files, and the supplied tools/VMEmulator for running and testing the compiled code.

Compiling and Running a Jack Program

0. Create a folder for your program. Let's call it the *program folder*.
1. Write your Jack program—a set of one or more Jack classes—each stored in a separate *ClassName*.jack text file. Put all these .jack files in the program folder.
2. Compile the program folder using the supplied Jack compiler. This will cause the compiler to translate all the .jack classes found in the folder into corresponding .vm files. If a compilation error is reported, debug the program and recompile until no error messages are issued.
3. At this point the program folder should contain your source .jack files along with the compiled .vm files. To test the compiled program, load the program folder into the supplied VM emulator, and run the loaded code. In case of run-time errors or undesired program behavior, fix the relevant file and go back to step 2.

Program examples: Your nand2tetris/project/09 folder includes the source code of a complete, three-class interactive Jack program (Square). It also includes the source code of the Jack programs discussed in this chapter.

Bitmap editor: If you develop a program that needs high-speed graphics, it is best to design *sprites* for rendering the key graphical elements of the program. For example, the output of the Sokoban application depicted in figure 9.9 consists of several repeating sprites. If you wish to design such sprites and write them directly into the screen memory map (bypassing the services of the OS Screen class, which may be too slow), you will find the projects/09/BitmapEditor tool useful.

A web-based version of project 9 is available at www.nand2tetris.org.

9.5 Perspective

Jack is an *object-based* language, meaning that it supports objects and classes but not inheritance. In this respect it is positioned somewhere between procedural languages like Pascal or C and object-oriented languages like Java or C++. Jack is certainly more simple-

minded than any of these industrial strength programming languages. However, its basic syntax and semantics are similar to those of modern languages.

Some features of the Jack language leave much to be desired. For example, its primitive type system is, well, rather primitive. Moreover, it is a weakly typed language, meaning that type conformity in assignments and operations is not strictly enforced. Also, you may wonder why the Jack syntax includes clunky keywords like `do` and `let`, why every subroutine must end with a `return` statement, why the language does not enforce operator priority, and so on—you may add your favorite complaint to the list.

All these somewhat tedious idiosyncrasies were introduced into Jack with one purpose: allowing the development of simple and minimal Jack compilers, as we will do in the next two chapters. For example, the parsing of a statement (in any language) is significantly easier if the first token of the statement reveals which statement we're in. That's why Jack uses a `let` keyword for prefixing assignment statements. Thus, although Jack's simplicity may be a nuisance when writing Jack *applications*, you'll be grateful for this simplicity when writing the Jack *compiler*, as we'll do in the next two chapters.

Most modern languages are deployed with a set of *standard classes*, and so is Jack. Taken together, these classes can be viewed as a portable, language-oriented operating system. Yet unlike the standard libraries of industrial-strength languages, which feature numerous classes, the Jack OS provides a minimal set of services, which is nonetheless sufficient for developing simple interactive applications.

Clearly, it would be nice to extend the Jack OS to provide concurrency for supporting multi-threading, a file system for permanent storage, sockets for communications, and so on. Although all these services can be added to the OS, readers will perhaps want to hone their programming skills elsewhere. After all, we don't expect Jack to be part of your life beyond Nand to Tetris. Therefore, it is best to view the Jack/Hack platform as a given environment and make the best out of it. That's precisely what programmers do when they write software for embedded devices and dedicated processors that operate in restricted environments. Instead of viewing the constraints imposed by the host platform as a problem, professionals view it as an opportunity to display their resourcefulness and ingenuity. That's what you are expected to do in project 9.