9. The High Level Language[1]

"High thoughts need a high language."

(Aristophanes, 448-380 BC)

This (work-in-progress) chapter presents an overview and specification of the *Jack* programming language. Jack is a simple object-based language that can be used to write high-level programs. It has the basic features and flavor of modern object-oriented languages like Java, with a much simpler syntax and no support for inheritance. 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.

The chapter starts with a very brief background. The bulk of the chapter describes and specifies the Jack language and its standard library (operating system). This is all the information needed for writing applications in the Jack language. Next, we illustrate some computer games written in Jack, and give general guidelines on how to write similar interactive applications over the Hack platform. All theses programs can be compiled using a *Jack compiler* that we provide, and then run on the computer hardware that we built in chapters 1-5. Throughout the chapter, our goal is not to turn you into a Jack programmer. Instead, our "hidden agenda" is to prepare you to write the compiler and operating system described in the chapters that lie ahead.

1. Background

It's important to note at the outset 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 will help us 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.

2. The Jack Language

We begin with some illustrations of Jack programming, and continue with a formal language specification. The former is all that you need in order to start writing Jack programs, and the latter is a complete language reference.

2.1 Examples

[1] From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2003, www.idc.ac.il/csd

Example 1: Hello World

Our first example is the classic Hello-World program:

```
/** Hello World program */
class Main {
   function void main(){
      do Output.printString ("HELLO WORLD");
      do Output.println();      // new line
      return;
    }
}
```

PROGRAM 1: Hello World.

This program consists of one class, called `Main`, which contains one function, called main, that contains a sequence of two invocations of functions from the standard library. The Main.main() method is where the execution starts in every program. You can see two comment formats: the /** */ documentation comment, and the //… one line comment.

Example 2: Fraction

The task: Every programming language has a fixed set of primitive data types, of which Jack supports three: `int`, `char`, and `boolean`. Suppose we wish to endow Jack with the added ability to handle *rational numbers*, i.e. 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 the class interface: A reasonable way to get started is to specify the set of properties and services that one would normally expect to see in a fraction abstraction. One such *Application Program Interface*, or *API*, is given in Program 2.

Fraction (partial API):

// A "Fraction" is an object representation of *n*/*m* where *n* and *m* are integers (e.g. 17/253)

`field int numerator, denominator:`	Represent *n* and *m*
`constructor Fraction new(int a, int b):`	Returns a new `Fraction` object.
`method int getNumerator():`	Returns the numerator of `this` object.
`method int getDenominator():`	Returns the denominator of `this` object.
`method Fraction plus(Fraction other):`	Returns the fraction sum of `this` fraction and the `other` one.
`method void print():`	Prints `this` object in the format "numerator/denominator".

PROGRAM 2: API for the Fraction abstraction. In Jack, operations on the current object (referred to as *this* object) are represented by *methods*, and class-level operations ("static methods" in, e.g., Java) are represented by *functions*.

Using the class (Example I): APIs mean different things to different people. If you are the programmer who has to *implement* the fraction abstraction, you can view its API as a contract that must be implemented in some way. Alternatively, if you are a programmer who needs to *use* fractions in your work, you can view the API as a library of fraction objects generators and fraction-related operations. Taking this latter view, consider the Jack code listed in Program 3.

```
/** Compute 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;
       }
}
```

PROGRAM 3: Using the Fraction abstraction in Jack programs

Program 3 illustrates several features of the Jack language: declaration and creation of new objects, use of the assignment statement `let`, and method calling using the `do` statement. As the code implies, users of the fraction abstraction don't have to know anything about its underlying *implementation.* Rather, they should be given access only to the fraction *interface*, or API, and then use it as a black box server of fraction-related operations.

Implementing the class: We now turn to the other player in our story -- the programmer who has to actually implement the fraction API in Jack. One possible implementation is given in Program 4. This example illustrates several additional features of object-oriented programming in Jack: a typical program structure (*classes*, *methods*, *constructors*, *functions*), as well as all statement types (*let*, *do*, *return, if, while*) available in Jack.

```
/** A Fraction (partial implementation) */
  class Fraction {

  field int numerator, denominator;

  /** Construct a new (reduced) fraction given a 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;
  }

 /* reduce this fraction - internal method (from the outside,
    a fraction always seems reduced.) */
 method void reduce() {
     var int g;
     let g = Fraction.gcd(numerator, denominator);
     if (g > 1) {
        let numerator = numerator / g;
        let denominator = denominator / g;
     }
     return;
  }

  /* compute the gcd of a and b - internal service function */
  function int gcd(int a, int b){
     var int r;
     // apply Euclid's algorithm
     while (~(b = 0)) {
        let r = a - (b * (a / b));  // r = remainder of division a/b
        let a = b;
        let b = r;
     }
     return a;
  }

// the code continues in PROGRAM 4 (part II)
```

PROGRAM 4 (part I): a Fraction class implementation.

```
// continuation of the PROGRAM 4 (part I) code:

  method int getNumerator(){
    return numerator;
  }

  method int getDenominator(){
    return denominator;
  }

  /** Return 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());
  }

  // more methods: minus, times, div, …

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

} // end Fraction
```

PROGRAM 4 (part II): a Fraction class implementation (continued)

We now turn to a formal description of the Jack language, focusing on its syntactic elements, program structure, variables, expressions, and statements.

2.2 Syntactic Elements

A Jack program is a sequence of tokens. Tokens are separated by an arbitrary amount of white space (including comments), that is ignored. Tokens can be *symbols*, *reserved words*, *constants*, and *identifiers*, as listed in Table 5.

White space and comments	Space characters, newline characters, and *comment*s are ignored. The following *comment* formats are supported:
	`//  comment to end of line`
	`/*  comment until closing */`
	`/** API documentation comment */`

Symbols	`( )` : arithmetic grouping, and parameter/argument-lists grouping
	`[ ]` : array indexing;
	`{ }` : program structure and statement grouping;
	`,` : 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	Positive *Integer* constants are in standard decimal notation, e.g. “`1984`”. Negative numbers like “`-13`” are not constants, but rather 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 can be obtained using the function calls `String.newLine()` and `String.doubleQuote()`). *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.

TABLE 5: Syntactic elements of the Jack language.

2.3 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 the subroutine declarations
     subroutine declarations  // a sequence of constructor, method, and function declarations
}
```

Each class declaration begins with a name through which the class can be globally accessed. Next comes an optional sequence of declarations of class-level *fields* and *static variables*, described below. Next comes a sequence of *subroutine* declarations, defining *methods*, *functions*, and *constructors*. A method "belongs" to an object and provides the objects' functionality, while a function "belongs" to the class in general and is not associated with a particular object (similar to Java's *static methods*). A constructor "belongs" to the class and generates object instances of this class.

Subroutine declarations have the following formats:

```
constructor type name (parameter-list) {
    declarations
    statements
}

method type name (parameter-list) {
    declarations
    statements
}

function type name (parameter-list) {
    declarations
    statements
}
```

Each subroutine has a *name* through which it can be accessed. The subroutine's *type* specifies the type of the value returned by the subroutine. If the subroutine returns no value, the subroutine type is declared `void`; otherwise it can be any of the primitive or object data types supported by the language (section 2.4). Constructors may have arbitrary names but must return the class's own type, i.e. the class name.

Each subroutine contains an arbitrary sequence of local variable declarations and then a sequence of statements.

Jack programs: 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 host platform will automatically start running the `Main.main` function.

2.4 Variables

Variables in Jack must be explicitly declared before they are used. There are several kinds of variables: fields, static variables, local variables, and parameters, each with its associated scope. Variables are typed.

Data Types

The type can either be primitive (`int`, `char`, `boolean`) as is pre-defined in the language specification, or an Object type (like `Employee`, `Car`, etc.) which is defined by the programmer, as needed.

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;`" will 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. Object types are references (as in Java). The declaration of an object variable creates only a reference and only allocated memory to hold the reference. Memory for the object itself is only allocated later, if and when the programmer actually *constructs* the variable. Example:

```
// The following 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 a null reference
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 references to these two objects.
let f = e;   // only the reference is copied
```

PROGRAM 6: Object types.

The Jack standard library defines two built-in object types that play a role in the language syntax: arrays and strings.

Arrays: Arrays are declared using the built-in class `Array`. Arrays are single 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; construction of an array is done using the `Array.new(length)` function; access to array elements is done using the `a[j]` notation. The following example reads a sequence of integers and then prints their average.

```
/** 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);
       let i = 0;
       while (i < length) {
          let a[i] = Keyboard.readInt("ENTER THE NEXT NUMBER: ");
          let i = i + 1;
       }
       let i = 0;
       let sum = 0;
       while (i < length) {
          let sum = sum + a[i];
          let i = i + 1;
       }
       do Output.printString("THE AVERAGE IS: ");
       do Output.printInt(sum / length);
       do Output.println();
       return;
   }
}
```

PROGRAM 7: Arrays.

Strings

Strings are declared using the built-in class String. The compiler also recognizes the syntax `"xxx"`. String contents can be accessed and modified using the methods of the `String` class (see below). Example:

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

Automatic conversions: The Jack language is only weakly typed. The language does not define the results of attempted assignment or conversion from one type to another, and different compilers may allow or forbid different conversions. (This incompleteness of the definition is in order to enable simple compilers to be valid while completely ignoring all typing issues.) All compilers allow, and automatically perform, the following assignments:

- Characters and Integers are automatically converted into each other as needed, according to the Unicode specification.
- An integer can be assigned to a reference variable (of any object type), in which case it is treated as an address in memory.
- Any Object type may be converted into an Array, and vice-versa. The conversion “transforms” the fields of the object into consecutive array entries.

Variable Scopes

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 that holds the sum of all the balances of all the accounts, where each account is represented as an object of the `BankAccount` class. *Field variables* are used to define the properties of individual objects of the class, e.g. `owner` and `balance` in our banking example. *Local variables*, used by subroutines, exist only as long as the subroutine is running, and *parameters* are used to pass arguments to subroutines. For example, the function signature `function void deposit(BankAccount b,int sum)` indicates that `b` and `sum` are parameters.

Table 8 gives a formal description of all the variable kinds supported by the Jack language.

Var. Kind	Definition and 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, but cannot be used in functions.
Local variables	**`var`** ***type name1*****`,`** ***name2*****`, ... ;`** Local variables are allocated on the stack when the subroutine is called and are freed when it returns. (like *local variables* of Java).	Subroutine declaration.	The subroutine in which they are declared.

Parameters	***type name1*****`,`** ***type name2*****`, ...`**
	e.g.:
	`function void drive (`**`Car c, int miles`**`)`
	`{ ... }`
	Used as inputs of subroutines.
		Appear in parameter-lists as part of subroutine declarations.	The subroutine in which they are defined.

TABLE 8: Variable kinds in the Jack language.

2.4 Statements

The Jack language features 5 statements. They are defined and described in Table 9.

Statement	Syntax	Description

let	`let` *variable*=*expression*`;`
	or
	`let` *variable* [*expression*]=*expression*`;`
		An assignment operation. The variable may be static, local, field, or a parameter. Alternatively the variable may be an array entry.

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

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

do	`do` *function-or-method-call*`;`	Used to call a function or method for their effect, ignoring the value they return, if any. The function or method call follows the syntax described in section 2.6.

return	`return` *expression*`;`
	or
	`return;`
		Used to return values from subroutines.
		The second form must be used by functions and methods that return a void value.
		Constructors must return the expression `this`.

TABLE 9: Jack statements.

2.5 Expressions

Expression Syntax: Jack expressions are defined recursively according to the rules given in Table 10.

A *Jack expression* is one of the following:

- A *constant*
- A *variable name* in scope (the variable may be *static*, *field*, *local*, or a *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 (see Section 2.6).
- An expression prefixed by one of the unary operators {-,~}:
  -*expression*: arithmetic negation
  ~*expression*: boolean negation (bitwise for integers)
- An expression of the form *expression operator expression* where *operator* is one of the binary operators {+,-,*,/,&,|,<,>,=}:

+ - * /	:	integer arithmetic operators
& |	:	boolean `And`, `Or` (bitwise for integers) operators
< > =	:	comparison operators

- An expression in parenthesis: (*expression*)

TABLE 10: Jack expressions are either atomic or derived recursively from simpler expressions

Order of evaluation and operator priority: 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. At the same time, the lack of operator priority makes the writing of Jack compilers simpler.

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, consider the function declaration "`function int sqrt(int n)`" in class `Math`, designed to return the integer part of the square root of its single parameter. Such a function can be invoked using calls like `Math.sqrt(17)`, `Math.sqrt(a*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, i.e. *className.subroutineName*(*argument-list*). Outside a class, the class functions and constructors are also called using the full-name syntax, while methods are called using the syntax *var.methodName(argument-list)*, where *var* is a previously defined object variable. Program 11 gives some examples.

```
class Bar {
...
}

class Foo {
  ...
  function void f() {
    var Bar b;
    ...
    ... g(5,7)    // call to method g of class Foo (on this object)
    ... Foo.p(2)  // call to function or constructor p of class Foo
    ... Bar.h(3)  // call to function or constructor h of class Bar
    ... b.q()     // call to method q of class Bar (on object b)
    ...
  }
  ...
}
```

PROGRAM 11: Subroutine call examples.

Object Construction and Disposal: As mentioned, when a variable of an object type is declared, only a reference to an object of this type is allocated. To actually create the object, a class constructor must be called. There must be at least one constructor for every class that defines a type. Constructors may have arbitrary names, but it is customary to call one of then "`new`". When constructors are used they are called just like any other class function using the format

```
let <var> = <class_name>.<constructor_name>(<parameter_list>);
```

When the constructor is called, the compiler automatically allocates memory space for the new object, and assigns the allocated space's base address to `this`. Then the constructor body is executed, in order to initialize the object into a valid state.

Objects can be de-allocated and their space reclaimed using the `Memory.deAlloc(object)` function of the standard library. Convention calls for every class to contain a `dispose()` method that properly encapsulates this de-allocation.

Example: Consider a class named `List`, designed to hold a linked-list. Each link in the list holds an element and a reference to the rest of the list. Program 12 gives a standard `List` implementation.

```
/** A List class (partial implementation) */
class List {
  field int data;
  field List next;

  constructor List new(int car, List cdr){
    let data = car;
    let next = cdr;
    return this;
  }
...

  method void dispose() {
     if (~(next = null)) {
        do next.dispose();
     }
     do Memory.deAlloc(this);
     return;
  }
}

// create a list holding the numbers (2,3,5):
function void create235(){
  var List v;
  let v=List.new(5,null);
  let v=List.new(2,List.new(3,v));
  // ...
  do v.dispose();
  return;
}
```

PROGRAM 12: Examples of object construction and destruction code in Jack. The function constructs the null-terminated linked-list `(2,3,5)` `and then disposes it.`

3. Jack Standard Library / Operating System

The Jack language comes with a standard library that may also be viewed as an interface to an underlying operating system. The library is a collection of Jack classes, and must be provided in every implementation of the Jack language. The standard library includes the following classes:

▪ **Math:**	Provides basic mathematical operations;
▪ **String:**	Implements the `String` type and basic string-related operations;
▪ **Array:**	Defines the Array type and allows construction and disposal of arrays;
▪ **Output:**	Handles text based output;
▪ **Screen:**	Handles graphic screen output;
▪ **Keyboard:**	Handles user input from the keyboard;
▪ **Memory:**	Handles memory operations;
▪ **Sys:**	Provides some execution-related services.

This section specifies the subroutines that are supposed to be in these classes.

Math

This class enables various mathematical operations.

- 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 the 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 at least of the prefix until a non numeric character is found).
- 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 **`moveCursor`**(int i, int j): Moves the cursor to the j’th column of the i’th row, and erases the character located 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 (over HACK: 256 rows * 512 columns).

- Function void **`clearScreen`**(): Erases the entire screen.
- Function void **`setColor`**(boolean b): Sets the screen 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 where the top left corner is (x1, y1) and the bottom right corner is (x2,y2).

- Function void **`drawCircle`**(int x, int y, int r): Draws a filled circle of radius r around (x,y)

Keyboard

This class allows reading inputs from the keyboard.

- 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, and 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 next line (until a newline character) from the keyboard, echoes the line to the screen, and returns its value. This method handles user backspaces.
- Function int **`readInt`**(String message): Prints the message on the screen, reads the next line (until a newline character) from the keyboard, echoes the line to the screen, and returns the integer until the first non numeric character in the line. This method handles user backspaces.

Memory

This class allows direct access to the main memory.

- Function int **`peek`**(int address): Returns the value of the main memory at this address.
- Function void **`poke`**(int address, int value): Sets the value of the main memory at this address to the given value.
- Function Array **`alloc`**(int size): Allocates the specified space on the heap and returns a reference to it.
- Function void **`deAlloc`**(Array o): De-allocates the given object and frees its memory space.

Sys

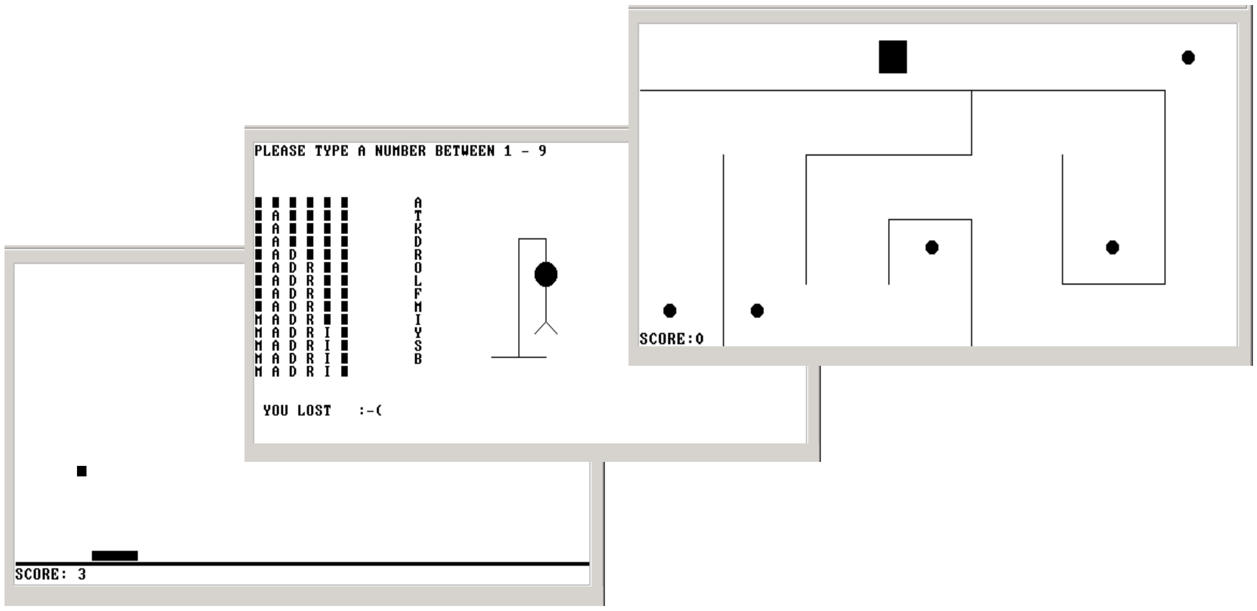
This class supports some execution-related services.

- Function void **`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.

4. Writing Jack Applications

Jack is a general-purpose language that can be implemented over different hardware platforms. In the next two chapters we will develop a Jack compiler that ultimately generates Hack code, and thus it is natural to discuss Jack applications in the context of the Hack platform.

Recall that the Hack computer is equipped with a 256 rows by 512 columns screen and a standard keyboard. These I/O devices, along with the Jack classes that drive them, enable the development of interactive programs with a graphical GUI. Figure 13 gives some examples.



```
SCORE: 3
```

```
PLEASE TYPE A NUMBER BETWEEN 1 - 9

■ ■ ■ ■ ■ ■      A
■ A ■ ■ ■ ■      T
■ A ■ ■ ■ ■      K
■ A ■ ■ ■ ■      D
■ A D ■ ■ ■      R
■ A D R ■ ■      O
■ A D R ■ ■      L
■ A D R ■ ■      F
■ A D R ■ ■      M
M A D R ■ ■      I
M A D R I ■      Y
M A D R I ■      S
M A D R I ■      B
M A D R I ■

 YOU LOST   :-(
```

```
SCORE:0
```

FIGURE 13: Screen shots of three computer games written in Jack, running on the Hack computer. Left to right: a single player "Pong" game, after scoring three points (the author's record), a "Hangman" game, where the user has to guess the name of the capital city hidden behind the squares (the 1-9 number determines the game level), and a maze game, in which the user scores points by moving the square over the dots.

The Pong game: A ball is moving on the screen randomly, bouncing off the screen "walls". The user can move a small bat horizontally by pressing the left and right arrow keys. Each time the bat hits the ball, the user scores a point and the bat shrinks a little, to make the game harder. If the user misses and the ball hits the bottom horizontal line, the game is over.

The Pong game provides a good illustration of Jack programming over the Hack platform. First, it is a non-trivial program, requiring several hundreds lines of Jack code organized in several classes. Second, the program has to carry out some non-trivial mathematical calculations, in order to compute the direction of the ball's movements. Third, the program must animate the movement of graphical objects on the screen (the bat and the ball), requiring extensive use of the language's graphics drawing methods. Finally, 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.

Other applications: Of course Pong is just one example of the numerous applications that can be written in Jack over the Hack platform. Since the Jack screen resembles that of a cellular telephone, it lends itself nicely to the computer games that one normally finds on cellular telephones, e.g. *Snake* and *Tetris*. In general, the more event- and GUI-driven is the application, the more "catchy" it will be.

Having said that, recall that the Jack/Hack platform is in fact a general-purpose computer. As such, one can use it to develop any application that involves inputs, outputs, and calculations, not necessarily in this order. For example, one can write a program that inputs student names and grades, and then prints the average grade and the name of the student who got the maximal grade, and so on. We now turn to describe how such applications can be planned and developed.

The development of Jack applications over a target platform requires careful planning (as always). In the specification stage, the application designer must consider the physical limitations of the target hardware, and plan accordingly. For example, the physical dimensions of the computer's screen limits the size of the graphical images that the program can handle. Likewise, one must consider the language's range of input/output commands, in order to have a realistic appreciation of what can and cannot be done.

Although the language's capabilities can be extended at will (by modifying its supporting libraries, also written in Jack), 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 course. 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, they view it 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.

Specification: The specification stage should start with some description of the application's behavior and, in the case of graphical and interactive applications, some hand-written drawings of typical screens. Using object-oriented analysis and design practices, the designer should then create an object-based architecture of the application. This requires the identification of *classes*, *fields*, and *subroutines*, resulting with a well-defined API (like Program 1).

Implementation: Next, one implements the API in Jack and tests it on the target platform. Following compilation into VM code, the program can be tested in two alternatives ways. Either the VM code can be executed directly on a VM emulator, or it can be translated into Hack code and ran on the hardware platform.

5. Perspective

The Jack programming language is certainly more "clunky" that what you would expect from a modern programming language. However, its basic features and semantic level are not so different from other modern programming languages. Jack is an "object-based" language: supports objects and classes but not inheritance. In this respect it is somewhere between procedural languages (like Pascal or C) and object-oriented languages (like Java or C++). Additionally, Jack's primitive type system is pretty weak, and moreover, Jack is not "strongly typed" – i.e. type conformity in assignments and operations is not strictly enforced. All these differences from normal programming languages are in order to simplify compiler construction.

The standard library -- operating system – is quite far from any reasonable operating system. The major deficiencies are the total lack of concurrency -- multi-threading or multi-processing, and the total lack of any permanent storage – files or even communication. The operating system does provide graphic and textual I/O similar to standard ones, although in very basic forms. It provides a standard implementation of strings, as well as standard memory allocation and de-allocation. Additionally, it provides the basic arithmetic operations of multiplication and division that are normally implemented in hardware.

6. Build It

Unlike most projects in the book, this project does not involve building part of the computer's hardware or software systems. Rather, you have to choose an application of your choice, specify it, and then implement it in Jack on the Hack platform.

Objective: The major objective of this project is to get acquainted with the Jack language, for two purposes. First, you have to know Jack intimately in order to write the Jack compiler in Projects 10 and 11. Second, you have to be familiar with Jack's supporting libraries in order to write the computer's operating system in Project 12. The best way to gain this knowledge is to write a Jack application.

Contract: Adopt or invent an application idea, e.g. a simple computer game or some other interactive program. Then specify and build the application.

Steps

1. Download the supplied `os.zip` file and extract its contents to a directory named `project9` on your computer (you may want to give this directory a more descriptive name, e.g. the name of your program). The resulting set of `.vm` files constitutes an implementation of the computer's operating system, including all the supporting Jack libraries.
2. Write your Jack program (set of one or more Jack classes) using a plain text editor. Put each class in a separate `ClassName.jack` file. Put all these `.jack` files in the same program directory described in step 1.
3. Compile your program using the supplied Jack Compiler. This is best done by applying the compiler to the name of the program directory. This will cause the compiler to translate all the `.jack` classes in that directory to corresponding `.vm` files. If a compilation error is reported, debug the program and re-compile until no error messages are issued.
4. At this point, the program directory should contain three sets of files: (a) your source `.jack` files, (b) a compiled `.vm` file for each one of your source `.jack` files, and (c) the supplied operating system `.vm` files. To test the compiled program, invoke the *VM Emulator* and direct it to load the program by selecting the program directory name. Then run the program. In case of run-time errors or undesired program behavior, fix the program and go to stage 3.

Deliverables: You are expected to deliver a `readme.txt` text file that tells users everything they have to know about using your program, and a zip file containing all your source Jack files. Do not submit any `.vm` files.