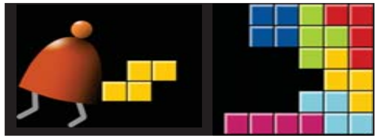
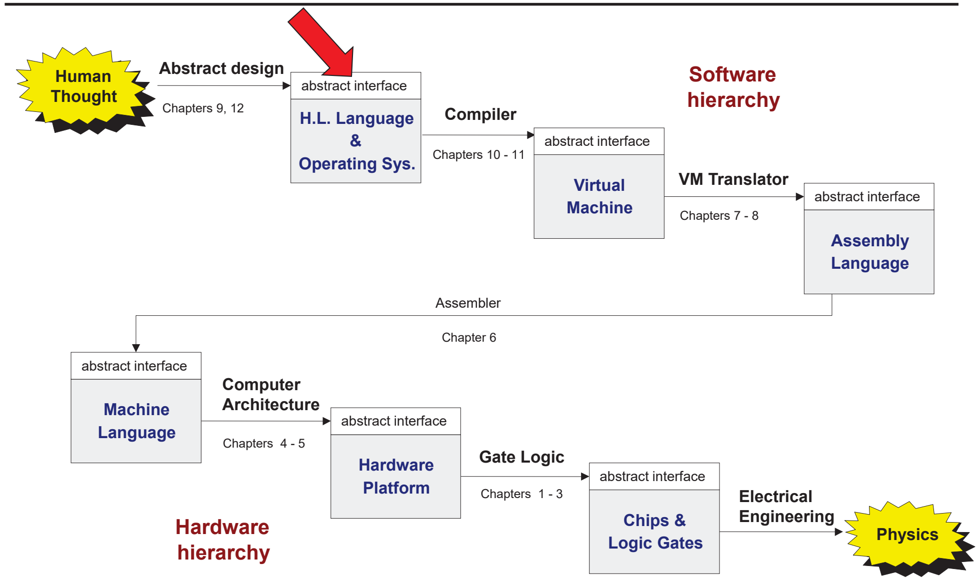
High-Level Language



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



Some milestones in the evolution of programming languages

- Machine language (binary code)
- Assembly language (low-level symbolic programming)
- Simple procedural languages, e.g. `Fortran`, `Basic`, `Pascal`, `C`

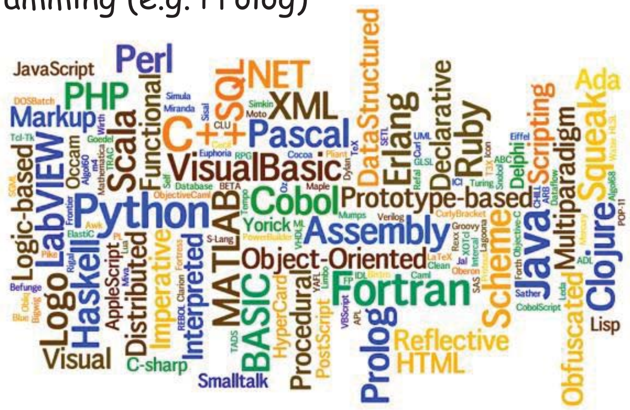
```
      PROGRAM TPK
C     THE TPK ALGORITHM
C     FORTRAN 77 STYLE
      REAL A(0:10)
      READ (5,*) A
      DO 10 I = 10, 0, -1
         Y = FUN(A(I))
         IF (Y .LT. 400) THEN
            WRITE(6,9) I,Y
 9          FORMAT(I10. F12.6)
         ELSE
            WRITE (6,5) I
 5          FORMAT(I10,' TOO LARGE')
         ENDIF
 10   CONTINUE
      END

      REAL FUNCTION FUN(T)
      REAL T
      FUN = SQRT(ABS(T)) + 5.0*T**3
      END
```


Some milestones in the evolution of programming languages

- Machine language (binary code)
- Assembly language (low-level symbolic programming)
- Simple procedural languages, e.g. `Fortran`, `Basic`, `Pascal`, `C`
- Simple object-based languages (without inheritance), e.g. early versions of `Visual Basic`, `JavaScript` ← **Jack**
- Fancy object-oriented languages (with inheritance): `C++`, `Java`, `C#`

Programming languages

- Procedural programming (e.g. C, Fortran, Pascal)
- Object-oriented programming (e.g. C++, Java, Python)
- Functional programming (e.g. Lisp, ML, Haskell)
- Logic programming (e.g. Prolog)



ML

- fun fac(x) =
  if x=0 then 1
  else x*fac(x-1);

- fun length(L) =
  if (L=nil) then 0
  else 1+length(tl(L));

Prolog

- Facts
  - human(kate).
  - human(bill).
  - Human(John).
  - likes(bill,kate).
  - likes(kate,john).
  - likes(john,kate).
- Rules
  - friend(X,Y) :- likes(X,Y),likes(Y,X).

Prolog

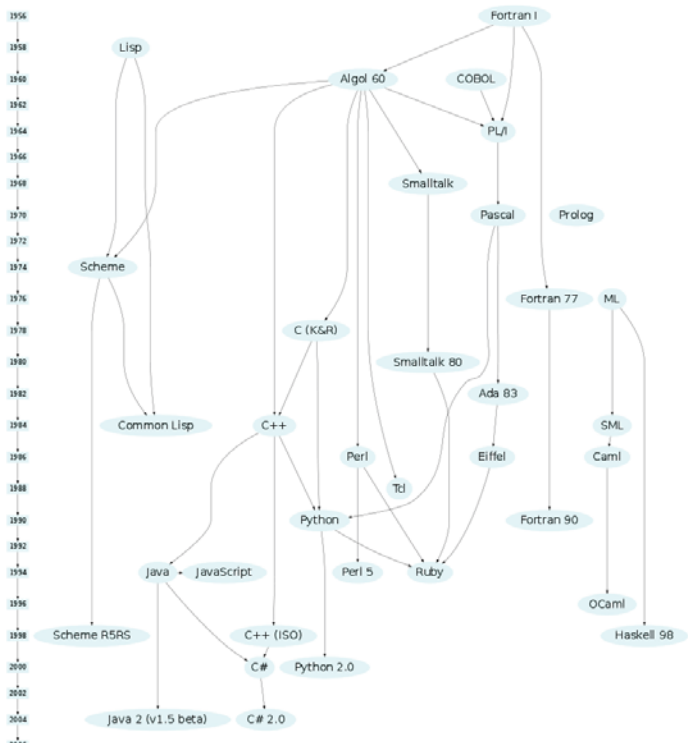
- Absolute value
  abs(X, Y) :- X <0, Y is –X.
  abs(X, X) :- X>=0.

  ?- abs(-9,8).
  No

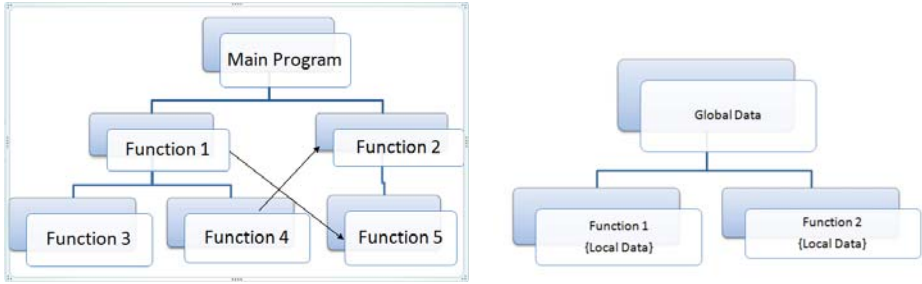
  ?- abs(-9,R).
  R=9

- Length of a list
  my_length([], 0).
  my_length([_|T],R) :- my_length(T, R1), R is R1+1.

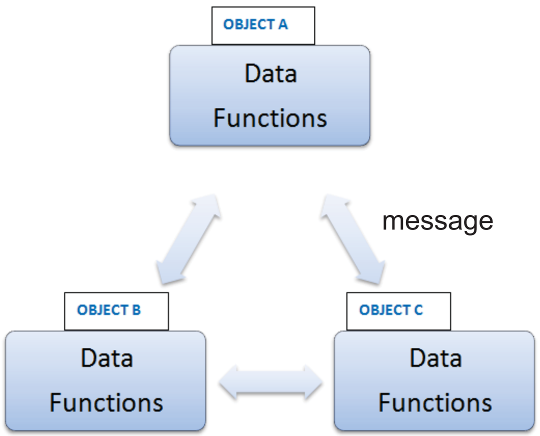
  ?- my_length([a, b, [c, d], e], R).
  R = 4



Procedure oriented programming



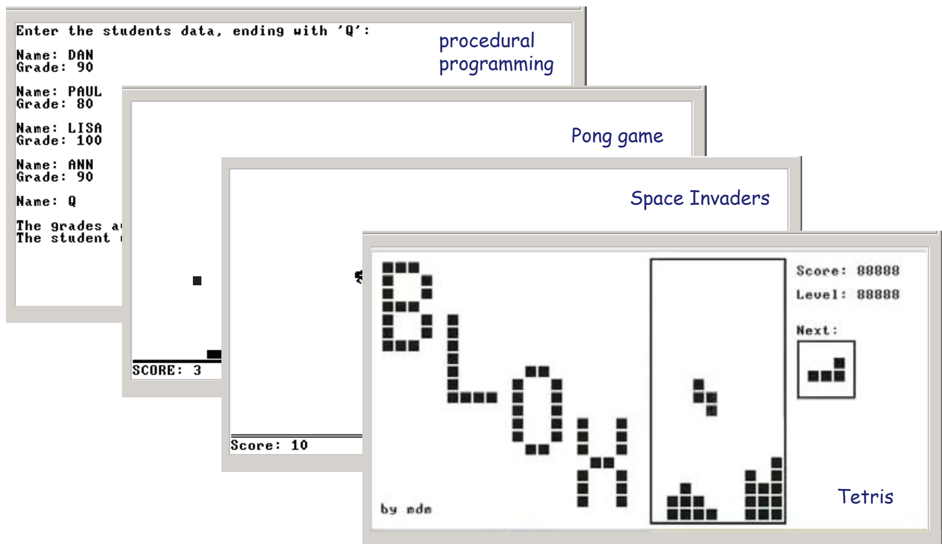
Object oriented programming



The Jack programming language

Jack: a simple, object-based, high-level language with a Java-like syntax

Some sample applications written in Jack:



Disclaimer

Although Jack is a real programming language, we don't view it as an *end*.

Rather, we use Jack as a *means* for teaching:

- How to build a compiler
- How the compiler and the language interface with the operating system
- How the topmost piece in the software hierarchy fits into the big picture

Jack can be learned (and un-learned) in one hour.

Roadmap for learning Jack

- Start with examples
  - Hello World
  - Procedure and array
  - Abstract data types
  - Linked list
  - ...
- Formal Jack Spec.
- More complex examples

Hello world

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

Some observations:

- Java-like syntax
- Classes
- Entry point: Main.main
- Typical comments format
- do for function calls
- Class_name.method_name
- Standard library a set of OS services (methods and functions) organized in 8 supplied classes: Math, String. Array, Output, Keyboard, Screen, Memory, Sys

Jack standard library aka language extensions aka Jack OS

```
class Math {
   function void init()
   function int abs(int x)
   function int multiply(int x, int y)
   function int divide(int x, int y)
   function int min(int x, int y)
   function int max(int x, int y)
   function int sqrt(int x)
}
```

Jack standard library aka language extensions aka Jack OS

```
Class String {
   constructor String new(int maxLength)
   method void   dispose()
   method int    length()
   method char   charAt(int j)
   method void   setCharAt(int j, char c)
   method String appendChar(char c)
   method void   eraseLastChar()
   method int    intValue()
   method void   setInt(int j)
   function char backSpace()
   function char doubleQuote()
   function char newLine()
}
```


Jack standard library aka language extensions aka Jack OS

```
Class Array {

   function Array new(int size)

   method void dispose()

}
```

```
class Memory {

   function int peek(int address)

   function void poke(int address, int value)

   function Array alloc(int size)

   function void deAlloc(Array o)

}
```


Jack standard library aka language extensions aka Jack OS

```
class Output {
   function void moveCursor(int i, int j)
   function void printChar(char c)
   function void printString(String s)
   function void printInt(int i)
   function void println()
   function void backSpace()
}
```

```
Class Screen {
   function void clearScreen()
   function void setColor(boolean b)
   function void drawPixel(int x, int y)
   function void drawLine(int x1, int y1, int x2, int y2)
   function void drawRectangle(int x1, int y1, int x2, int y2)
   function void drawCircle(int x, int y, int r)
}
```


Jack standard library aka language extensions aka Jack OS

```
Class Keyboard {
   function char keyPressed()
   function char readChar()
   function String readLine(String message)
   function int readInt(String message)
}
```

```
Class Sys {
   function void halt():
   function void error(int errorCode)
   function void wait(int duration)
}
```

Typical programming tasks in Jack

Jack can be used to develop any app that comes to my mind, for example:

- Array processing — a program storing numbers in an array
- Procedural programming: — a program that computes 1 + 2 + ... + n
- Object-oriented programming: — a class representing bank accounts
- Abstract data type representation: — a class representing fractions (like 2/5)
- Data structure representation: — a class representing linked lists

We will now discuss the above examples

As we do so, we'll begin to unravel how the magic of a high-level object-based language is delivered by the compiler and by the VM

These insights will serve us in the next lectures, when we build the Jack compiler.

Array example

```
class Main {
  function void main () {
    var Array a;
    var int length, i, sum;

    let length = Keyboard.readInt(“#number:”)
    let a = Array.new(length);
    let i = 0;
    let sum = 0;

    while (i < length) {
      let a[i] = Keyboard.readInt("next: ");
      let sum = sum + a[i];
      let i = i+1;
    }

    do Output.printString("The average: ");
    do Output.printInt(sum / length);
    do Output.println();
    return;
  }
}
```

- var: variable declaration
- type: int, Array
- let: assignment
- Array: provided by OS. No type for an array. Actually, it can contain any type and even different types in an array.
- Primitive types: int, boolean, char.
- All types in Jack occupy one word. When declaring a variable of primitive types, the space is reserved. For other types, a reference is reserved.

Procedural programming example

```
class Main {

  /** Sums up 1 + 2 + 3 + ... + n */
  function int sum (int n) {
    var int sum, i;
    let sum = 0;
    let i = 1;
    while (~(i > n)) {
      let sum = sum + i;
      let i = i + 1;
    }
    return sum;
  }

  function void main () {
    var int n;
    let n = Keyboard.readInt("Enter n: ");
    do Output.printString("The result is: ");
    do Output.printInt(sum(n));
    return;
  }
}
```

Jack program = a collection of one or more classes

Jack class = a collection of one or more subroutines

Execution order: when we execute a Jack program, `Main.main()` starts running.

Jack subroutine:

- method
- constructor
- function (static method)
- (the example on the left has functions only, as it is "object-less")

Object-oriented programming example

The BankAccount class (skeletal)

```
/** Represents a bank account.
    A bank account has an owner, an id, and a balance.
    The id values start at 0 and increment by 1 each
    time a new account is created. */

class BankAccount {

    /** Constructs a new bank account with a 0 balance. */
    constructor BankAccount new(String owner)

    /** Deposits the given amount in this account. */
    method void deposit(int amount)

    /** Withdraws the given amount from this account. */
    method void withdraw(int amount)

    /** Prints the data of this account. */
    method void printInfo()

    /** Disposes this account. */
    method void dispose()
}
```


Object-oriented programming example (continues)

```
/** Represents a bank account. */
class BankAccount {
  // class-level variable
  static int newAcctId;

  // Private variables(fields/properties)
  field int id;
  field String owner;
  field int balance;

  /** Constructs a new bank account */
  constructor BankAccount new (String
   owner) {
      let id = newAcctId;
      let newAcctId = newAcctId + 1;
      let this.owner = owner;
      let balance = 0;
      return this;   // 2
  }
  // More BankAccount methods.
}
```

```
// Code in any other class:
var int x;
var BankAccount b;   // 1
let b = BankAccount.new("joe");   // 3
```

Explain `b = BankAccount.new("joe")`

Calls the constructor (which creates a new BankAccount object)

Explain `return this`

The constructor returns the RAM base address of the memory block that stores the data of the newly created BankAccount object

Explain `b = BankAccount.new("joe")`

stores in variable b a pointer to the object's base memory address

Object-oriented programming example (continues)

```
/** Represents a bank account. */
class BankAccount {
  // class-level variable
  static int newAcctId;

  // Private variables(fields/properties)
  field int id;
  field String owner;
  field int balance;

  /** Constructs a new bank account */
  constructor BankAccount new (String
   owner) {
      let id = newAcctId;
      let newAcctId = newAcctId + 1;
      let this.owner = owner;
      let balance = 0;
      return this;   // 2
  }
  // More BankAccount methods.
}
```

```
// Code in any other class:
var int x;
var BankAccount b;   // 1
let b = BankAccount.new("joe");   // 3
```

Behind the scene (following compilation):

```
// b = BankAccount.new("joe")
push "joe"
call BankAccount.new
pop b
```

Explanation: the calling code pushes an argument and calls the constructor; the constructor's code (not shown above; the compiler generates `Memory.alloc(n)` for constructors) creates a new object, pushes its base address onto the stack, and returns;

The calling code then pops the base address into a variable that will now point to the new object.

Object-oriented programming example (continues)

```
class BankAccount {
  static int nAccounts;

  field int id;
  field String owner;
  field int balance;

  // Constructor ... (omitted)

  /** Handles deposits */
  method void deposit (int amount) {
      let balance = balance+amount;
      return;
  }

  /** Handles withdrawls */
  method void withdraw (int amount){
      if (~(amount > balance)) {
          let balance = balance-amount;
      }
      return;
  }
  // More BankAccount methods.
}
```

```
...
var BankAccount b1, b2;
...
let b1 = BankAccount.new("joe");
let b2 = BankAccount.new("jane");
do b1.deposit(5000);
do b1.withdraw(1000);
...
```

Explain `do b1.deposit(5000)`

- In Jack, void methods are invoked using the keyword `do` (a compilation artifact)
- The object-oriented method invocation style `b1.deposit(5000)` is a fancy way to express the procedural semantics `deposit(b1,5000)`

Behind the scene (following compilation):

```
// do b1.deposit(5000)
push b1
push 5000
call BankAccount.deposit
```

Object-oriented programming example (continues)

```
class BankAccount {
  static int nAccounts;

  field int id;
  field String owner;
  field int balance;

  // Constructor ... (omitted)

  /** Prints information about this account. */
  method void printInfo () {
      do Output.printInt(id);
      do Output.printString(owner);
      do Output.printInt(balance);
      return;
  }

  /** Disposes this account. */
  method void dispose () {
      do Memory.deAlloc(this);
      return;
  }
  // More BankAccount methods.
}
```

```
// Code in any other class:
...
var int x;
var BankAccount b;

let b = BankAccount.new("joe");
// Manipulates b...
do b.printInfo();
do b.dispose();
```

Explain

`do b.dispose()`

Jack has no garbage collection; The programmer is responsible for explicitly recycling memory resources of objects that are no longer needed. If you don't do so, you may run out of memory.

Object-oriented programming example (continues)

```
class BankAccount {
  static int nAccounts;

  field int id;
  field String owner;
  field int balance;

  // Constructor ... (omitted)

  /** Prints information about this account. */
  method void printInfo () {
      do Output.printInt(id);
      do Output.printString(owner);
      do Output.printInt(balance);
      return;
  }

  /** Disposes this account. */
  method void dispose () {
      do Memory.deAlloc(this);
      return;
  }
  // More BankAccount methods.
}
```

```
// Code in any other class:
...
var int x;
var BankAccount b;

let b = BankAccount.new("joe");
// Manipulates b...
do b.printInfo();
do b.dispose();
```

Explain

`do Memory.deAlloc(this)`

This is a call to an OS function that knows how to recycle the memory block whose base-address is `this`. We will write this function when we develop the OS (project 12).

Abstract data type example

The Fraction class API (method signatures)

```
/** A fraction consists of a numerator and a denominator, both int values */

class Fraction {
    /** Constructs a fraction from the given data */
    constructor Fraction new(int numerator, int denominator)

    /** Reduces this fraction, e.g. changes 20/100 to 1/5. */
    method void reduce()

    /** Accessors
    method int getNumerator()
    method int getDenominator()

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

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

    /** Prints this fraction */
    method void print()

    /** Disposes this fraction */
    method void dispose()
}
```


Abstract data type example (continues)

```
class Fraction {
    field int numerator, denominator;

    constructor Fraction new (int numerator, int denominator) {
        let this.numerator = numerator;
        let this.denominator = denominator;
        do reduce() // Reduces the new fraction
        return this
    }

    /** Reduces this fraction */
    method void reduce () { // Code omitted }

    // A static method computing the greatest common denominator of a and b.
    function int gcd (int a, int b) { // Code omitted }

    method int getNumerator () {
        return numerator;
    }

    method int getDenominator () {
        return denominator;
    }

    // More Fraction methods follow.
```

```
// Code in any other class:
...
var Fraction a, b;
let a = Fraction.new(2,5);
let b = Fraction.new(70,210);
do b.print() // prints "1/3"
...
// (print method in next slide)
```

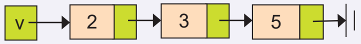
Abstract data type example (continues)

```
    ...

    // Constructor and previously defined methods omitted

    /** Returns the sum of this fraction 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());
    }

    // Similar fraction arithmetic methods follow, code omitted.

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

}
```

```
// Code in any other class:
var Fraction a, b, c;
let a = Fraction.new(2,3);
let b = Fraction.new(1,5);
// computes c = a + b
let c = a.plus(b);
do c.print(); // prints "13/15"
```


Data structure example

```
/** Represents a sequence of int values, implemented as a linked list.
    The list consists of an atom, which is an int value,
    and a tail, which is either a list or a null value.  */
class List {
    field int data;
    field List next;

    /* Creates a new list */
    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();
        }
        do Memory.deAlloc(this);
        return;
    }
    ...
}  // class List.
```




```
// Code in any other class:
...
// Creates a list holding 2,3, and 5:
var List v;
let v = List.new(5 , null);
let v = List.new(2 , List.new(3,v));
...
```


Jack language specification

- Syntax
- Program structure
- Data types
- Variable kinds
- Expressions
- Statements
- Subroutine calling

(for complete language specification, see the book).

Jack 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.

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

Jack syntactic elements

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

Jack syntactic elements

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.

Jack program structure

```
class ClassName {
  field variable declarations;
  static variable declarations;
  constructor type { parameterList ) {
    local variable declarations;
    statements
  }
  method type { parameterList ) {
    local variable declarations;
    statements
  }
  function type { parameterList ) {
    local variable declarations;
    statements
  }
}
```

About this spec:

- Every part in this spec can appear 0 or more times
- The order of the `field` / `static` declarations is arbitrary
- The order of the subroutine declarations is arbitrary
- Each *type* is either `int`, `boolean`, `char`, or a class name.

A Jack program:

- Each class is written in a separate file (compilation unit)
- Jack program = collection of one or more classes, one of which must be named `Main`
- The `Main` class must contain at least one method, named `main()`

Jack data types

Primitive types (Part of the language; Realized by the compiler):

- `int` 16-bit 2's complement (from -32768 to 32767)
- `boolean` `0` and `-1`, standing for `true` and `false`
- `char` unicode character ('a', 'x', '+', '%', ...)

Abstract data types (Standard language extensions; Realized by the OS / standard library):

- `String`
- `Array`

`...` (extensible)

Application-specific types (User-defined; Realized by user applications):

- `BankAccount`
- `Fraction`
- `List`
- `Bat / Ball` ... (as needed)

Jack data types

Jack is 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 it.

```
var char c; var String s;
Let c = 33;  // ‘A’
// Equivalently
Let s = “A”;  let c=s.charAt(0);
```

```
var Array a;
Let a = 5000;
Let a[100] = 77;  // RAM[5100]=77
```

```
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)
```


Jack variable kinds and scope

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.

Jack Statements (five types)

```
let varName = expression;
or
let varName[expression] = expression;
```

```
if (expression) {
    statements
}
else {
    statements
}
```

```
while (expression) {
    statements
}
```

```
do function-or-method-call;
```

```
return expression;
or
return;
```


Jack expressions

A Jack *expression* is any one of the following:

- A constant
- A variable name in scope (the variable may be static, field, local, or a parameter)
- The keyword `this`, denoting the current object
- An array element using the syntax *arrayName*[*expression*], where *arrayNname* 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* (logical negation)
- An expression of the form *expression op expression* where *op* is one of the following:
  - `+ - * /` (integer arithmetic operators)
  - `& |` (boolean and and or operators, bit-wise)
  - `< > =` (comparison operators)
- (*expression*) (an expression within parentheses)

Jack subroutine calls

General syntax: `subroutineName(arg0, arg1, …)`
where each argument is a valid Jack expression

Parameter passing is *by-value* (primitive types) or *by-reference* (object types)

Example 1:

Consider the function (static method): `function int sqrt(int n)`

This function can be invoked as follows:

```
sqrt(17)
sqrt(x)
sqrt((b * b) – (4 * a * c))
sqrt(a * sqrt(c - 17) + 3)
```

etc. In all these examples the argument value is computed and passed by-value

Example 2:

Consider the method: `method Matrix plus (Matrix other);`

If `u` and `v` were variables of type `Matrix`, this method can be invoked using: `u.plus(v)`

The `v` variable is passed by-reference, since it refers to an object.

Noteworthy features of the Jack language

- The (cumbersome) `let` keyword, as in `let x = 0;`
- The (cumbersome) `do` keyword, as in `do reduce();`
- No operator priority: (language does not define, compiler-dependent)
  `1 + 2 * 3` yields `9`, since expressions are evaluated left-to-right;
  To effect the commonly expected result, use `1 + (2 * 3)`
- Only three primitive data types: `int`, `boolean`, `char`;
  In fact, each one of them is treated as a 16-bit value
- No casting; a value of any type can be assigned to a variable of any type
- Array declaration: `Array x;` followed by `x = Array.new();`
- Static methods are called `function`
- Constructor methods are called `constructor`;
  Invoking a constructor is done using the syntax *ClassName*`.new(`*argsList*`)`

Q: Why did we introduce these features into the Jack language?

A: To make the writing of the Jack compiler easy!

Any of these language features can be modified, with a reasonable amount of work, to make them conform to a more typical Java-like syntax.

The Jack grammar

Lexical elements: The Jack language includes five types of terminal elements (tokens):

keyword: **'class'** | **'constructor'** | **'function'** | **'method'** | **'field'** | **'static'** | **'var'** | **'int'** | **'char'** | **'boolean'** | **'void'** | **'true'** | **'false'** | **'null'** | **'this'** | **'let'** | **'do'** | **'if'** | **'else'** | **'while'** | **'return'**

symbol: '{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~'

integerConstant: A decimal number in the range 0 .. 32767.

StringConstant: '"' A sequence of Unicode characters not including double quote or newline '"'

identifier: A sequence of letters, digits, and underscore ('_') not starting with a digit.

'x':	x appears verbatim
x:	x is a language construct
x?:	x appears 0 or 1 times
x*:	x appears 0 or more times
x|y:	either x or y appears
(x,y):	x appears, then y.

The Jack grammar

Program structure: A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax:

class: **'class'** className '{' classVarDec* subroutineDec* '}'

classVarDec: (**'static'** | **'field'**) type varName (',' varName)* ';'

type: **'int'** | **'char'** | **'boolean'** | className

subroutineDec: (**'constructor'** | **'function'** | **'method'**) (**'void'** | type) subroutineName '(' parameterList ')' subroutineBody

parameterList: ((type varName) (',' type varName)*)?

subroutineBody: '{' varDec* statements '}'

varDec: **'var'** type varName (',' varName)* ';'

className: identifier

subroutineName: identifier

varName: identifier

'x':	x appears verbatim
x:	x is a language construct
x?:	x appears 0 or 1 times
x*:	x appears 0 or more times
x|y:	either x or y appears
(x,y):	x appears, then y.

The Jack grammar

Statements:

statements: statement*

statement: letStatement | ifStatement | whileStatement | doStatement | returnStatement

letStatement: `'let'` varName (`'['` expression `']'`)? `'='` expression `';'`

ifStatement: `'if'` `'('` expression `')'` `'{'` statements `'}'` (`'else'` `'{'` statements `'}'`)?

whileStatement: `'while'` `'('` expression `')'` `'{'` statements `'}'`

doStatement: `'do'` subroutineCall `';'`

ReturnStatement `'return'` expression? `';'`

- 'x': x appears verbatim
- x: x is a language construct
- x?: x appears 0 or 1 times
- x*: x appears 0 or more times
- x|y: either x or y appears
- (x,y): x appears, then y.

The Jack grammar

Expressions:

expression: term (op term)*

term: integerConstant | stringConstant | keywordConstant | varName | varName `'['` expression `']'` | subroutineCall | `'('` expression `')'` | unaryOp term

subroutineCall: subroutineName `'('` expressionList `')'` | (className | varName) `'.'` subroutineName `'('` expressionList `')'`

expressionList: (expression (`','` expression)*)?

op: `'+'` | `'-'` | `'*'` | `'/'` | `'&'` | `'|'` | `'<'` | `'>'` | `'='`

unaryOp: `'-'` | `'~'`

KeywordConstant: `'true'` | `'false'` | `'null'` | `'this'`

- 'x': x appears verbatim
- x: x is a language construct
- x?: x appears 0 or 1 times
- x*: x appears 0 or more times
- x|y: either x or y appears
- (x,y): x appears, then y.

VM programming: multiple functions

Compilation:

- A Jack application is a set of 1 or more class files (just like .java files).
- When we apply the Jack compiler to these files, the compiler creates a set of 1 or more .vm files (just like .class files). Each method in the Jack app is translated into a VM function written in the VM language
- Thus, a VM file consists of one or more VM functions.

VM programming: multiple functions (files)

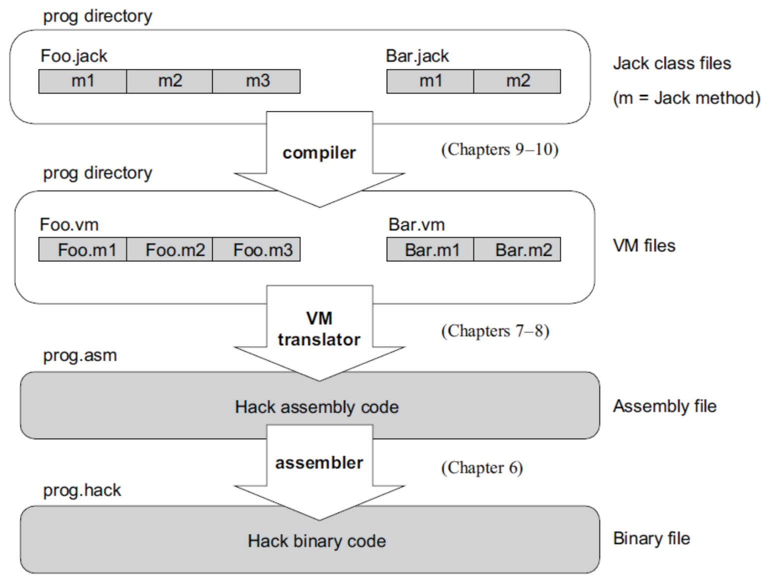


Figure 7.8 Program elements in the Jack-VM-Hack platform.

VM programming: multiple functions (memory)

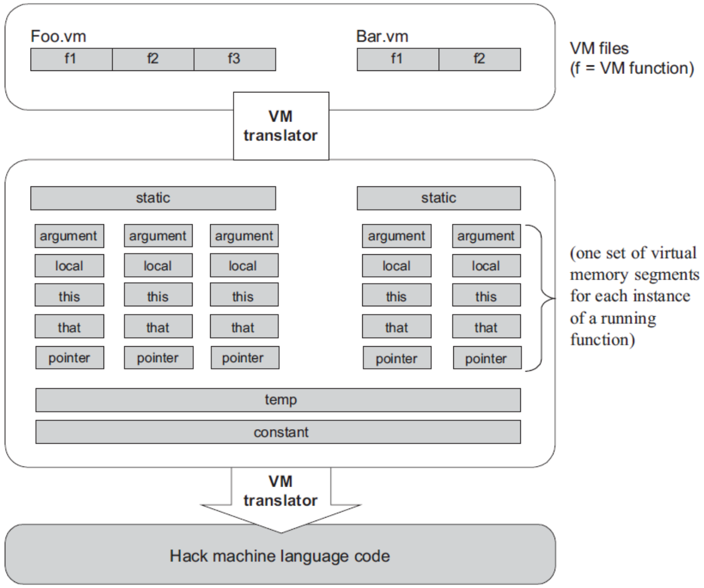


Figure 7.7 The virtual memory segments are maintained by the VM implementation.

A simple game: square

- (Demo)
- Use Square as an example.
- Design a class: think of its
  - States: data members
  - Behaviors: function members
- Square
  - x, y, size
  - MoveUp, MoveDown, IncSize, …

Perspective

- Jack is an object-based language: no inheritance
- Primitive type system (3 types)
- Standard library
- Our hidden agenda: gearing up to learn how to develop the ...
  - Compiler (projects 10 and 11)
  - OS (project 12).