9 High-Level Language

High thoughts need a high language.
—Aristophanes (448–380 BC)

All the hardware and software systems presented so far in the book were low-level, meaning that humans are not expected to interact with them directly. In this chapter we present a high-level language, called Jack, designed to enable human programmers write high-level programs. Jack is a simple object-based language. It has the basic features and flavor of modern languages like Java and C#, with a much simpler syntax and no support for inheritance. In spite of this simplicity, Jack is a general-purpose language that can be used to create numerous applications. In particular, it lends itself nicely to simple interactive games like Snake, Tetris, and Pong—a program whose complete Jack code is included in the book's software suite.

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, written in Jack. 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. Instead, our hidden agenda is to prepare you to develop the compiler and operating system that lie ahead.

If you have any experience with a modern object-oriented programming language, you will immediately feel at home with Jack. Therefore, the Background section starts the chapter with some typical programming examples, and the Specification section proceeds with a full functional description of the language and its standard library. The Implementation section gives some screen shots of typical Jack applications and offers general guidelines on how to write similar programs over the Hack platform. The final Project section provides additional details about compiling and debugging Jack programs.

All the programs shown in the chapter can be compiled by the Jack compiler supplied with the book. The resulting VM code can then run as is on the supplied VM emulator. Alternatively, one can further translate the compiled VM code into binary code, using the VM translator and the assembler built in chapters 7–8 and 6, respectively. The resulting machine code can then be executed as is on the hardware platform that we built in chapters 1–5.

It's important to reiterate that in and by itself, Jack is a rather uninteresting and simple-minded language. However, this simplicity has a purpose. First, you can learn (and unlearn) Jack very quickly—in about an hour. Second, the Jack language was carefully planned to lend itself nicely to simple compilation techniques. As a result, one 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 engineering principles underlying modern languages like Java and C#. Rather than taking the compilers and run-time environments of these languages for granted, we will build a Jack compiler and a run-time environment ourselves, beginning in the next chapter. For now, let's take Jack out of the box.

9.1 Background

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

9.1.1 Example 1: Hello World

When we tell the Jack run-time environment to run a given 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.

Jack is equipped with a *standard library* whose complete API is given in section 9.2.7. This library extends the basic language with various abstractions and services such as arrays, strings, mathematical functions, memory management, and input/output functions. Two such functions are invoked by the program in figure

```
/** Hello World program. */
class Main {
  function void main() {
    /* Prints some text using the standard library. */
    do Output.printString("Hello World");
    do Output.println();    // New line
    return;
  }
}
```

Figure 9.1 Hello World.

9.1, effecting the "`Hello world`" printout. The program also demonstrates the three comment formats supported by Jack.

9.1.2 Example 2: Procedural Programming and Array Handling

Jack is equipped with typical language constructs for procedural programming. It also includes basic commands for declaring and manipulating arrays. Figure 9.2 illustrates both of these features, in the context of inputting and computing the average of a series of numbers.

Jack programs declare and construct arrays using the built-in `Array` class, which is part of the standard Jack library. Note that Jack arrays are not typed and can include anything—integers, objects, and so forth.

9.1.3 Example 3: Abstract Data Types

Every programming language has a fixed set of primitive data types, of which Jack supports three: `int`, `char`, and `boolean`. Programmers can extend this basic repertoire by creating new classes that represent abstract data types, as needed. For example, suppose we wish to endow Jack with the ability to handle *rational numbers*, namely, objects of the form n/m where n and m are integers. This can be done by creating a stand-alone class, designed to provide a fraction abstraction for Jack programs. Let us call this class `Fraction`.

Defining a Class Interface A reasonable way to get started is to specify the set of properties and services expected from a fraction abstraction. One such *Application Program Interface* (API), is given in figure 9.3a.

```
/** Computes the average of a sequence of integers. */
class Main {
  function void main() {
    var Array a;
    var int length;
    var int i, sum;

    let length = Keyboard.readInt("How many numbers? ");
    let a = Array.new(length); // Constructs the array
    let i = 0;

    while (i < length) {
      let a[i] = Keyboard.readInt("Enter the next 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 Procedural programming and array handling.

In Jack, operations on the current object (referred to as `this`) are represented by *methods*, whereas class-level operations (equivalent to *static methods* in Java) are represented by *functions*. Operations that create new objects are called *constructors*.

Using Classes APIs mean different things to different people. If you are the programmer who has to *implement* the fraction class, you can view its API as a contract that must be implemented, one way or another. Alternatively, if you are a programmer who needs to *use* fractions in your work, you can view the API as a documentation of a fraction server, designed to generate fraction objects and supply fraction-related operations. Taking this latter view, consider the Jack code listed in figure 9.3b.

Figure 9.3b illustrates an important software engineering principle: Users of any given abstraction don't have to know anything about its underlying *implementation*.

```
// A Fraction is an object representation of n/m where n and m are integers.
field int numerator, denominator        // Fraction object properties
constructor Fraction new(int a, int b)  // Returns a new Fraction object
method int getNumerator()               // Returns the numerator of this
                                        // fraction
method int getDenominator()             // Returns the denominator of this
                                        // fraction
method Fraction plus(Fraction other)    // Returns the sum of this fraction
                                        // and another fraction, as a
                                        // fraction
method void print()                     // Prints this fraction in the
                                        // format "numerator/denominator"
// Additional fraction-related services are specified here, as needed.
```

Figure 9.3a Fraction class API.

```
// Computes the sum of 2/3 and 1/5.
class Main {
  function void main() {
    var Fraction a, b, c;
    let a = Fraction.new(2,3);
    let b = Fraction.new(1,5);
    let c = a.plus(b);  // Compute c = a + b
    do c.print();  // Should print the text "13/15"
    return;
  }
}
```

Figure 9.3b Using the Fraction abstraction.

```
/** Provides the Fraction type and related services. */
class Fraction {
  field int numerator, denominator;

  /** Constructs a new (and reduced) fraction from given
   *  numerator and denominator. */
  constructor Fraction new(int a, int b) {
    let numerator = a;  let denominator = b;
    do reduce();  // If a/b is not reduced, reduce it
    return this;
  }

  /** 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 denominator of a and b. */
  function int gcd(int a, int b){
    var int r;
    while (~(b = 0)) {            // Apply Euclid's algorithm.
      let r = a - (b * (a / b));  // r=remainder of a/b
      let a = b; let b = r; }
    return a;
  }

  /** Accessors. */
  method int getNumerator() { return numerator; }
  method int getDenominator() { return denominator; }
```

Figure 9.3c A possible Fraction class implementation.

```
  /** Returns the sum of this fraction and another one.
  method Fraction plus(Fraction other){
    var int sum;
    let sum = (numerator * other.getDenominator()) +
              (other.getNumerator() * denominator());
    return Fraction.new(sum, denominator *
           other.getDenominator());
  }

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

  /** Prints this fraction. */
  method void print() {
    do Output.printInt(numerator);
    do Output.printString("/");
    do Output.printInt(denominator);
    return;
  }
} // Fraction class
```

Figure 9.3c (continued)

Rather, they can be given access only to the abstraction's *interface*, or class API, and then use it as a black box server of abstraction-related operations.

Implementing the Class We now turn to the other player in our story—the programmer who has to actually implement the fraction abstraction. A possible Jack implementation is given in figure 9.3c.

Figure 9.3c illustrates the typical Jack program structure: *classes*, *methods*, *constructors*, and *functions*. It also demonstrates all the statement types available in the language: `let`, `do`, `if`, `while`, and `return`.

9.1.4 Example 4: Linked List Implementation

A *linked list* (or simply *list*) is a chain of objects, each consisting of a data element and a reference (pointer) to the rest of the list. Figure 9.4 shows a possible Jack class implementation of the linked list abstraction. The purpose of this example is to illustrate typical object handling in the Jack language.

```
/** The List class provides a linked list abstraction. */
class List {
  field int data;
  field List next;

  /* Creates a new List object. */
  constructor List new(int car, List cdr) {
    let data = car;
    let next = cdr;
    return this;
  }

  /* Disposes this List by recursively disposing its tail. */
  method void dispose() {
    if (~(next = null)) {
       do next.dispose();
    }
    // Use an OS routine to recycle the memory held by this
    // object.
    do Memory.deAlloc(this);
    return;
  }

  // More List-related methods come here

 } // class List
```

```
/* Creates a list holding the numbers (2,3,5).
   (this code can appear in any class). */
function void create235() {
  var List v;
  let v = List.new(5,null);
  let v = List.new(2,List.new(3,v));
  ... // Does something with the list
  do v.dispose();
  return;
}
```

Figure 9.4 Object handling in a linked list context.

9.2 The Jack Language Specification

We now turn to a formal and complete description of the Jack language, organized by its syntactic elements, program structure, variables, expressions, and statements. This language specification should be viewed 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, which are ignored. Tokens can be *symbols*, *reserved words*, *constants*, and *identifiers*, as listed in figure 9.5.

9.2.2 Program Structure

The basic programming unit in Jack is a *class*. Each class resides in a separate file and can be compiled separately. Class declarations have the following format:

```
class name {
    Field and static variable declarations  // Must precede subroutine declarations.
    Subroutine declarations  // Constructor, method and function declarations.
}
```

Each class declaration specifies a name through which the class can be globally accessed. Next comes a sequence of zero or more *field* and *static variable* declarations. Then comes a sequence of one or more *subroutine* declarations, each defining a *method*, a *function*, or a *constructor*. Methods "belong to" objects and provide their functionality, while functions "belong to" the class in general and are not associated with a particular object (similar to Java's *static methods*). A constructor "belongs to" the class and, when called, generates object instances of this class.

All subroutine declarations have the following format:

```
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* describing the value returned by

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 */
/** API documentation comment */
```

Symbols

`( )`	Used for grouping arithmetic expressions and for enclosing parameter-lists and argument-lists
`[ ]`	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 must be positive and in standard decimal notation, e.g., `1984`. Negative integers like `-13` are not constants but rather expressions consisting of a unary minus operator applied to an integer constant.

String constants are enclosed within two quote (“) characters and may contain any characters except *newline* or *double-quote*. (These characters are supplied by the functions `String.newLine()` and `String.doubleQuote()` from the standard library.)

Boolean constants can be `true` or `false`.

The constant `null` 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. Thus `x` and `X` are treated as different identifiers.

Figure 9.5 Jack syntactic elements.

the subroutine. If the subroutine returns no value, the type is declared `void`; otherwise, it can be any of the primitive data types supported by the language, or any of the class types supplied by the standard library, or any of the class types supplied by other classes in the application. Constructors may have arbitrary names, but they must return an object of the class type. Therefore the type of a constructor must always be the name of the class to which it belongs.

Following its header specification, the subroutine declaration contains a sequence of zero or more local variable declarations, then a sequence of zero or more statements.

As in Java, a *Jack program* is a collection of one or more classes. One class must be named `Main`, and this class must include at least one function named `main`. When instructed to execute a Jack program that resides in some directory, the Jack runtime environment will automatically start running the `Main.main` function.

9.2.3 Variables

Variables in Jack must be explicitly declared before they are used. There are four kinds of variables: *field*, *static*, *local*, and *parameter* variables, each with its associated scope. Variables must be typed.

Data Types Each variable can assume either a *primitive* data type (`int`, `char`, `boolean`), as predefined in the Jack language specification, or an *object* type, which is the name of a class. The class that implements this type can be either part of the Jack standard library (e.g., `String` or `Array`), or it may be any other class residing in the program directory.

Primitive Types Jack features three primitive data types:

- `int:` 16-bit 2's complement
- `boolean:` *false* and *true*
- `char:` unicode character

Variables of primitive types are allocated to memory when they are declared. For example, the declarations `var int age; var boolean gender;` cause the compiler to create the variables `age` and `gender` and to allocate memory space to them.

Object Types Every class defines an object type. As in Java, the declaration of an object variable only causes the creation of a reference variable (pointer). Memory

```
// This code assumes the existence of Car and Employee classes.
// Car objects have model and licensePlate fields.
// Employee objects have name and Car fields.
var Employee e, f;  // Creates variables e, f that contain null references
var Car c;          // Creates a variable c that contains a null reference
...
let c = Car.new("Jaguar","007")    // Constructs a new Car object
let e = Employee.new("Bond",c)     // Constructs a new Employee object
// At this point c and e hold the base addresses of the memory segments
// allocated to the two objects.
let f = e;  // Only the reference is copied - no new object is constructed.
```

Figure 9.6 Object types (example).

for storing the object itself is allocated later, if and when the programmer actually *constructs* the object by calling a constructor. Figure 9.6 gives an example.

The Jack standard library provides two built-in object types (classes) that play a role in the language syntax: `Array` and `String`.

Arrays Arrays are declared using a built-in class called `Array`. Arrays are one-dimensional and the first index is always 0 (multi-dimensional arrays may be obtained as arrays of arrays). Array entries do not have a declared type, and different entries in the same array may have different types. The declaration of an array only creates a reference, while the actual construction of the array is done by calling the `Array.new(length)` constructor. Access to array elements is done using the `a[j]` notation. Figure 9.2 illustrates working with arrays.

Strings Strings are declared using a built-in class called `String`. The Jack compiler recognizes the syntax `"xxx"` and treats it as the contents of some `String` object. The contents of `String` objects can be accessed and modified using the methods of the `String` class, as documented in its API. Example:

```
var String s;
var char c;
...
let s = "Hello World";
let c = s.charAt(6);      // "W"
```

Type Conversions The Jack language is weakly typed. The language specification does not define the results of attempted assignment or conversion from one type to another, and different Jack compilers may allow or forbid them. (This underspecification is intentional, allowing the construction of minimal Jack compilers that ignore typing issues.)

Having said that, all Jack compilers are expected to allow, and automatically perform, the following assignments:

- Characters and integers are automatically converted into each other as needed, according to the Unicode specification. Example:

```
var char c;  var String s;
let c = 33;  // 'A'
// Equivalently:
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 treated as an address in memory. Example:

```
var Array a;
let a = 5000;
let a[100] = 77; // Memory address 5100 is set to 77
```

- An object variable (whose type is a class name) may be converted into an `Array` variable, and vice versa. The conversion allows accessing the object fields as array entries, and vice versa. Example:

```
// Assume that class Complex has two int fields: re and im.
var Complex c; var Array a;
let a = Array.new(2);
let a[0] = 7; let a[1] = 8;
let c = a; // c==Complex(7,8)
```

Variable Kinds and Scope Jack features four kinds of variables. *Static variables* are defined at the class level and are shared by all the objects derived from the class. For example, a `BankAccount` class may have a `totalBalance` static variable holding the sum of balances of all the bank accounts, each account being an object derived from the `BankAccount` class. *Field variables* are used to define the properties of individual objects of the class, for example, account `owner` and `balance`. *Local variables*, used by subroutines, exist only as long as the subroutine is running, and

Variable kind	Definition/Description	Declared in	Scope
Static variables	**`static`** *type name1*, *name2*, **`...;`** Only one copy of each static variable exists, and this copy is shared by all the object instances of the class (like *private static variables* in Java)	Class declaration.	The class in which they are declared.
Field variables	**`field`** *type name1*, *name2*, **`...;`** Every object instance of the class has a private copy of the field variables (like *private object variables* in Java)	Class declaration.	The class in which they are declared, except for functions.
Local variables	**`var`** *type name1*, *name2*, **`...;`** Local variables are allocated on the stack when the subroutine is called and freed when it returns (like *local variables* in Java)	Subroutine declaration.	The subroutine in which they are declared.
Parameter variables	*type name1*, *name2*, **`...`** Used to specify inputs of subroutines, for example: `function void drive (Car c, int miles)`	Appear in parameter lists as part of subroutine declarations.	The subroutine in which they are declared.

Figure 9.7 Variable kinds in the Jack language (throughout the table, *subroutine* is either a *function*, a *method*, or a *constructor*).

parameter variables are used to pass arguments to subroutines. For example, our `BankAccount` class may include the method signature `method void transfer-(BankAccount from, int sum)`, declaring the two parameters `from` and `sum`. Thus, if `joeAccount` and `janeAccount` were two variables of type `BankAccount`, the command `joeAccount.transfer(janeAccount,100)` will effect a transfer of 100 from Jane to Joe.

Figure 9.7 gives a formal description of all the variable kinds supported by the Jack language. The *scope* of a variable is the region in the program in which the variable name is recognized.

Statement	Syntax	Description
`let`	**`let`** *variable* `=` *expression* `;` or **`let`** *variable* `[` *expression* `] =` *expression* `;`	An assignment operation (where *variable* is either single-valued or an array). The variable kind may be *static*, *local*, *field*, or *parameter*.
`if`	**`if`** `(` *expression* `) {` *statements* `}` **`else`** `{` *statements* `}`	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.
`return`	**`Return`** *expression* `;` or **`return;`**	Used to return a value from a subroutine. The second form must be used by functions and methods that return a void value. Constructors must return the expression `this`.

Figure 9.8 Jack statements.

9.2.4 Statements

The Jack language features five generic statements. They are defined and described in figure 9.8.

9.2.5 Expressions

Jack expressions are defined recursively according to the rules given in figure 9.9.

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 *name*[*expression*], where *name* 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 (bit-wise for integers);
- An expression of the form *expression operator expression* where *operator* is one of the following binary operators:
  `+ - * /` Integer arithmetic operators;
  `& |` Boolean And and Boolean Or (bit-wise for integers) operators;
  `< > =` Comparison operators;
- `(`*expression*`)`: An expression in parentheses.

Figure 9.9 Jack expressions.

Operator Priority and Order of Evaluation Operator priority is *not* defined by the language, except that expressions in parentheses are evaluated first. Thus an expression like `2+3*4` may yield either 20 or 14, whereas `2+(3*4)` is guaranteed to yield 14. The need to use parentheses in such expressions makes Jack programming a bit cumbersome. However, the lack of formal operator priority is intentional, since it simplifies the writing of Jack compilers. Of course, different language implementations (compilers) can specify an operator priority and add it to the language documentation, if so desired.

9.2.6 Subroutine Calls

Subroutine calls invoke methods, functions, and constructors for their effect, using the general syntax *subroutineName*(*argument-list*). The number and type of the arguments must match those of the subroutine's parameters, as defined in its declaration. The parentheses must appear even if the argument list is empty. Each argument may be an expression of unlimited complexity. For example, the `Math` class, which

```
class Foo {
  // Some subroutine declarations - code omitted
  ...
  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 g(5,7)          // Calls method g of class Foo (on this object)
    do Foo.p(2)        // Calls function p of class Foo
    do Bar.h(3)        // Calls function h of class Bar
    let b = Bar.r(4);  // Calls constructor or function r of class Bar
    do b.q()           // Calls method q of class Bar (on object b)
    Let i = w(b.s(3),  Foo.t())  // Calls method w on this object,
                                 // method s on object b and function
                                 // or constructor t of class Foo
    ...
  }
}
```

Figure 9.10 Subroutine call examples.

is part of Jack's standard library, contains a square root function whose declaration is `function int sqrt(int n)`. Such a function can be invoked using calls like `Math.sqrt(17)`, or `Math.sqrt((a * Math.sqrt(c - 17) + 3)`, and so on.

Within a class, methods are called using the syntax *methodName*(*argument-list*), while functions and constructors must be called using their full names, namely, *className.subroutineName*(*argument-list*). Outside a class, the class functions and constructors are also called using their full names, while methods are called using the syntax *varName.methodName*(*argument-list*), where *varName* is a previously defined object variable. Figure 9.10 gives some examples.

Object Construction and Disposal Object construction is a two-stage affair. When a program declares a variable of some object type, only a reference (pointer) variable is created and allocated memory. 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` from figure 9.3c) must contain at least one constructor. Constructors may have arbitrary names, but it is customary to

call one of them `new`. Constructors are called just like any other class function using the format:

`let` *varName* = *className.constructorName*(*parameter-list*);

For example, `let c = Circle.new(x,y,50)` where `x`, `y`, and 50 are the screen location of the circle's center and its radius. When a constructor is called, the compiler requests the operating system to allocate enough memory space to hold the new object in memory. The OS returns the base address of the allocated memory segment, and the compiler assigns it to `this` (in the circle example, the value of `this` is assigned to `c`). Next, the constructed object is typically initialized to some valid state, effected by the Jack commands found in the constructor's body.

When an object is no longer needed in a program, it can be disposed. In particular, objects can be de-allocated from memory and their space reclaimed using the `Memory.deAlloc(object)` function from the standard library. Convention calls for every class to contain a `dispose()` method that properly encapsulates this de-allocation. For example, see figure 9.4.

9.2.7 The Jack Standard Library

The Jack language comes with a collection of built-in classes that extend the language's capabilities. This standard library, which can also be viewed as a basic operating system, must be provided in every Jack language implementation. The standard library includes the following classes, all implemented in Jack:

- *Math:* provides basic mathematical operations;
- *String:* implements the `String` type and string-related operations;
- *Array:* implements the `Array` type and array-related operations;
- *Output:* handles text output to the screen;
- *Screen:* handles graphic output to the screen;
- *Keyboard:* handles user input from the keyboard;
- *Memory:* handles memory operations;
- *Sys:* provides some execution-related services.

Math This class enables various mathematical operations.

- function void **`init`**(): for internal use only.
- function int **`abs`**(int x): returns the absolute value of x.

- function int **multiply**(int x, int y): returns the product of x and y.
- function int **divide**(int x, int y): returns the integer part of x/y.
- 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 implements the `String` data type and various string-related operations.

- constructor String **new**(int maxLength): constructs a new empty string (of length zero) that can contain at most maxLength characters;
- method void **dispose**(): disposes this string;
- method int **length**(): returns the length of this string;
- method char **charAt**(int j): returns the character at location j of this string;
- method void **setCharAt**(int j, char c): sets the j-th element of this string to c;
- method String **appendChar**(char c): appends c to this string and returns this string;
- method void **eraseLastChar**(): erases the last character from this string;
- method int **intValue**(): returns the integer value of this string (or of the string prefix until a non-digit character is detected);
- method void **setInt**(int j): sets this string to hold a representation of j;
- function char **backSpace**(): returns the backspace character;
- function char **doubleQuote**(): returns the double quote (“) character;
- function char **newLine**(): returns the newline character.

Array This class enables the construction and disposal of arrays.

- function Array **new**(int size): constructs a new array of the given size;
- method void **dispose**(): disposes this array.

Output This class allows writing text on the screen.

- function void **init**(): for internal use only;
- function void **moveCursor**(int i, int j): moves the cursor to the j-th column of the i-th row, and erases the character displayed there;

- function void **printChar**(char c): prints c at the cursor location and advances the cursor one column forward;
- function void **printString**(String s): prints s starting at the cursor location and advances the cursor appropriately;
- function void **printInt**(int i): prints i 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 allows drawing graphics on the screen. Column indices start at 0 and are left-to-right. Row indices start at 0 and are top-to-bottom. The screen size is hardware-dependant (in the Hack platform: 256 rows by 512 columns).

- function void **init**(): for internal use only;
- function void **clearScreen**(): erases the entire screen;
- function void **setColor**(boolean b): sets a color (white = false, black = true) to be used for all further `drawXXX` commands;
- function void **drawPixel**(int x, int y): draws the (x,y) pixel;
- function void **drawLine**(int x1, int y1, int x2, int y2): draws a line from pixel (x1,y1) to pixel (x2,y2);
- 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);
- function void **drawCircle**(int x, int y, int r): draws a filled circle of radius r <= 181 around (x,y).

Keyboard This class allows reading inputs from a standard keyboard.

- function void **init**(): for internal use only;
- function char **keyPressed**(): returns the character of the currently pressed key on the keyboard; if no key is currently pressed, returns 0;
- function char **readChar**(): waits until a key is pressed on the keyboard and released, then echoes the key to the screen and returns the character of the pressed key;
- function String **readLine**(String message): prints the message on the screen, reads the line (text until a newline character is detected) from the keyboard, echoes the line to the screen, and returns its value. This function also handles user backspaces;

- function int **readInt**(String message): prints the message on the screen, reads the line (text until a newline character is detected) from the keyboard, echoes the line to the screen, and returns its integer value (until the first nondigit character in the line is detected). This function also handles user backspaces.

Memory This class allows direct access to the main memory of the host platform.

- function void **init**(): for internal use only;
- function int **peek**(int address): returns the value of the main memory at this address;
- function void **poke**(int address, int value): sets the contents of the main memory at this address to value;
- function Array **alloc**(int size): finds and allocates from the heap a memory block of the specified size and returns a reference to its base address;
- function void **deAlloc**(Array o): De-allocates the given object and frees its memory space.

Sys This class supports some execution-related services.

- function void **init**(): calls the `init` functions of the other OS classes and then calls the `Main.main()` function. For internal use only;
- function void **halt**(): halts the program execution;
- function void **error**(int errorCode): prints the error code on the screen and halts;
- function void **wait**(int duration): waits approximately duration milliseconds and returns.

9.3 Writing Jack Applications

Jack is a general-purpose programming language that can be implemented over different hardware platforms. In the next two chapters we will develop a *Jack compiler* that ultimately generates binary Hack code, and thus it is natural to discuss Jack applications in the Hack context. This section illustrates briefly three such applications and provides general guidelines about application development on the Jack-Hack platform.

Examples Four sample applications are illustrated in figure 9.11. The *Pong* game, whose Jack code is supplied with the book, provides a good illustration of Jack programming over the Hack platform. The Pong code is not trivial, requiring

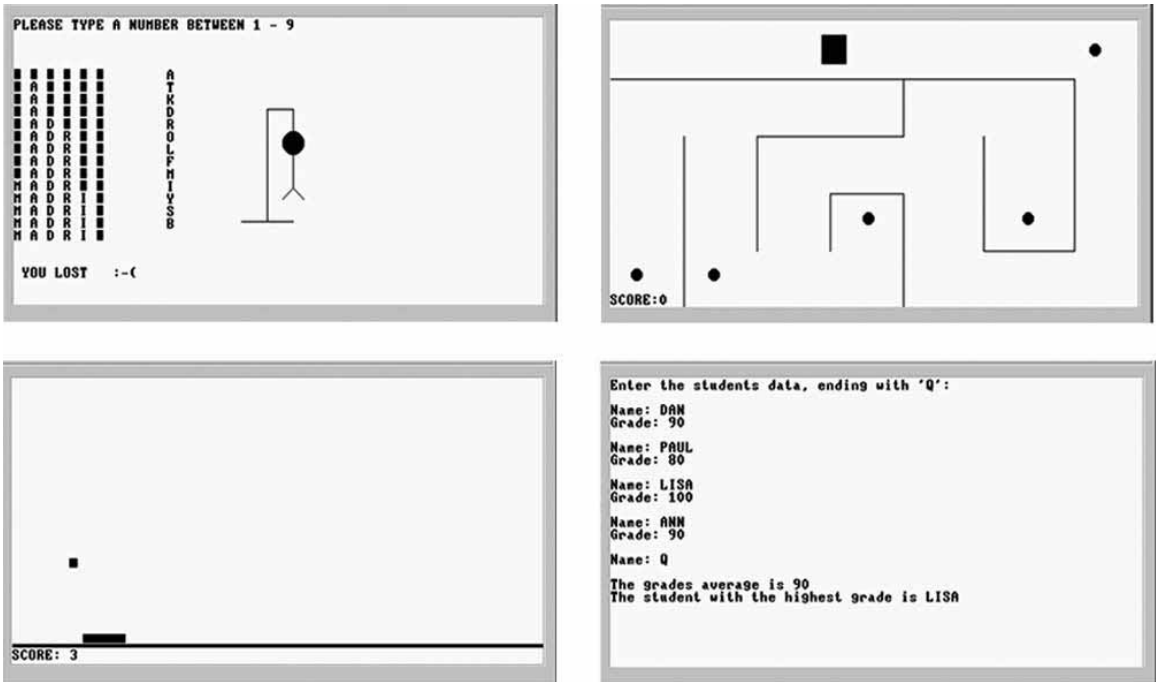

Figure 9.11 Screen shots of sample Jack applications, running on the Hack computer. Hangman, Maze, Pong, and a simple data processing program.

several hundred lines of Jack code organized in several classes. Further, the program has to carry out some nontrivial mathematical calculations in order to compute the direction of the ball's movements. The program must also animate the movement of graphical objects on the screen, requiring extensive use of the language's graphics drawing services. And, in order to do all of the above quickly, the program must be efficient, meaning that it has to do as few real-time calculations and screen drawing operations as possible.

Application Design and Implementation The development of Jack applications over a hardware platform like Hack requires careful planning (as always). First, the application designer must consider the physical limitations of the hardware, and plan accordingly. For example, 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.

As usual, the design process normally starts with a conceptual description of the application's behavior. In the case of graphical and interactive programs, this may take the form of hand-written drawings of typical screens. In simple applications, one can proceed to implementation using procedural programming. In more complex tasks, it is advisable to first create an object-based design of the application. This entails the identification of *classes*, *fields*, and *subroutines*, possibly leading to the creation of some API document (e.g., figure 9.3a).

Next, one can proceed to implement the design in Jack and compile the class files using a Jack compiler. The testing and debugging of the code generated by the compiler depend on the details of the target platform. In the Hack platform supplied with the book, testing and debugging are normally done using the supplied VM emulator. Alternatively, one can translate the Jack program all the way to binary code and run it directly on the Hack hardware, or on the CPU emulator supplied with the book.

The Jack OS Jack programs make an extensive use of the various abstractions and services supplied by the language's standard library, also called the Jack OS. This OS is itself implemented in Jack, and thus its executable version is a set of compiled `.vm` files—just like the user program (following compilation). Therefore, before running any Jack program, you must first copy into the program directory the `.vm` files comprising the Jack OS (supplied with the book). The chain of command is as follows: The computer is programmed to first run the `Sys.init`. This OS function, in turn, is programmed to start running your `Main.main` function. This function will then call various subroutines from both the user program and from the OS, and so on.

Although the standard library of the Jack language can be extended, readers will perhaps want to hone their programming skills elsewhere. After all, we don't expect Jack to be part of your life beyond this book. 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 constrains imposed by the host platform as a problem, professionals view it as an opportunity to display their resourcefulness and ingenuity. That's why some of the best programmers in the trade were first trained on primitive computers.

9.4 Perspective

Jack is an "object-based" language, meaning that it supports objects and classes, but not inheritance. In this respect it is located somewhere between procedural languages

like Pascal or C and object-oriented languages like Java or C++. Jack is certainly more "clunky" than any of these industrial-strength programming languages. However, its basic syntax and semantics are not very different from 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, one may wonder why the Jack syntax includes keywords like `do` and `let`, why curly brackets must be used even in single statement blocks, and why the language does not enforce a formal operator priority.

Well, all these deviations from normal programming languages were introduced into Jack with one purpose: to allow the development of elegant and simple Jack compilers, as we will do in the next two chapters. For example, when parsing a statement (in any language), it is much easier to handle the code if the first token of the statement indicates which statement we're in. That's why the Jack syntax includes the `do` and `let` keywords, and so on. Thus, although Jack's simplicity may be a nuisance when writing a *Jack application*, you will probably be quite grateful for it while writing the *Jack compiler* in the next two chapters.

Most modern languages are deployed with *standard libraries*, and so is Jack. As in Java and C#, this library can also be viewed as an interface to a simple and portable operating system. In the Jack-Hack platform, the services supplied by this OS are extremely minimal. They include no concurrency to support multi-threading or multi-processing, no file system to support permanent storage, and no communication. At the same time, the Jack OS provides some classical OS services like graphic and textual I/O (in very basic forms), standard implementation of strings, and standard memory allocation and de-allocation. Additionally, the Jack OS implements various mathematical functions, including multiplication and division, normally implemented in hardware. We return to these issues in chapter 12, where we will build this simple operating system as the last module in our computer system.

9.5 Project

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, for example, a simple computer game or some other interactive program. Then design and build the application.

Resources You will need three tools: the Jack compiler, to translate your program into a set of `.vm` files, the VM emulator, to run and test your translated program, and the Jack Operating System.

The Jack OS The Jack Operating System is available as a set of `.vm` files. These files constitute an implementation of the standard library of the Jack programming language. In order for any Jack program to execute properly, the compiled `.vm` files of the program must reside in a directory that also contains all the `.vm` files of the Jack OS. When an OS-oriented error is detected by the Jack OS, it displays a numeric error code (rather than text, which wastes precious memory space). A list of all the currently supported error codes and their textual descriptions can be found in the file `projects/09/OSerrors.txt`.

Compiling and Running a Jack Program

0. Each program must be stored in a separate directory, say `Xxx`. Start by creating this directory, then copy all the files from `tools/OS` into it.

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 `Xxx` directory.

2. Compile your program using the supplied Jack compiler. This is best done by applying the compiler to the name of the program directory (`Xxx`). This will cause the compiler to translate all the `.jack` classes found in the directory into corresponding `.vm` files. If a compilation error is reported, debug the program and recompile `Xxx` until no error messages are issued.

3. At this point the program directory should contain three sets of files: (i) your source `.jack` files, (ii) the compiled `.vm` files, one for each of your `.jack` class files, and (iii) additional `.vm` files, comprising the supplied Jack OS. To test the compiled program, invoke the VM emulator and load the entire `Xxx` program directory. Then run the program. In case of run-time errors or undesired program behavior, fix the program and go to stage 2.

A Sample Jack Program The book's software suite includes a complete example of a Jack application, stored in `projects/09/Square`. This directory contains the source Jack code of three classes comprising a simple interactive game.