Agenda

Jack: A simple, high-level programming language

- Hello World
- Basic language constructs
- Object-oriented programming: object and list examples

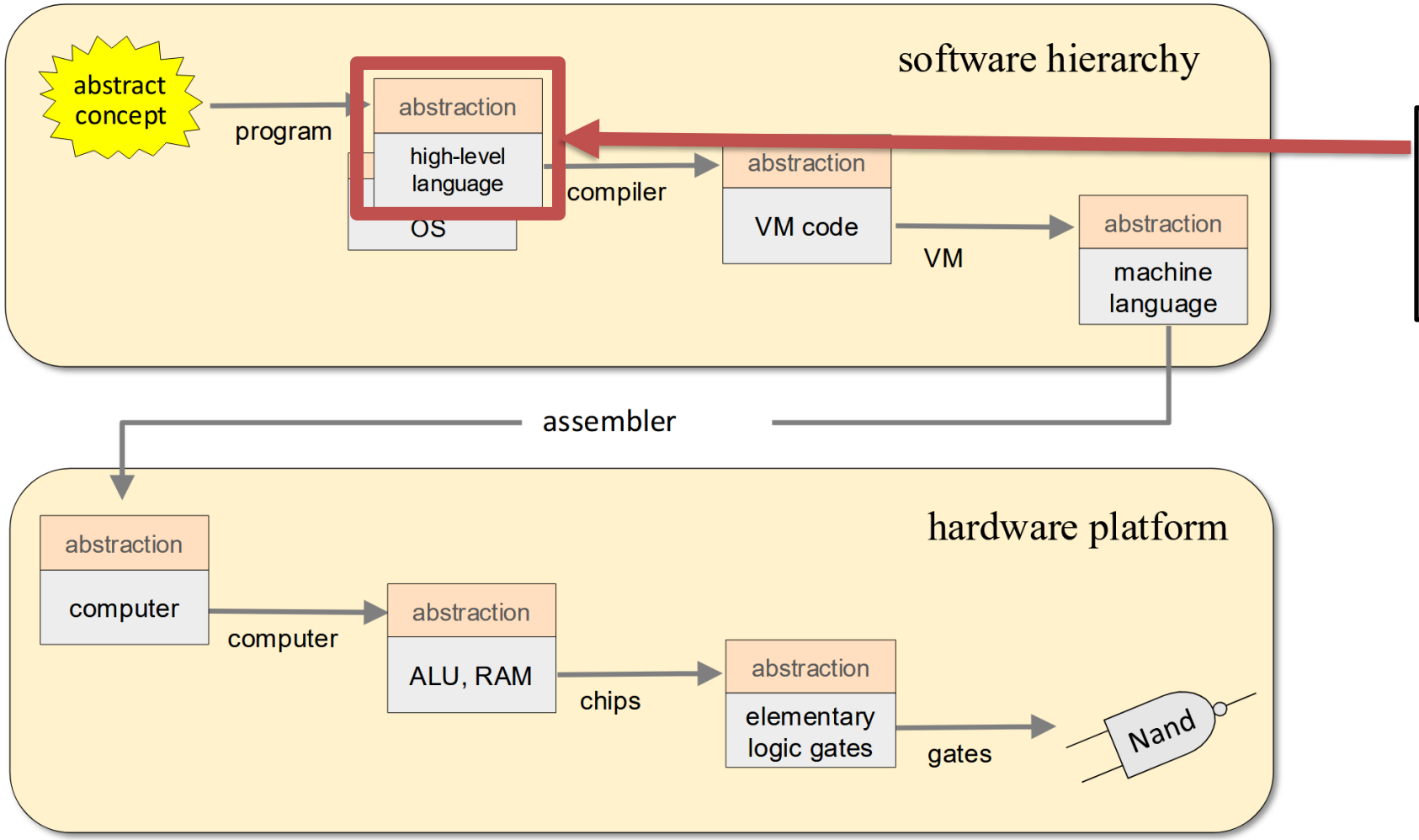
Formal specification

Built-in features (via OS system library)

- strings
- math
- graphics
- input/output

(with slides from nand2tetris)

The big picture: Hack



Next: A high-level language that makes programming easier

Goals

Understand how high-level languages ...

- handle primitive and class types
- deal with strings, arrays, and lists
- create, represent, and dispose objects
- interact with the host OS

Hello World

```
class Main {
  function void main() {
    do Output.printString("Hello World");
    do Output.println();
    return;
  }
}
```

Need to define one main function

OS files need to be included with the user program

Our convention will be to put applications in their own directory

To compile to vm code:

./JackCompiler.sh <DirectoryName>

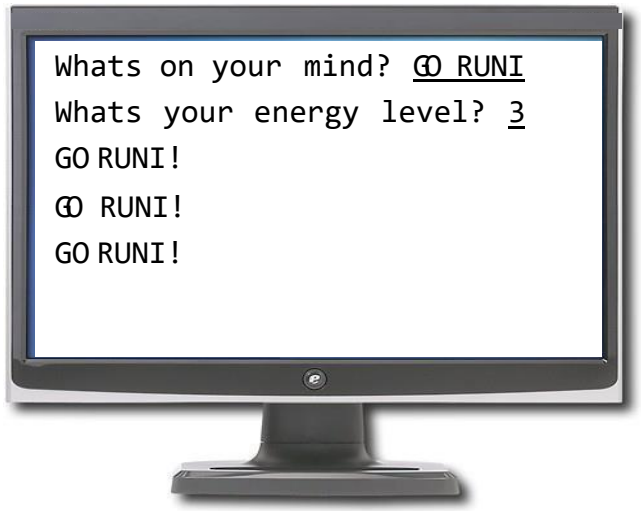
To test, open the directory in the VM emulator:

./**VMEmulator.sh**

Your goal is for your compiled machine code to run in your simulated computer.

Jack program example

```
/** Performs some interaction with the user.*/
class Main {
  function void main() {
    var String s;
    var int energy, i;
    let s = Keyboard.readLine("Whats on your mind?");
    let s = s.appendChar(33); // the character '!'
    let energy = Keyboard.readInt("Whats your energy level?");
    let i = 0;
    while (i < energy) {
       do Output.printString(s);
       do Output.println();
       let i = i + 1;
    }
    return;
  }
}
```



Basic language constructs

```
/** Performs some interaction with the user.*/
class Main {
  function void main() {
    var String s;
    var int energy, i;
    let s = Keyboard.readLine("Whats on your mind?");
    let s = s.appendChar(33); // the character '!'
    let energy = Keyboard.readInt("Whats your energy level?");
    let i = 0;
    while (i < energy) {
       do Output.printString(s);
       do Output.println();
       let i = i + 1;
    }
    return;
  }
}
```

Comments

```
/** API block comment */

/* block comment */

// comment to end of line
```

White space

(ignored)

Basic language constructs

```
/** Performs some interaction with the user.*/
class Main {
  function void main() {
    var String s;
    var int energy, i;
    let s = Keyboard.readLine("Whats on your mind?");
    let s = s.appendChar(33); // the character '!'
    let energy = Keyboard.readInt("Whats your energy level?");
    let i = 0;
    while (i < energy) {
       do Output.printString(s);
       do Output.println();
       let i = i + 1;
    }
    return;
  }
}
```

Program structure

Jack program: One or more Jack classes, one of which must be named `Main`

`Main` must have at least one function, named `main`

Program's entry point:
`Main.main`

Basic language constructs

```
/** Performs some interaction with the user.*/
class Main {
  function void main() {
    var String s;
    var int energy, i;
    let s = Keyboard.readLine("Whats on your mind?");
    let s = s.appendChar(33); // the character '!'
    let energy = Keyboard.readInt("Whats your energy level?");
    let i = 0;
    while (i < energy) {
       do Output.printString(s);
       do Output.println();
       let i = i + 1;
    }
    return;
  }
}
```

Data types

Built-in primitives:

- `int`
- `char`
- `boolean`

Class types:

- Standard library types, like `String`
- Programmer-defined types

Basic language constructs

```
/** Performs some interaction with the user.*/
class Main {
  function void main() {
    var String s;
    var int energy, i;
    let s = Keyboard.readLine("Whats on your mind?");
    let s = s.appendChar(33); // the character '!'
    let energy = Keyboard.readInt("Whats your energy level?");
    let i = 0;
    while (i < energy) {
       do Output.printString(s);
       do Output.println();
       let i = i + 1;
    }
    return;
  }
}
```

Control flow

- if
- while
- do

Basic language constructs

```
/** Performs some interaction with the user.*/
class Main {
  function void main() {
    var String s;
    var int energy, i;
    let s = Keyboard.readLine("Whats on your mind?");
    let s = s.appendChar(33); // the character '!'
    let energy = Keyboard.readInt("Whats your energy level?");
    let i = 0;
    while (i < energy) {
       do Output.printString(s);
       do Output.println();
       let i = i + 1;
    }
    return;
  }
}
```

Input / output

`Keyboard` (OS class):

library of methods for reading from the keyboard

`Output` (OS class):

library of methods for writing text to the screen

Lecture plan

High-level programming (tutorial)

- ✓ Program example
- ✓ Basic language constructs
- ➡ Object-based programming
  - Example: Points
  - Example: Lists

We assume basic OOP and recursion knowledge, at the level of an Introduction to CS course.

The Jack language (specification)

- The language
- The operating system

Application development (project 9)

- Jack applications
- Application example
- Graphics optimization

Procedural programming

```
/** Performs some interaction with the user.*/
class Main {
  function void main() {
    var String s;
    var int energy, i;
    let s = Keyboard.readLine("Whats on your mind?");
    let s = s.appendChar(33); // the character '!'
    let energy = Keyboard.readInt("Whats your energy level?");
    let i = 0;
    while (i < energy) {
       do Output.printString(s);
       do Output.println();
       let i = i + 1;
    }
    return;
  }
}
```

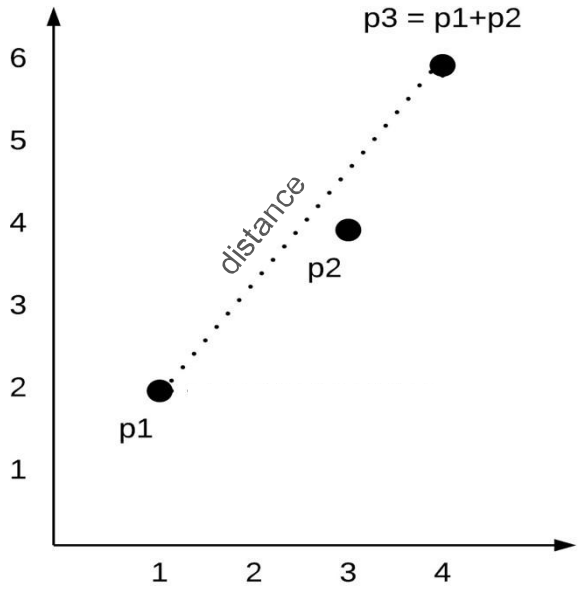
Simple program:

- Procedural
- One class / one function
- No objects

OO Programming

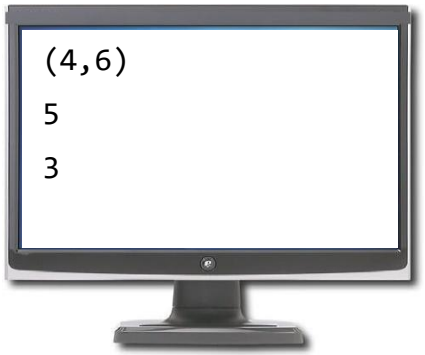
Example

A `Point` class that represents and manipulates 2D points



Client code

```
...
var Point p1, p2, p3;
let p1 = Point.new(1,2);
let p2 = Point.new(3,4);
let p3 = p1.plus(p2);
do p3.print();
do Output.printInt(p1.distance(p3));
do Output.printInt(Point.getPointCount());
...
```



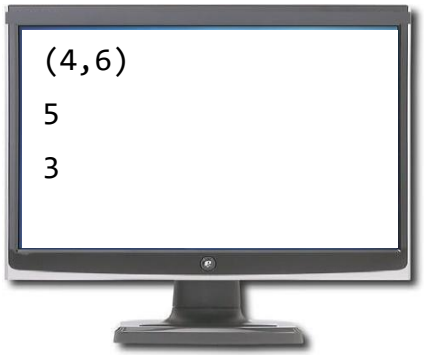
OO Programming

`Point` class API

```
/** Represents a 2D point. */
class Point {

    /** Constructs a point from the given coordinates
    */
    constructor Point new(int ax, int ay)

    /** Returns the number of points constructed so
    far */
    function int getPointCount()

    /** Returns the sum of this point and the other
    point */
    method Point plus(Point other)

    /** Returns the Euclidean distance between this
       point and the other point */
    method int distance(Point other)

    /** Prints this point, as "(x,y)" */
    method void print() {

    // More Point methods...

}
```

Client code (example)

```
...
var Point p1, p2, p3;
let p1 = Point.new(1,2);
let p2 = Point.new(3,4);
let p3 = p1.plus(p2);
do p3.print();
do Output.printInt(p1.distance(p3));
do Output.printInt(Point.getPointCount());
...
```



Uses `Point` as an abstract data type

OO Programming: Point class

```
/** Represents a 2D point. */
class Point {

    // The coordinates of this point:
    field int x, y;

    // The number of point objects constructed so far:
    static int pointCount;

    /** Constructs a point and
        initializes it with the given
        coordinates */
    constructor Point new(int ax, int ay) {
        let x = ax;
        let y = ay;
        let pointCount = pointCount + 1;
        return this;
    }

    /** Returns the x coordinate of this point */
    method int getx() { return x; }

    /** Returns the y coordinate of this point */
    method int gety() { return y; }

    /** Returns the number of Points constructed so far */
    function int getPointCount() {
        return pointCount;
    }

    // Class declaration continues on the right.
```

```
    /** Returns a point which is this
        point plus the other point */
    method Point plus(Point other) {
        return Point.new(x + other.getx(),
                         y + other.gety());
    }

    /** Returns the Euclidean distance between
        this point and the other point */
    method int distance(Point other) {
        var int dx, dy;
        let dx = x - other.getx();
        let dy = y - other.gety();
        return Math.sqrt((dx*dx) + (dy*dy));
    }

    /** Prints this point, as "(x,y)" */
    method void print() {
        do Output.printString("(");
        do Output.printInt(x);
        do Output.printString(",");
        do Output.printInt(y);
        do Output.printString(")");
        return;
    }

} // End of Point class declaration.
```

Client code (example)

```
...
var Point p1, p2, p3;
let p1 = Point.new(1,2);
let p2 = Point.new(3,4);
let p3 = p1.plus(p2);
do p3.print();
do Output.printInt(p1.distance(p3));
do Output.printInt(Point.getPointCount());
...
```

OO Programming: Point class

Point class

```
/** Represents a 2D point. */
class Point {

   // The coordinates of this point:
   field int x, y;

   // The number of point objects constructed so far:
   static int pointCount;

   /** Constructs a point and initializes it with the given coordinates */
   constructor Point new(int ax, int ay) {
       let x = ax; let y = ay;
       let pointCount = pointCount + 1; return this;
   }

   /** Returns the x coordinate of this point */
   method int getx() { return x; }

   /** Returns the y coordinate of this point */
   method int gety() { return y; }

   /** Returns the number of Points constructed so far */
   function int getPointCount() {
      return pointCount;
   }

   // Class declaration continues on the right.
```

Jack class declaration

A sequence of:

- *field* declarations,
- *static* variable declarations,
- *subroutine* declarations (constructors, methods, functions)

OO Programming: Point class

Point class

```
/** Represents a 2D point. */
class Point {
   // The coordinates of this point:
   field int x, y;
   // The number of point objects constructed so far:
   static int pointCount;
   /** Constructs a point and initializes it with the
       given coordinates */
   constructor Point new(int ax, int ay) {
       let x = ax; let y = ay;
       let pointCount = pointCount + 1; return
       this;
   }
   /** Returns the x coordinate of this point */
   method int getx() { return x; }
   /** Returns the y coordinate of this point */
   method int gety() { return y; }
   /** Returns the number of Points constructed so far */
   function int getPointCount() {
      return pointCount;
   }
   // Class declaration continues on the right.
```

Visibility

- All fields are private
- All methods are public

Accessing a field x

- Of the current object: Simply access x (same as accessing this.x)
- Of another object: Use a get / set method, e.g. anotherPoint.getx()

OO Programming: Creating objects

Point class

```
/** Represents a 2D point. */
class Point {

    // The coordinates of this point:
    field int x, y;

    // The number of point objects constructed so
    far:
    static int pointCount;

    /** Constructs a point and
        initializes it with the given
        coordinates */
   constructor Point new(int ax, int ay) {
         let x = ax;
         let y = ay;
         let pointCount = pointCount + 1;
         return this;
    }

    // More Point methods...
```

```
...
var Point p1, p2;
let p1 = Point.new(1,2);
let p2 = Point.new(3,4);
...
```

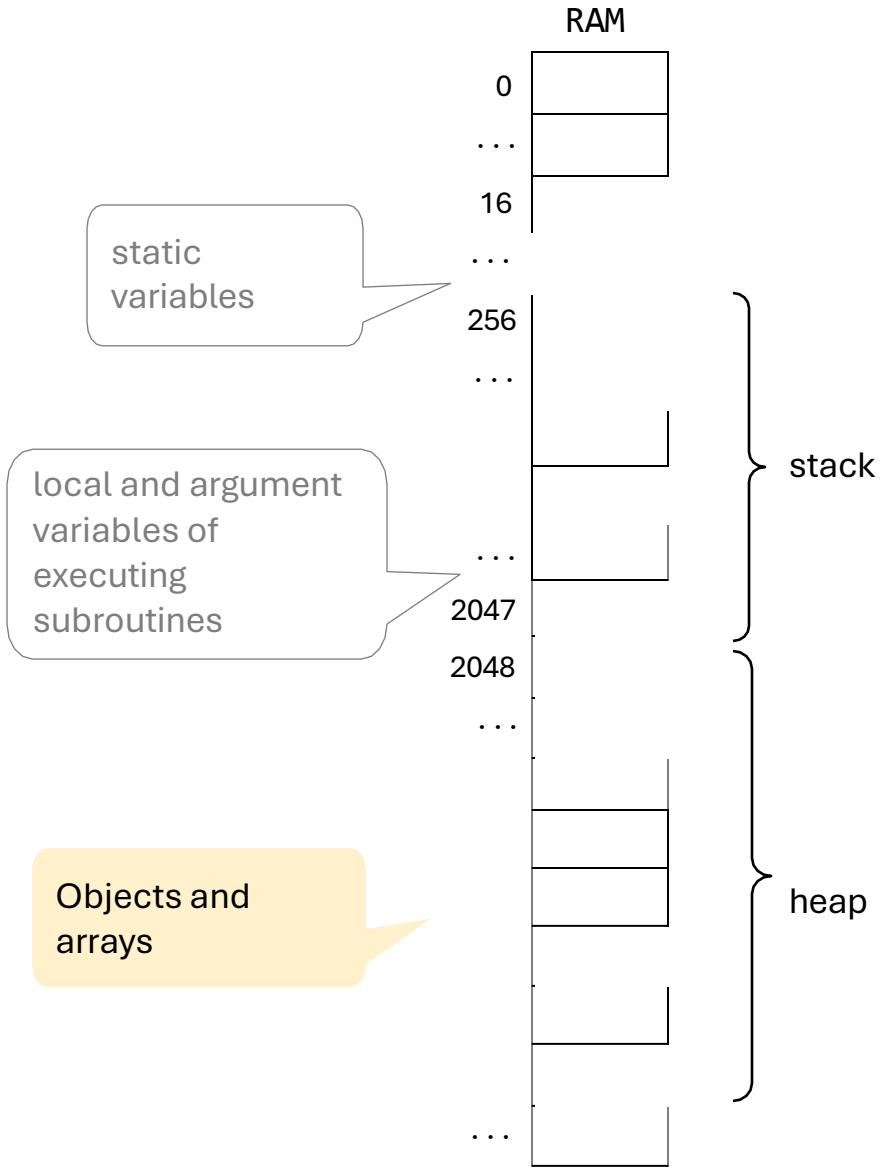
OO Programming: Creating objects

Point class

```
/** Represents a 2D point. */
class Point {

    // The coordinates of this point:
    field int x, y;

    // The number of point objects constructed so far:
    static int pointCount;

    /** Constructs a point and initializes it
        with the given coordinates */
    constructor Point new(int ax, int ay) {
        let x = ax;
        let y = ay;
        let pointCount = pointCount + 1;
        return this;
    }

    // More Point methods...
```

Client code

```
...
var Point p1, p2;
let p1 = Point.new(1,2);
let p2 = Point.new(3,4);
...
```



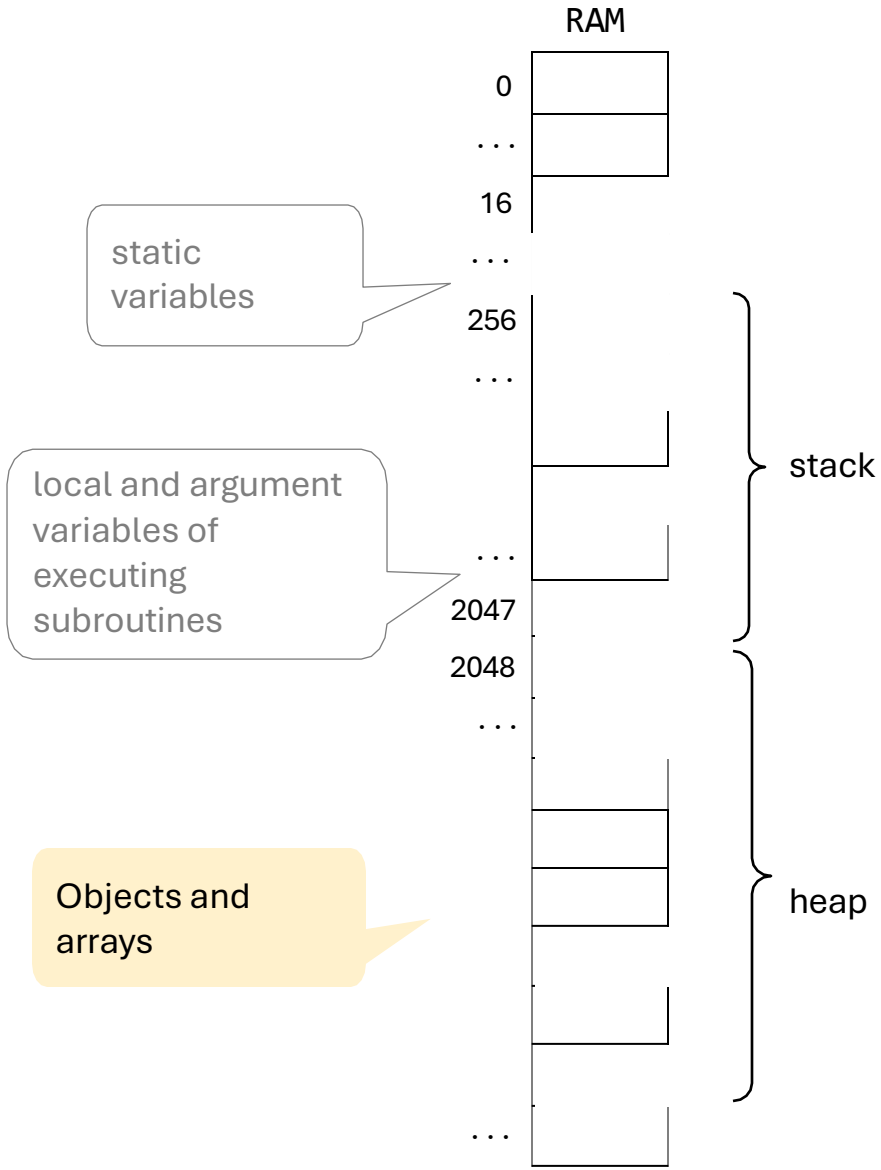
Example: Creating objects

Point class

```
/** Represents a 2D point. */
class Point {

    // The coordinates of this point:
    field int x, y;

    // The number of point objects constructed so far:
    static int pointCount;

    /** Constructs a point and initializes it
        with the given coordinates */
    constructor Point new(int ax, int ay) {
        let x = ax;
        let y = ay;
        let pointCount = pointCount + 1;
        return this;
    }

    // More Point methods...
```

Client code

```
...
var Point p1, p2;
let p1 = Point.new(1,2);
let p2 = Point.new(3,4);
...
```



Suppose:

SP is 256
LCL is 256
p1 (local 0) has location 2048
p2 (local 1) has location 2068.

Draw their data in RAM

OO Programming: Destroying objects

Point class

```
/** Represents a 2D point. */
class Point {

    // The coordinates of this point:
    field int x, y;

    // The number of point objects constructed so far:
    static int pointCount;

    /** Constructs a point and
        initializes it with the given
        coordinates */
    constructor Point new(int ax, int ay) {
        let x = ax;
        let y = ay;
        let pointCount = pointCount + 1;
        return this;
    }

    /** Disposes this point. */
    method void dispose() {
        // Calls an OS function that frees the memory
        // used by this object
        do Memory.deAlloc(this);.
        return;
    }
}
```

Best practice

1. When an object is no longer needed, dispose it
   (Jack has no garbage collection)
2. If you write a class that has one or more constructors, write also a dispose method.

Client code

```
...
var Point p1;
let p1 = Point.new(1,2);
...
// When the object is no longer needed:
do p1.dispose();
...
```

Lists

List: The value `null`, or a value followed by a list

list examples	commonly documented as
`null`	`()`
`(3, null)`	`(3)`
`(3, (5, null))`	`(3, 5)`
`(3, (5, (2, null)))`	`(3, 5, 2)`

List representation

`List` class

```
/** A linked list of integers. */
class List {

    /** Creates a List */
    constructor List new(int car, List cdr)

    /** Prints this list */
    method void print() {

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

    // More list processing methods
}
```

Abstraction

Lists are represented as *instances* (objects) of a `List` class;

List representation

`List` class

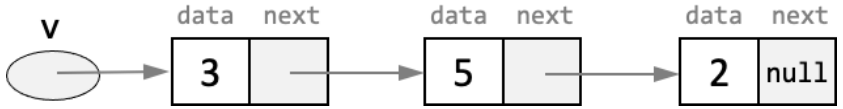
```
/** A linked list of integers. */
class List {
   field int data;    // an int value,
   field List next;  // followed by a list of int values

   /** Creates a List */
   constructor List new(int car, List cdr)

   /** Prints this list */
   method void print() {

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

   // More list processing methods
}
```

Abstraction

Lists are represented as *instances* (objects) of a `List` class;

Implementation

Recursive definition:
An `int`, followed by a `List`



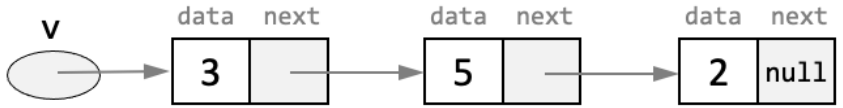
List processing

`List` class

```
/** A linked list of integers. */
class List {
   field int data;   // an int value,
   field List next; // followed by a list of int values

   /** Creates a List */
   constructor List new(int car, List cdr)

   /** Prints this list */
   method void print() {

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

   // More list processing methods
}
```

Client code

```
// Builds, prints, and disposes a list
class Main {
   function void main() {
      // Creates the list (3, 5, 2)
      var List v;
      let v = List.new(2,null);
      let v = List.new(5,v);
      let v = List.new(3,v);
      do v.print(); // Prints 3, 5, 2
      do v.dispose();
      return;
   }
}
```



We'll discuss methods for:

- Constructing a list
- Iterating a list
- Disposing s list

<u>The goal</u>

Illustrating how *objects* are created and managed.

List construction

`List` class

```
/** A linked list of integers. */
class List {
   field int data;   // an int value,
   field List next; // followed by a list of int values

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

   // More List methods
}
```

Client code

```
// Builds, prints, and disposes a list
class Main {
   function void main() {
      // Creates the list (3, 5, 2)
      var List v;
      let v = List.new(2,null);
      let v = List.new(5,v);
      let v = List.new(3,v);
      ...
      ...
      ...
   }
}
```

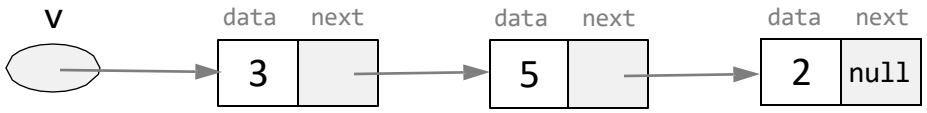
Visualize the creation of the linked list

List iteration

```
/** A linked list of integers. */
class List {
   field int data;      // an int value,
   field List next;     // followed by a list of int values.

   /** Constructor (code omitted) */

   /** Accessors */
   method int getData() { return data; }
   method int getNext() { return next; }

   /** Prints this list */
   method void print() {
      var List current;    // creates a List variable and initializes
      let current = this;  // it to the first element of this list

      while (~(current = null)) {
         do  Output.printInt(current.getData());
         do Output.printChar(32);    // prints  space
         let current = current.getNext();
      }
      return;
   } // More List methods...
```

client code

```
...
var List v;
// populates the list (code omitted)
...
do v.print();
do Output.printChar(33); // prints '!'
...
```



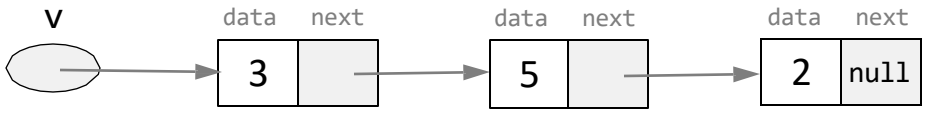
List iteration

```
/** A linked list of integers. */
class List {
   field int data;      // an int value,
   field List next;     // followed by a list of int values.

   /** Constructor (code omitted) */

   /** Accessors */
   method int getData() { return data; }
   method int getNext() { return next; }

   /** Prints this list */
   method void print() {
      var List current;    // creates a List variable and initializes
      let current = this;  // it to the first element of this list

      while (~(current = null)) {
         do  Output.printInt(current.getData());
         do Output.printChar(32);    // prints  space
         let current = current.getNext();
      }
      return;
   } // More List methods...
```

client code

```
...
var List v;
// populates the list (code omitted)
...
do v.print();
do Output.printChar(33); // prints '!'
...
```

Visualize the execution of the print method



List disposal: Recursive access

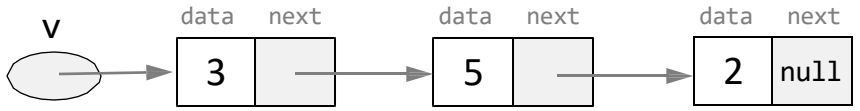
`List` class

```
/** A linked list of integers. */
class List {
   field int data;    // an int value
   field List next;    // followed by a list of int values

   // Constructor and other List methods (code omitted)

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Client

```
var List v;
// builds the list (2, 3, 5), code
omitted
...
do v.dispose();
...
```

We'll use the dispose method to illustrate recursive list processing.



List disposal: Recursive access

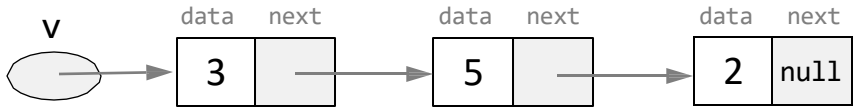
`List` class

```
/** A linked list of integers. */
class List {
   field int data;    // an int value
   field List next;    // followed by a list of int values

   // Constructor and other List methods (code omitted)

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Client

```
var List v;
// builds the list (2, 3, 5), code
omitted
...
do v.dispose();
...
```

Visualize the execution of dispose



Syntax

```
/** Procedural processing example */
class Main {
    /* Inputs numbers and computes their average */
    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 a number: ");
         let sum = sum + a[i];
         let i = i + 1;
      }
      ...
}
```

Syntax elements

- White space / comments
- keywords
- Symbols
- Constants
- Identifiers

Syntax

```
/** Procedural processing example */
class Main {
   /* Inputs numbers and computes their average */
   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 a number: ");
        let sum = sum + a[i];
        let i = i + 1;
     }
     ...
}
```

Syntax elements

- White space / comments
- keywords
- Symbols
- Constants
- Identifiers

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 */
```

Syntax

```
/** Procedural processing example */
class Main {
   /* Inputs numbers and computes their average */
   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 a number: ");
        let sum = sum + a[i];
        let i = i + 1;
     }
     ...
}
```

Syntax elements

- White space / comments
- keywords
- Symbols
- Constants
- Identifiers

`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

Syntax

```
/** Procedural processing example */
class Main {
   /* Inputs numbers and computes their average */
   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 a number: ");
        let sum = sum + a[i];
        let i = i + 1;
     }
     ...
}
```

Syntax elements

- White space / comments
- keywords
- Symbols
- Constants
- Identifiers

() 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.

Syntax

```
/** Procedural processing example */
class Main {
    /* Inputs numbers and computes their average */
    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 a number: ");
         let sum = sum + a[i];
         let i = i + 1;
      }
      ...
}
```

Syntax elements

- White space / comments
- keywords
- Symbols
- Constants
- Identifiers

- *Integer constants* are values in the range `0` to `32767`. 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 double quote (") characters and may contain any character except newline or double quote, which can be obtained by calling the OS functions `String.newLine()` and `String.doubleQuote()`.
- *Boolean constants* can be `true` or `false`.
- The constant `null` represents a null reference.

Syntax

```
/** Procedural processing example */
class Main {
    /* Inputs numbers and computes their average */
    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 a number: ");
         let sum = sum + a[i];
         let i = i + 1;
      }
      ...
}
```

Syntax elements

- White space / comments
- keywords
- Symbols
- Constants
- Identifiers

Identifiers are composed from arbitrarily long sequences of letters (A to Z, a to z), digits (0 to 9), and "_". The first character must be a letter or "_".

The language is case sensitive: x and X are treated as different identifiers.

Variables

Variable kind	Description	Declared in	Scope
static variables	`static` *type varName*1, *varName2*, … ; 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 varName*1, *varName2*, … ; Every object (instance of the class) has a private copy of the field variables (like *member variables* in Java)	class declaration	The class in which they are declared, except for functions, where they are undefined.
local variables	`var` *type varName*1, *varName2*, … ; Local variables are created just before the subroutine starts running and are disposed when it returns (like *local variables* in Java)	subroutine declaration	The subroutine in which they are declared.
parameter variables	*type varName*1, *varName2*, … Used to pass arguments to the subroutine. Treated like local variables whose values are initialized "from the outside", just before the subroutine starts running.	subroutine signature	The subroutine in which they are declared.

Statements

Statement	Syntax	Description

`let`	`let` *varName* `=` *expression*`;`
	or
	`let` *varName*`[`*expression1*`] =`
	expression2`;`
		An assignment operation (where *varName* is either single-valued or an array). The variable kind may be *static*, *local*, *field*, or *parameter*.

`if`	`if (`*expression*`) {`
	statements1
	`}`
	`else {`
	statements2
	`}`
		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`.

Expressions

A *Jack expression* is one of the following:

- A *constant*
- A *variable name* in scope. The variable may be *static*, *field*, *local*, or *parameter*
- The `this` keyword, denoting the current object (cannot be used in functions)
- An *array element* using the syntax *Arr*`[`*expression*`]`,
  where *Arr* is a variable name of type `Array` in scope
- A *subroutine call* that returns a non-void type
- An expression prefixed by one of the unary operators `-` or `~`:
  `-` *expression*: arithmetic negation
  `~` *expression*: boolean negation (bit-wise for integers)
- An expression of the form *expression op expression*
  where *op* is one of the following binary operators:
  `+ - * /` Integer arithmetic operators
  `& |` Boolean And and Boolean Or (bit-wise for integers) operators
  `< > =` Comparison operators
- `(`*expression*`)`: An expression in parenthesis

Exercise: What type of variable?

Exercise: What type of expression?

The Hack character set

key	code
(space)	32
!	33
"	34
#	35
$	36
%	37
&	38
‘	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

key	code
0	48
1	49
…	…
9	57

:	58
;	59
<	60
=	61
>	62
?	63
@	64

key	code
A	65
B	66
C	…
…	…
Z	90

[	91
/	92
]	93
^	94
_	95
`	96

key	code
a	97
b	98
c	99
…	…
z	122

{	123
|	124
}	125
~	126

key	code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
…	…
f12	152

Strings

Examples

```
...
var String s;   // Creates an object variable (pointer)
var char c;     // Creates a primitive variable
...

// Sets s to "ABC"
let s = String.new(3);
let s = s.appendChar(65);
let s = s.appendChar(66);
let s = s.appendChar(67);

// Alternatively, the Jack compiler allows:
let s = "ABC";

// Gets the value at index 1 (the char code of
'B')let c = s.charAt(1);
```

Exact specs: OS `String` class API.

Classes declaration

```
class Foo {
    field variable declarations
    static variable
    declarations subroutine
    declarations
}
```

- Jack program = collection of one or more Jack classes
- Class = basic compilation unit
- Each class `Foo` is stored in a separate `Foo.jack` file
- The class name's first character must be an uppercase letter.

Subroutine declaration

```
constructor | method | function type subroutineName (parameter-list) {
    local variable declarations
    statements
}
```

Jack subroutines

Constructors: create new objects

Methods: operate on the current

object **Functions**: static methods

Subroutine types and return values

Method and function type can be either `void`, a primitive data type, or a class name;

Each subroutine must end with the statement `return` *value*, or `return`.

Subroutine calls

Subroutine call syntax: *subroutineName*(*argument-list*)

The number and type of arguments must agree with those of the subroutine's parameters; Each argument is an expression of unlimited complexity.

Examples:

```
class Foo {
    ...
    method void f() {
       var Bar b;          // Declares a local variable of class type Bar
       var int i;          // Declares a local variable of primitive type
       ...                 int

       do Foo.p(3);        // Calls function p of the current class;
                           // (a function name must include the class name)
       do g();             // Calls method g of the current class on the this object;
                           // Note: Cannot be called from within a function (static
                           method)
       do Bar.h();         // Calls function h of class Bar

       let b = Bar.r();    // Calls function or constructor r of class Bar

       do b.q();           // Calls method q of class Bar on object b (which is of type
       Bar)

       ...
```

Arrays

Examples

```
...
var Array arr;
var String s;
let s = "Hello World!"
...
let arr = Array.new(4);
let arr[0] = 12;
let arr[1] = false;
let arr[2] = Point.new(5,6);
let arr[3] = s;
...
```

Jack arrays are ...

- Implemented as instances (objects) of an OS class named `Array`
- Untyped

A multi-dimensional array is obtained by creating an array of arrays.

Casting

Characters and integers can be converted into each other:

```
var char c;
// Sets c to 'A'
let c = 'A'; // Not supported by the Jack language
let c = 65;  // 'A'
// Sets c to 'A' (workaround, if needed)
var String s;
let s = "A";  let c = s.charAt(0);
```

An integer can be assigned to a reference variables, in which case it is treated as a memory address:

```
var Array arr;       // Creates a pointer variable
let arr = 5000;      // OK...
let arr[100] = 17; // Sets RAM[5100] to 17
```

An object can be converted into an array, and vice versa:

```
var Array arr;
let arr = Array.new(2);
let arr[0] = 2; let arr[1] = 5;
var Point p;     // A Point object has two int coordinates
let p = arr;     // Sets p to the base address of the memory block representing the array [2,5]
do p.print()     // Prints "(2,5)"  (using the print method of the Point class)
```

Particular syntax features

`var`: must precede local variable declaration: `var int x;`

`let`: must precede assignments: `let x = 0;`

`do`: must precede calling a method or a function outside an expression: `do reduce();`

`{ }`: must be used to enclose a code block, even if it contains a single statement:
`if (a > 0) { return a; } else { return -a; }`

A function or a method declaration must end with `return;` or with `return` *expression*`;`

A constructor declaration must end with `return this;`

No operator priority:

The value of this expression is unpredictable: `2 + 3 * 4`

To enforce operator priority, use parentheses: `2 + (3 * 4)`

Jack is weakly typed, allowing exotic casting.

Some of these features...

- Make the writing of Jack programs (project 9) a bit harder;
- Make the writing of a Jack compiler (projects 10-11) much easier.

The Jack OS standard class library

```
// Inputs numbers and computes their average
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 a number:");
        let sum = sum + a[i];
        let i = i + 1;
     }
     do Output.printString("The average is ");
     do Output.printInt(sum / length);
     return;
   }
}
```

Most programming languages (Java, Python, Jack...) are simple

What makes them powerful and seemingly complex is an open-ended *standard class library*

The standard class library can be seen as a language-specific OS.

The Jack OS

Jack OS

A collection of supplied Jack classes;

Implements a standard class library (similar (in concept) to Java's standard class library)

Implemented in Jack (.vm files that are compiled into your executable)

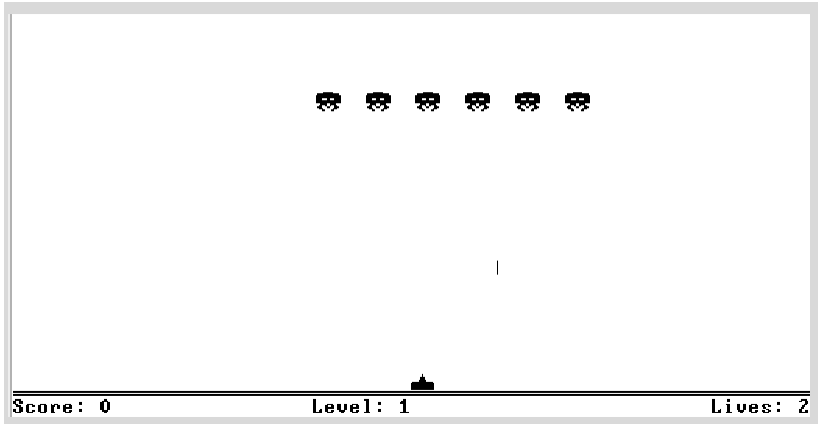
Purpose

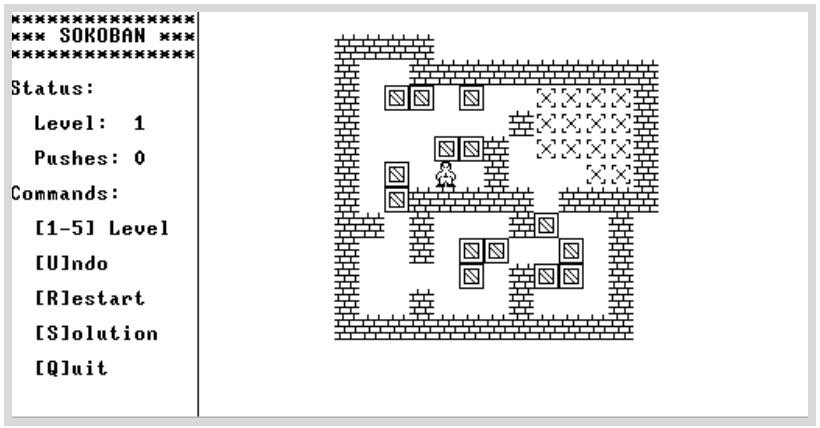
Closes gaps between high-level programs and the host hardware;

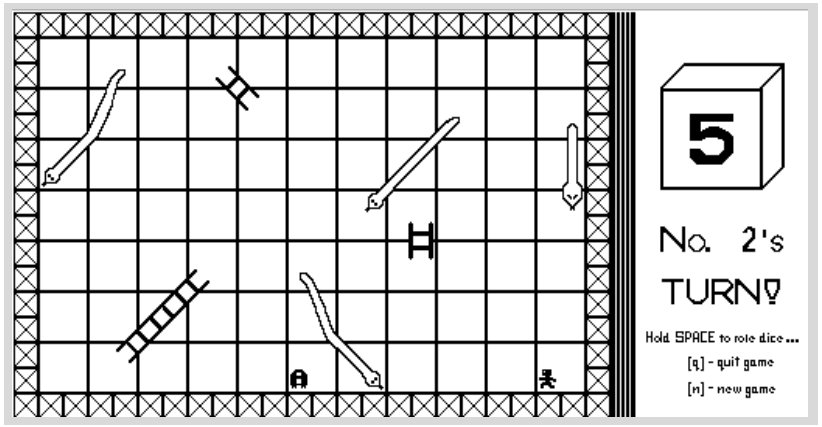
Provides efficient implementations of commonly-used functions and ADTs.

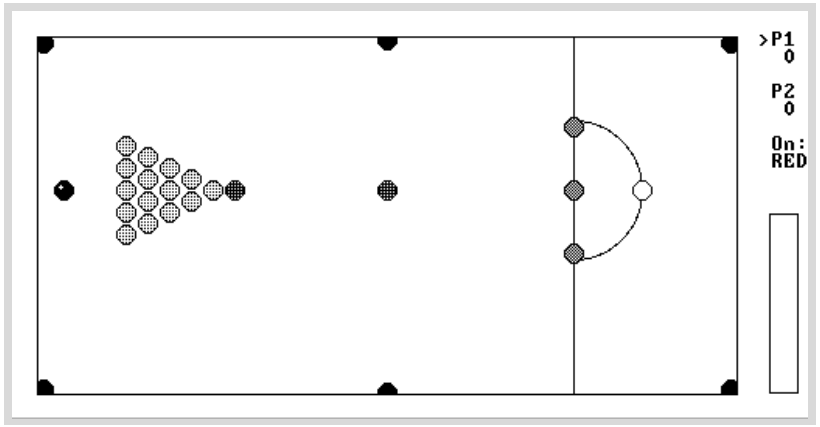
Functionality: Sys, Math, String, Keyboard, Output, Screen, Array, Mem (See the book for details)

Sample Jack programs









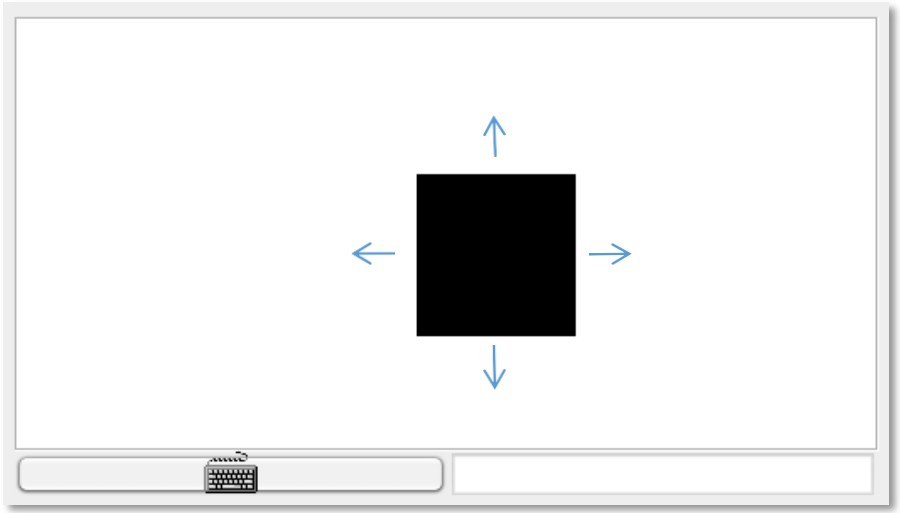
More examples: Search “Nand to Tetris game” in Youtube.

Program example: Square Dance

`Square`: Represents a graphical `Square` object;

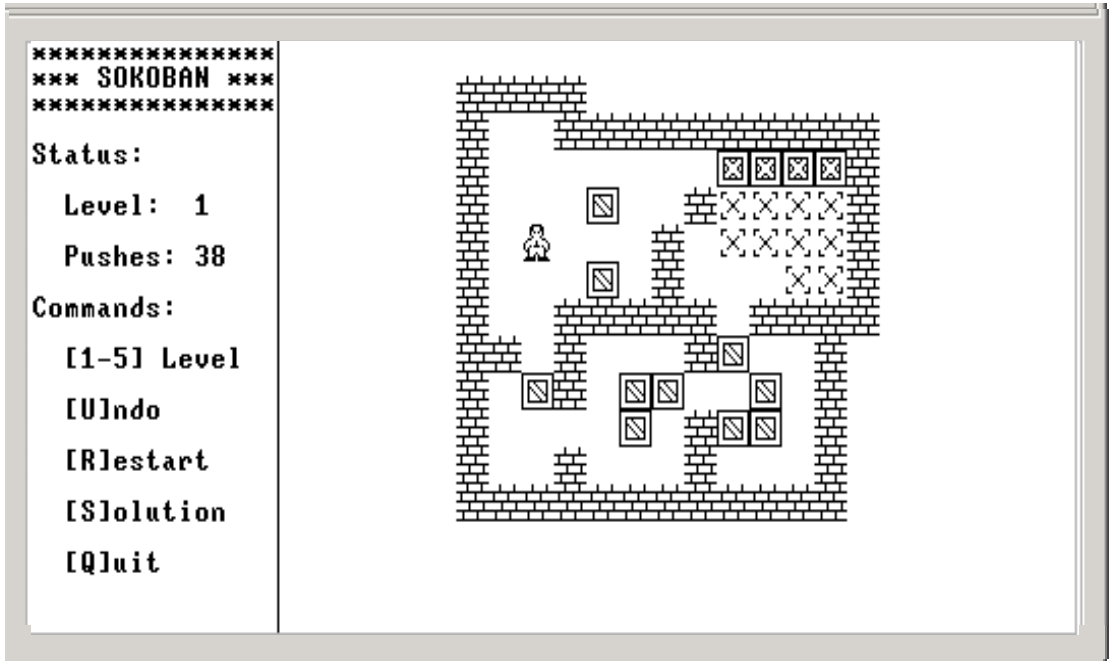
`SquareGame`: Creates a `Square`, then enters a loop that captures, and responds to, the user's inputs;

`Main:` Creates a `SquareGame`, and launches the game.

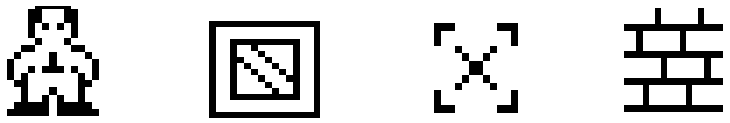


Optimizing graphics: Sprites

Sokoban (by Golan Parashi)



Basic graphical elements (*sprites*):



Standard drawing: Using the OS library `Screen`

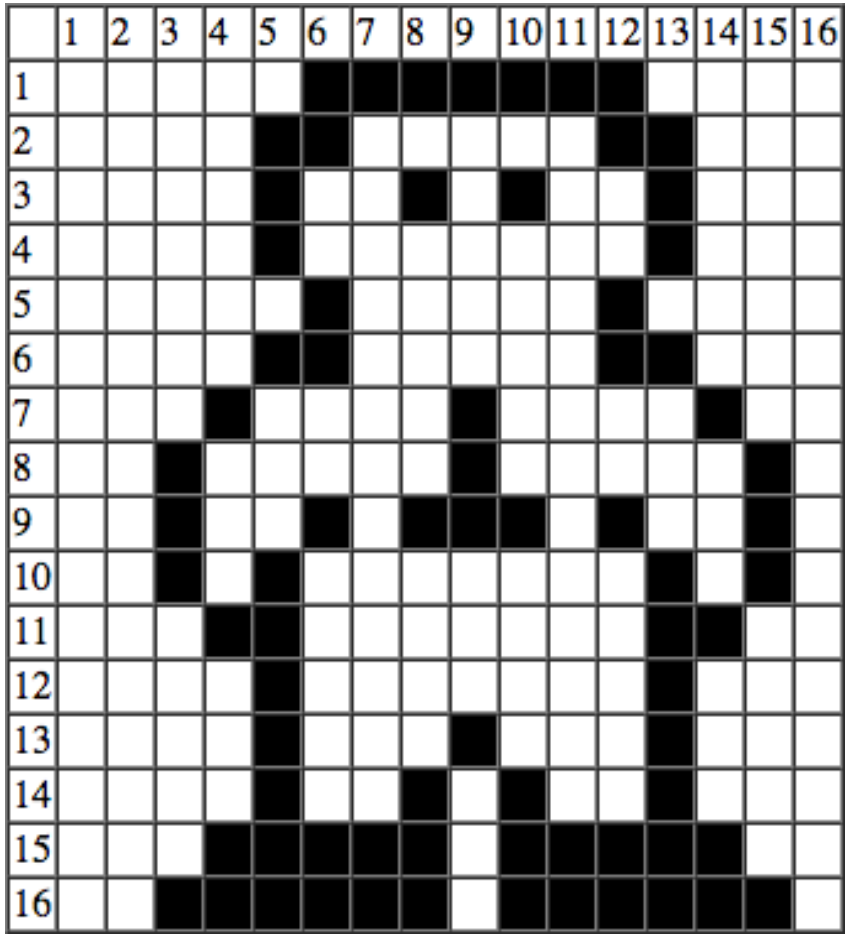


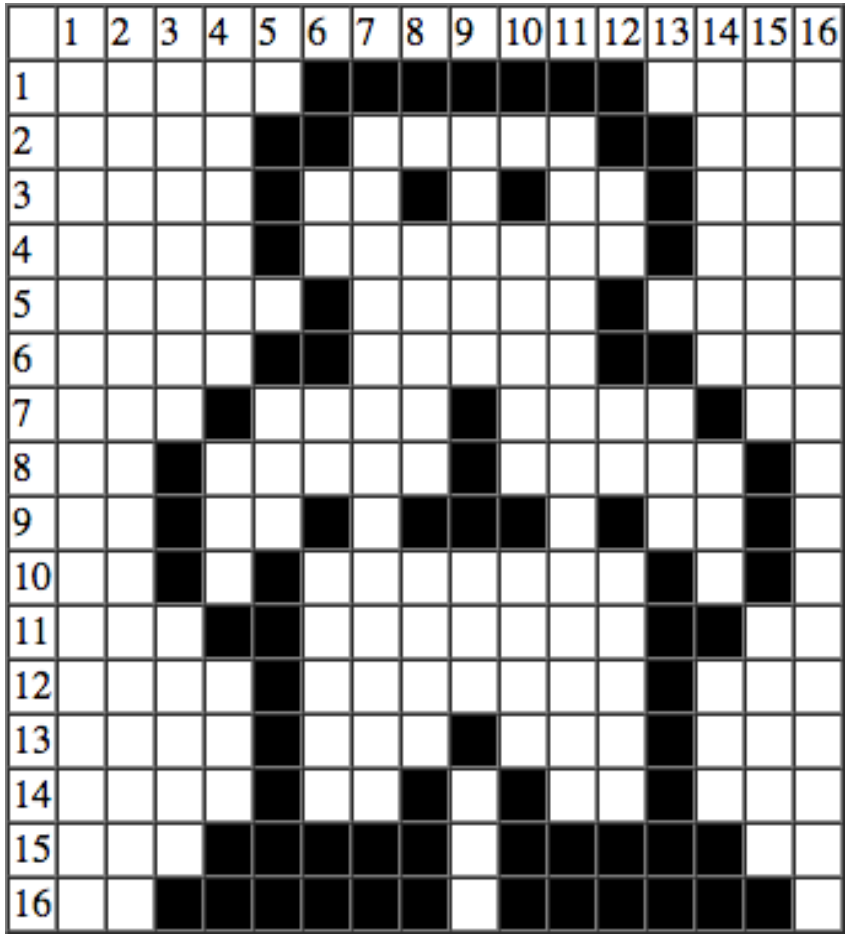
Image drawing code

```
// Draws the top row
do Screen.drawPixel(6,1);
do Screen.drawPixel(7,1);
...
do Screen.drawPixel(12,1);
...
// Draws the bottom row
do Screen.drawPixel(3,16);
...
do Screen.drawPixel(15,16);
```

Efficiency

75 pixel drawing operations

Optimized drawing: Writing directly to the screen memory map



```
0000111111100000 = 4064
0001100000110000 = 6192
0001001010010000 = 4752
...
```

Image drawing code

```
// Draws the sprite
do Memory.poke(addr0, 4064);
do Memory.poke(addr1, 6192);
do Memory.poke(addr2, 4752);
...
```

Efficiency

16 memory write operations

Best Practice

For simple graphics, use the standard OS library (`Screen`)

For high-performance sprites, use the bitmap editor, and `Memory.poke`