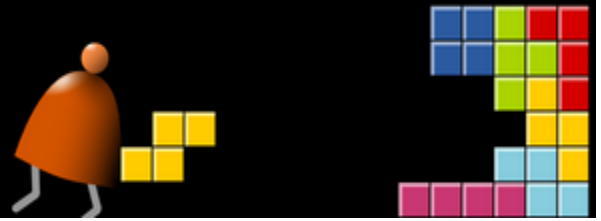


Lecture 9

High-Level Language

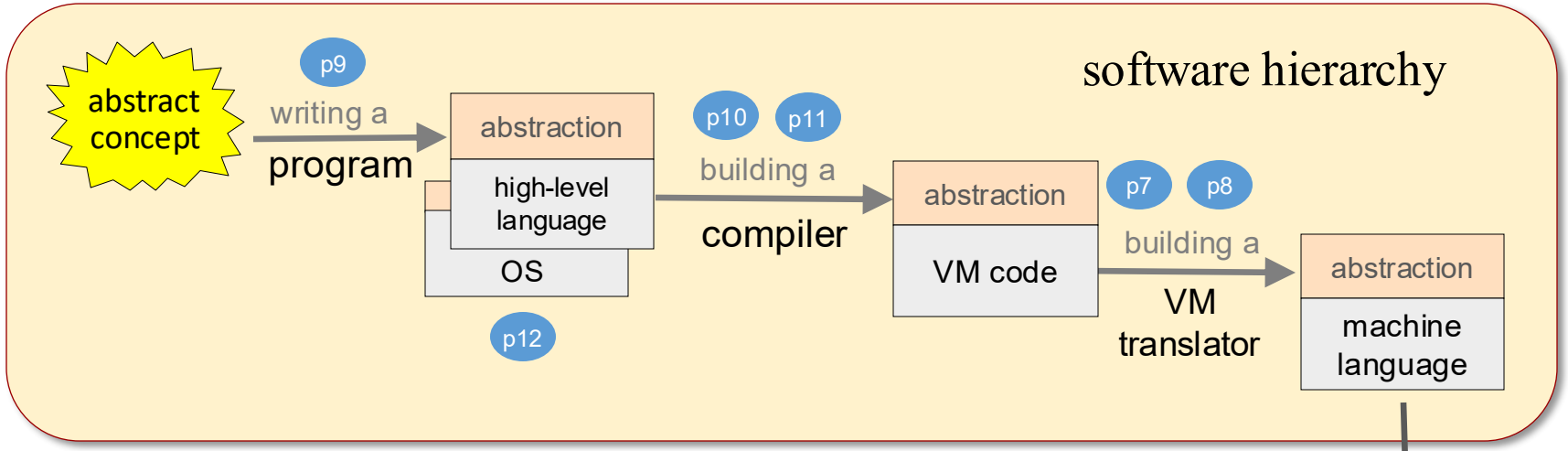
Slide deck for Chapter 9 of the book

The Elements of Computing Systems (2nd edition)

By Noam Nisan and Shimon Schocken

MIT Press

Nand to Tetris Roadmap: Part II



Work plan

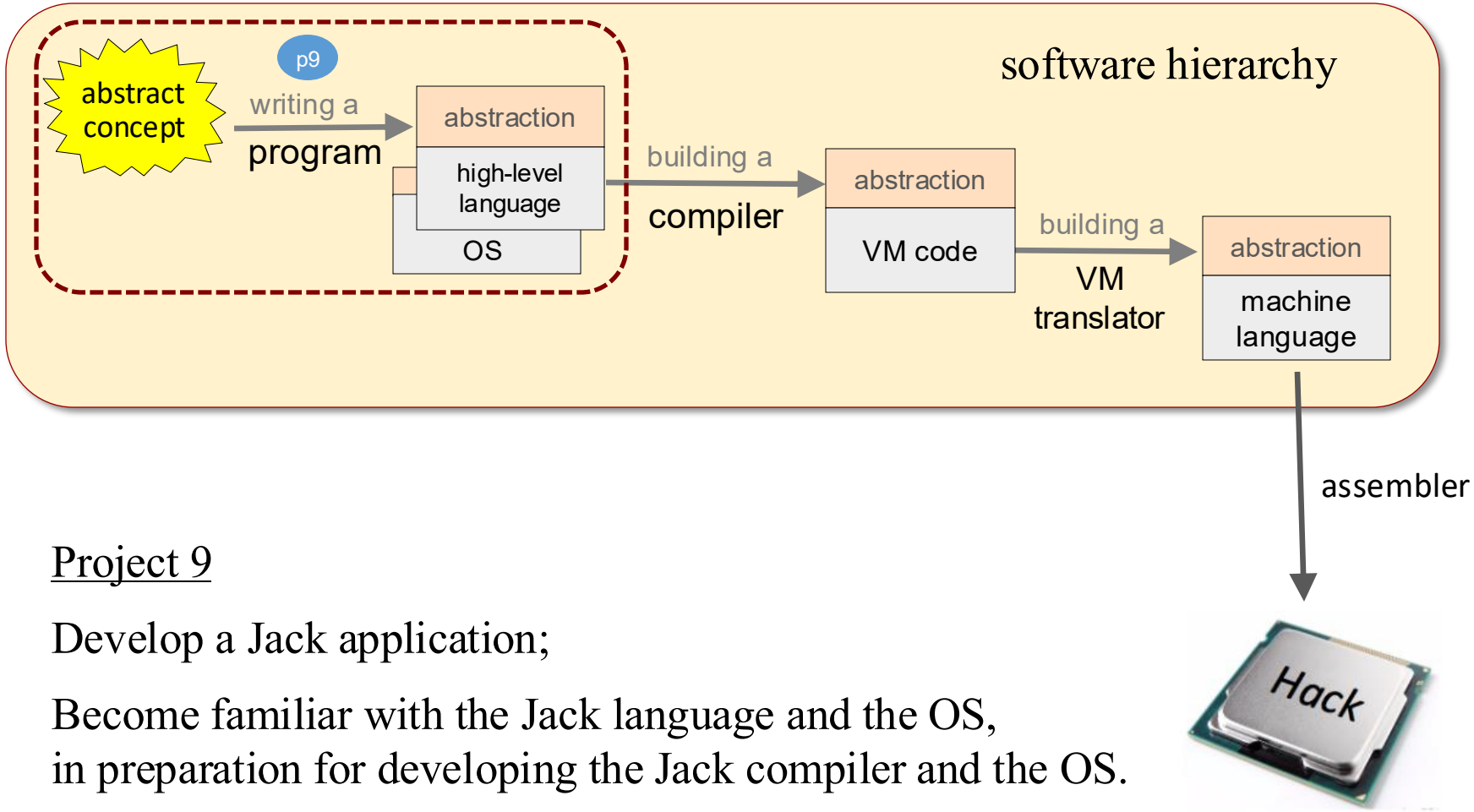
Projects 7,8: VM translator (compiler's backend)

Projects 10,11: Compiler (compiler's frontend)

Project 9: High-level language

Project 12: Operating system.

Nand to Tetris Roadmap: Part II



Project 9

Develop a Jack application;

Become familiar with the Jack language and the OS,
in preparation for developing the Jack compiler and the OS.

Take home lessons

Jack is a cool language

That can be learned in one hour.
But, the goal is not learning Jack. The goal is to ...

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

Plus, get a hands-on experience with ...

- Abstraction / implementation
- OO programming
- Application design
- Optimization.

Lecture plan

High-level programming (tutorial)

- Program example
- Basic language constructs
- Object-based programming

The Jack language (specification)

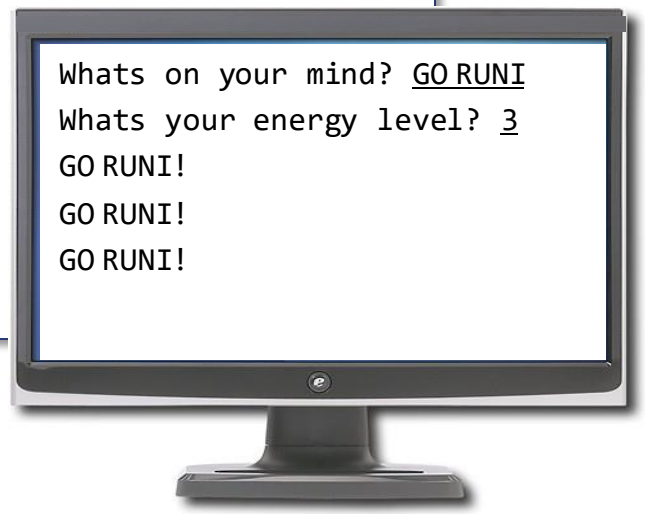
- The language
- The operating system

Application development (project 9)

- Jack applications
- Application example
- Graphics optimization

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;
  }
}
```



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;
  }
}
```

Jack:

A simple, Java-like language,

Object-based,

Multi-purpose,

Interactive.

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;
  }
}
```

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

Primitive:

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

Class types:

- Standard library types, like `String`;
- Programmer-defined types,
  Defined and used as needed.

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`

(used to call methods outside an expression)

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

The Jack language (specification)

- The language
- The operating system

Application development (project 9)

- Jack applications
- Application example
- Graphics optimization

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;
  }
}
```

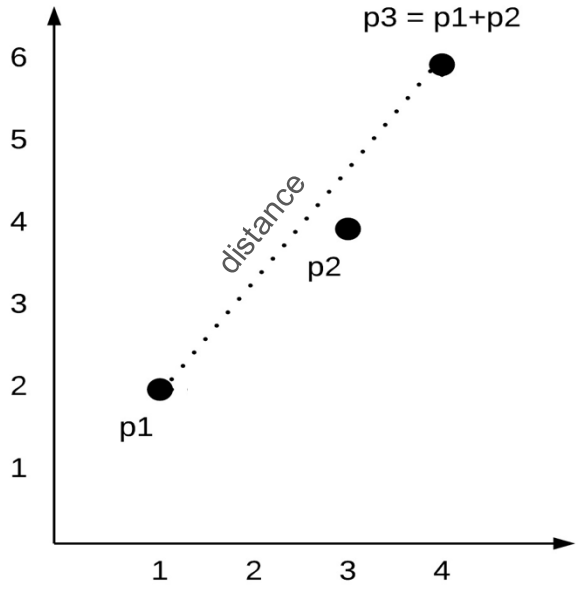
Simple program:

- Procedural
- One class / one function
- No objects

OO Programming

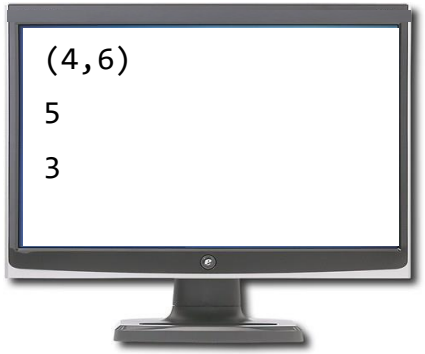
Example

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



Client code (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 API

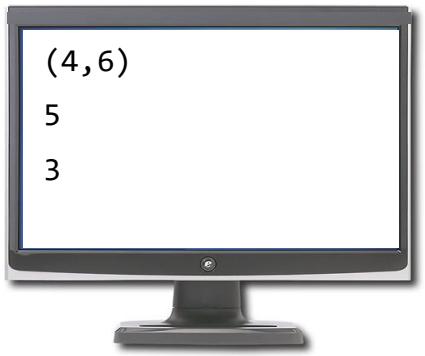
```
/** Represents a 2D point. */
class Point {

    /** 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…

}
```

Point abstraction

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 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 construct
    function int getPointCou

    /** Returns the s
    metho

                                    istance between this point
                        oint */
     int distance(Point other)

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

    // More Point methods…

}
```

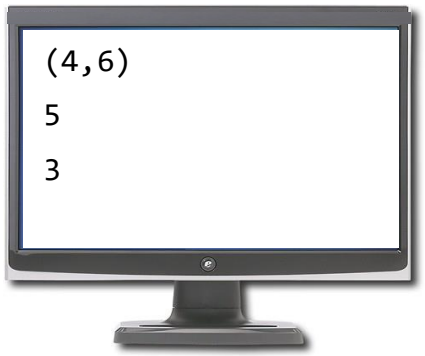
Let's open the black box…

Point abstraction

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

```
/** 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 th
```

object properties

class variables

Constructors

Create and initialize objects;

A Jack class has 0 or more constructors; one of them is normally named `new`.

Methods

Operate on the current object

Functions

Static methods that operate on no particular object

Jack class declaration

A sequence of:

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

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.
```

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…
```

Client code

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

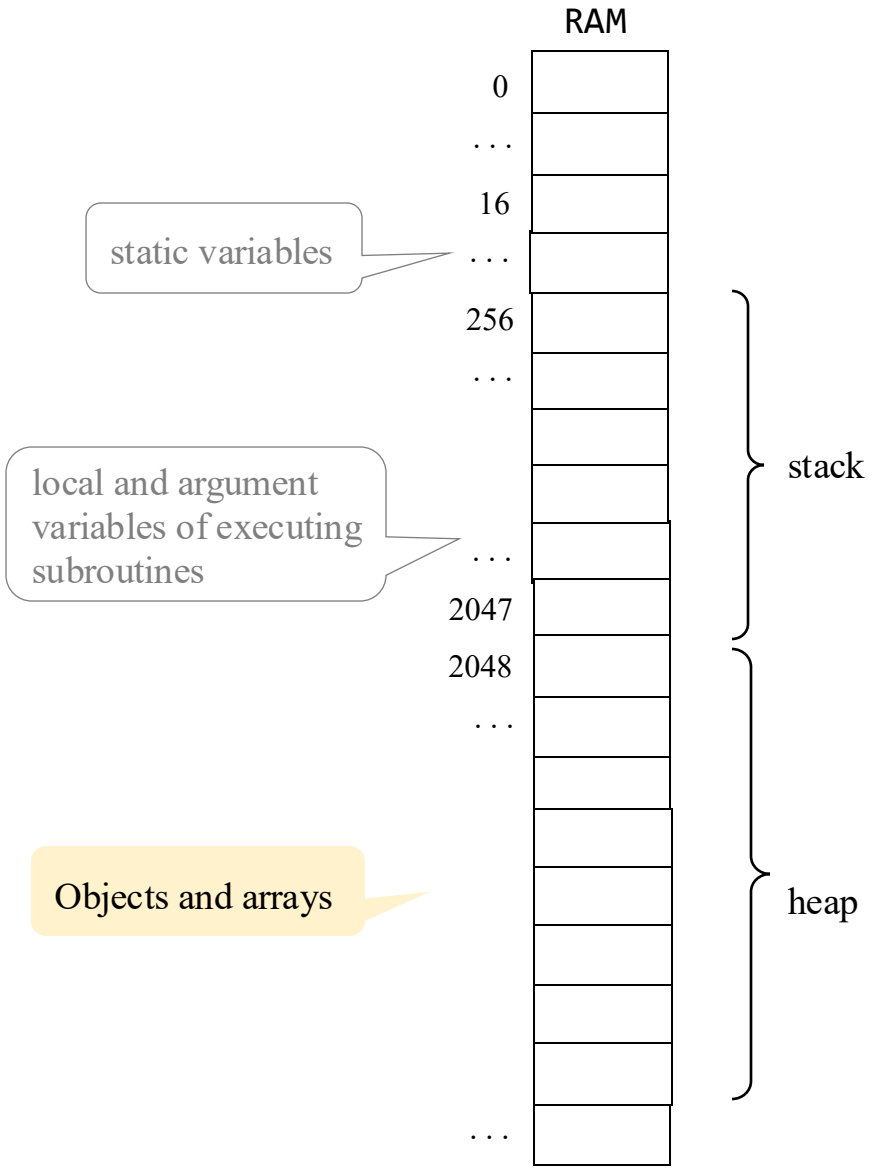
OO Programming: Creating objects

`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);
...
```



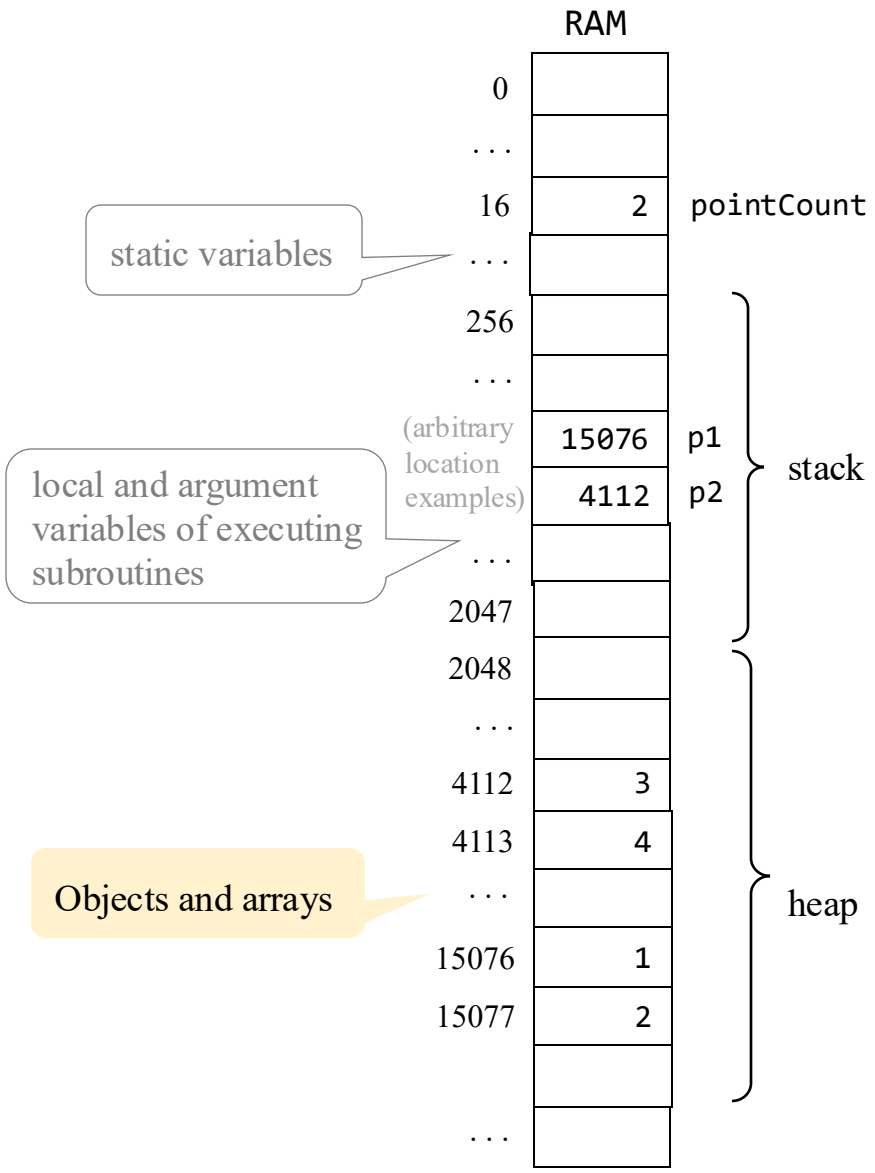
OO Programming: Creating objects

`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);
...
```



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…
```

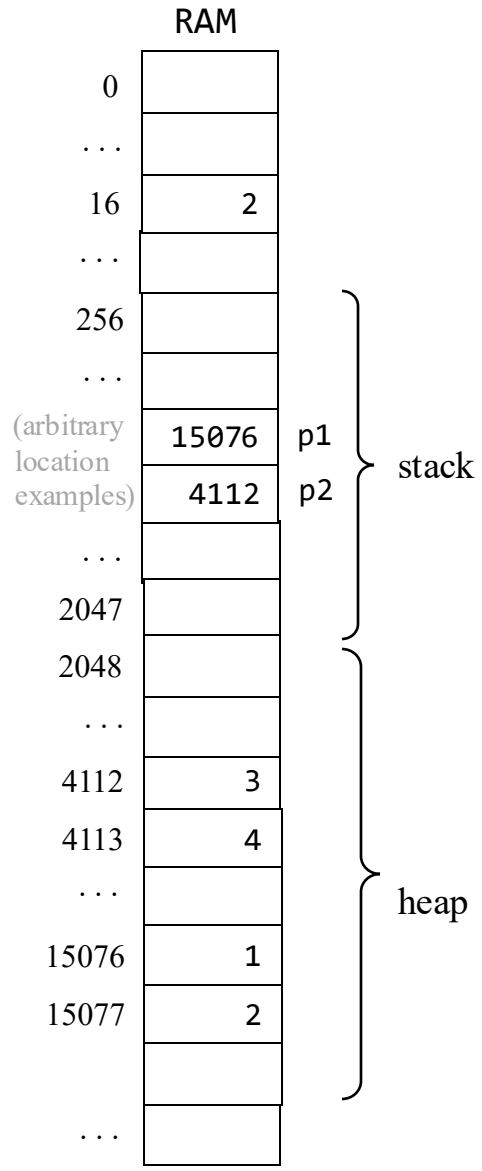
`this`: Contains the base address of the object created by the constructor

Constructors must end with the statement `return this`

Client code

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

The client-side variables `p1` and `p2` end up pointing to the objects returned by the constructor calls.



OO Programming: Destructing 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();
...
```

Lecture plan

High-level programming (tutorial)

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

The Jack language (specification)

- The language
- The operating system

Application development (project 9)

- Jack applications
- Application example
- Graphics optimization

Lecture plan

High-level programming (tutorial)

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

The Jack language (specification)

- The language
- The operating system

Application development (project 9)

- Jack applications
- Application example
- Graphics optimization

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

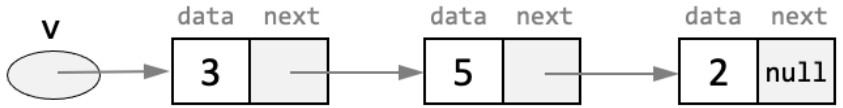
```
/** A linked list of integers. */
class List {
   field int data;   // an int value,
   field List next;  // followed by a list of int values

   /** 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`



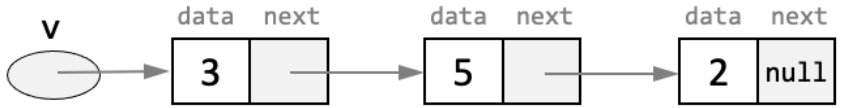
List processing

`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

The goal

Illustrating how *objects* are created and managed.

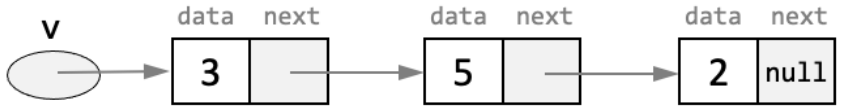
List construction

`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)

   /** 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

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);
      ...
      ...
      ...
   }
}
```

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);
      ...
      ...
      ...
   }
}
```

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);
      ...
      ...
      ...
   }
}
```

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);
      ...
      ...
      ...
   }
}
```

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);
      ...
      ...
```

The *compiled constructor* includes low-level code (not shown here) that:

1. Calls an OS function that allocates a memory block for the new object;
2. Creates a local variable `this` and sets it to the base address of the new block.

(the low-level code will be implemented when we develop the compiler)

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);
      ...
      ...
      ...
   }
}
```

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);
      ...
      ...
      ...
   }
}
```

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
}
```

When the constructor terminates, it returns the address of the newly constructed object to the caller.

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);
      ...
      ...
      ...
   }
}
```

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);
      ...
      ...
      ...
   }
}
```

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);
      ...
      ...
      ...
   }
}
```

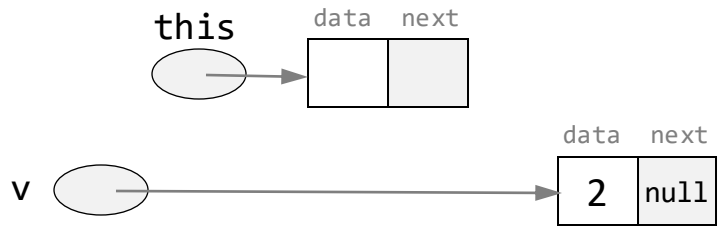
List construction

`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);
      ...
      ...
      ...
   }
}
```



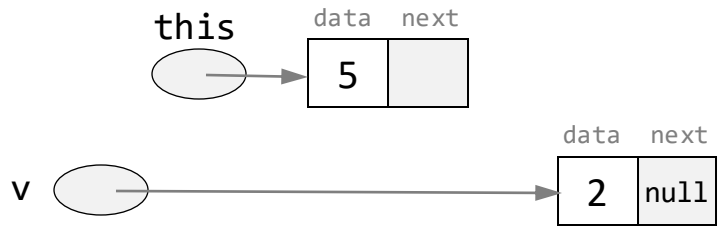
List construction

`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);
      ...
      ...
      ...
   }
}
```



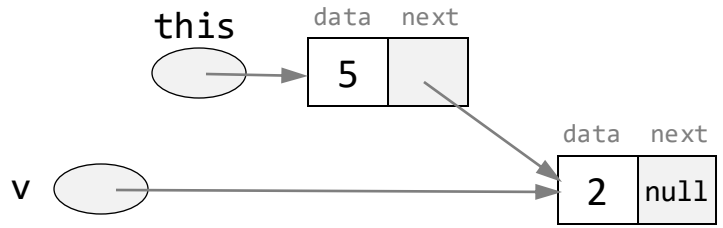
List construction

`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);
      ...
      ...
      ...
   }
}
```



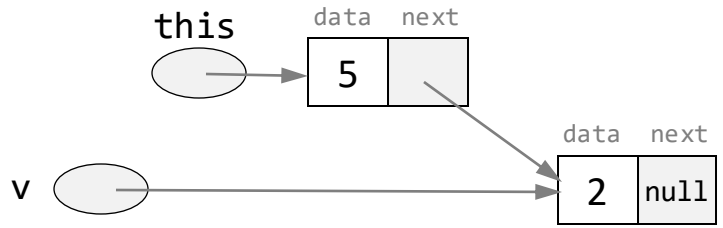
List construction

`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);
      ...
      ...
      ...
   }
}
```



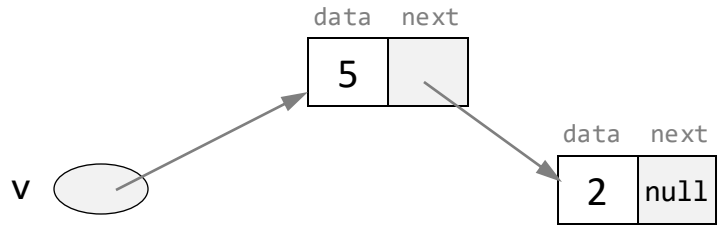
List construction

`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);
      ...
      ...
      ...
   }
}
```



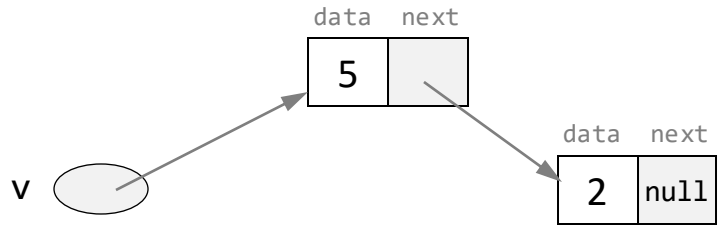
List construction

`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);
      ...
      ...
      ...
   }
}
```



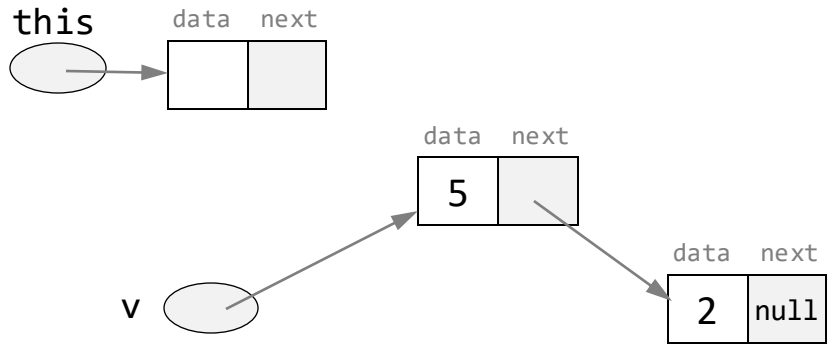
List construction

`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);
      ...
      ...
      ...
   }
}
```



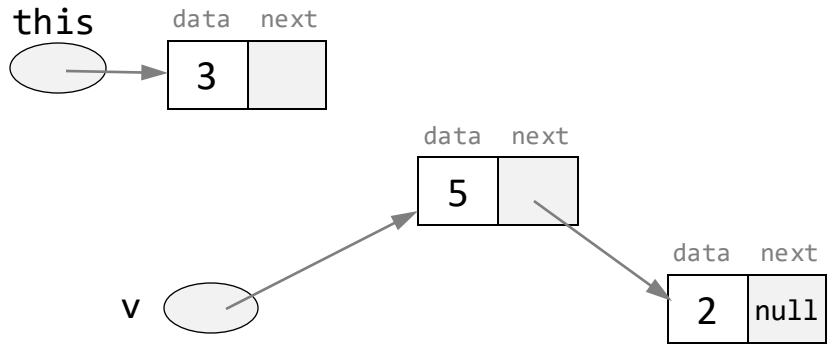
List construction

`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);
      ...
      ...
      ...
   }
}
```



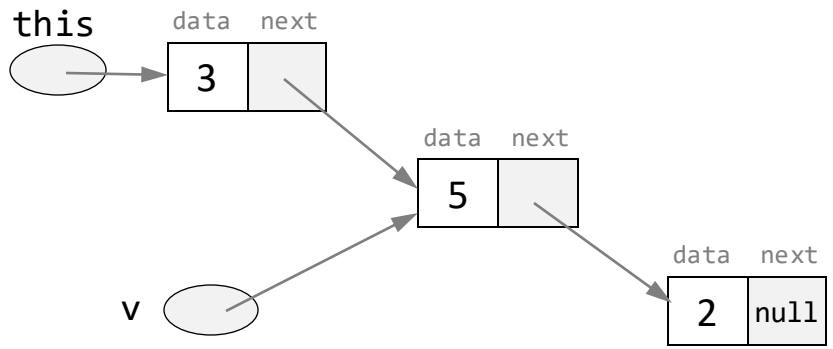
List construction

`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);
      ...
      ...
      ...
   }
}
```



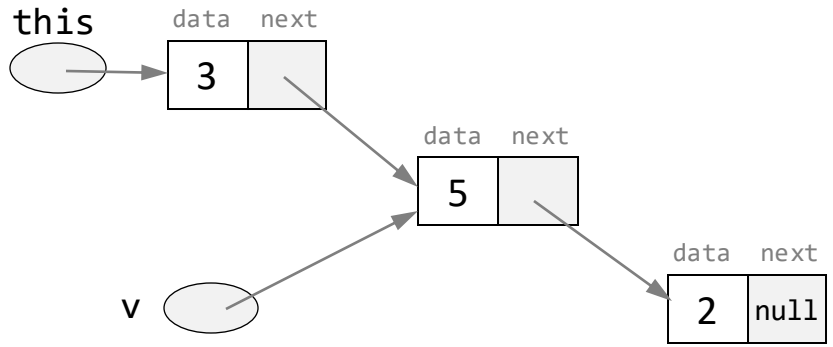
List construction

`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);
      ...
      ...
      ...
   }
}
```



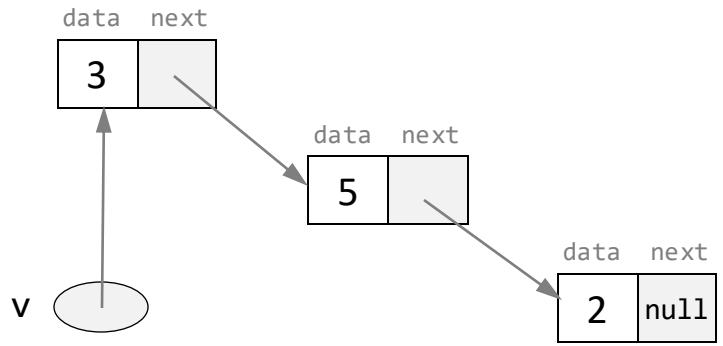
List construction

`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);
      ...
      ...
      ...
   }
}
```



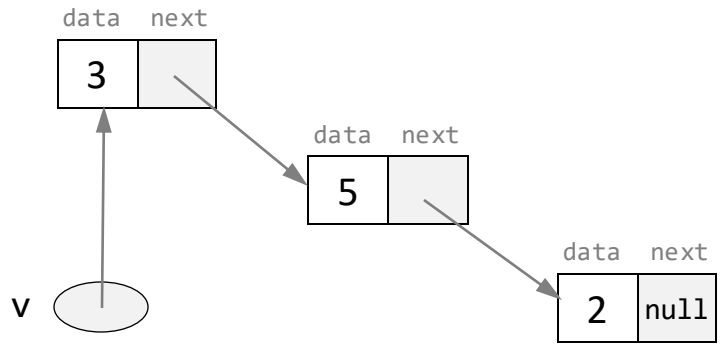
List construction

`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);
      ...
      ...
      ...
   }
}
```



We constructed the list `(3,5,2)`.

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);
      ...
      ...
      ...
   }
}
```

Aside: There are better and safer ways to build lists...

- A constructor that takes an *array* of values as a parameter and returns a *list* of values
- An *append* method that adds a value to the current list
- Etc.

Best practice: Move the list construction / manipulation code from the client to the `List` class

(this discussion belongs to an OOP course).

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
}
```

We'll discuss methods for:

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

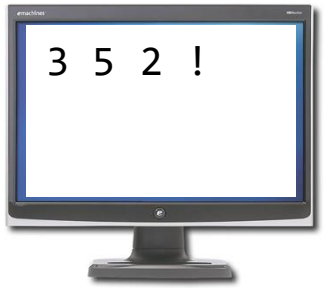
List processing

`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

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



We'll discuss methods for:

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

To illustrate *iteration*, we'll trace a method that iterates through, and prints, the list's elements.

Iterating a list

`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
}
```

client code

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

Iterating a list

`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 (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 a 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 '!'
...
```

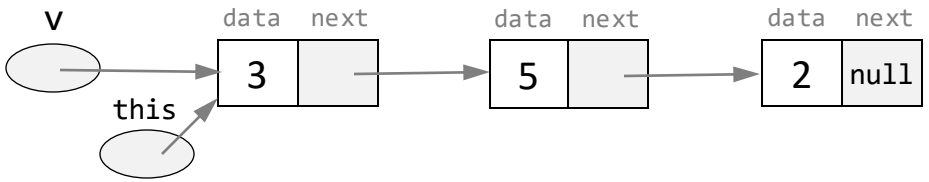
Iterating a list

`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 (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 a 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 '!'
...
```



The *compiled method* includes low-level code (not shown here) that:

1. Creates a local variable named `this`;
2. Sets `this` to the object on which the method was called.

(the low-level code will be implemented when we develop the compiler)

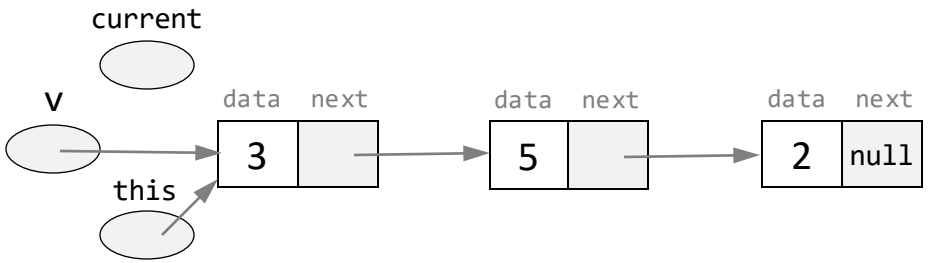
Iterating a list

`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 (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 a 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 '!'
...
```



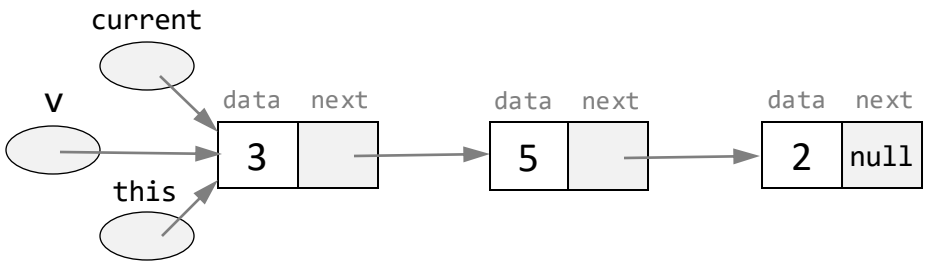
Iterating a list

`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 (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 a 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 '!'
...
```



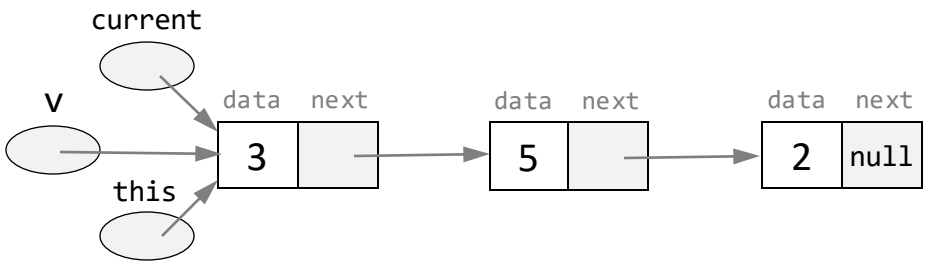
Iterating a list

`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 (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 a 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 '!'
...
```



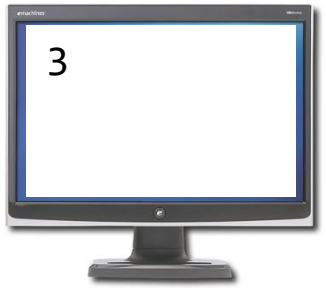
Iterating a list

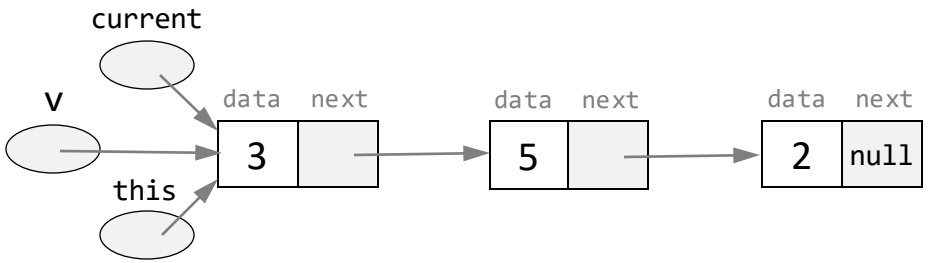
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





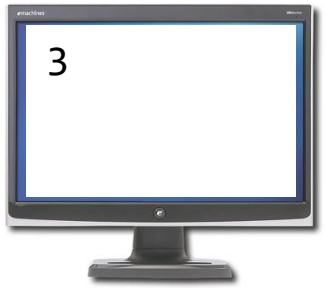
Iterating a list

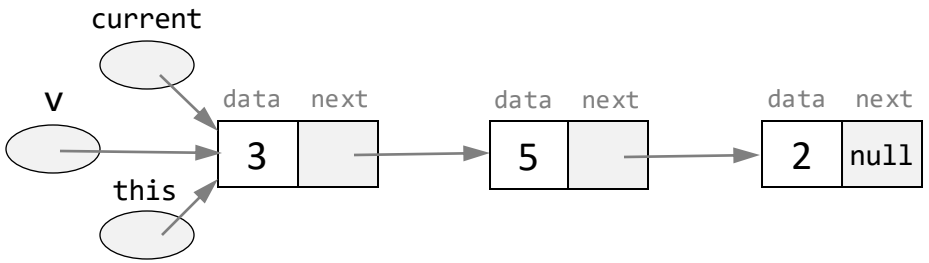
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





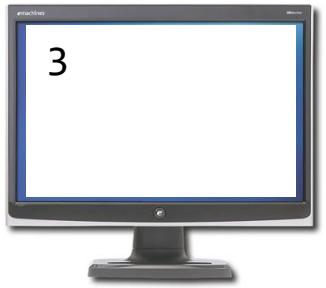
Iterating a list

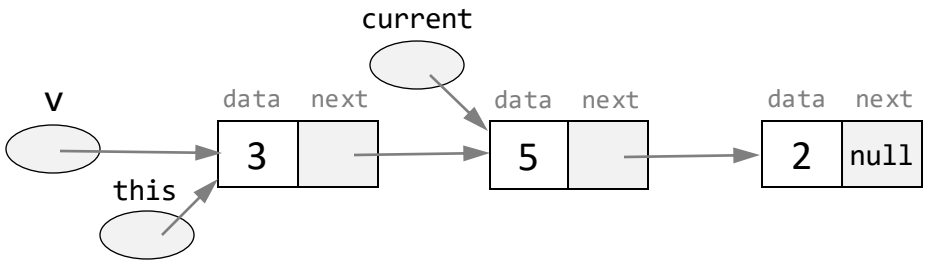
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





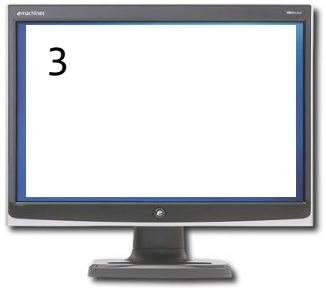
Iterating a list

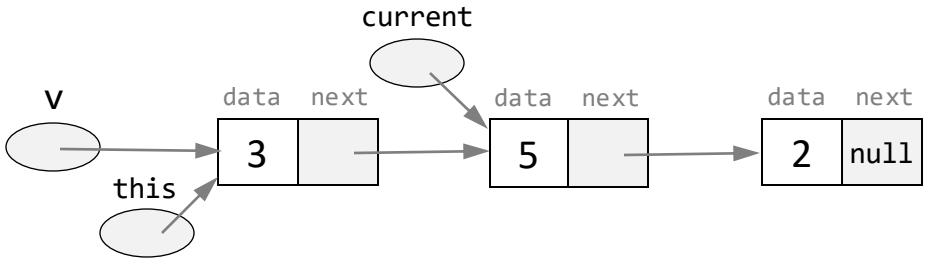
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





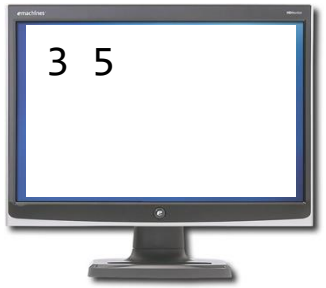
Iterating a list

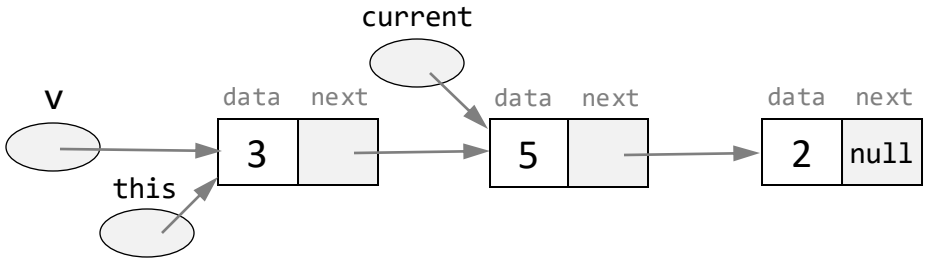
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





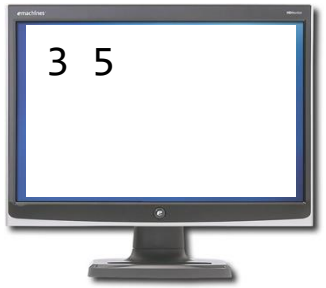
Iterating a list

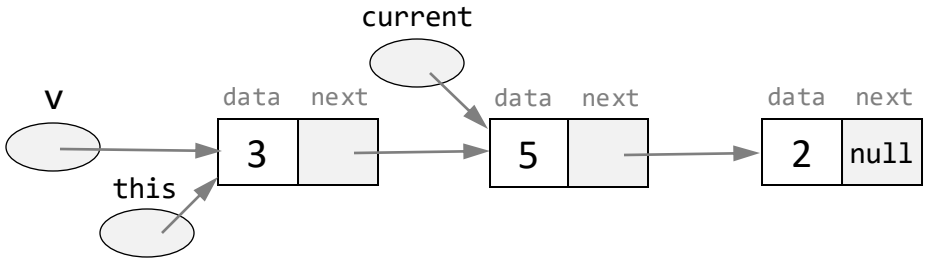
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





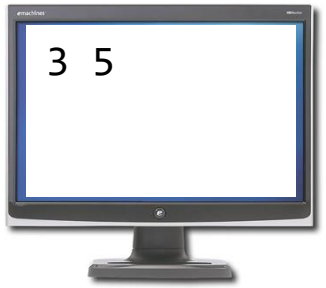
Iterating a list

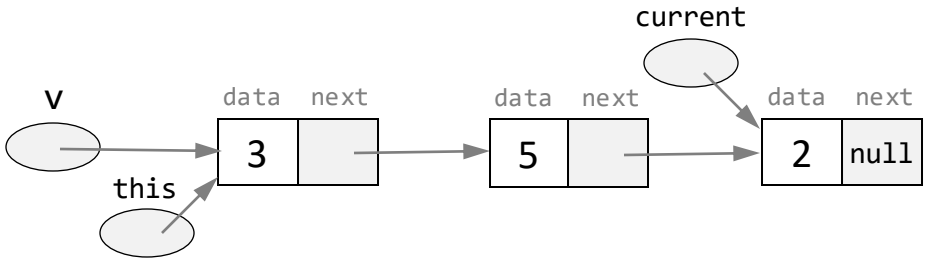
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





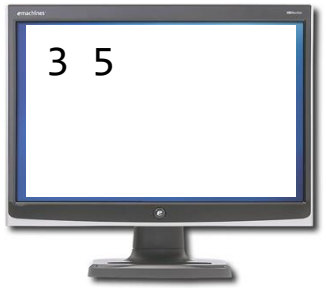
Iterating a list

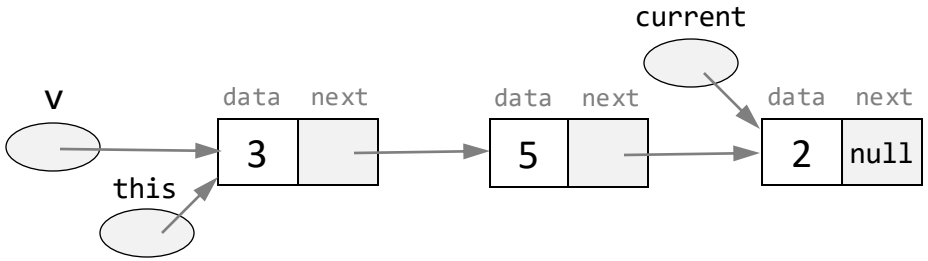
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





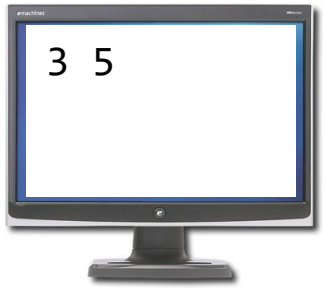
Iterating a list

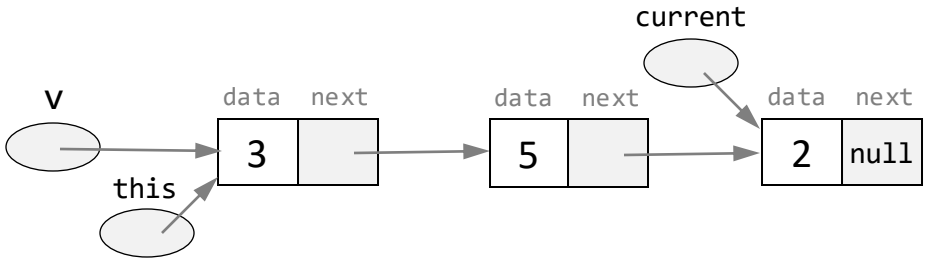
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





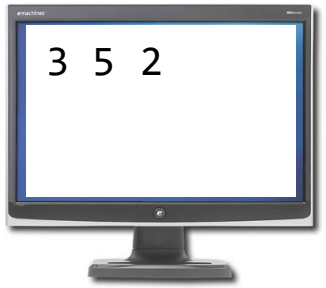
Iterating a list

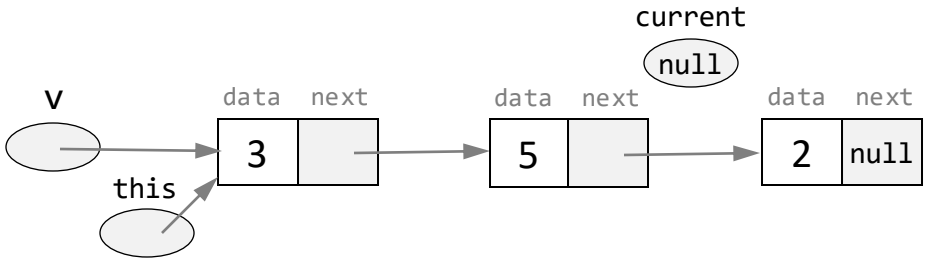
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





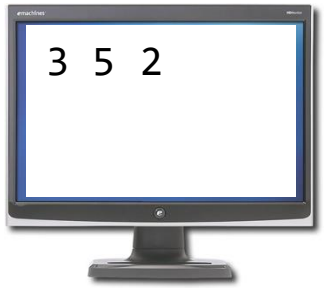
Iterating a list

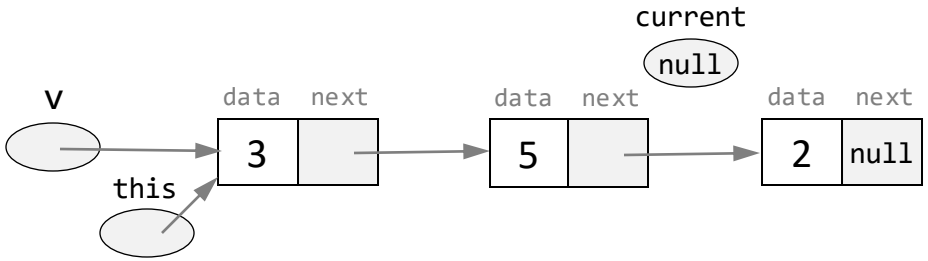
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





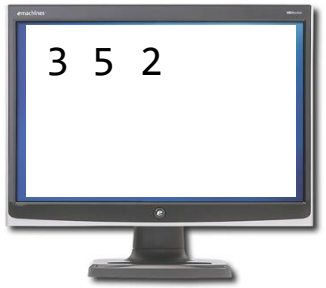
Iterating a list

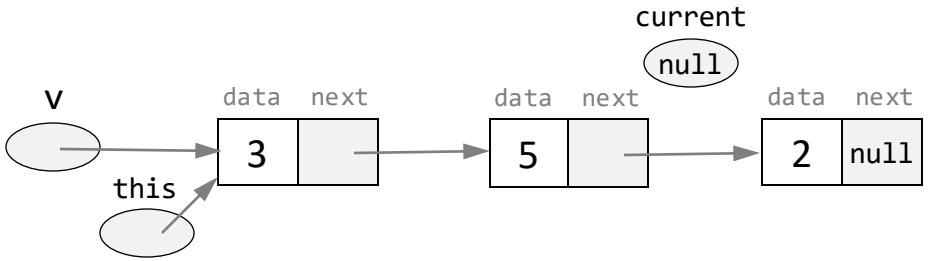
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





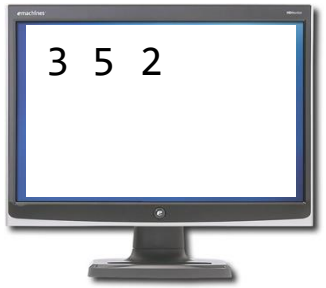
Iterating a list

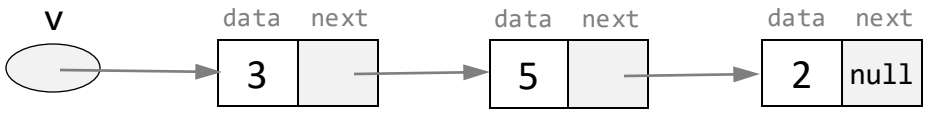
`List` class

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

   /** Constructor (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 a 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 '!'
...
```





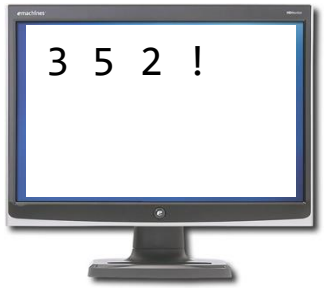
Iterating a list

`List` class

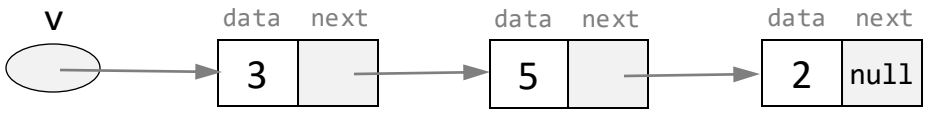
```
/** A linked list of integers. */
class List {
   field int data;      // an int value,,
   field List next;     // followed by a list of int values.

   /** Constructor (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 a 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 '!'
...
```



We've iterated through all the list elements.



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
}
```

We'll discuss methods for:

- Constructing a list
- Iterating a list
- Disposing s 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

```
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.

We'll discuss methods for:

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

List processing: Recursive access

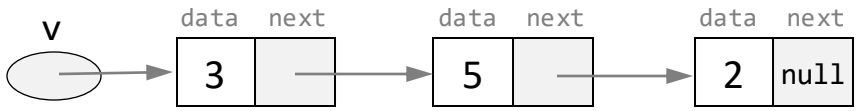
`List` class

```
/** 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 code

```
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 processing: Recursive access

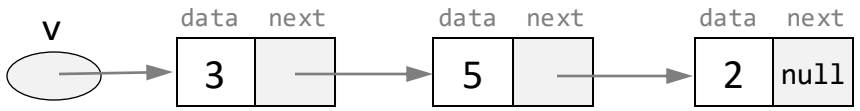
`List` class

```
/** 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 code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
... return site 0
```

We mark return sites within the code, to help the code tracing



List processing: Recursive access

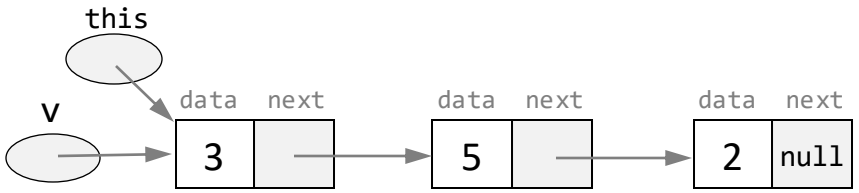
`List` class

```
/** 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;
   }

}
```

When a method starts executing, a local `this` variable is created, and is set to the object on which the method was called

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



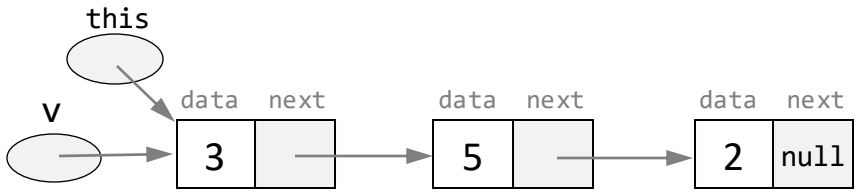
List processing: Recursive access

`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 code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



List processing: Recursive access

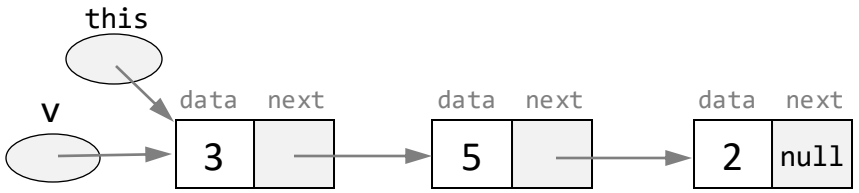
`List` class

```
/** 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();
      } return site 1
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Calls `dispose` recursively, on the object `this.next`

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



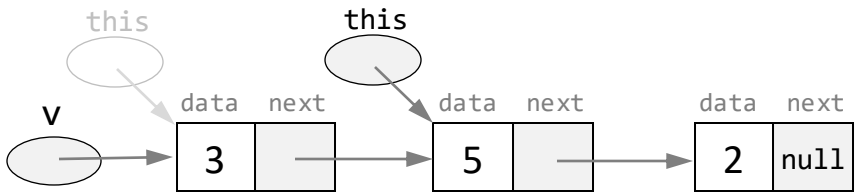
List processing: Recursive access

`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();
      } return site 1
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



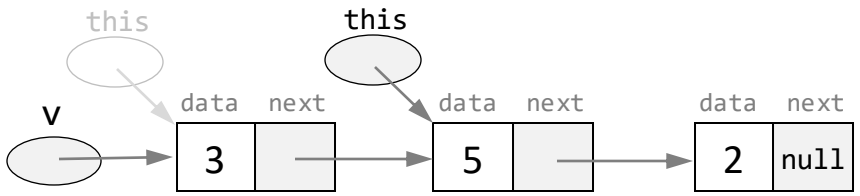
List processing: Recursive access

`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();
      } return site 1
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
... return site 0
```



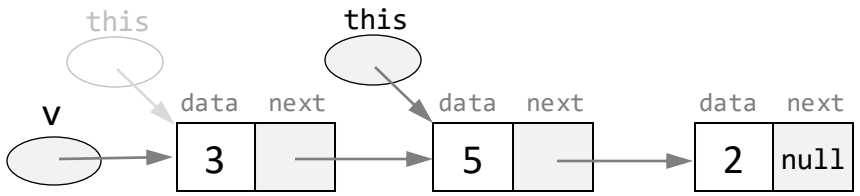
List processing: Recursive access

`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();
      } return site 1   return site 2
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



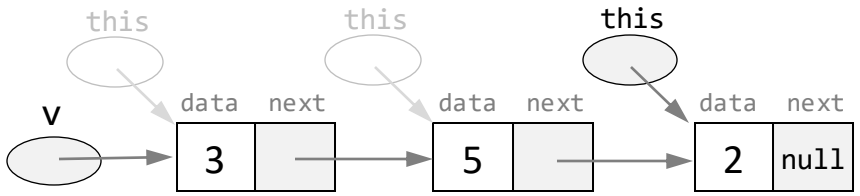
List processing: Recursive access

`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();
      } return site 1   return site 2
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



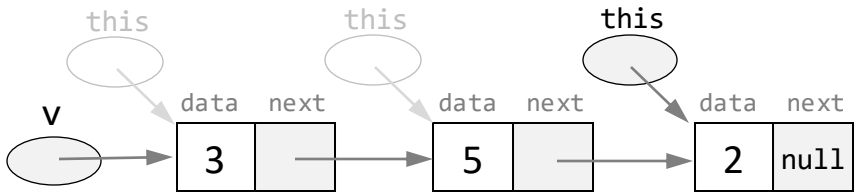
List processing: Recursive access

`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();
      } return site 1   return site 2
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



List processing: Recursive access

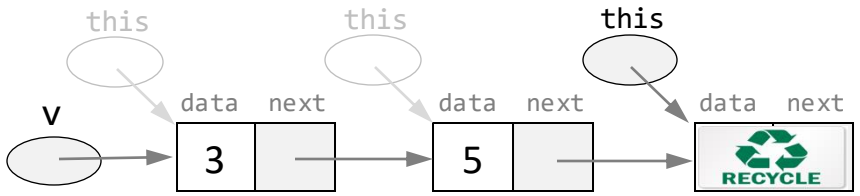
`List` class

```
/** 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();
      } return site 1   return site 2
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Calls an OS method to free the memory block beginning at base address `this`

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



List processing: Recursive access

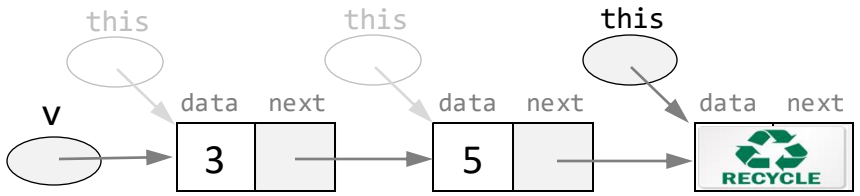
`List` class

```
/** 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();
      } return site 1   return site 2
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Returns to the most recent return address

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



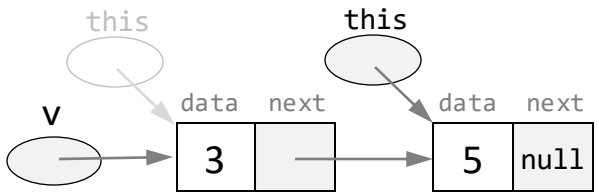
List processing: Recursive access

`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();
      } return site 1
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
... return site 0
```



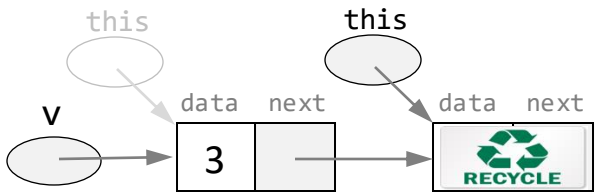
List processing: Recursive access

`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();
      } return site 1
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
... return site 0
```



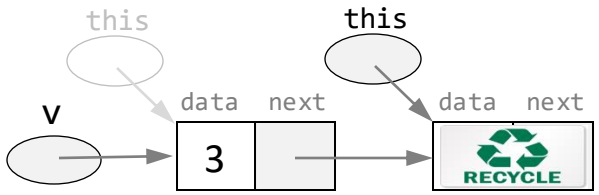
List processing: Recursive access

`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();
      } return site 1
      // Frees the memory of this list
      do Memory.deAlloc(this);
      return;
   }

}
```

Client code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



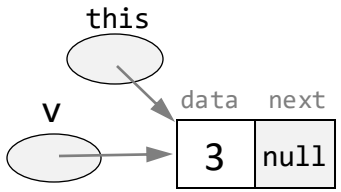
List processing: Recursive access

`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 code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



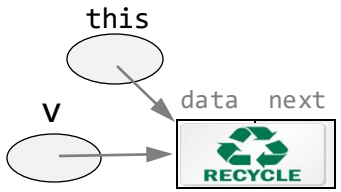
List processing: Recursive access

`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 code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



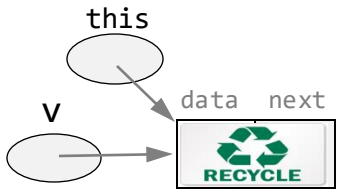
List processing: Recursive access

`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 code

```
var List v;
// builds the list (2, 3, 5), code omitted
...
do v.dispose();
...  return site 0
```



List processing: Recursive access

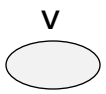
`List` class

```
/** 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 code

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

The list was emptied, its objects recycled.



Lecture plan

✓ High-level programming (tutorial)

- Program example
- Basic language constructs
- Object-based programming

Jack language specification (reference)

- The language
  - Typical features
  - Jack-specific features
- The operating system

Application development (project 9)

- Jack applications
- Application example
- Graphics optimization

Lecture plan

✓ High-level programming (tutorial)

- Program example
- Basic language constructs
- Object-based programming

➡ Jack language specification (reference)

- The language
  - Typical features
  - Jack-specific features
- The operating system

The language specs should be used as a technical reference, as needed.

Application development (project 9)

- Jack applications
- Application example
- Graphics optimization

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

Data types

Primitive types

`int`

`char`

`boolean`

Class types

From the Jack standard library, like `String`

Defined by programmers, as needed, like `Point`

Data types

Primitive types

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

Class types

- From the Jack standard library, like `String`
- Defined by programmers, as needed, like `Point`

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.

Lecture plan

✓ High-level programming (tutorial)

- Program example
- Basic language constructs
- Object-based programming

Jack language specification (reference)

- The language
  - ✓ Typical features
  - ➡ Jack-specific features
- The operating system

Application development (project 9)

- Jack applications
- Application example
- Graphics optimization

Classes

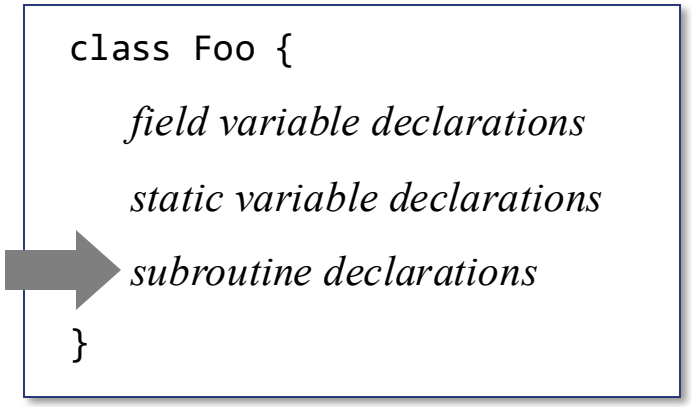
Class declaration

```
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.

Classes

Class declaration



- 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.

Subroutines

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.

Lecture plan

✓ High-level programming (tutorial)

- Program example
- Basic language constructs
- Object-based programming

Jack language specification (reference)

✓ The language

➡ The operating system

Application development (project 9)

- Jack applications
- Application example
- Graphics optimization

The Jack OS

Jack code

```
// 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 code

```
// 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;
   }
}
```

Jack OS

A collection of supplied Jack classes;

Similar (in concept) to Java's standard class library

Purpose

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

Provides efficient implementations of commonly-used functions and ADTs.

The Jack OS

OS classes

`Math:`	Common mathematical operations
`String:`	Common string processing
`Array:`	Used to construct and dispose arrays
`Output:`	Handles textual output
`Screen:`	Handles graphical output
`Keyboard:`	Handles input
`Memory:`	Memory management services
`Sys:`	Execution relates services

Complete OS API: book / website

Two views on the OS

Abstraction (API):

How to *use* the OS — this lecture

Implementation:

How to *build* the OS — chapter 12

The Jack OS

OS classes:

`Math:`	Common mathematical operations
`String:`	Common string processing
`Array:`	Used to construct and dispose arrays
`Output:`	Handles textual output
`Screen:`	Handles graphical output
`Keyboard:`	Handles input
`Memory:`	Memory management services
`Sys:`	Execution relates services

```
class Math {

   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)

}
```

Usage example:

```
int b;
let b = Math.sqrt(Math.multiply(x,y)));
...
```

The Jack OS

OS classes:

Math:	Common mathematical operations
String:	Common string processing
Array:	Used to construct and dispose arrays
Output:	Handles textual output
Screen:	Handles graphical output
Keyboard:	Handles input
Memory:	Memory management services
Sys:	Execution relates services

```
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()
}
```

Usage example:

```
// Gets the last character in string str
let c = str.charAt(str.length()-1);
...
```

The Jack OS

OS classes

`Math:`	Common mathematical operations
`String:`	Common string processing
`Array:`	Used to construct and dispose arrays
`Output:`	Handles textual output
`Screen:`	Handles graphical output
`Keyboard:`	Handles input
`Memory:`	Memory management services
`Sys:`	Execution relates services

```
Class Array {

   function Array new(int size)

   method void dispose()

}
```

Usage example:

```
// Declares and creates an array of 10 elements
// (Jack arrays are untyped)
Array arr;
let arr = Array.new(10);
...
```

The Jack OS

OS classes

`Math:`	Common mathematical operations
`String:`	Common string processing
`Array:`	Used to construct and dispose arrays
`Output:`	Handles textual output
`Screen:`	Handles graphical output
`Keyboard:`	Handles input
`Memory:`	Memory management services
`Sys:`	Execution relates services

```
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()
}
```

Screen: 23 rows of 64 characters, black and white

Font: Fixed, defined in the `Output` class

Usage example:

```
// (Note: the ASCII code of the character a is 97)
// Prints 97, then prints the character a, twice
do Output.printInt(97);
do Output.printChar(97);
do Output.printString("a");
...
```

The Jack OS

OS classes

Math:	Common mathematical operations
String:	Common string processing
Array:	Used to construct and dispose arrays
Output:	Handles textual output
Screen:	Handles graphical output
Keyboard:	Handles input
Memory:	Memory management services
Sys:	Execution relates services

```
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)
}
```

Screen: 256 rows of 512 pixels, black and white

Usage example:

```
// Draws a filled circle of radius 50 pixels,
// centered at (100, 100), using the current color
do Screen.drawCircle(100,100,50);
...
```

The Jack OS

OS classes

- `Math:` Common mathematical operations
- `String:` Common string processing
- `Array:` Used to construct and dispose arrays
- `Output:` Handles textual output
- `Screen:` Handles graphical output
- `Keyboard:` Handles input
- `Memory:` Memory management services
- `Sys:` Execution relates services

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

Reads data from the standard keyboard.

Usage example:

```
...
let age = Keyboard.readInt("enter your age: ");
...
```

The Jack OS

OS classes

`Math:`	Common mathematical operations
`String:`	Common string processing
`Array:`	Used to construct and dispose arrays
`Output:`	Handles textual output
`Screen:`	Handles graphical output
`Keyboard:`	Handles input
`Memory:`	Memory management services
`Sys:`	Execution relates services

```
class Memory {
   function int peek(int address)
   function void poke(int address, int value)
   function Array alloc(int size)
   function void deAlloc(Array o)
}
```

Usage example:

```
// Sets RAM[513] to RAM[512]
do Memory.poke(513,Memory.peek(512));
...
```

The Jack OS

OS classes

`Math:`	Common mathematical operations
`String:`	Common string processing
`Array:`	Used to construct and dispose arrays
`Output:`	Handles textual output
`Screen:`	Handles graphical output
`Keyboard:`	Handles input
`Memory:`	Memory management services
`Sys:`	Execution relates services

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

Usage example:

```
// Pauses for one second (1000 milliseconds)
do Sys.wait(1000);
...
```

The Jack OS

OS classes

`Math:`	Common mathematical operations
`String:`	Common string processing
`Array:`	Used to construct and dispose arrays
`Output:`	Handles textual output
`Screen:`	Handles graphical output
`Keyboard:`	Handles input
`Memory:`	Memory management services
`Sys:`	Execution relates services

Complete OS API: book / website

Lecture plan

✓ High-level programming (tutorial)

- Program example
- Basic language constructs
- Object-based programming

✓ Jack language specification (reference)

- The language
- The operating system

➡ Application development (project 9)

- Jack applications
- Application example
- Graphics optimization

Lecture plan

✓ High-level programming (tutorial)

- Program example
- Basic language constructs
- Object-based programming

✓ Jack language specification (reference)

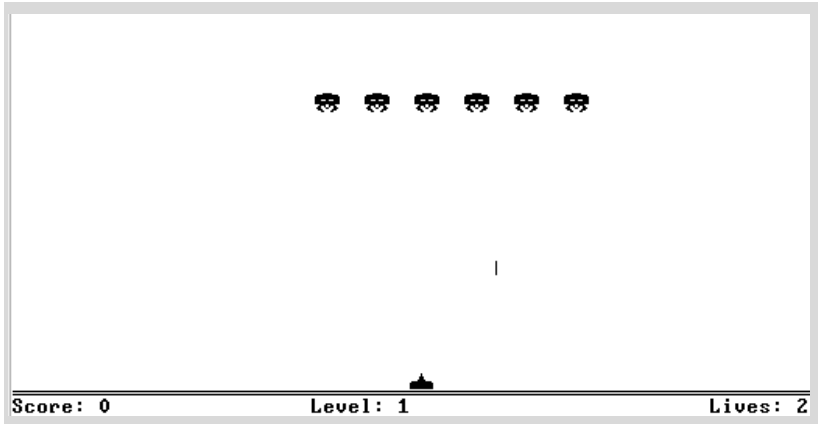
- The language
- The operating system

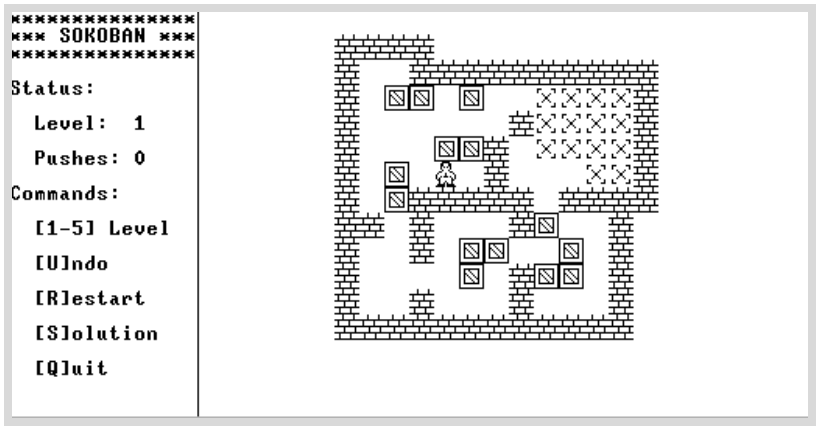
Application development (project 9)

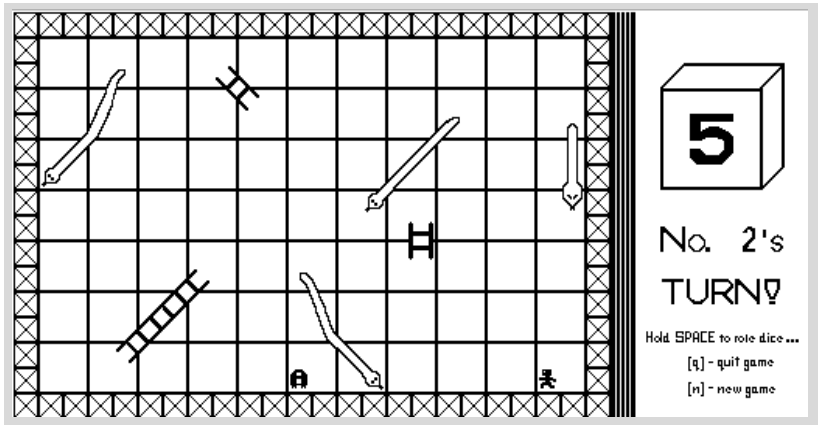
➡ Jack applications

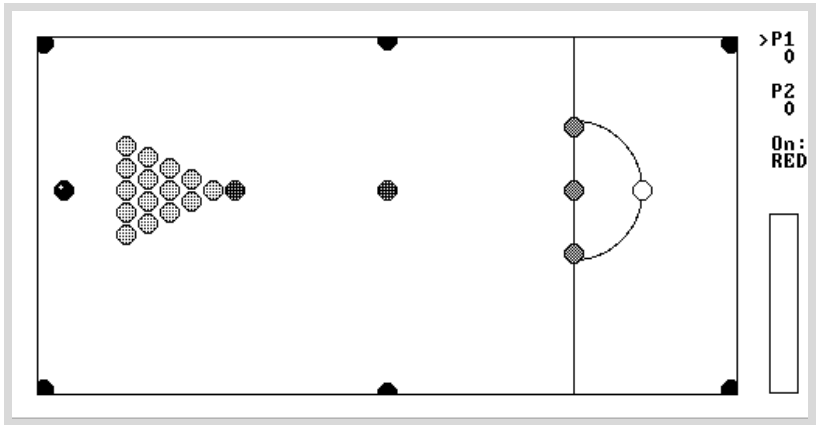
- Application example
- Graphics optimization

Sample Jack programs









More examples: Search "Nand to Tetris game" in Youtube.

Sample Jack programs (source code: `nand2tetris/projects/9`)

`List`:	Illustrates list processing
`Average`:	Illustrates array processing
`ComplexArrays`:	Illustrates 1D and 2D arrays
`ConvertToBin`:	Illustrates algebraic operations, and working with `peek` and `poke`
`Fraction`:	Represents fractions as objects
`Square`:	Simple, interactive, multi-class OO application
`Pong`:	Complete, interactive, multi-class OO application

A useful set of programming examples for project 9.

Compiling / executing a Jack program

If using the Nand2Tetris IDE (recommended)

Compiling

The compiler handles only folders.

When given a folder name, it loads and compiles all the `.jack` files in the given folder.

Executing the compiled program

Click the "load" button of the compiler's dashboard;

The compiled code will be loaded into the VM emulator, in no particular file order.

The resulting code base includes all the VM functions in all the `.vm` files that were loaded into the compiler (the notion of individual files no longer exists at the VM level).

Compiling / executing a Jack program

If using the desktop Nand2Tetris software package:

Compiling a single Jack file / folder that has one or more `.jack` files:

`% JackCompiler.sh` *filename*`.jack` / *folderName*

(replace `.sh` with `.bat`, depending on your PC/OS)

Executing the compiled program

`% VMEmulator.sh` (replace `.sh` with `.bat`, depending on your PC/OS)

Load the entire folder (or single file) into the VM emulator

If a "confirmation message" is displayed, click "yes"

If the program is interactive (like `Square`):

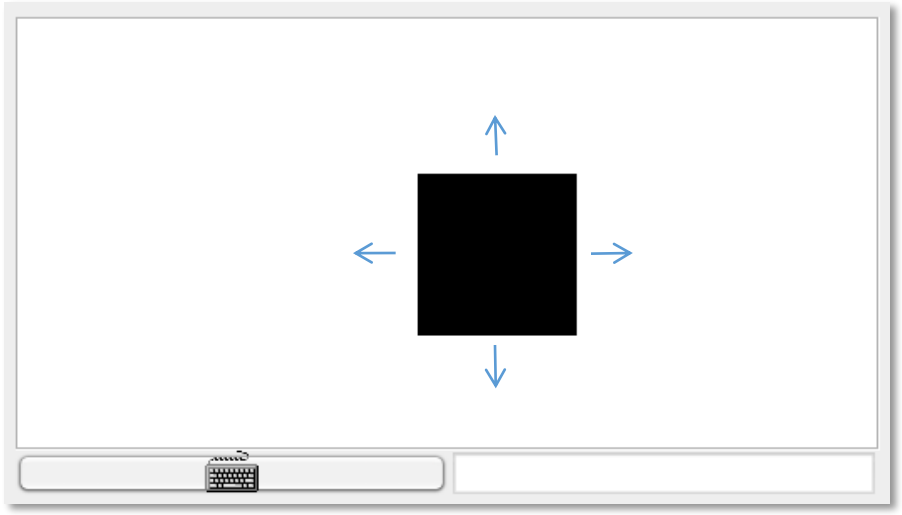
Select "no animation" from the "Animate" menu (meaning: No tracing of the program's logic)

Click the "fast forward" icon (or select "run" from the "Run" menu)

To pause / stop execution, select the pause / stop icons

To control the execution speed, stop and use the speed slider.

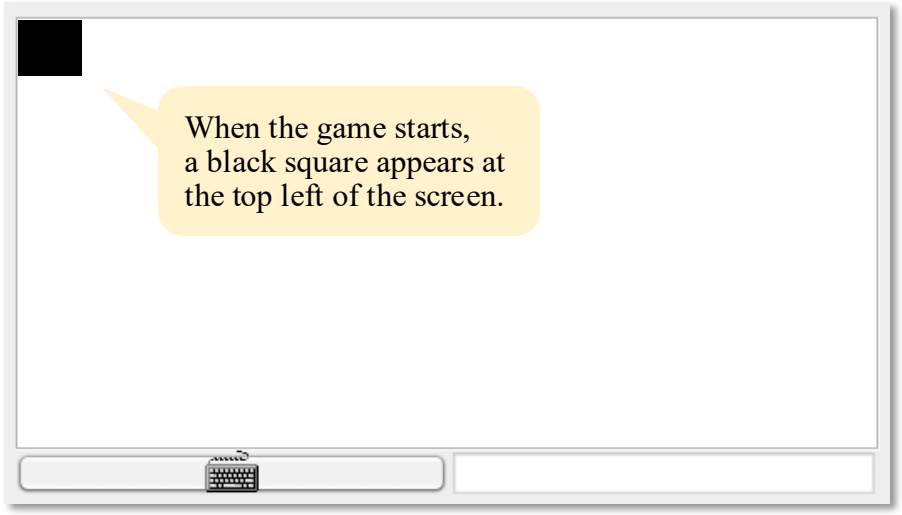
Program example: Square Dance



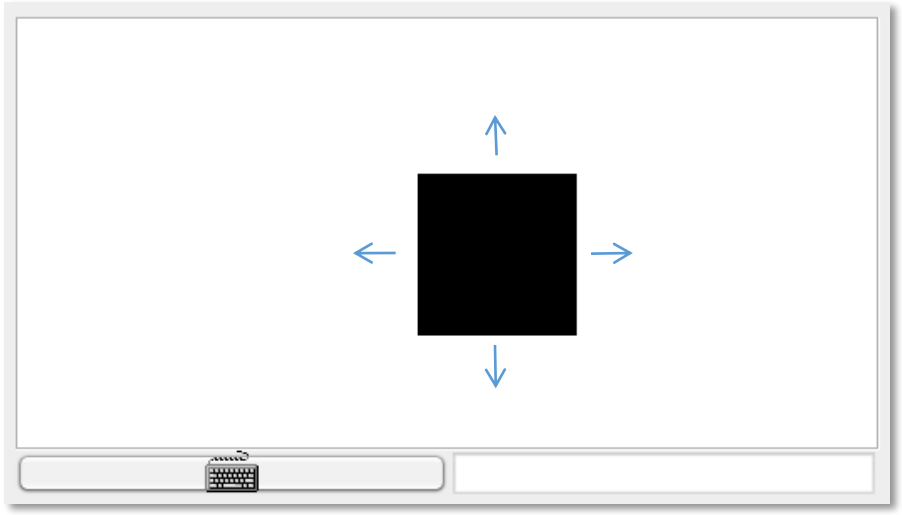
Purpose: Learning...

- How to design / build an interactive OO application
- How to implement graphical objects and animation
- How to use the Jack OS.

Program example: Square Dance



Program example: Square Dance



Usage

up arrow: the square moves up, until another key is pressed

down arrow: the square moves down, until another key is pressed

left arrow: the square moves left, until another key is pressed

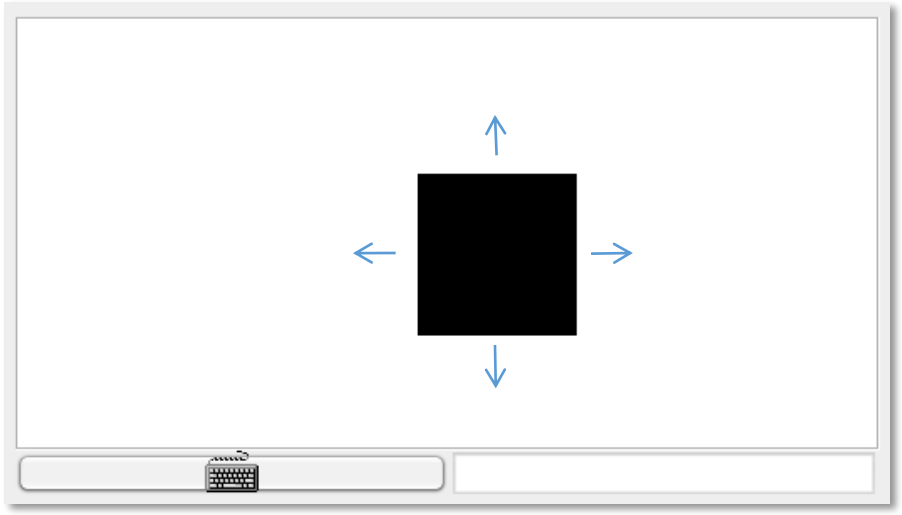
right arrow: the moves right, until another key is pressed

x key: the square's size increases a little (2 pixels)

z key: the square's size decreases a little (2 pixels)

q: game over.

Program example: Square Dance



Design: three Jack classes:

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

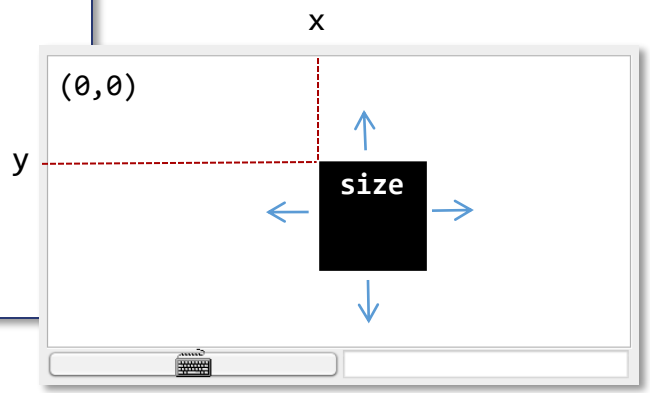
`SquareGame`: Creates a `Square`, then enters a loop that captures, and responds to, the user's inputs;

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

Program example: Square Dance / class Square

Square API

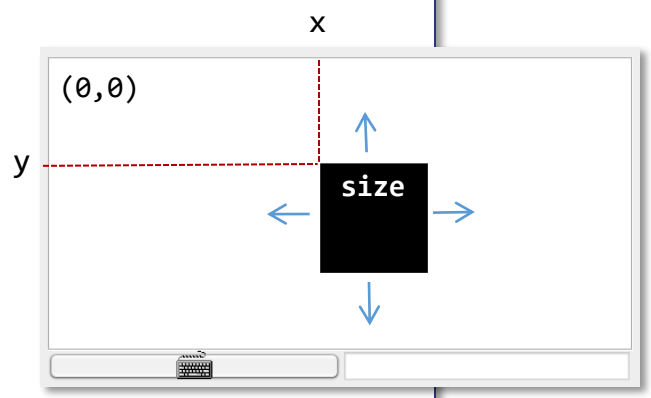
```
/** Implements a graphical square.
 *  The square has top-left x and y coordinates, and a size. */
class Square {

   /** Constructs a new square with a given location and size */
   constructor Square new(int ax, int ay, int asize)

   /** Disposes this square */
   method void dispose()

   /** Draws this square in its current (x,y) location */
   method void draw()

   /** Erases this square */
   method void erase()

   /** Increments this square's size by 2 pixels */
   method void incSize()

   /** Decrements this square's size by 2 pixels */
   method void decSize()

   /** Moves this square up by 2 pixels */
   method void moveUp()

   /** Moves this square down by 2 pixels */
   method void moveDown()

   /** Moves this square left by 2 pixels */
   method void moveLeft()

   /** Moves this square right by 2 pixels */
   method void moveRight()
}
```



Program example: Square Dance / class Square

Square.jack

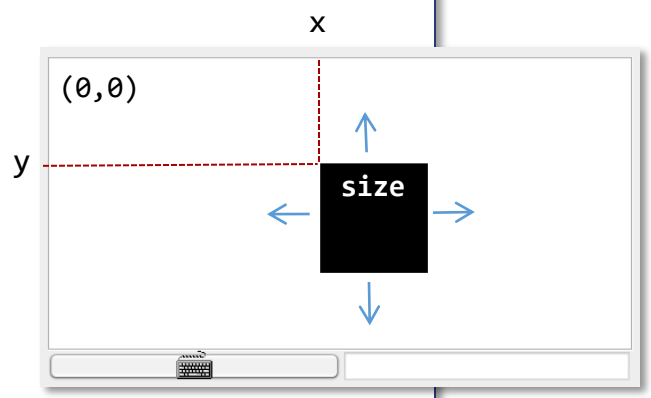
```
/** Implements a graphical square.
 *  The square has top-left x and y coordinates, and a size. */
class Square {

   field int x, y; // screen location of the top-left corner of this square
   field int size; // length of this square, in pixels
```



Program example: Square Dance / class `Square`

`Square.jack`

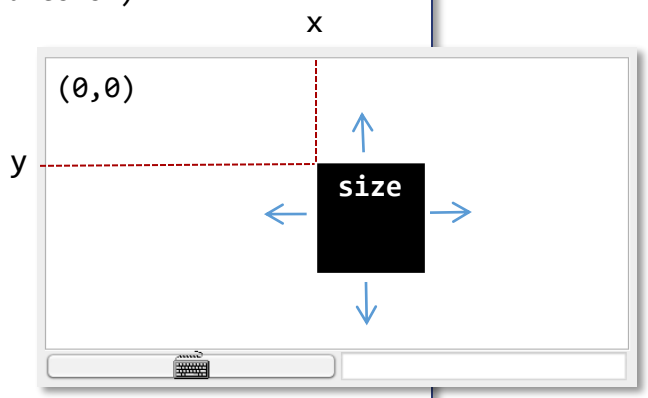
```
/** Implements a graphical square.
 *  The square has top-left x and y coordinates, and a size. */
class Square {

   field int x, y; // screen location of the top-left corner of this square
   field int size; // length of this square, in pixels

   /** Constructs and draws a new square with a given location and size. */
   constructor Square new(int ax, int ay, int asize) {
      let x = ax;
      let y = ay;
      let size = asize;
      do draw();
      return this;
   }

   ...
```



Program example: Square Dance / class Square

Square.jack

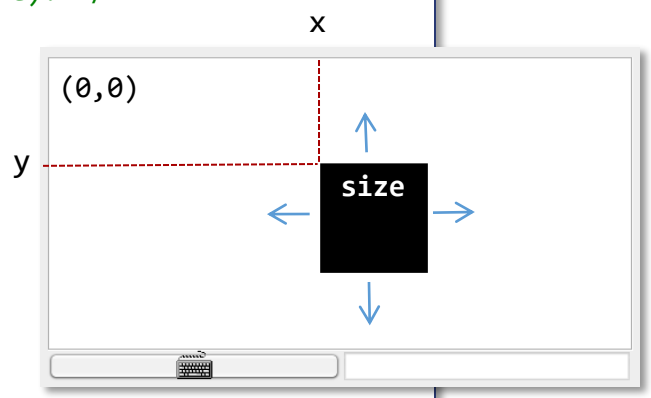
```
/** Implements a graphical square.
 *  The square has top-left x and y coordinates, and a size. */
class Square {

   field int x, y; // screen location of the top-left corner of this square
   field int size; // length of this square, in pixels

   ...

   /** Draws this square in its current (x,y) location */
   method void draw() {
      // Draws the square using the color black
      do Screen.setColor(true);
      do Screen.drawRectangle(x, y, x + size, y + size);
      return;
   }

   /** Erases this square. */
   method void erase() {
      // Draws the square using the color white (background color)
      do Screen.setColor(false);
      do Screen.drawRectangle(x, y, x + size, y + size);
      return;
   }

   ...
```



Program example: Square Dance / class Square

Square.jack

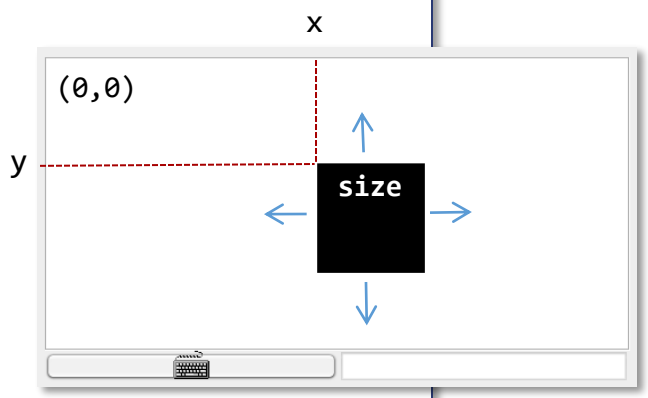
```
/** Implements a graphical square.
 *  The square has top-left x and y coordinates, and a size. */
class Square {

   field int x, y; // screen location of the top-left corner of this square
   field int size; // length of this square, in pixels

   ...

   /** Increments this square size by 2 pixels (if possible). */
   method void incSize() {
      if (((y + size) < 254) & ((x + size) < 510)) {
         do erase();
         let size = size + 2;
         do draw();
      }
      return;
   }

   /** Decrements this square size by 2 pixels (if possible). */
   method void decSize() {
      if (size > 2) {
         do erase();
         let size = size - 2;
         do draw();
      }
      return;
   }

   ...
```



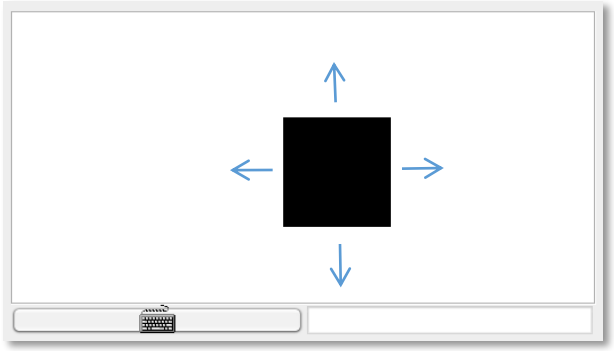
Program example: Square Dance / class Square

Square.jack

```
/** Implements a graphical square.
 *  The square has top-left x and y coordinates, and a size. */
class Square {

   field int x, y; // screen location of the top-left corner of this square
   field int size; // length of this square, in pixels

   ...

   /** Moves this square up by 2 pixels (if possible). */
   method void moveUp() {
      if (y > 1) {
         // Erases the bottom two rows of this square in its current location
         do Screen.setColor(false);
         do Screen.drawRectangle(x, (y + size) - 1, x + size, y + size);
         let y = y - 2;
         // Draws the top two rows of this square in its new location
         do Screen.setColor(true);
         do Screen.drawRectangle(x, y, x + size, y + 1);
      }
      return;
   }

   method void moveDown()  { // similar }

   method void moveLeft()  { // similar }

   method void moveRight() { // similar }

} // class Square
```



Program example: Square Dance



Design: 3 Jack classes:

`Square`: Represents a `Square` object

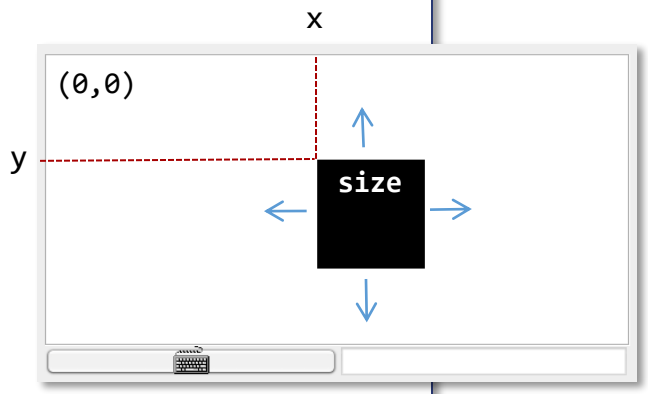
`SquareGame`: Creates a `Square`, then enters a loop that captures the user's inputs and manipulates the square accordingly

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

Program example: Square Dance / class `SquareGame`

`SquareGame.jack`

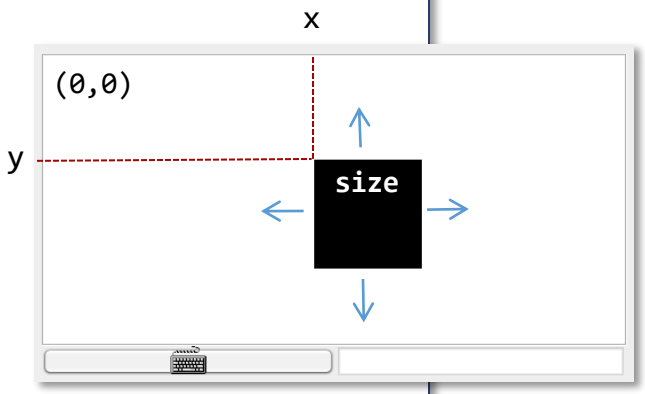
```
/** Implements a square dance game.
 *  The game has a graphical square object, and a direction (none, up, down, left, right).
 *  The square is always moving in its current direction. */
class SquareGame {

   field Square square; // the square of this game
   field int direction; // the current direction: 0=none, 1=up, 2=down, 3=left, 4=right
```



Program example: Square Dance / class SquareGame

SquareGame.jack

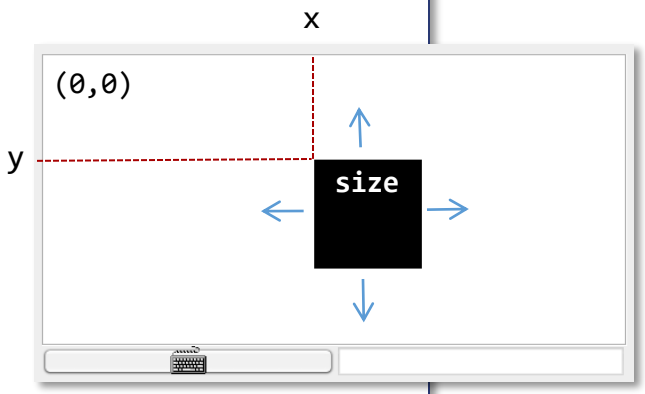
```
/** Implements a square dance game.
 *  The game has a graphical square object, and a direction (none, up, down, left, right).
 *  The square is always moving in its current direction. */
class SquareGame {

   field Square square; // the square of this game
   field int direction; // the current direction: 0=none, 1=up, 2=down, 3=left, 4=right

   /** Constructs a new square dance game.
    *  The initial square is located in (0,0), has size 30, and is not moving. */
   constructor SquareGame new() {
      let square = Square.new(0, 0, 30);
      let direction = 0;
      return this;
   }

   /** Disposes this game. */
   method void dispose() {
      do square.dispose();
      do Memory.deAlloc(this);
      return;
   }

   ...
```



Program example: Square Dance / class SquareGame

SquareGame.jack

```
/** Implements a square dance game.
 *  The game has a graphical square object, and a direction (none, up, down, left, right).
 *  The square is always moving in its current direction. */
class SquareGame {

   field Square square; // the square of this game
   field int direction; // the current direction: 0=none, 1=up, 2=down, 3=left, 4=right

   /** Constructs a new square dance game.
    *  The initial square is located in (0,0), has size 30, and is not moving. */
   constructor SquareGame new() {
      let square = Square.new(0, 0, 30);
      let direction = 0;
      return this;
   }

   /** Disposes this game. */
   method void dispose() {
      do square.dispose();
      do Memory.deAlloc(this);
      return;
   }

   /** Moves the square in the current direction. */
   method void moveSquare() {
      if (direction = 1) { do square.moveUp(); }
      if (direction = 2) { do square.moveDown(); }
      if (direction = 3) { do square.moveLeft(); }
      if (direction = 4) { do square.moveRight(); }

      do Sys.wait(5);  // delays the next movement
      return;
   }
   ...
```

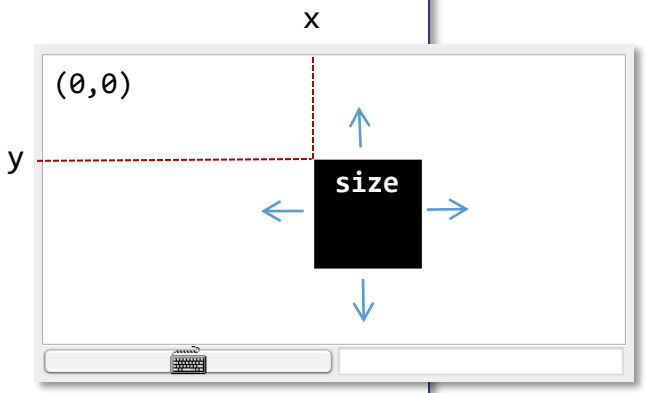


Program example: Square Dance / class `SquareGame`

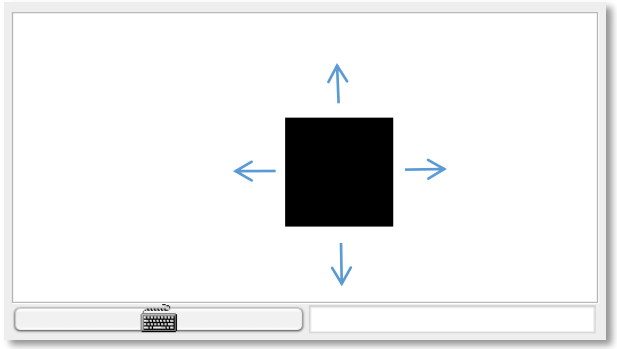
`SquareGame.jack`

```
/** Implements a square game... */
class SquareGame {

   field Square square; // the square of this game
   field int direction; // the current direction: 0=none, 1=up, 2=down, 3=left, 4=right

   ...
   /** Runs the game: handles the user's inputs, and moves the square accordingly */
   method void run() {
      var char key;  // the key currently pressed by the user
      var boolean exit;
      let exit = false;

      while (~exit) {
         // waits for a key to be pressed
         while (key = 0) {
            let key = Keyboard.keyPressed();
            do moveSquare();
         }
         if (key = 81)  { let exit = true; }     // q key
         if (key = 90)  { do square.decSize(); } // z key
         if (key = 88)  { do square.incSize(); } // x key
         if (key = 131) { let direction = 1; }   // up
         if (key = 133) { let direction = 2; }   // down
         if (key = 130) { let direction = 3; }   // left
         if (key = 132) { let direction = 4; }   // right

         // waits for the key to be released
         while (~(key = 0)) {
            let key = Keyboard.keyPressed();
            do moveSquare();
         }
      } // while
      return;
   }
}  // SquareGame class
```

typical handling of keyboard events in interactive Jack apps



Program example: Square Dance



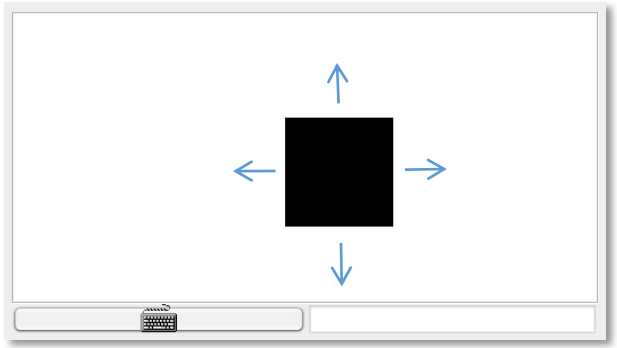
Design: 3 Jack classes:

`Square`: Represents a `Square` object

`SquareGame`: Creates a `Square`, then enters a loop that captures the user's input and moves the square / resizes / quits accordingly

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

Program example: Square Dance



Main.jack

```
/** Main class of the Square Dance game. */
class Main {

  /** Initializes a new game and starts it. */
  function void main() {
     var SquareGame game;
     let game = SquareGame.new();
     do game.run();
     do game.dispose();
     return;
  }
}
```

Design: 3 Jack classes:

Square: Represents a `Square` object

SquareGame: Creates a `Square`, then enters a loop that captures the user's input and moves the square / resizes / quits accordingly

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

Lecture plan

✓ High-level programming (tutorial)

- Program example
- Basic language constructs
- Object-based programming

✓ Jack language specification (reference)

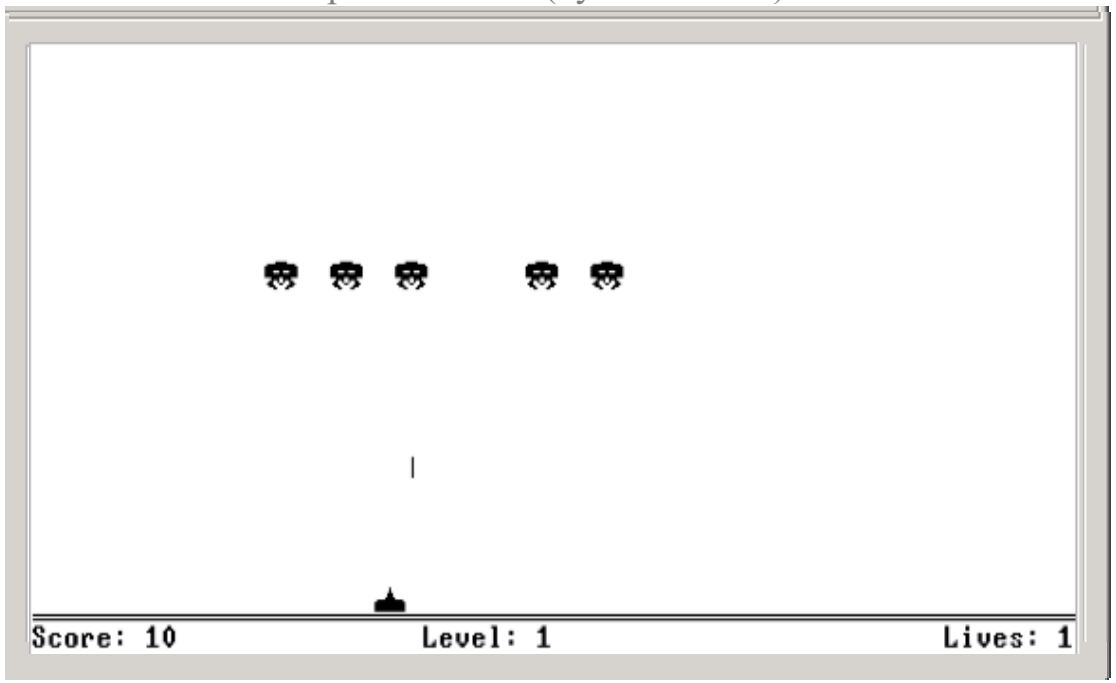
- The language
- The operating system

Application development (project 9)

- Jack applications
- Application example
- Graphics optimization

Sprites

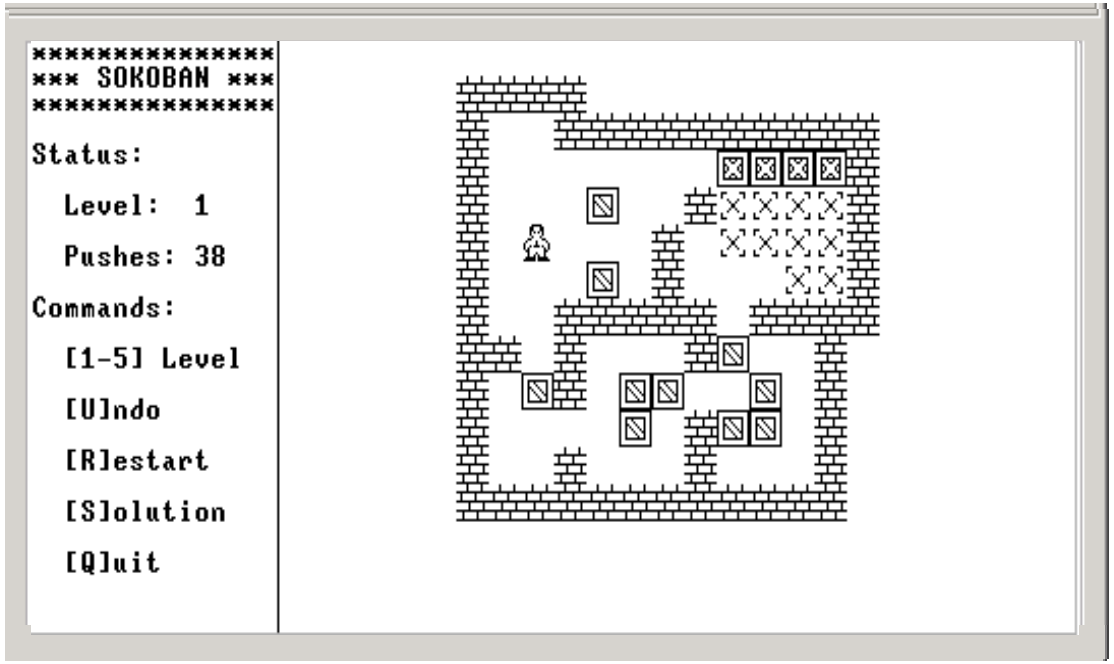
Space Invaders (by Ran Navok)



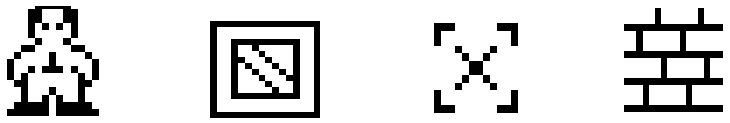
Basic graphical elements (*sprites*):

Sprites

Sokoban (by Golan Parashi)



Basic graphical elements (*sprites*):



Standard drawing: Using the OS library `Screen`

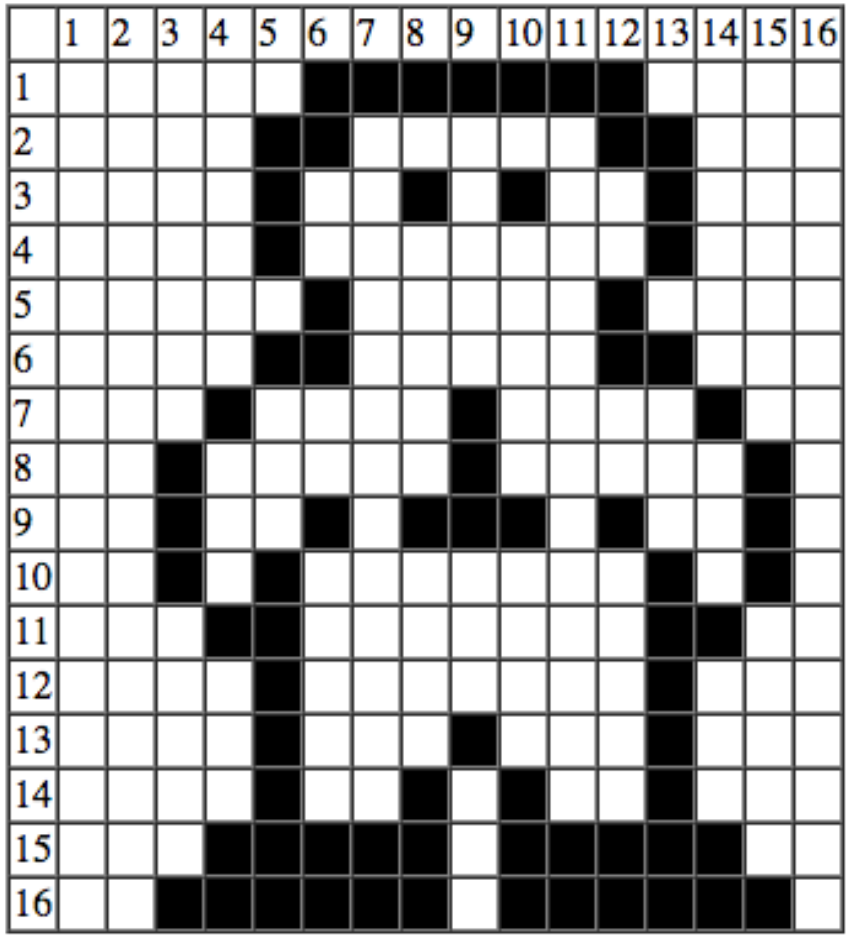


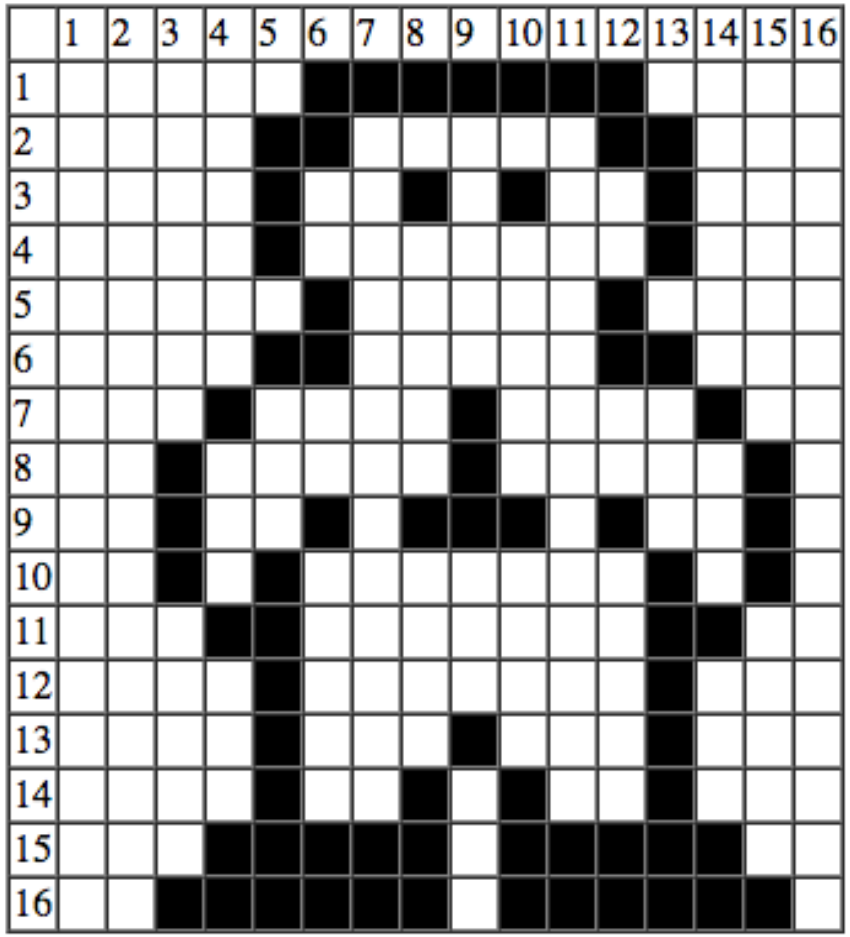
Image drawing code

```
// 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

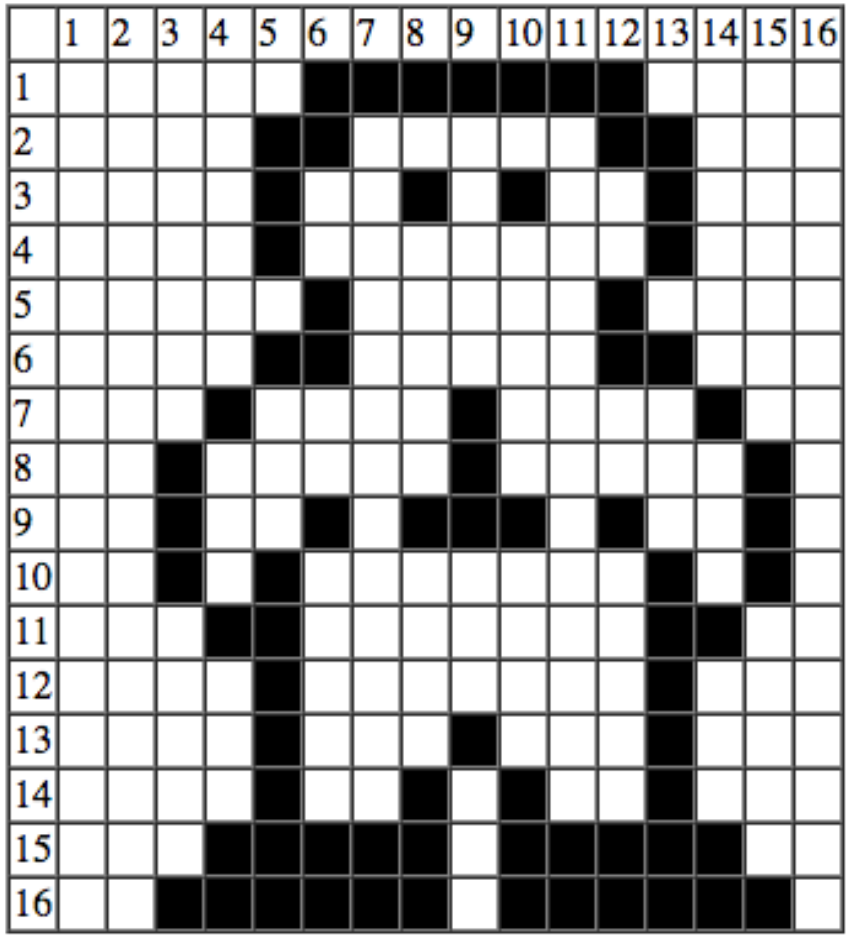
75 pixel drawing operations

Optimized drawing: Writing directly to the screen memory map



```
0000111111100000 = 4064
0001100000110000 = 6192
0001001010010000 = 4752
...
...
0111111011111100 = 32508
```

Optimized drawing: Writing directly to the screen memory map



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

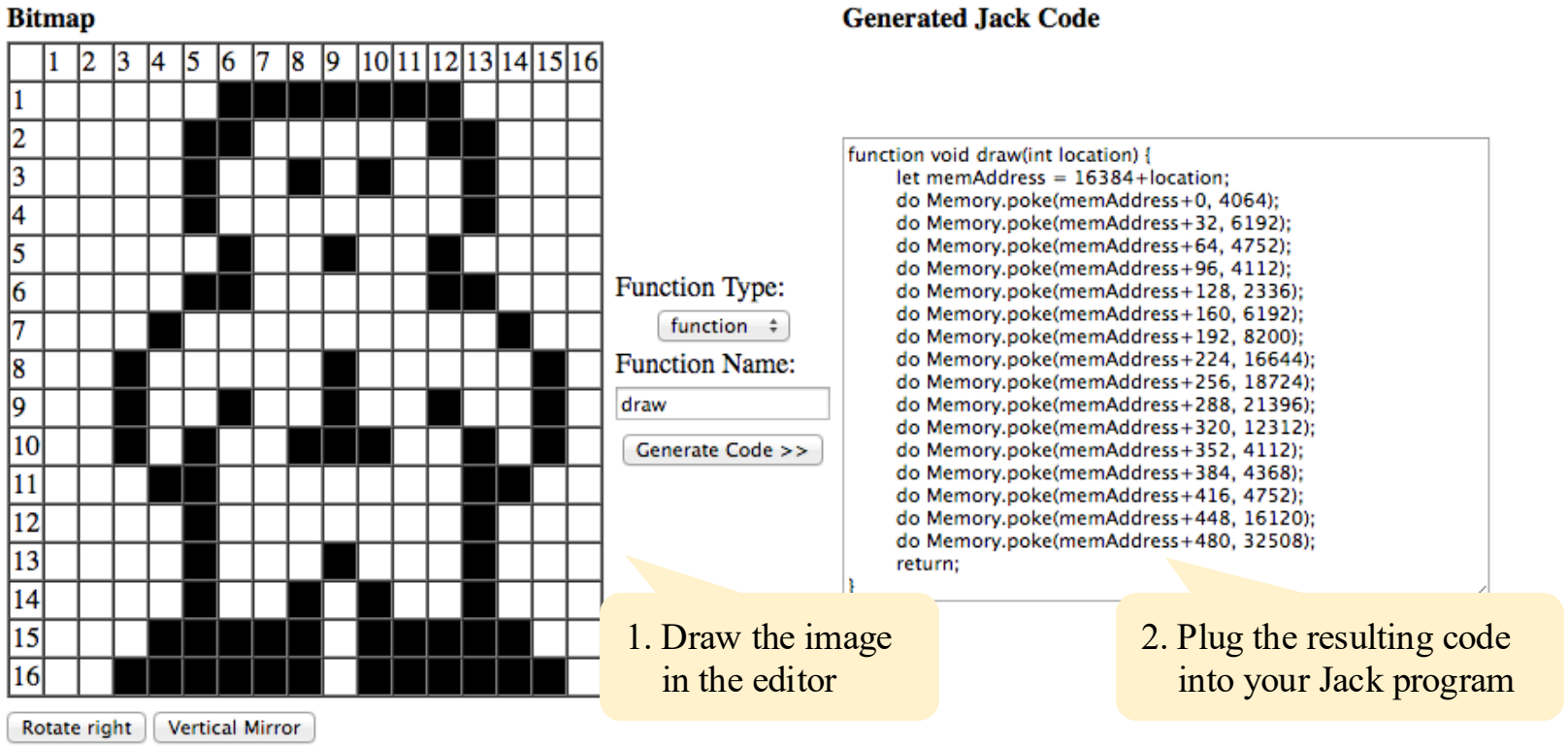
Image drawing code

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

Efficiency

16 memory write operations

Bitmap editor



Developed by Golan Parashi (desktop version) and Eric Umble (IDE version)

Best Practice

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

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

Perspective

Jack is a simple language,

Featuring essential elements of:

- Procedural programming
- OO programming

Motivation:

Simple Java-like language

Limitations

- Primitive type system
- Few control structures
- Some peculiar syntax
- No inheritance

Motivation

a minimal language that can be implemented by a simple compiler

Other features

- Weakly typed
- Full memory access

Motivation

Gives the programmer full control of the computer platform, especially for writing the OS.