Chapter 9

High Level Language

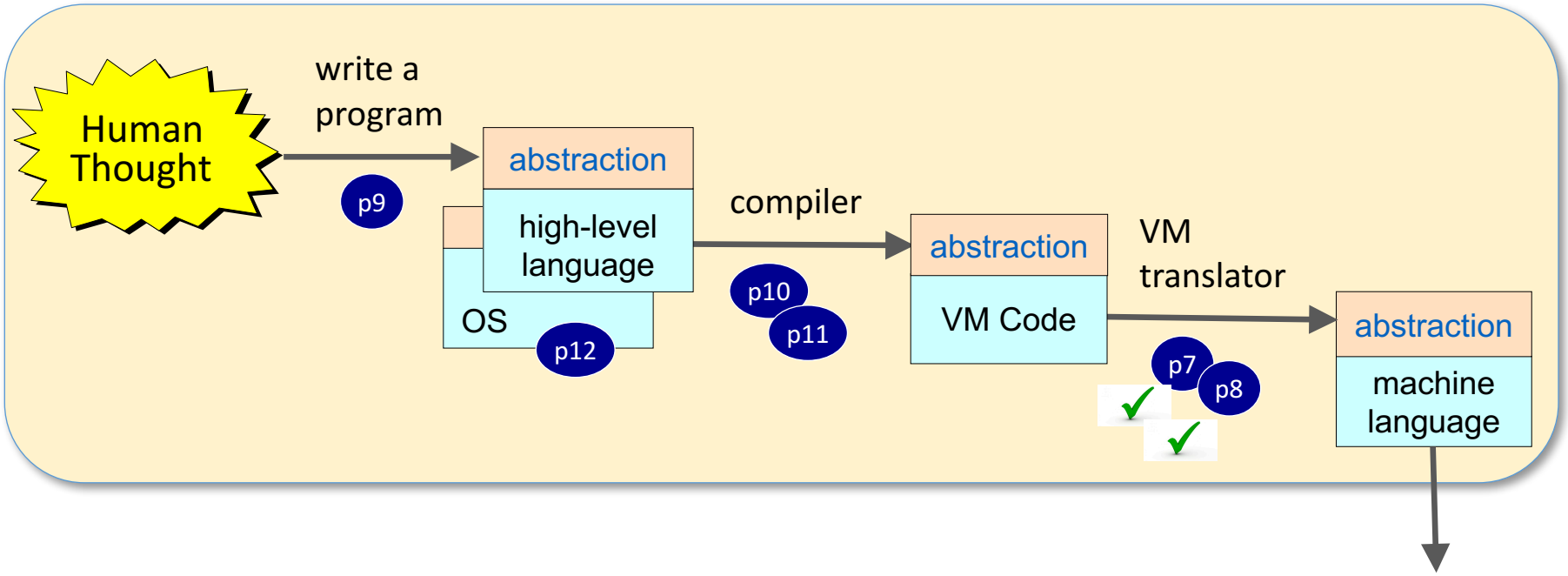
These slides support chapter 9 of the book

The Elements of Computing Systems

By Noam Nisan and Shimon Schocken

MIT Press

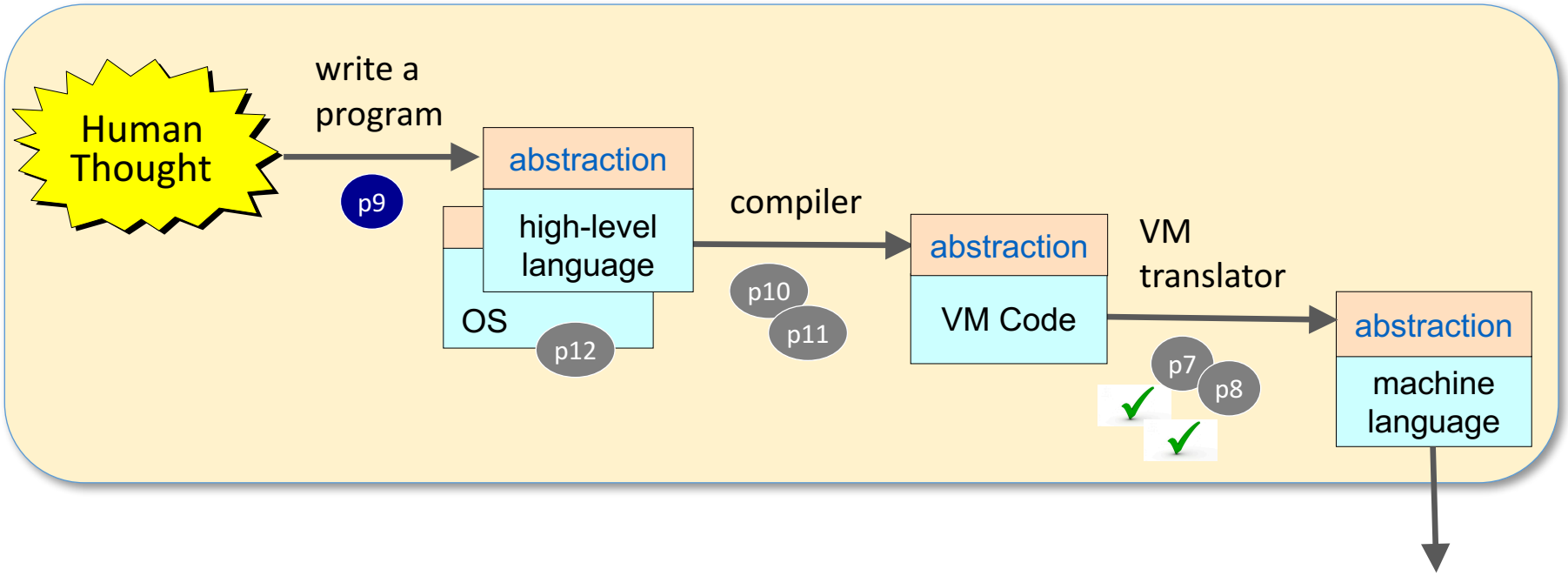
Road map



Agenda

- ✓ Projects 7,8: developing a VM translator
- Projects 10,11: developing a Jack compiler
- Module 12: developing an operating system
- Project 9: Jack overview

Road map



Agenda

- Projects 7,8: developing a VM translator
- Projects 10,11: developing a Jack compiler
- Module 12: developing an operating system
- **Project 9: Jack overview**

Jack in a nutshell (using an example we saw before)

```
/** Demo: working with Point objects. */
class Main {
  function void main() {
     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.println();
     do Output.printInt(p1.distance(p3));
     return;
  }
}
```

```
/** Represents a Point object. */
class Point {
  field int x, y;
  static int pointCount;
  /** Constructs a new point */
  constructor Point new(int ax, int ay) {
      let x = ax;
      let y = ay;
      let pointCount = pointCount + 1;
      return this;
  }
  // More Point methods...
}
```

- A simple, Java-like language
- Object-based, no inheritance
- Multi-purpose
- Lends itself to interactive apps
- Can be learned in about an hour

Take home lessons

An inside view of how high-level OO languages ...

- are designed
- handle primitive types and class types
- create, represent, and dispose objects
- deal with strings, arrays, and lists
- interact with the host OS
- ... and many more issues

Additional experience with ...

- abstrtaction – implementation
- OO programming
- application design and implementation.

High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

Jack language specification

- Syntax
- Data types
- Classes
- Methods

Hello world

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

Language constructs

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

- Comments:

  ```
  /** API block comment */

  /* block comment */

  // in-line comment
  ```

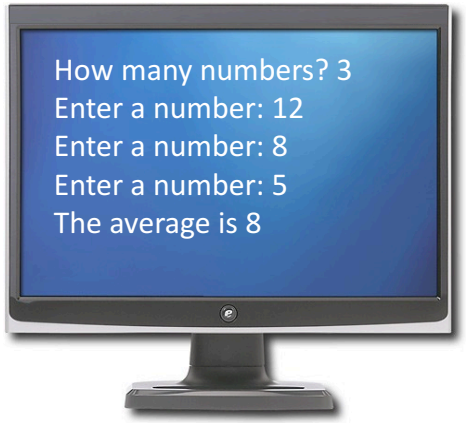
- White space

  (ignored)

Procedural processing

Jack code

```
// Inputs some 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;
   }
}
```



Language constructs

Jack code

```
// Inputs some 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;
   }
}
```

- A Jack program is a collection of one or more Jack classes, one of which must be named `Main`
- The `Main` class must have at least one function, named `main`
- Program's entry point: `Main.main`

Language constructs

Jack code

```
// Inputs some 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 data types:

- Primitive:
  - `int`
  - `char`
  - `boolean`
- Class types:
  - OS: `Array`, `String`, ...
  - Additional ADT's can be defined and used, as needed

Language constructs

Jack code

```
// Inputs some 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;
   }
}
```

Flow of control:

- `if / if...else`
- `while`
- `do`

Language constructs

Jack code

```
// Inputs some 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;
   }
}
```

Arrays:

- `Array` is implemented as part of the standard class library
- Jack arrays are not typed

Language constructs

Jack code

```
// Inputs some 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;
   }
}
```

OS services:

- `Keyboard.readInt`
- `Output.printString`
- `Output.printInt`
- More...

High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

Jack language specification

- Syntax
- Data types
- Classes
- Methods

OO programming

`Fraction` API

```
class Fraction {

    /** Constructs a (reduced) fraction from the given numerator and denominator */
    constructor Fraction new(int x, int y)

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

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

    /** Disposes this fraction. */
    method void dispose()

    /** Prints this fraction in the format x/y. */
    method void print()
}
```

Example: fractions

```
2/3 + 1/5

(2/7 * 3/4) + 1/2

8/9 / (5/6 – 2/19)
```

OO programming: using a class

`Fraction` API

```
class Fraction {

    /** Constructs a (reduced) fraction from the given numerator and denominator */
    constructor Fraction new(int x, int y)

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

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

    /** Disposes this fraction. */
    method void dispose()

    /** Prints this fraction in the format x/y. */
    method void print()
}
```

Jack class

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

Abstraction – implementation

- Users of an abstraction need know nothing about its implementation
- All they need is the class interface (API)

OO programming: building a class

`Fraction` class

```
/** Represents the Fraction type and related operations. */
class Fraction {

   field int numerator, denominator;

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

   method int getDenominator() {
      return denominator;
   }

   // More Fraction methods follow.

} // Fraction class
```

field, aka *property*, aka *member variable*

the only way to access field values from outside the class is through accessors

Some Jack class

```
class Foo {
   ...
   var Fraction x;
   let x = Fraction.new(5,17);
   do Output.printInt(x.numerator);       // not allowed
   do Output.printInt(x.getNumerator());  // ok
   ...
```

OO programming: building a class

`Fraction` class

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

   /** Constructs a (reduced) fraction from the given numerator and denominator. */
   constructor Fraction new(int x, int y) {
      let numerator = x;  let denominator = y;
      do reduce();     // reduces the fraction
      return this;     // returns the base address of the new object
   }

   // Reduces this fraction.
   method void reduce() {
      var int g;
      let g = Fraction.gcd(numerator, denominator);
      if (g > 1) {let numerator = numerator / g; let denominator = denominator / g;}
      return;
   }

   // Computes the greatest common divisor of the given integers.
   function int gcd(int a, int b) {
      var int r;
      while (~(b = 0)) {              // applies Euclid's algorithm.
         let r = a - (b * (a / b));   // r  = remainder of the integer division a/b
         let a = b; let b = r; }
      return a;
   }
}      // More Fraction code in next slide
```

- `this`: a reference to the current object (base address)
- a constructor must return the (base address of) the newly created object

a subroutine must terminate with a `return` command

Jack subroutines:

- methods
- constructors
- functions

OO programming: building a class

`Fraction` class

```
/** Represents the Fraction type and related operations. */
class Fraction {

   field int numerator, denominator;

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

   // More Fraction methods follow.

} // Fraction class
```

OO programming: building a class

`Fraction` class

```
/** Represents the Fraction type and related operations. */
class Fraction {

   field int numerator, denominator;

   /** Prints this fraction in the format x/y. */
   method void print() {
      do Output.printInt(numerator); do Output.printString(”/”);
      do Output.printInt(denominator);
      return;
   }

   // More Fraction methods follow.

} // Fraction class
```

OO programming: building a class

`Fraction` class

```
/** Represents the Fraction type and related operations. */
class Fraction {

   field int numerator, denominator;

   /** Disposes this fraction. */
   method void dispose() {
      do Memory.deAlloc(this);   // uses an OS routine to recycle the object's memory.
      return;
   }

} // Fraction class
```

Garbage collection

- Jack has no garbage collection
- Objects must be disposed explicitly
- Best practice: every class that has a constructor should also feature a dispose method.

OO programming: object representation

Fraction class

```
/** Represents the Fraction type and related operations. */
class Fraction {

   field int numerator, denominator;

   /** Constructs a (reduced) fraction from the given numerator and denominator */
   constructor Fraction new(int x, int y) {
      let numerator = x;  let denominator = y;
      do reduce();
      return this;
   }

   // More Fraction methods
}
```

The compiled constructor's code includes OS calls that allocate and manage RAM space for representing the new object

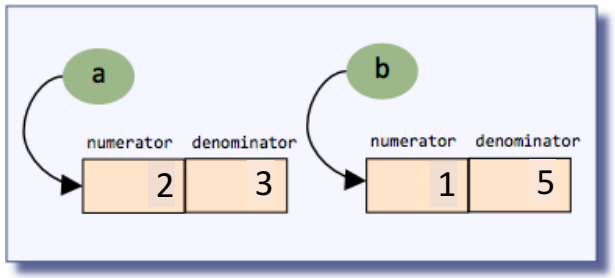
Issues:
- allocating memory
- de-allocating memory

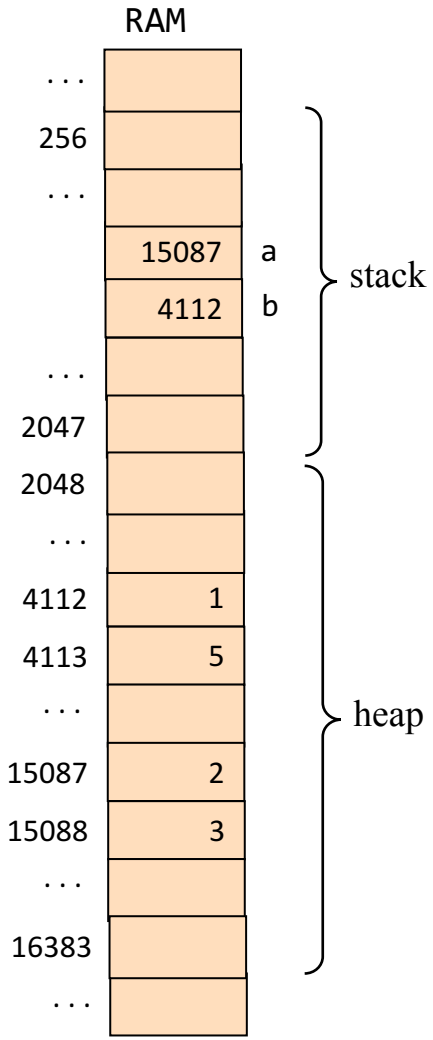
(handled by the compiler and the OS)

Client code

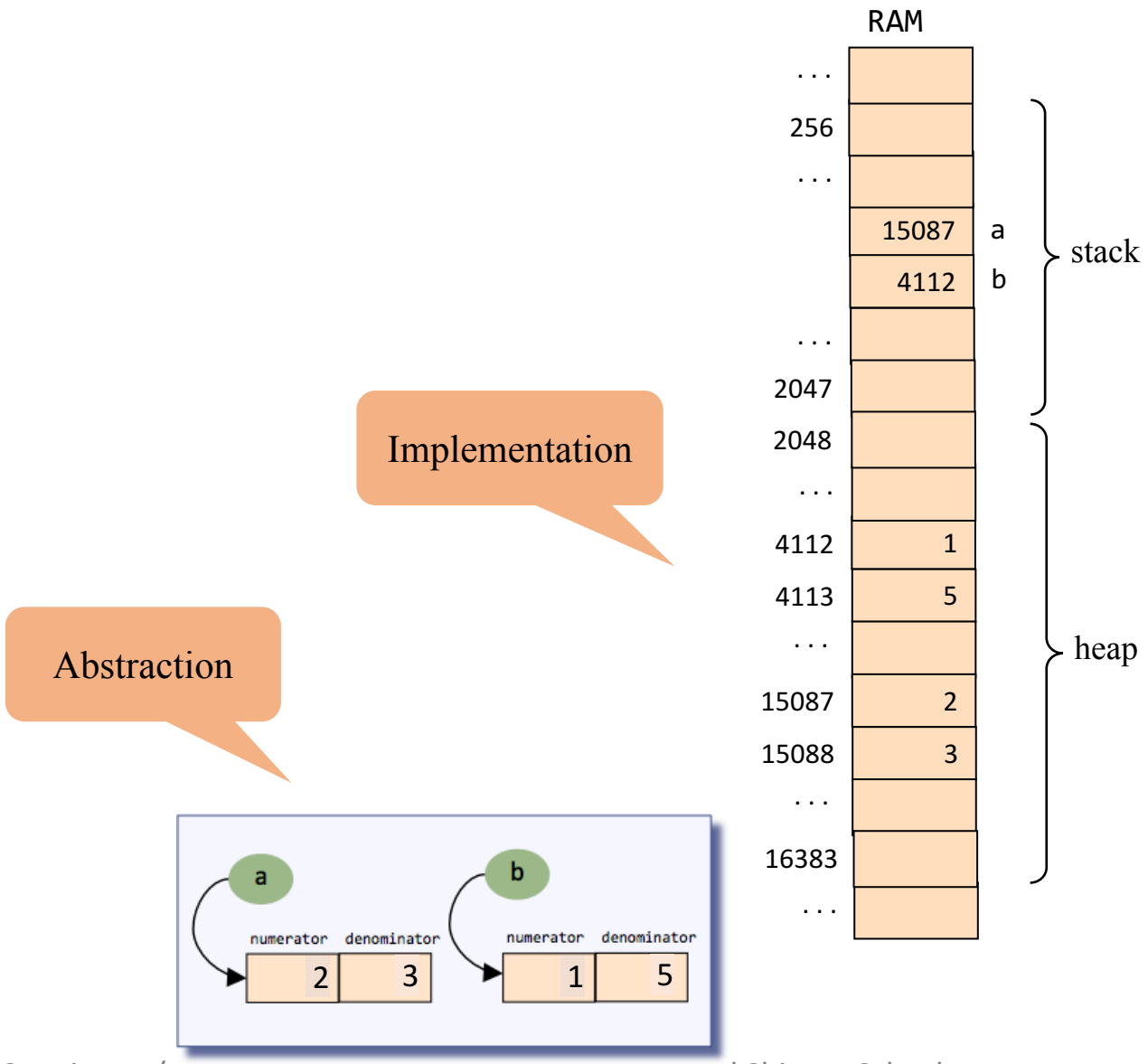
```
...
var Fraction a, b, c;
...
let a = Fraction.new(2,3);
let b = Fraction.new(1,5);
...
```

The client's view





OO programming: object representation



High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

Jack language specification

- Syntax
- Data types
- Classes
- Methods

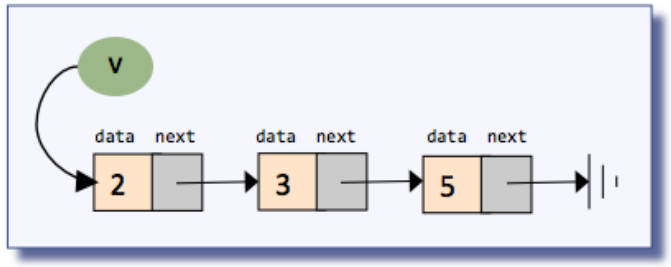
List processing

List definition

- the atom `null`, or
- an atom, followed by a list

Examples:	Abbreviated as:
`null`	`()`
`(5, null)`	`(5)`
`(3, (5, null))`	`(3, 5)`
`(2, (3, (5, null)))`	`(2, 3, 5)`

List example



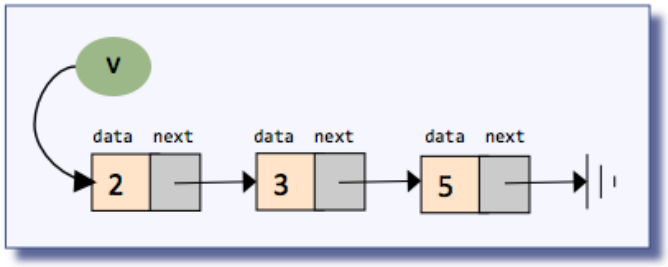
The list (2, (3, (5, null))),
commonly abbreviated as (2, 3, 5)

List processing

`List` API (partial)

```
/** Represents 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() {
}
```

List example



The list (2, (3, (5, null))),
commonly abbreviated as (2, 3, 5)

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

List processing: creation

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.
```

client code

```
...

var List v;

let v = List.new(5,null);

let v = List.new(3, v));

let v = List.new(2, v));

...
```

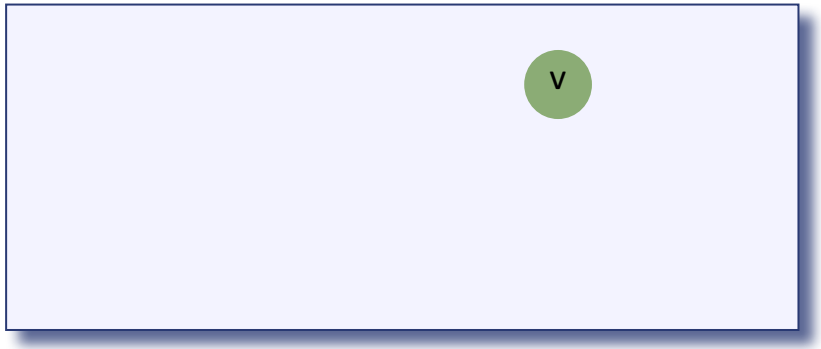
List processing: creation

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

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

   // More List methods
}
```

client code

```
...
var List v;
let v = List.new(5,null);
let v = List.new(3, v));
let v = List.new(2, v));
...
```



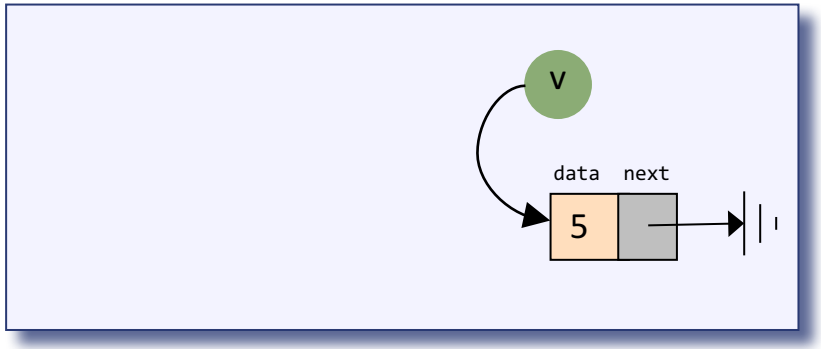
List processing: creation

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

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

   // More List methods
}
```

client code

```
...

var List v;

let v = List.new(5,null);

let v = List.new(3, v));

let v = List.new(2, v));

...
```



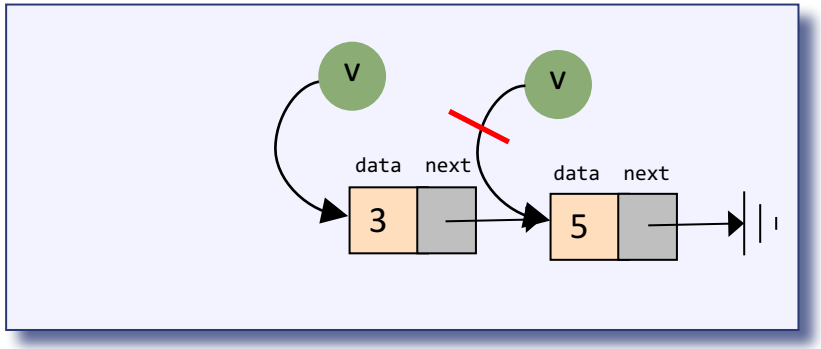
List processing: creation

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

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

   // More List methods
}
```

client code

```
...

var List v;

let v = List.new(5,null);

let v = List.new(3, v));

let v = List.new(2, v));

...
```



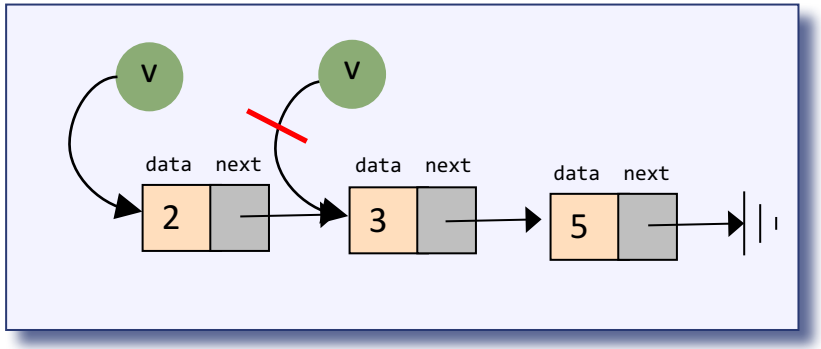
List processing: creation

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

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

   // More List methods
}
```

client code

```
...

var List v;

let v = List.new(5,null);

let v = List.new(3, v));

let v = List.new(2, v));

...
```



List processing: creation

`List` class

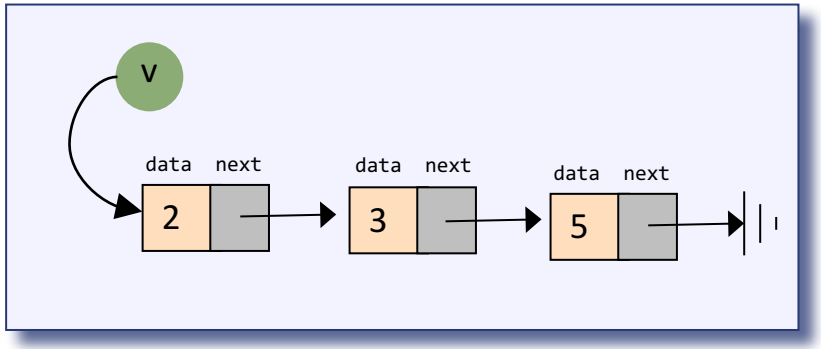
```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

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

   // More List methods
}
```

client code

```
...

var List v;

let v = List.new(5,null);

let v = List.new(3, v));

let v = List.new(2, v));

...
```

equivalent to:

```
var List v;
let v = List.new(5,null);
let v = List.new(2,List.new(3,v));
```



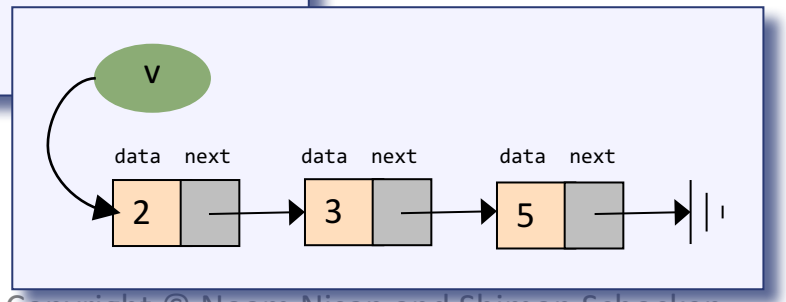
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

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

      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }

      return;
   }

   // More list methods

}
```

client code

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



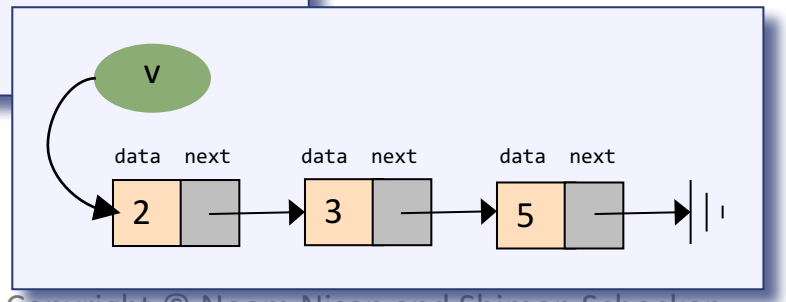
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

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

      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }

      return;
   }

   // More list methods

}
```

client code

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



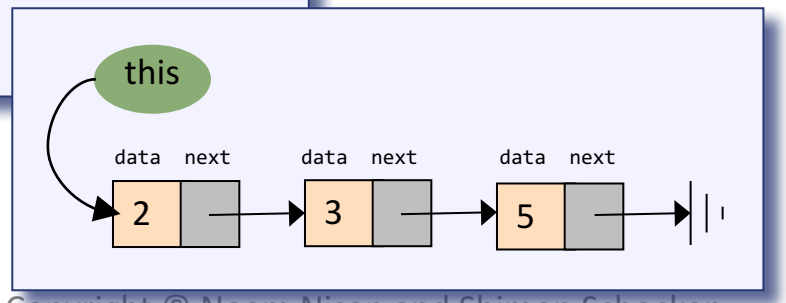
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

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

      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }

      return;
   }

   // More list methods

}
```

client code

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



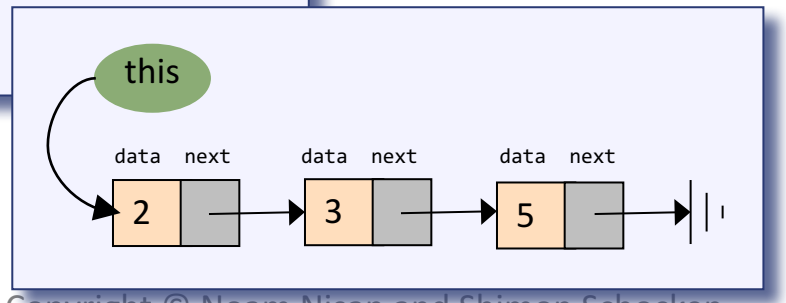
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Prints this list. */
   method void print() {
      var List current;      // creates a List variable and
      let current = this;    // initializes it to the first item of this list
      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }
      return;
   }

   // More list methods

}
```

client code

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



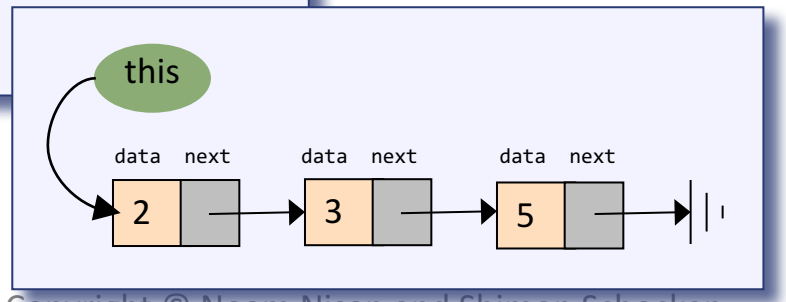
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Prints this list. */
   method void print() {
      var List current;      // creates a List variable and
      let current = this;    // initializes it to the first item of this list
      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }
      return;
   }

   // More list methods

}
```

client code

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



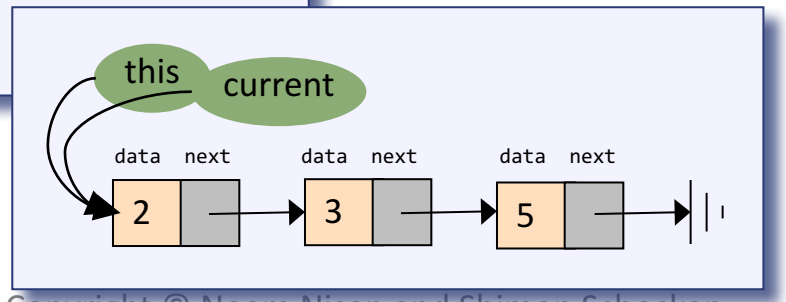
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Prints this list. */
   method void print() {
      var List current;      // creates a List variable and
      let current = this;    // initializes it to the first item of this list
      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }
      return;
   }

   // More list methods

}
```

client code

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



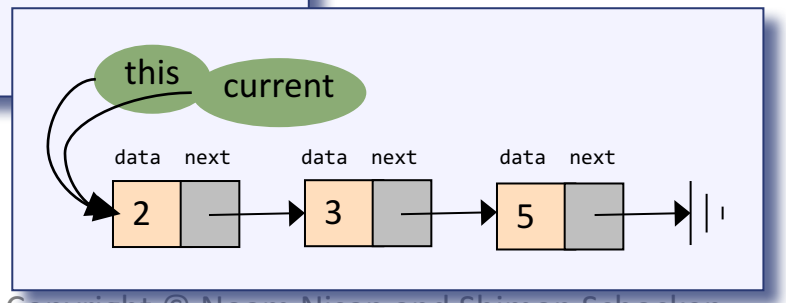
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Prints this list. */
   method void print() {
      var List current;     // creates a List variable and
      let current = this;   // initializes it to the first item of this list
      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }
      return;
   }

   // More list methods

}
```

client code

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



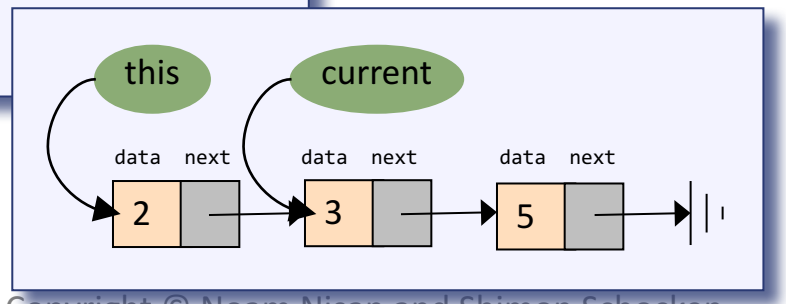
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Prints this list. */
   method void print() {
      var List current;     // creates a List variable and
      let current = this;   // initializes it to the first item of this list
      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }
      return;
   }

   // More list methods

}
```

client code

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



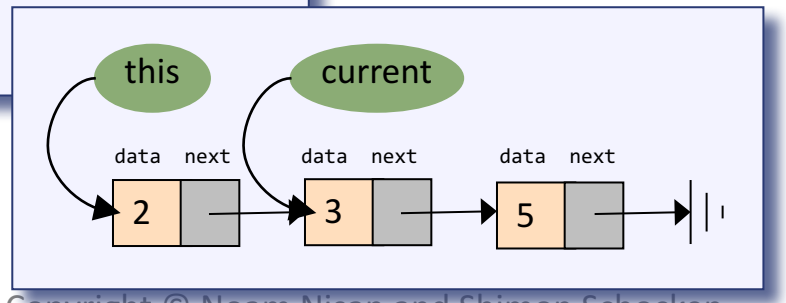
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Prints this list. */
   method void print() {
      var List current;     // creates a List variable and
      let current = this;   // initializes it to the first item of this list
      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }
      return;
   }

   // More list methods

}
```

client code

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



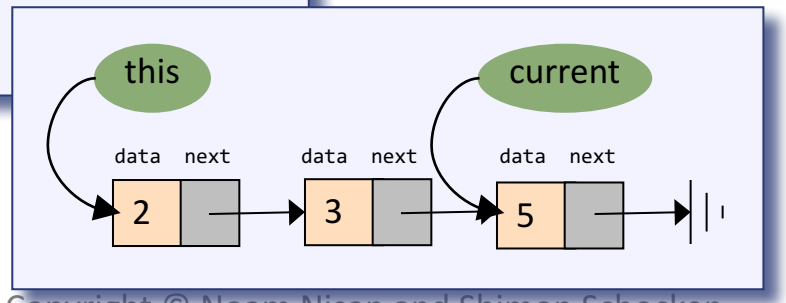
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Prints this list. */
   method void print() {
      var List current;     // creates a List variable and
      let current = this;   // initializes it to the first item of this list
      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }
      return;
   }

   // More list methods

}
```

client code

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



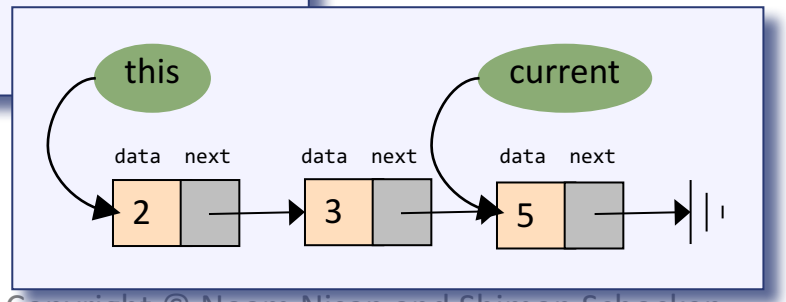
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Prints this list. */
   method void print() {
      var List current;     // creates a List variable and
      let current = this;   // initializes it to the first item of this list
      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }
      return;
   }

   // More list methods

}
```

client code

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



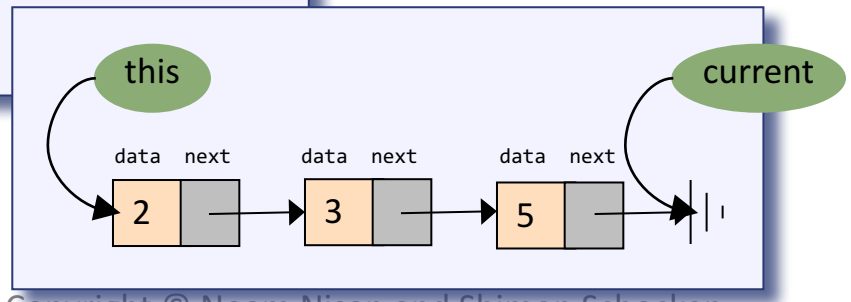
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Prints this list. */
   method void print() {
      var List current;     // creates a List variable and
      let current = this;   // initializes it to the first item of this list
      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }
      return;
   }

   // More list methods

}
```

client code

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



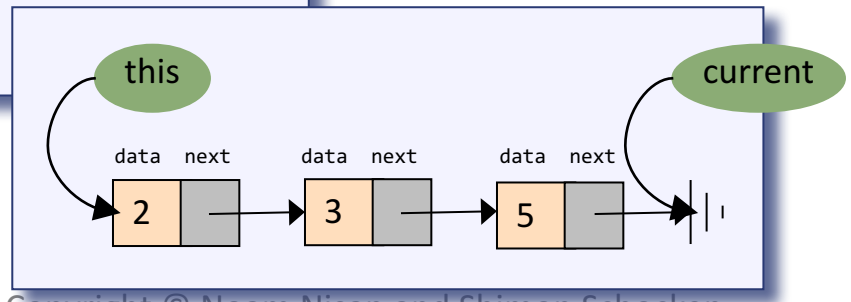
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

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

      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }

      return;
   }

   // More list methods

}
```

client code

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



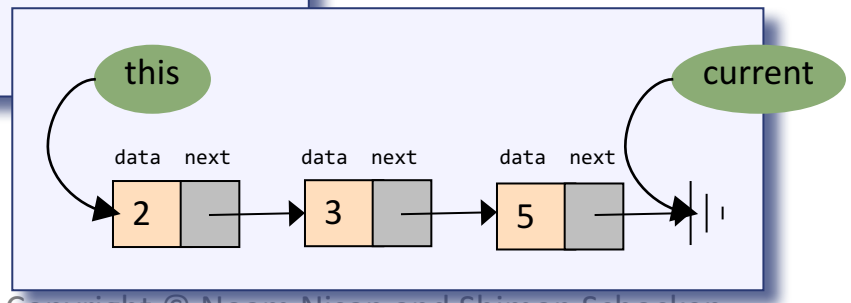
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

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

      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }

      return;
   }

   // More list methods

}
```

client code

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



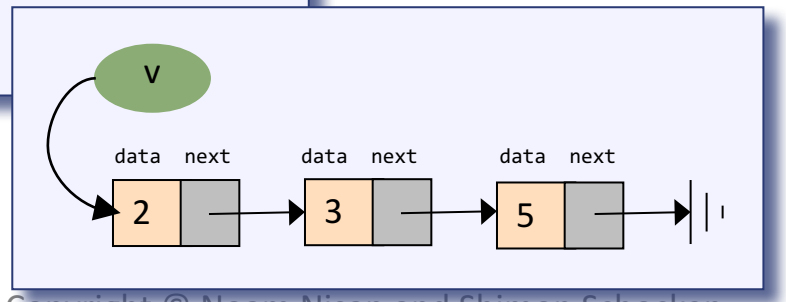
List processing: sequential access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

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

      while (~(current = null)) {
         do Output.printInt(current.getData());
         do Output.printChar(32);    // prints a space
         do current = current.getNext();
      }

      return;
   }

   // More list methods

}
```

client code

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



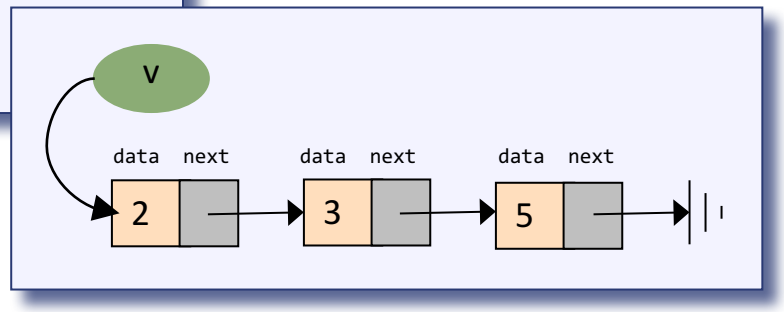
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



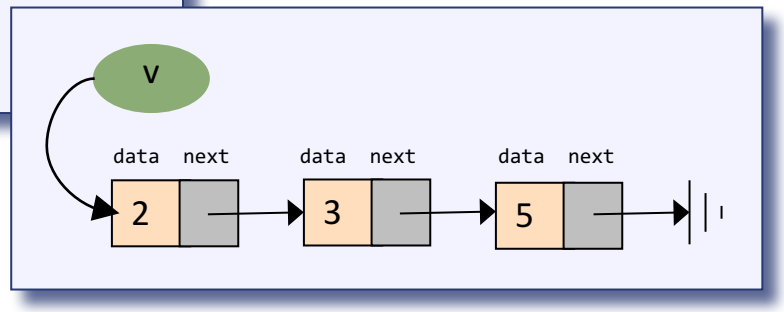
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



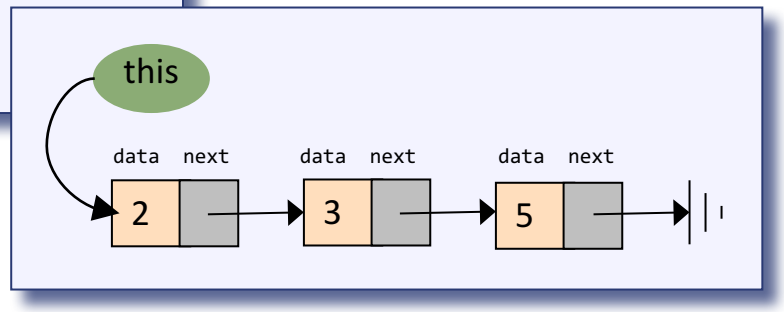
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



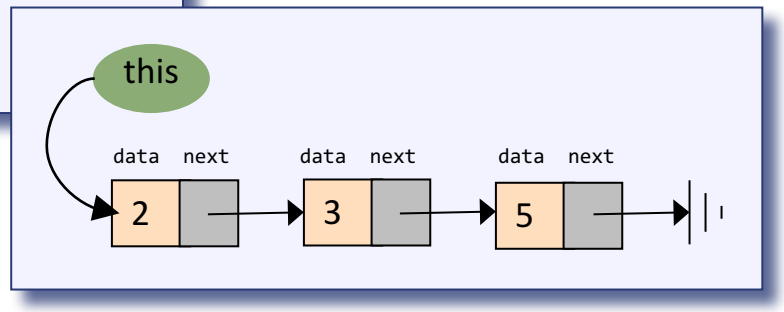
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



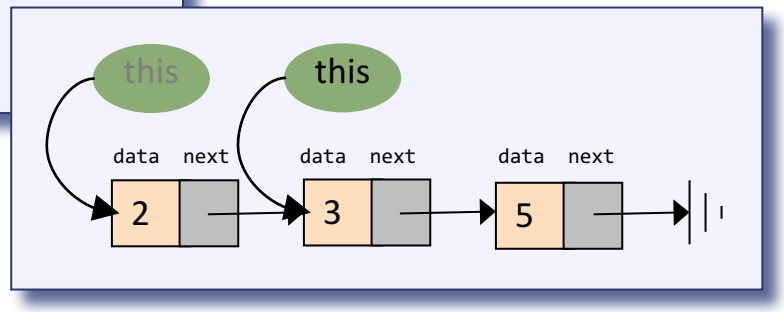
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



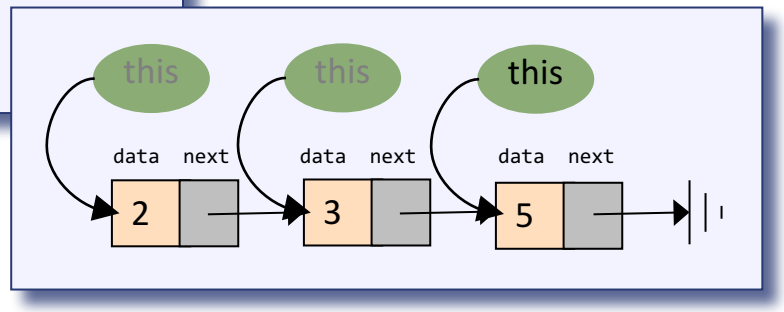
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



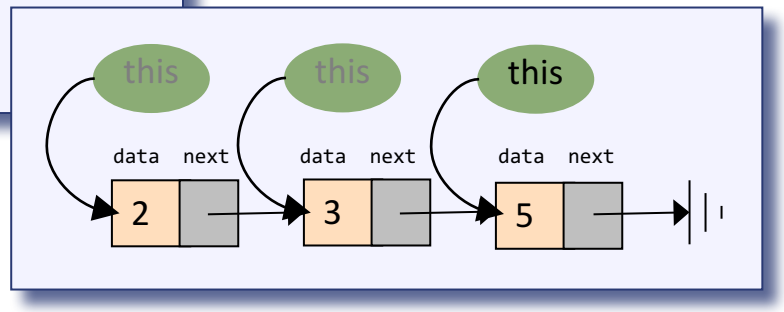
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



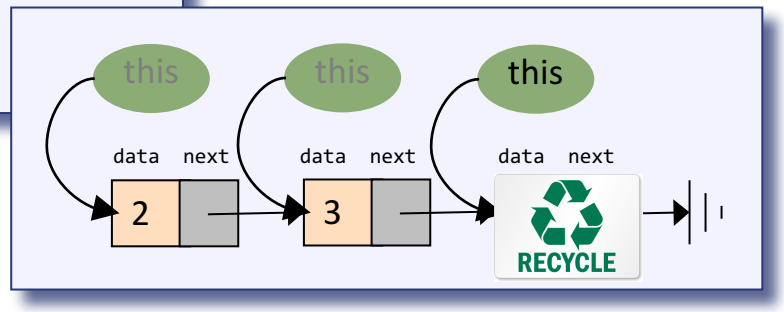
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



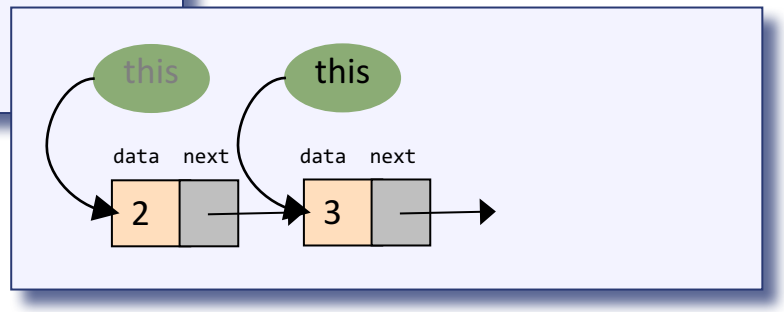
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



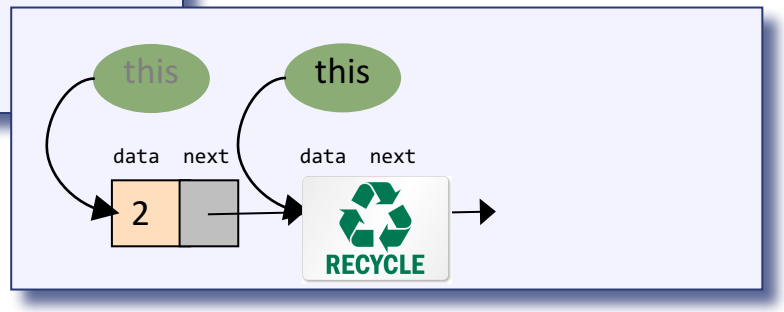
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



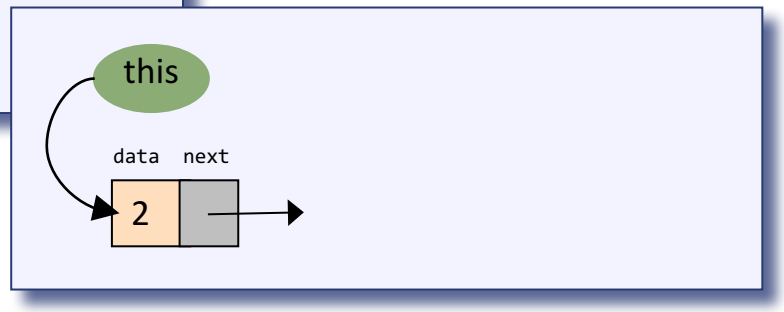
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



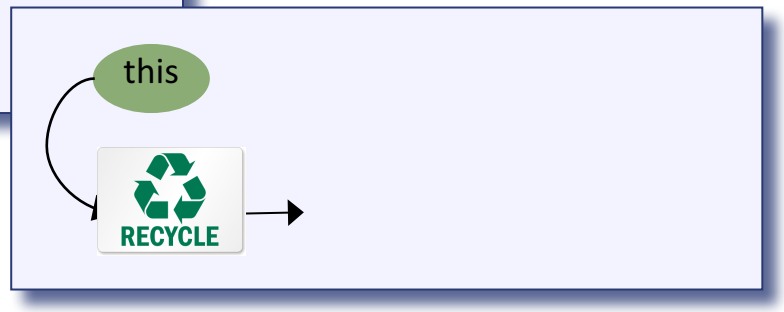
List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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



List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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

List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

client code

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

List processing: recursive access

`List` class

```
/** Represents a linked list of integers. */
class List {
   field int data;      // a list consists of a data field,
   field List next;     // followed by a list.

   /** Disposes this list */
   // by recursively disposing its tail
   method void dispose() {
      if (~(next = null)) {
         do next.dispose();
      }
      // Uses an OS routine to recycle this list
      do Memory.deAlloc(this);
      return;
   }
}
```

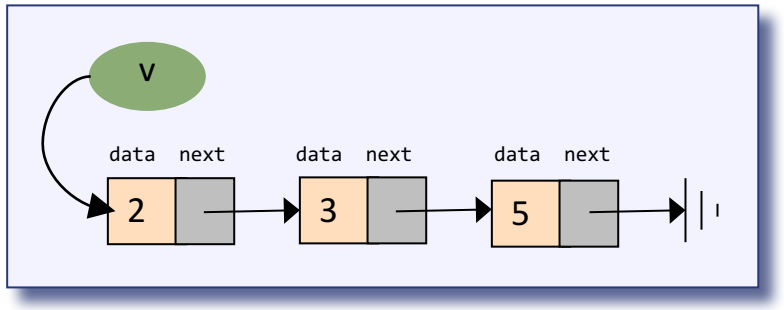
client code

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

Recap

We saw examples of:

- Building a list
- Processing a list, sequentially
- Processing a list, recursively
- Disposing a list



Who and what makes the magic work?

- When an object is constructed or disposed, the OS is responsible for allocating and reclaiming, respectively, the necessary RAM space
- List processing is implemented by pointer manipulation
- The pointer manipulation is done by the code that is generated by the compiler.

High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

Jack language specification

- Syntax
- Data types
- Classes
- Methods

Syntax

```
/** Procedural processing example */
class Main {
    /* Inputs some 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: white space / comments

```
/** Procedural processing example */
class Main {
   /* Inputs some 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: keywords

```
/** Procedural processing example */
class Main {
   /* Inputs some 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: symbols

```
/** Procedural processing example */
class Main {
   /* Inputs some 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: constants

```
/** Procedural processing example */
class Main {
   /* Inputs some 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* must be positive and in standard decimal notation, e.g., `1984`. Negative integers like `-13` are not constants but rather expressions consisting of a unary minus operator applied to an integer constant.
- *String constants* are enclosed within two quote (") characters and may contain any character except newline or double-quote. (These characters are supplied by the functions `String.newLine()` and `String.doubleQuote()` from the standard library.)
- *Boolean constants* can be true or false.
- The `null` constant signifies a null reference.

Syntax: identifiers

```
/** Procedural processing example */
class Main {
    /* Inputs some 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-Z, a-z), digits (0-9), and "_". The first character must be a letter or "_".

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

Recap

```
/** Procedural processing example */
class Main {
    /* Inputs some 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

High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

Jack language specification

- Syntax
- Data types
- Classes
- Methods

Data types

```
/** Procedural processing example */
class Main {
    /* Inputs some 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;
      }
      ...
}
```

Primitive types

- `int`:
  Non-negative 2's-complement 16-bit integer, i.e. an integer in the range 0,..., 32767
- `boolean`:
  `true` or `false`
- `char`:
  Integer values representing characters

Class types

- OS types: `String`, `Array`
- User-defined types: `Fraction`, `List`, ...

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

Type conversions

Characters and integers can be converted into each other, as needed:

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

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 memory address 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 Fraction x; // a Fraction object has two int fields: numerator and denominator.
let x = arr;    // sets x to the base address of the memory block representing the array [2,5]
do x.print()    // prints 2/5 (using the print method of the Fraction class)
```

High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

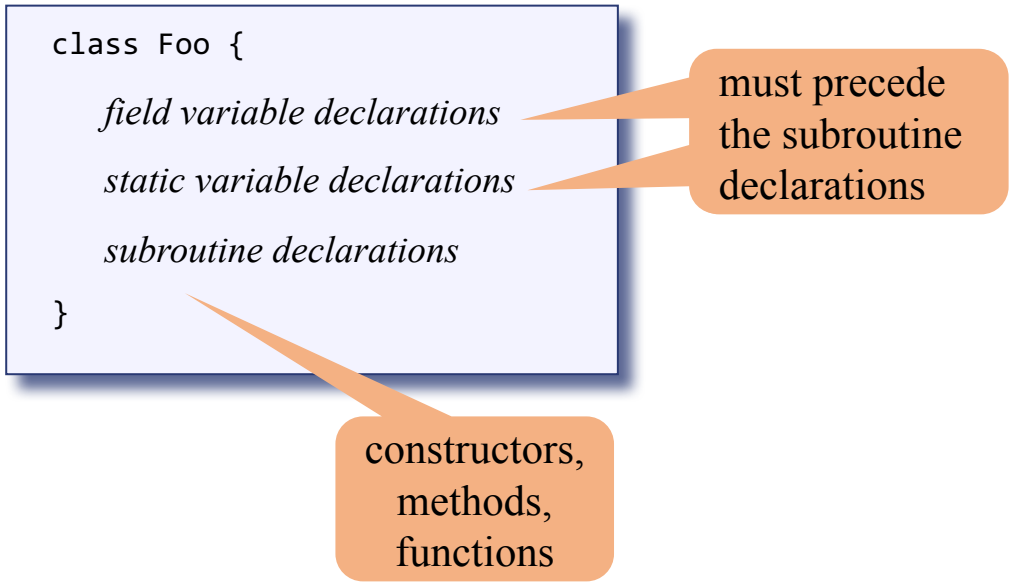
Jack language specification

- Syntax
- Data types
- Classes
- Methods

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.

`Foo.jack` file



Utility classes

`Math` class API (example)

Provides various mathematical operations.

- `function int abs(int x)`: returns the absolute value of `x`.
- `function int multiply(int x, int y)`: returns the product of `x` and `y`.
- `function int divide(int x, int y)`: returns the integer part of `x/y`.
- `function int min(int x, int y)`: returns the minimum of `x` and `y`.
- `function int max(int x, int y)`: returns the maximum of `x` and `y`.
- `function int sqrt(int x)`: returns the integer part of the square root of `x`.

- A class that contains only functions ("static methods")
  can be called a "utility class"
- (no fields, constructors, or methods)
- Offers a "library of services"

Classes that represent entities (objects)

- Examples: `Fraction`, `List`, `String`, ...
- A class that contains at least one method
- Used to represent an object type and operations on this object
- Typically contains fields and methods
- Can also contain functions, recommended for "helper" purpose only

Best practice:

Don't mix library functionality and object representation in the same class.

Typical bad example....

`Fraction` class

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

   /** Constructs a (reduced) fraction from the given numerator and denominator. */
   constructor Fraction new(int x, int y) {
      let numerator = x;  let denominator = y;
      do reduce();    // reduces the fraction
      return this;    // returns the base address of the new object
   }

   // Reduces this fraction.
   method void reduce() {
      var int g;
      let g = Fraction.gcd(numerator, denominator);
      if (g > 1) {let numerator = numerator / g; let denominator = denominator / g;}
      return;
   }

   // Computes the greatest common divisor of the given integers.
   function int gcd(int a, int b) {
      var int r;
      while (~(b = 0)) {              // applies Euclid's algorithm.
         let r = a - (b * (a / b));   // r  = remainder of the integer division a/b
         let a = b; let b = r; }
      return a;
   }

   // More Fraction methods
```

After refactoring:

`Fraction` class

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

   /** Constructs a (reduced) fraction from the given numerator and denominator. */
   constructor Fraction new(int x, int y) {
      let numerator = x;  let denominator = y;
      do reduce();    // reduces the fraction
      return this;    // returns the base address of the new object
   }

   // Reduces this fraction.
   method void reduce() {
      var int g;
      let g = Math.gcd(numerator, denom
      if (g > 1) {let numerator = numer
      return;
   }

   // More Fraction methods

}  // End of Fraction class
```

```
/** Library of mathematical functions. */
class Math {
   ...
   // Computes the greatest common divisor of the given integers.
   function int gcd(int a, int b) {
      var int r;
      while (~(b = 0)) {
         let r = a - (b * (a / b));
         let a = b; let b = r;
      }
      return a;
   }

   // More math functions

}  // End of Math class
```

Jack application

- A Jack *program*, or *application*, is a collection of one or more Jack classes, one of which must be named `Main`
- The `Main` class must have at least one function, named `main`
- Program's entry point: `Main.main`

Jack's standard class library / OS

Jack code

```
// Inputs some 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;
   }
}
```

OS purpose:

- Closes gaps between high-level programs and the host hardware
- Provides efficient implementations of commonly-used functions
- Provides efficient implementations of commonly-used ADTs

OS implementation:

- A collection of classes
- Similar to Java's standard class library

Jack's standard class library / OS

OS class	*Services*
`Math`	Common mathematical operations: `multiply(int,int)`, `sqrt(int)`, etc.
`String`	Represents string objects and related methods: `length()`, `charAt(int)`, etc.
`Array`	Represents array objects and related operations: `new(int)`, `dispose()`.
`Output`	Supports text output to the screen: `printString(String)`, `printInt(int)`, `println()`, etc.
`Screen`	Supports graphics output to the screen: `drawPixel(int,int)`, `setColor(boolean)`, `drawCircle(int,int,int)`, etc.
`Keyboard`	Supports input from the keyboard: `readLine(String)`, `readInt(String)`, etc.
`Memory`	Facilitates access to the host RAM: `peek(int)`, `poke(int,int)`, `alloc(int)`, `deAlloc(Array)`.
`Sys`	Supports execution-related services: `halt()`, `wait(int)`, etc.

Complete OS API: Nand2Tetris book / website

High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

Jack language specification

- Syntax
- Data types
- Classes
- Methods

Classes

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

Classes

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

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 `return` *value* or `return`.

Constructors

Constructor declaration

```
constructor ClassName constructorName (parameter-list) {
    local variable declarations
    statements
}
```

Constructors

- 0, 1, or more in a class
- Common name: `new`
- The constructor's type must be the name of the constructor's class
- The constructor must return a reference to an object of the class type.

```
class Point {
   field int x;
   field int y;
   ...
   /* Creates a Point. */
   constructor Point new(int ax, int ay) {
      let x = ax;
      let y = ay;
      return this;
   }
   ...
}
```

Variables

Variable kinds:

- *field* variables:
  object properties, can be manipulated by the class constructors and methods
- *static* variables:
  class-level variables,
  can be manipulated by the class subroutines
- *local* variables:
  used by subroutines, for local computations
- *parameter* variables:
  used to pass values to subroutines,
  behave like local variables

```
/** Represents a Point object. */
class Point {
   field int x, y;
   static int pointCount;
   ...
   method int bla(int z) {
      int foo;
      let foo = z * (x + y);
      return foo;
   }
   ...
}
```

Variables must be ...

- Declared before they are used
- Typed.

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

Subroutine calls

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 g();          // calls method g of the current class on this object
       do Foo.p(3);     // calls function p of the current class
       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 the b object
       let i = w(b.s(), Foo.t()); // calls method w on this object. The arguments are
                                  // the results of calling method s on object b,
                                  // and function or constructor t of class Foo
       ...
     }
}
```

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

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

Strings

Examples:

```
...
var String s;   // Creates an object variable (pointer), initialized to null
var char c;     // Creates a primitive variable, initialized to zero
...
// Suppose we want s to refer to the string "Hello World!":
let s = String.new(12);
// Followed by a loop that uses String's appendChar method to set s to "Hello World!"
// (loop code not shown)

// Alternatively, the Jack compiler allows using:
let s = "Hello World";  // "syntactic sugar"

// Accessing some character within a string:
let c = s.charAt(6);    // Sets c to the numeric value representing 'W'.
```

For more String and character operations, see the `String` class API

Arrays

Examples:

```
...
var Array arr;
var String helloWorld;
let helloWorld = “Hello World!”
...
let arr = Array.new(4);
let arr[0] = 12;
let arr[1] = false;
let arr[2] = Fraction.new(314/100);
let arr[3] = helloWorld;
...
```

- Jack arrays are ...
  - instances (objects) of the OS class `Array`
  - not typed
  - uni-dimensional
- Multi-dimensional arrays can be obtained by using an arrays of arrays.

End note: peculiar features of the Jack language

- The keyword `let`:
  must be used in assignments: `let x = 0;`
- The keyword `do`:
  must be used for calling a method or a function outside an expression:
  `do reduce();`
- The body of a statement must be within curly brackets, even if it contains a single statement: `if (a > 0) {return a;} else {return -a;}`
- All subroutine must end with a `return`
- No operator priority:
  - The following value if unpredictable: `2 + 3 * 4`
  - To enforce priority of operations, use parentheses: `2 + (3 * 4)`
- The language is weakly typed.

These concessions make the life of the compiler writer much easier.

High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

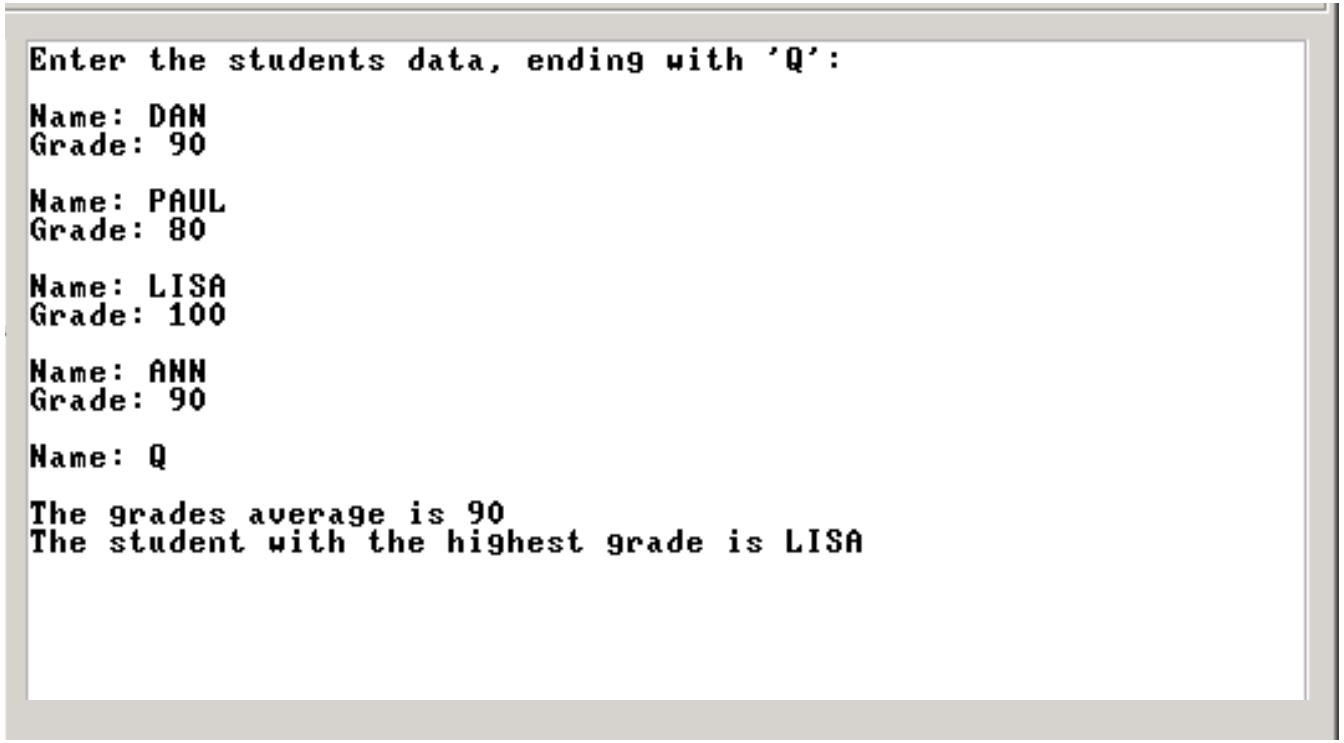
Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

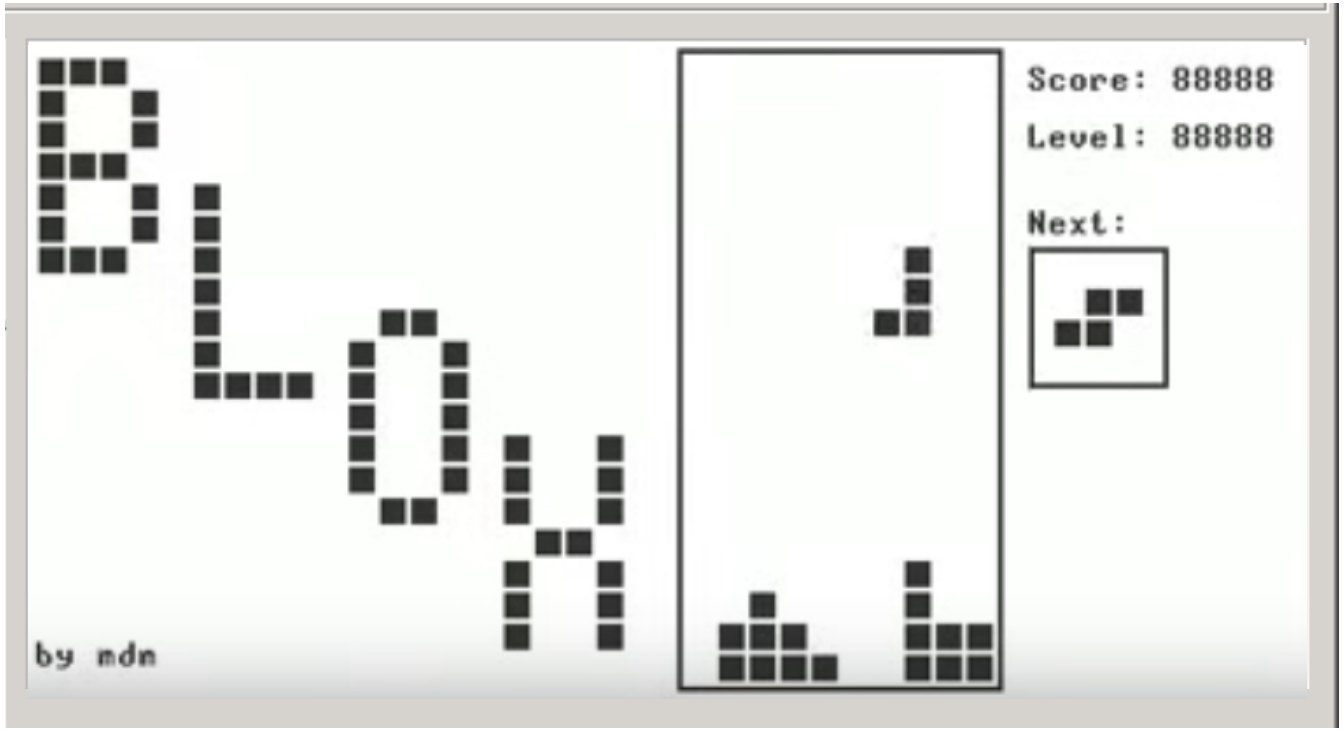
Jack language specification

- Syntax
- Data types
- Classes
- Methods

Sample applications: Stats

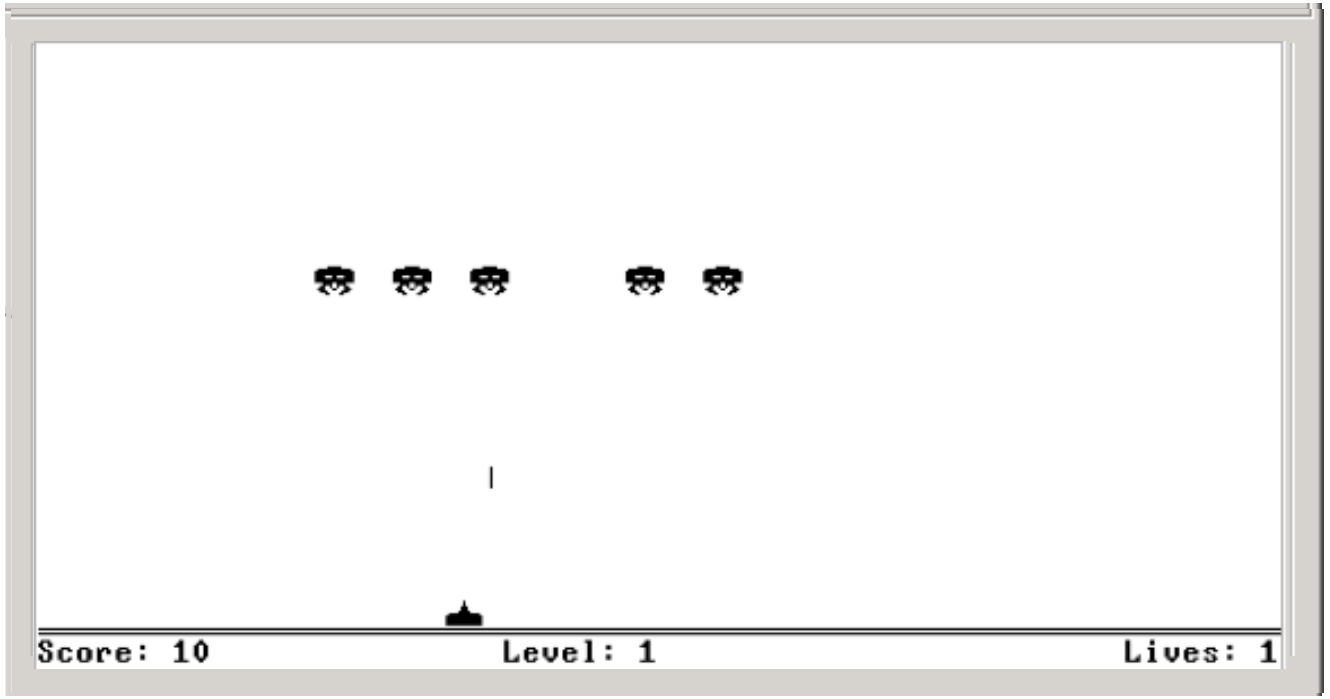


Sample applications: Tetris (by Marc Domink Migge)

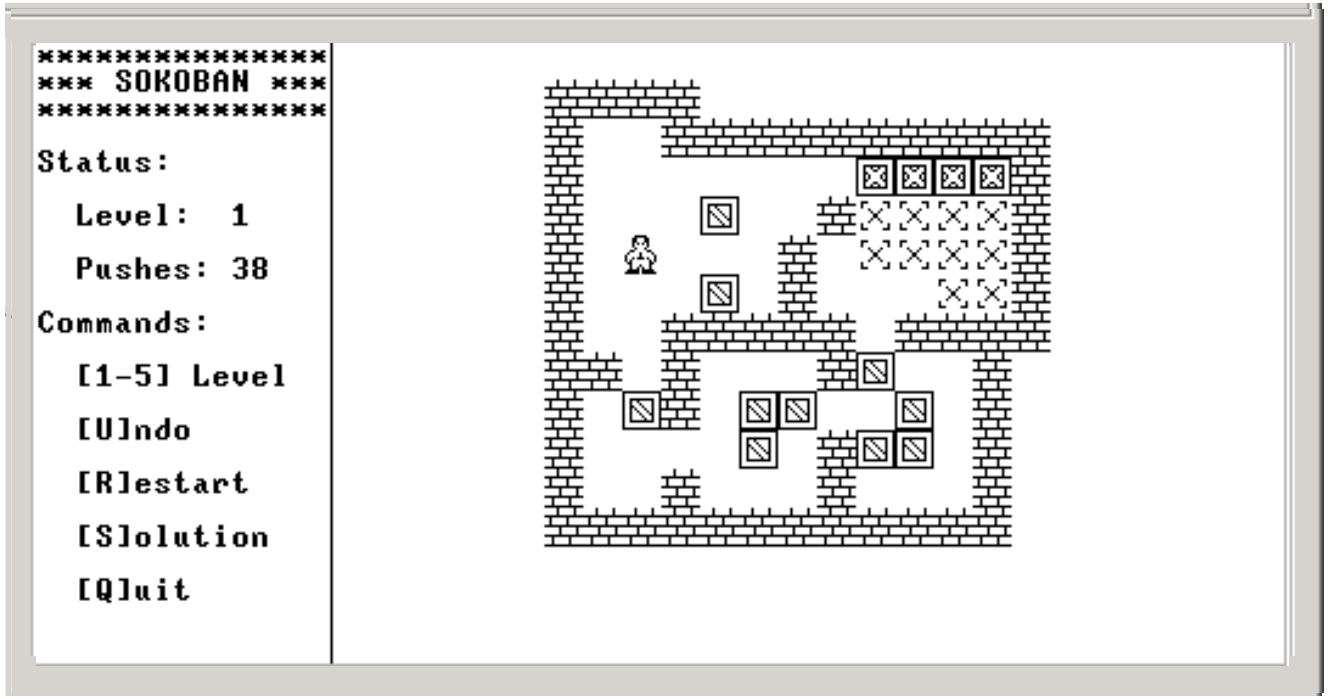


Sample applications: Bouncing Ball (by Gavin Stewart)

Sample applications: Space Invaders (by Ran Navok)



Sample applications: Sokoban (by Golan Parashi)



Developing a Jack application

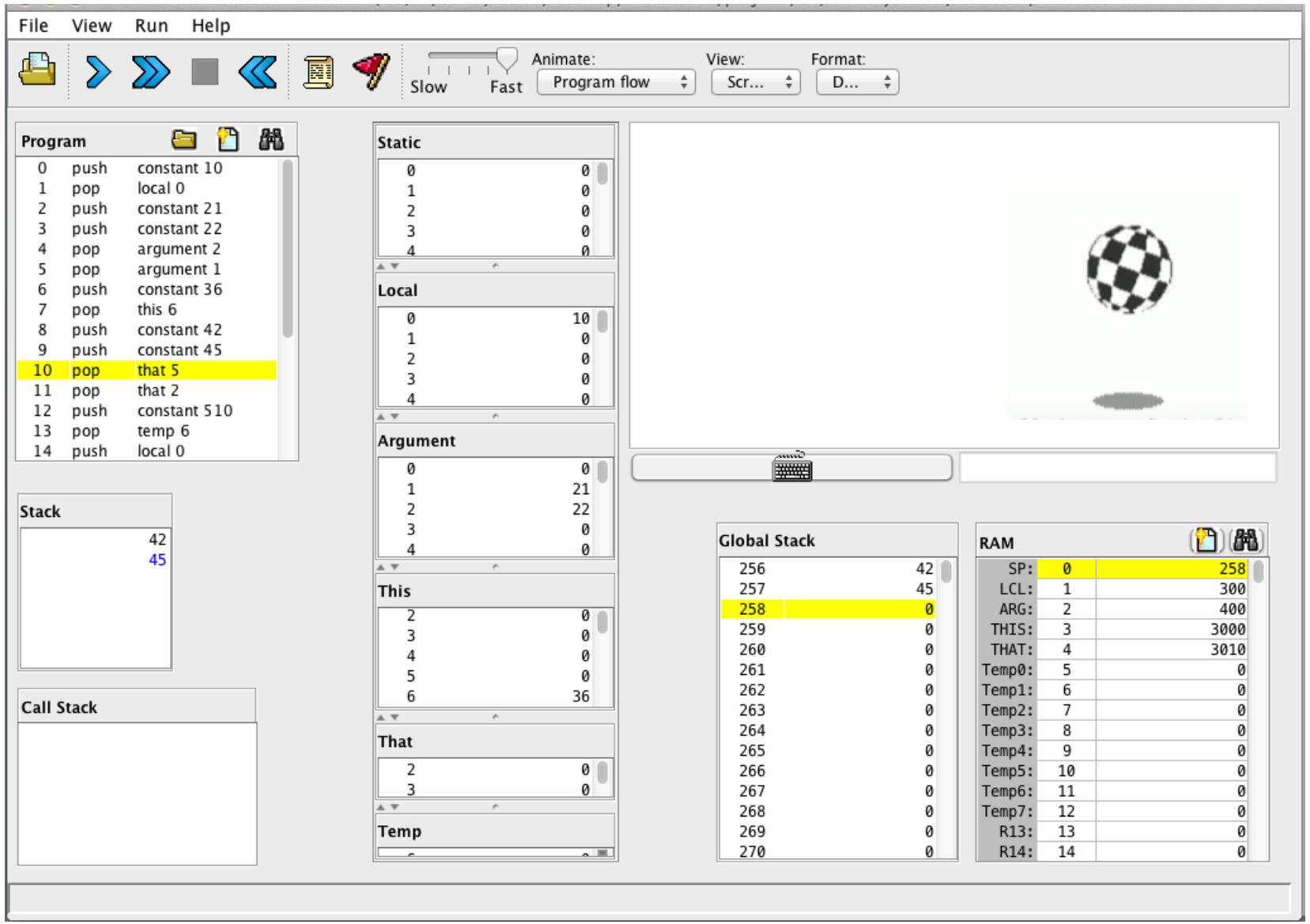
Put all the app files in one directory, whose name is the app name

Write / edit your Jack class files using a standard text editor

Compile your Jack files / directory using the supplied `JackCompiler` (available in `nand2tetris/tools`)

Execute your app by loading the app directory (which now contains the compiled `.vm` files) into the supplied VM emulator, and running the code

Running a Jack application



High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

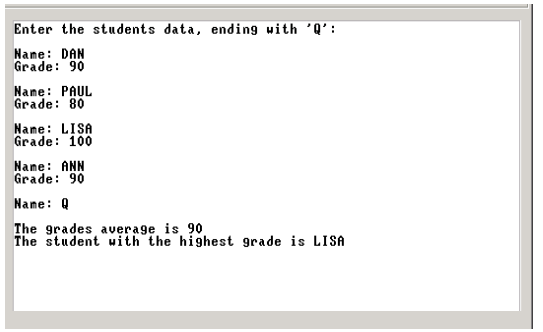
Jack language specification

- Syntax
- Data types
- Classes
- Methods

Handling output: text

Textual apps:

- Screen: 23 rows of 64 characters, b&w
- Font: featured by the Jack OS
- Output: Jack OS `Output` class



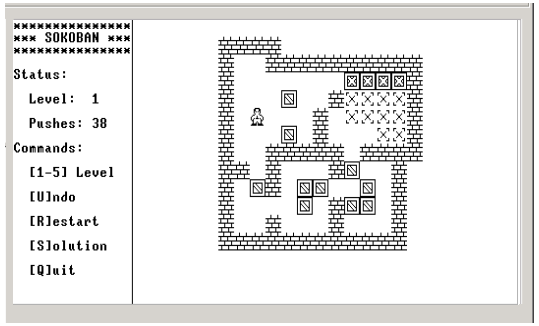

```
class Output {
   function void moveCursor(int i, int j)
   function void printChar(char c)
   function void printString(String s)
   function void printInt(int i)
   function void println()
   function void backSpace()
}
```

OS class, for handling textual output

Handling output: graphics

Graphical apps:

- Screen: 256 rows of 512 pixels, b&w
- Output: Jack OS `Screen` class (or do your own)



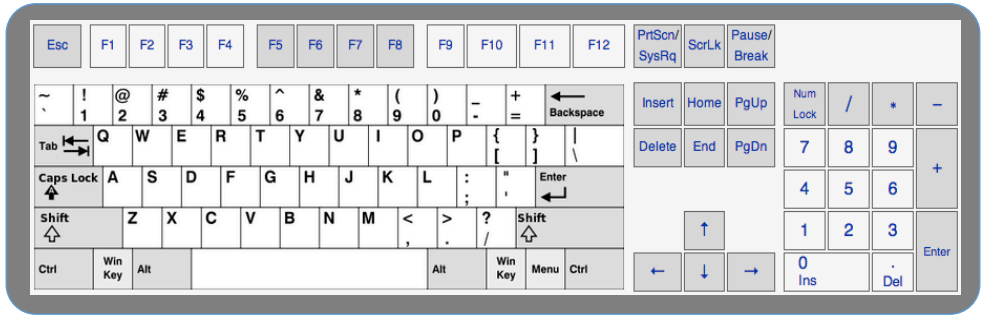

```
Class Screen {
   function void clearScreen()
   function void setColor(boolean b)
   function void drawPixel(int x, int y)
   function void drawLine(int x1, int y1, int x2, int y2)
   function void drawRectangle(int x1, int y1, int x2, int y2)
   function void drawCircle(int x, int y, int r)
}
```

OS class, for handling graphical output

Handling inputs

Input device:

- Standard keyboard
- Input programming: use the OS `Keyboard` class



```
Class Keyboard {

   function char keyPressed()

   function char readChar()

   function String readLine(String message)

   function int readInt(String message)

}
```

OS class, for handling input from the keyboard

The Jack 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

`Keyboard.keypress()`

returns the code of the currently pressed key, or `0` when no key is pressed

The Jack OS: Math

```
class Math {

   function void init()

   function int abs(int x)

   function int multiply(int x, int y)

   function int divide(int x, int y)

   function int min(int x, int y)

   function int max(int x, int y)

   function int sqrt(int x)

}
```

The Jack OS: `String`

```
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()
}
```

The Jack OS: `Array`

```
Class Array {

   function Array new(int size)

   method void dispose()

}
```

The Jack OS: Memory

```
class Memory {

   function int peek(int address)

   function void poke(int address, int value)

   function Array alloc(int size)

   function void deAlloc(Array o)

}
```

The Jack OS: Sys

```
Class Sys {

   function void halt():

   function void error(int errorCode)

   function void wait(int duration)

}
```

Sample Jack programs

- `Square`: a simple, interactive, multi-class OO application
- `Pong`: a complete, interactive, multi-class OO application
- `Average`: illustrates simple array processing
- `ComplexArrays`: illustrates various array manipulations, including two-dimensional arrays
- `ConvertToBin`: illustrates algebraic operations, and working with `peek` and `poke`

Code: `nand2tetris/projects/11`

Best practice

General

- Watch some existing Jack programs (see "cool stuff" in `www.nand2tetris.org`)
- Play with the supplied programs, and review their code (e.g. `Square` and `Pong`)
- Understand the UX limitations of the Jack I/O
- Plan your app carefully (OO design and testing strategy)
- Implement, test, and ... have fun!

Technical

- Writing: Write / edit your Jack class files in a standard text editor;
  The OS API is supplied in `nand2tetris/projects/09`
- Optimizing: later
- Documenting: use standard practice
- Compiling: use the supplied `JackCompiler` (available in `nand2tetris/tools`)
- Executing: load the app directory (which now contains the compiled `.vm` files) into the supplied VM emulator, and run the code

  (remember the emulator's speed and animation controls).

High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

Jack language specification

- Syntax
- Data types
- Classes
- Methods

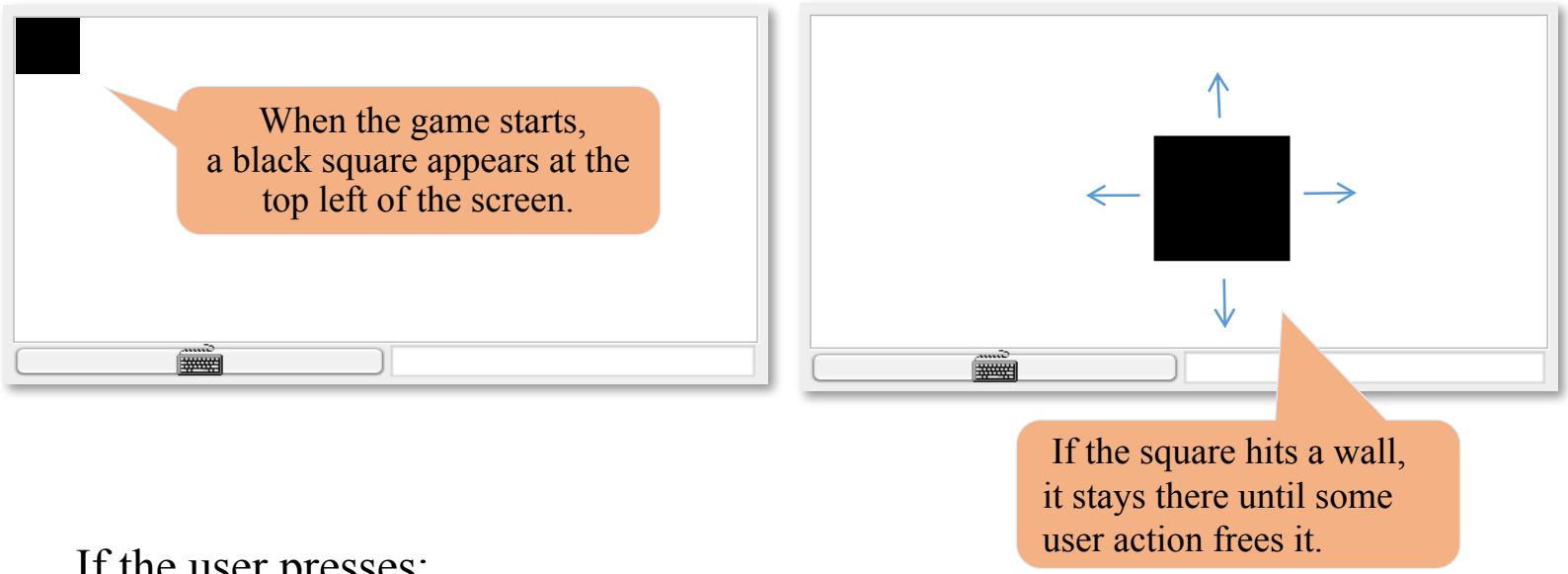
Objectives

The `Square` code review illustrates:

- OO design
- A typical interactive application
- Handling inputs and outputs
- Using the OS

Demo

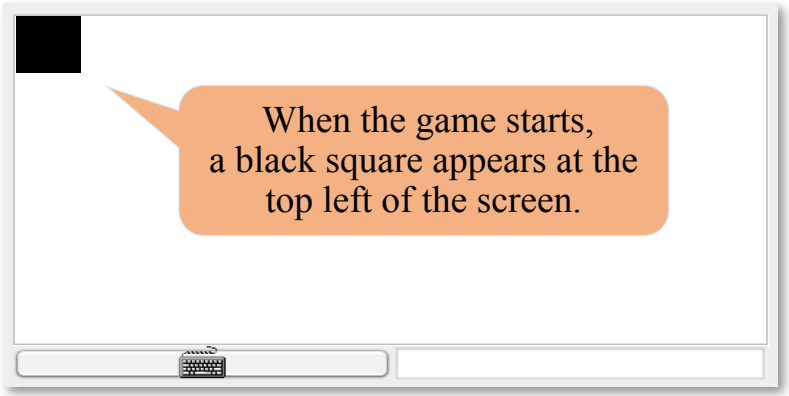
Square Dance game

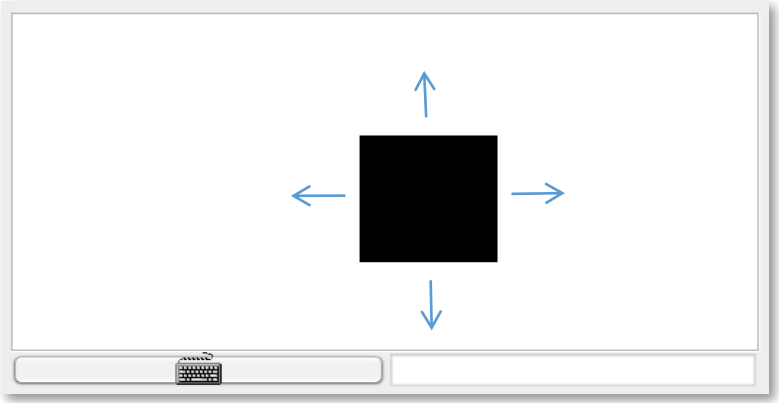


If the user presses:

- up arrow: the square starts moving up, until another key is pressed
- down arrow: the square starts moving down, until another key is pressed
- left arrow: the square starts moving left, until another key is pressed
- right arrow: the square starts moving 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.

App design

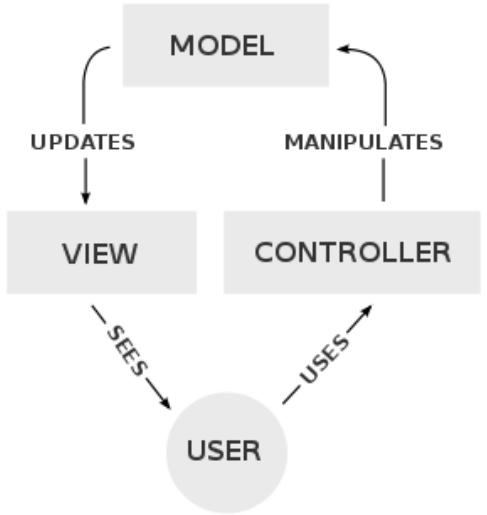




Three Jack classes:

- `Square`: represents a graphical square
- `SquareGame`: captures user's inputs and moves the square accordingly (in a loop)
- `Main`: starts the app, initializes the game, and launches it

MVC model

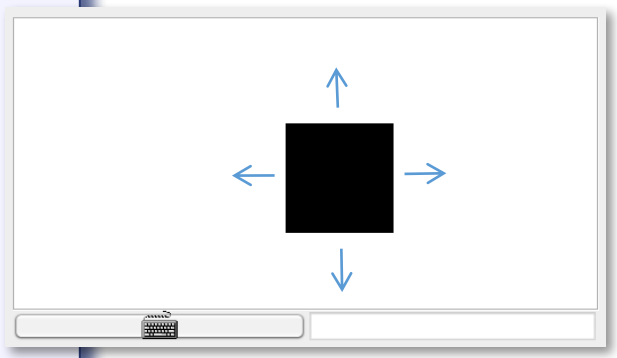


(source: Wikipedia)

Square class API

Square API

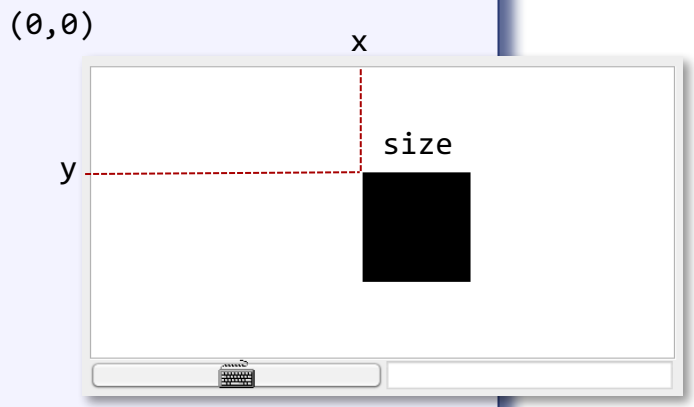
```
/** Implements a graphical square. */
class Square {

   /** 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 on the screen */
   method void draw()

   /** Erases this square from the screen */
   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()
}
```



Square class

Square.jack

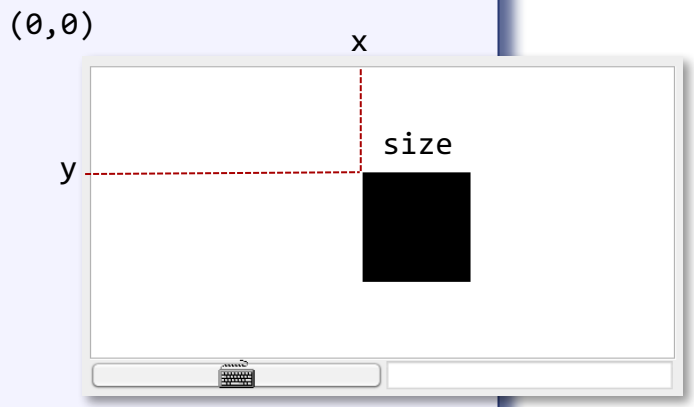
```
/** Implements a graphical square. */
class Square {

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

   /** Constructs 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;
   }

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

   ...
```



Square class

Square.jack

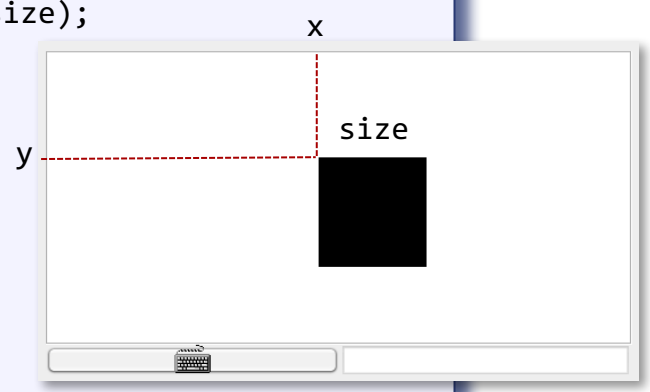
```
/** Implements a graphical square. */
class Square {

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

   ...

   /** Draws the square on the screen. */
   method void draw() {
      do Screen.setColor(true);
      do Screen.drawRectangle(x, y, x + size, y + size);
      return;
   }

   /** Erases the square from the screen. */
   method void erase() {
```



Square class

Square.jack

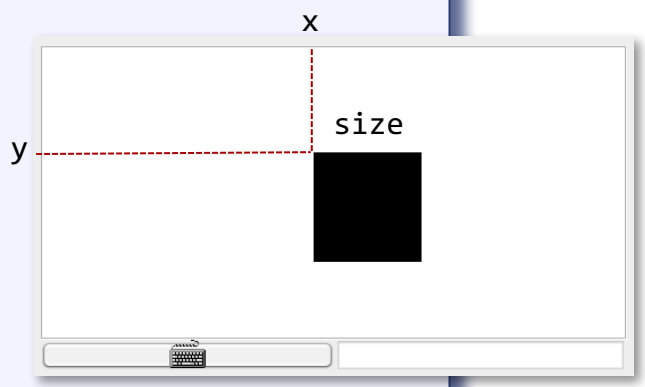
```
/** Implements a graphical square. */
class Square {

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

   ...

   /** Draws the square on the screen. */
   method void draw() {
      do Screen.setColor(true);
      do Screen.drawRectangle(x, y, x + size, y + size);
      return;
   }

   /** Erases the square from the screen. */
   method void erase() {
      do Screen.setColor(false);
      do Screen.drawRectangle(x, y, x + size, y + size);
      return;
   }

   ...
```



Square class

Square.jack

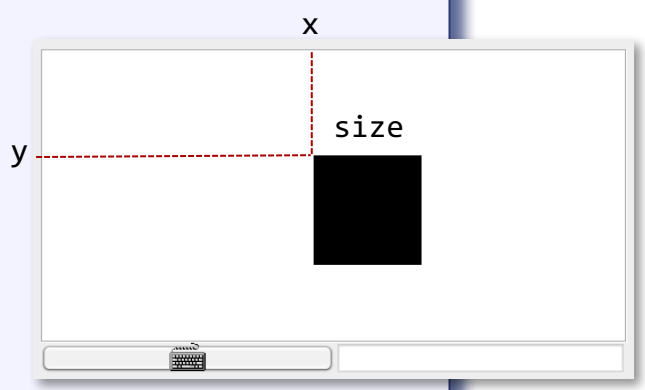
```
/** Implements a graphical square. */
class Square {

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

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

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

   ...
```



Square class

Square.jack

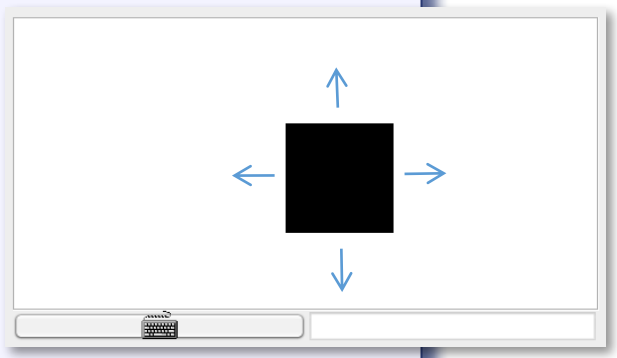
```
/** Implements a graphical square. */
class Square {

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

   /** Moves the square up by 2 pixels. */
   method void moveUp() {
      if (y > 1) {
         do Screen.setColor(false);
         do Screen.drawRectangle(x, (y + size) - 1, x + size, y + size);
         let y = y - 2;
         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
```



SquareGame class

SquareGame.jack

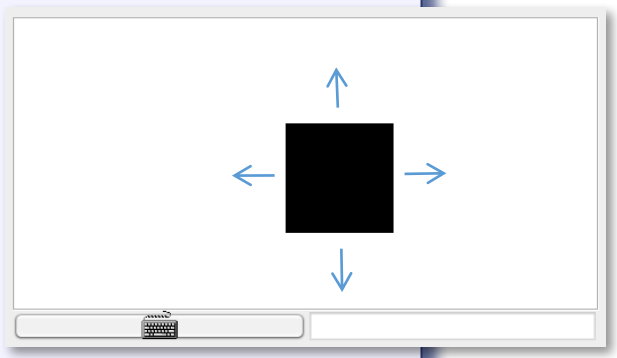
```
/** Implements a square game. */
class SquareGame {

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

   /** Constructs a new Square Game. */
   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;
   }

   ...
```



SquareGame class

SquareGame.jack

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

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

   /** 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;
   }

   ...
```

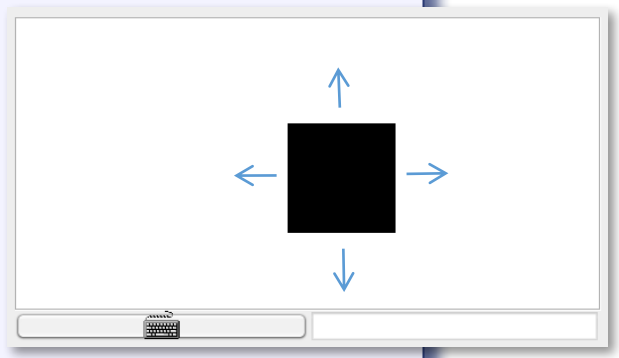


SquareGame class

SquareGame.jack

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

   field Square square; // the square of this game
   field int direction; // the square's 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 arrow
         if (key = 133) { let direction = 2; }   // down arrow
         if (key = 130) { let direction = 3; }   // left arrow
         if (key = 132) { let direction = 4; }   // right arrow

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

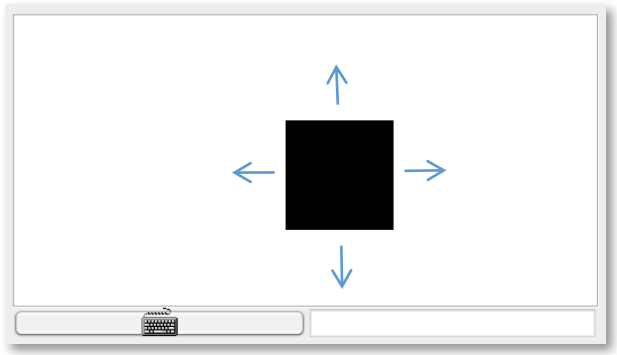
typical handling of "keyboard events" in interactive Jack apps



Main class

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;
    }
}
```



Recap

Important issues:

- Understanding the UI style
- Getting to know the Jack OS
- OO design
- Testing stratgey

High level language: lecture plan

High level programming

- Hello world
- Procedural programming
- Object-based programming
- List processing

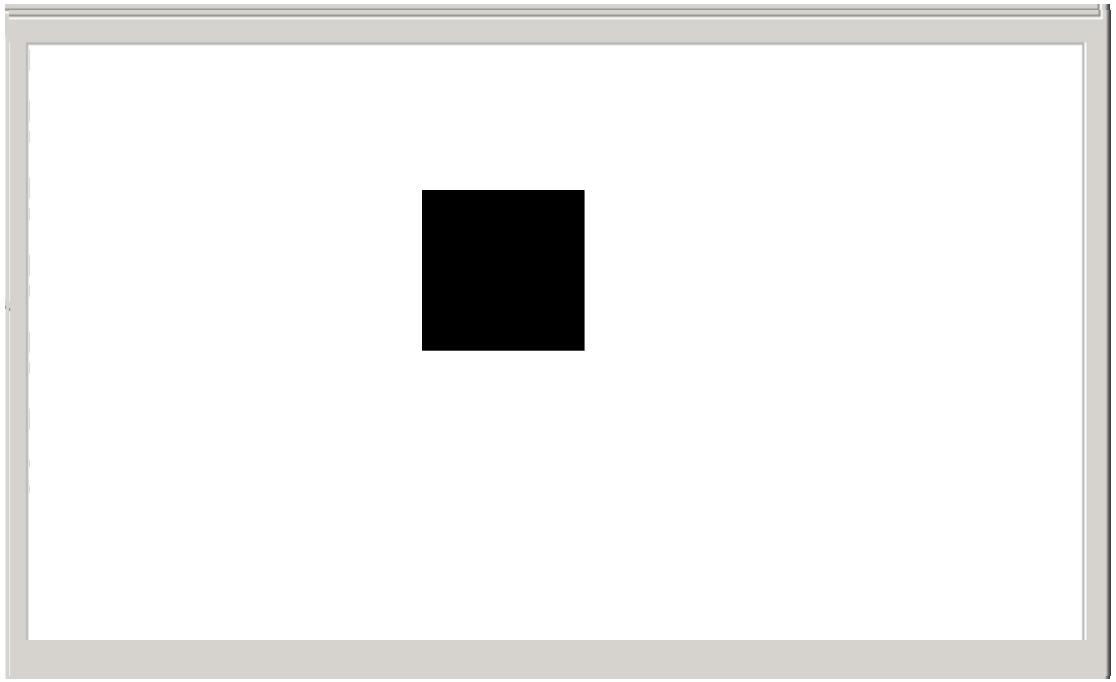
Application development

- Jack applications
- Using the OS
- Application example
- Graphics optimization

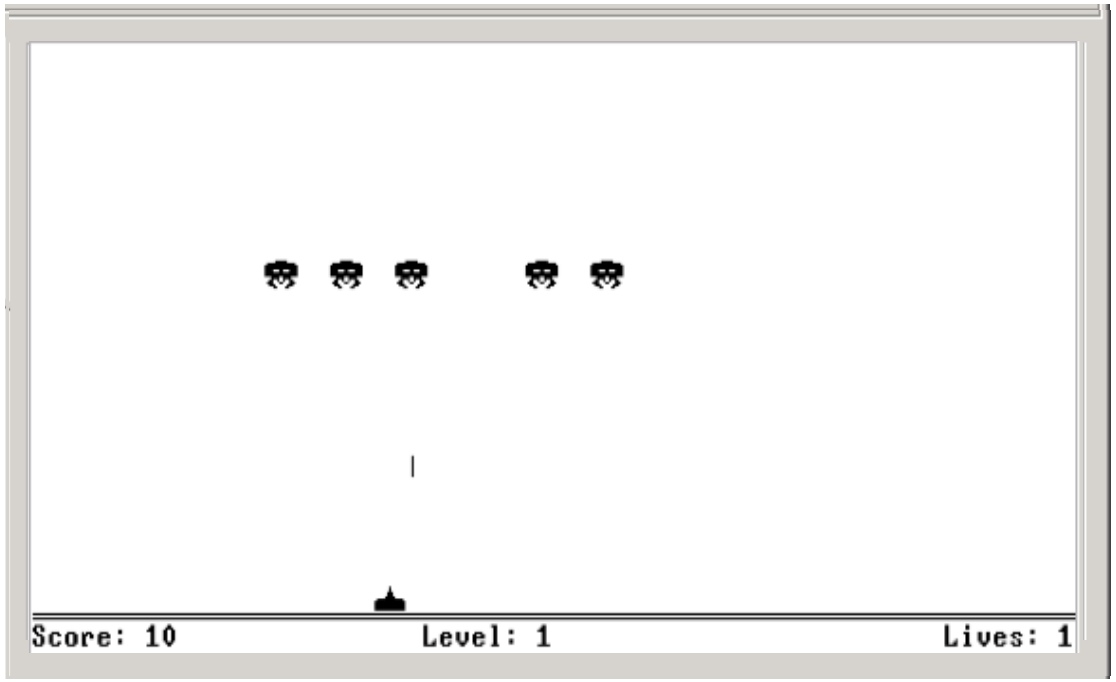
Jack language specification

- Syntax
- Data types
- Classes
- Methods

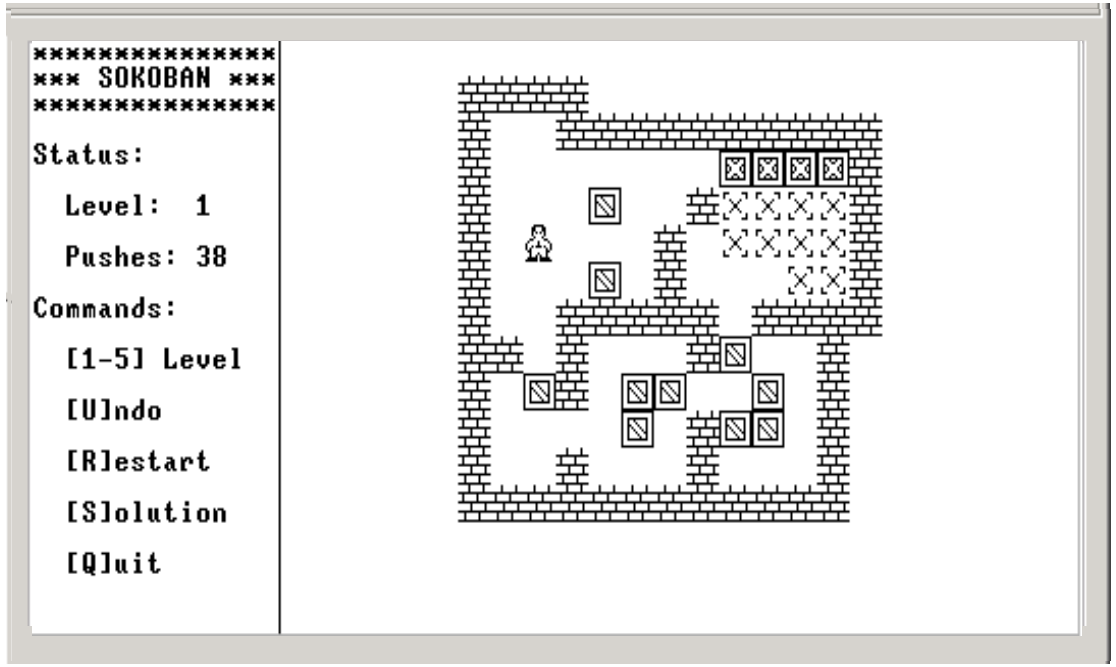
Square dance

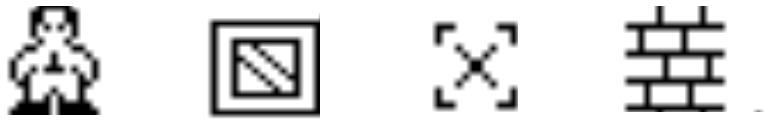


Space Invaders (by Ran Navok)



Sokoban (by Golan Parashi)





Handling sprites

Sprite

A two-dimensional bitmap, typically integrated into a larger scene

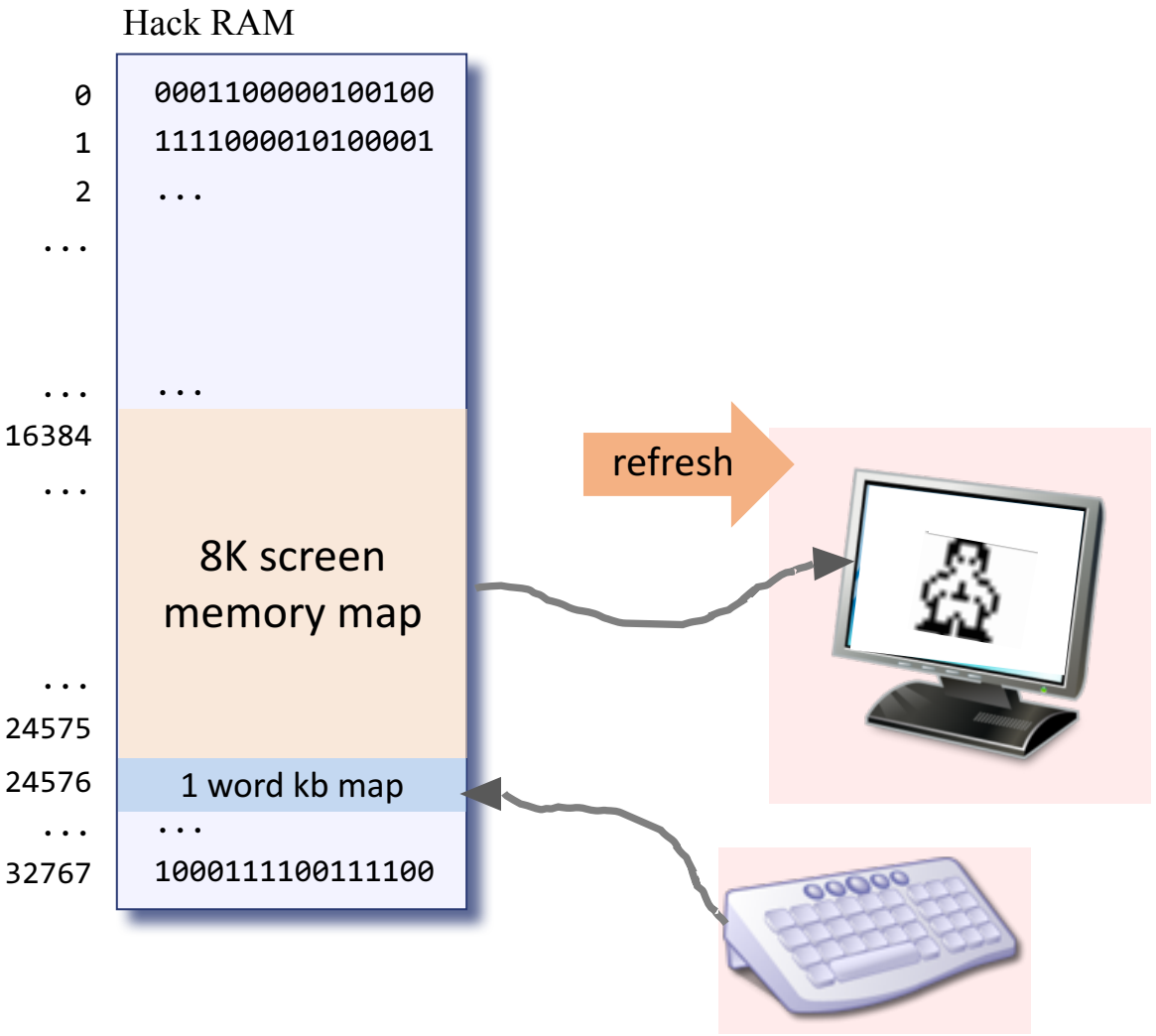
Challenges

- Drawing sprites quickly
- Creating smooth animations

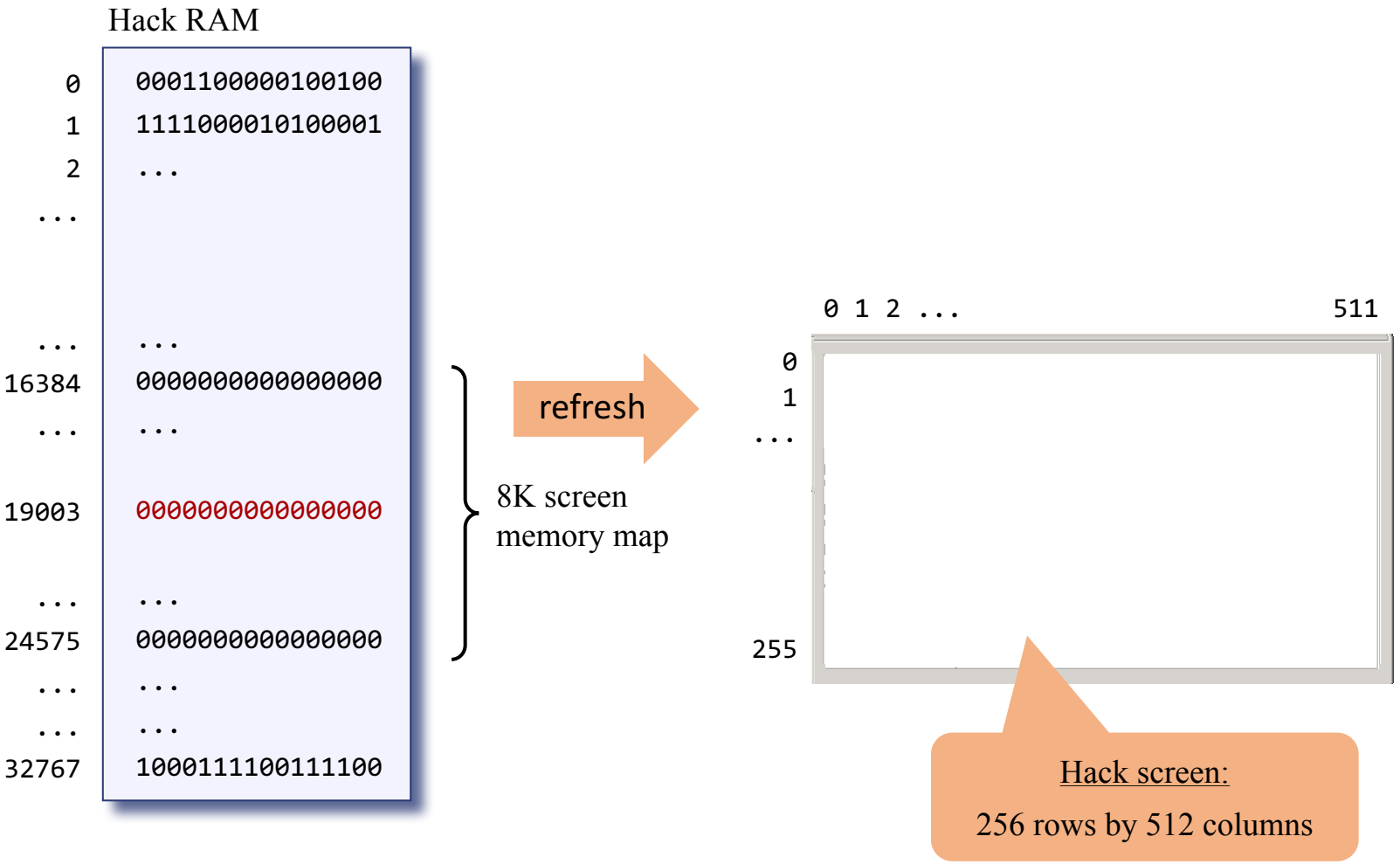
Solutions

- Use the standard OS graphics library
- Use your own graphics functions

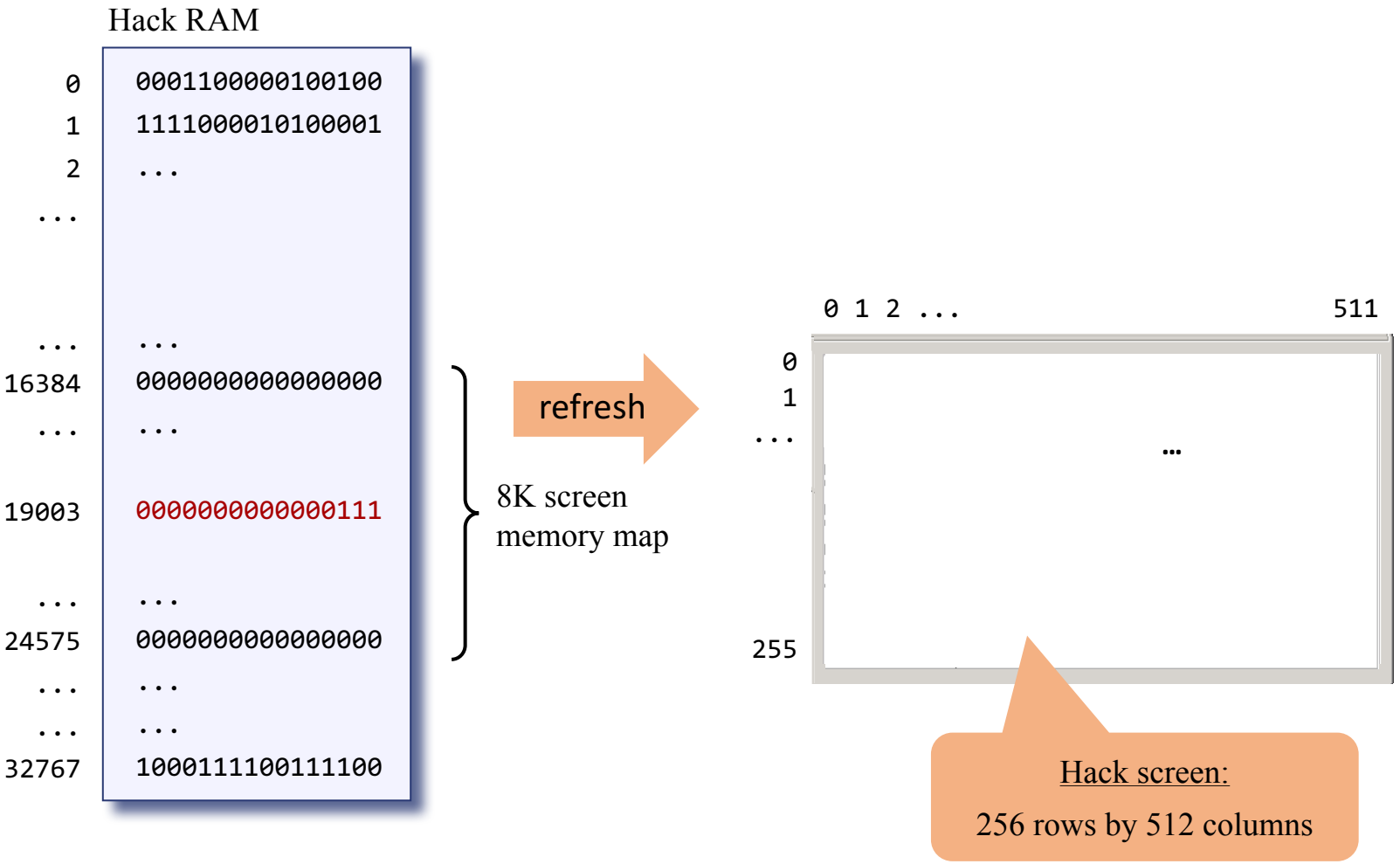
Hack I/O


Hack RAM
0
1
2
...
...
...
16384
...
...
24575
24576
...
32767
0001100000100100
1111000010100001
...
...
8K screen
memory map
1 word kb map
...
1000111100111100
refresh

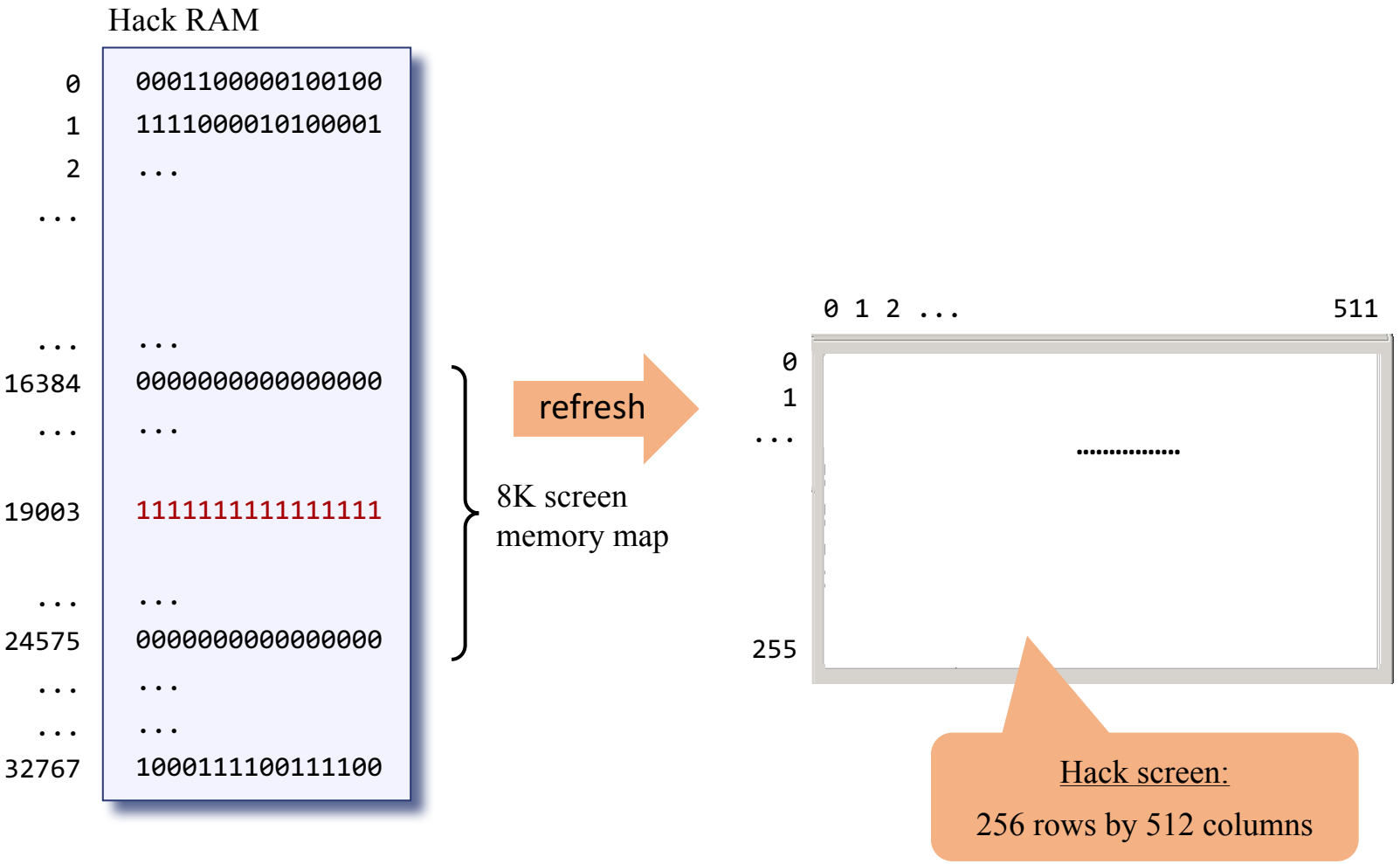
Memory map


Hack RAM
0 0001100000100100
1 1111000010100001
2 ...
...
...
...
16384 0000000000000000
...
19003 0000000000000000
...
24575 0000000000000000
...
...
32767 1000111100111100
refresh
8K screen memory map
0 1 2 ... 511
0
1
...
255
Hack screen:
256 rows by 512 columns

Memory map


Hack RAM
0 0001100000100100
1 1111000010100001
2 ...
...
...
...
16384 0000000000000000
...
19003 0000000000000111
...
24575 0000000000000000
...
...
32767 1000111100111100
refresh
8K screen
memory map
0 1 2 ... 511
0
1
...
255
...
Hack screen:
256 rows by 512 columns

Memory map


Hack RAM
0 0001100000100100
1 1111000010100001
2 ...
...
...
...
16384 0000000000000000
...
19003 1111111111111111
...
24575 0000000000000000
...
...
32767 1000111100111100
refresh
8K screen memory map
0 1 2 ... 511
0
1
...
255
Hack screen:
256 rows by 512 columns

Accessing memory: read / write

Hack RAM

0	0001100000100100
1	1111000010100001
2	...
...	
...	...
16384	0000000000000000
...	...
19003	0000000000000111
...	...
24575	0000000000000000
...	...
...	...
32767	1000111100111100

The OS `Memory` class API

- `function int peek(int address)`
- `function void poke(int address, int value)`

...

Jack code

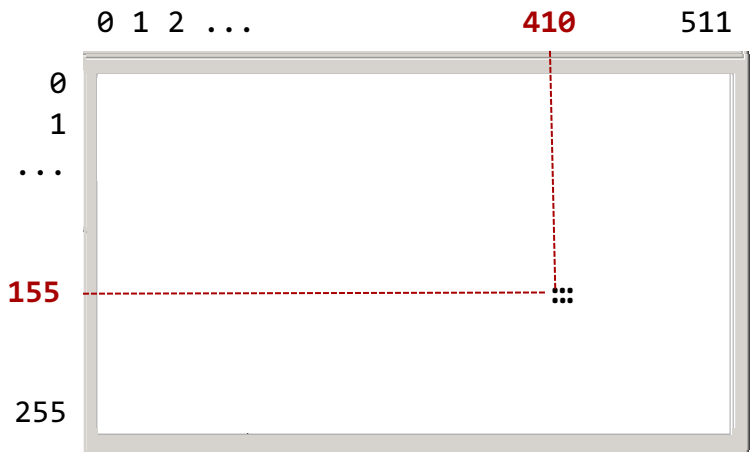
```
let x = Memory.peek(19003)
// x will be set to 7
```

```
do Memory.poke(19003,-1)
// RAM[19003] will be set to
// 1111111111111111
```

Drawing pixels

Jack code

```
// draws the image
do Screen.drawPixel(410,155);
do Screen.drawPixel(411,155);
do Screen.drawPixel(412,155);
do Screen.drawPixel(410,156);
do Screen.drawPixel(411,156);
do Screen.drawPixel(412,156);
```



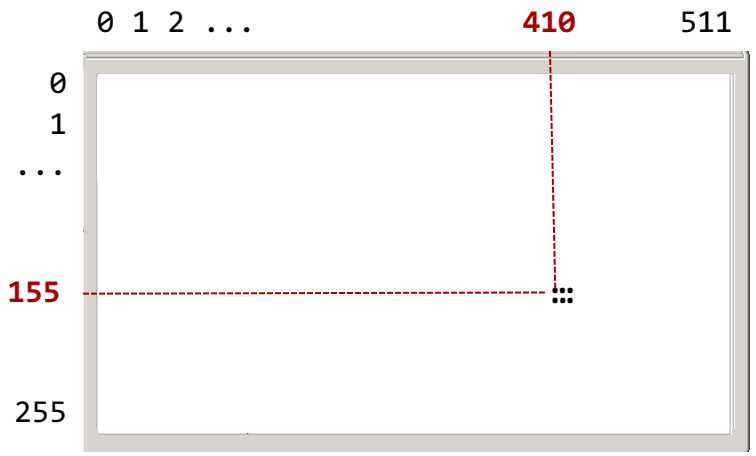
The OS `Screen` class API:

- `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):`

Drawing pixels

Jack code

```
// draws the image
do Screen.drawPixel(410,155);
do Screen.drawPixel(411,155);
do Screen.drawPixel(412,155);
do Screen.drawPixel(410,156);
do Screen.drawPixel(411,156);
do Screen.drawPixel(412,156);
```




```
// draws the image
do Screen.drawLine(410,155,412,155);
do Screen.drawLine(410,156,412,156);
```

```
// draws the image
do Screen.drawRectangle(410,155,412,156);
```

Standard drawing

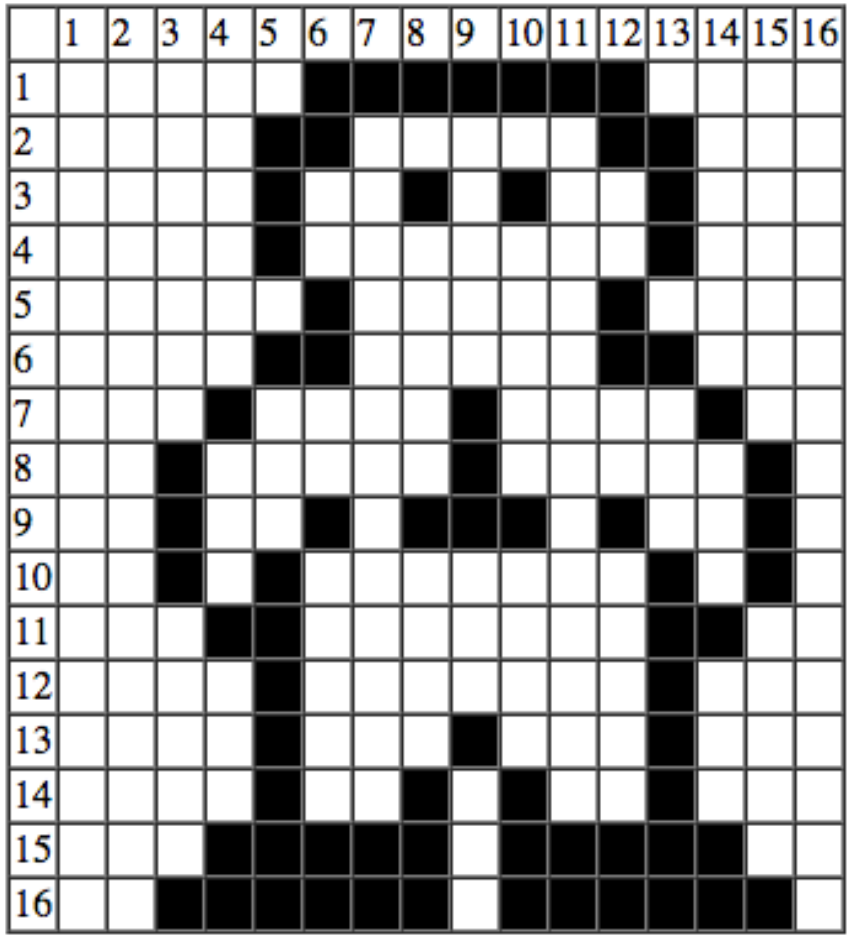


Image drawing code:

```
// 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);
```

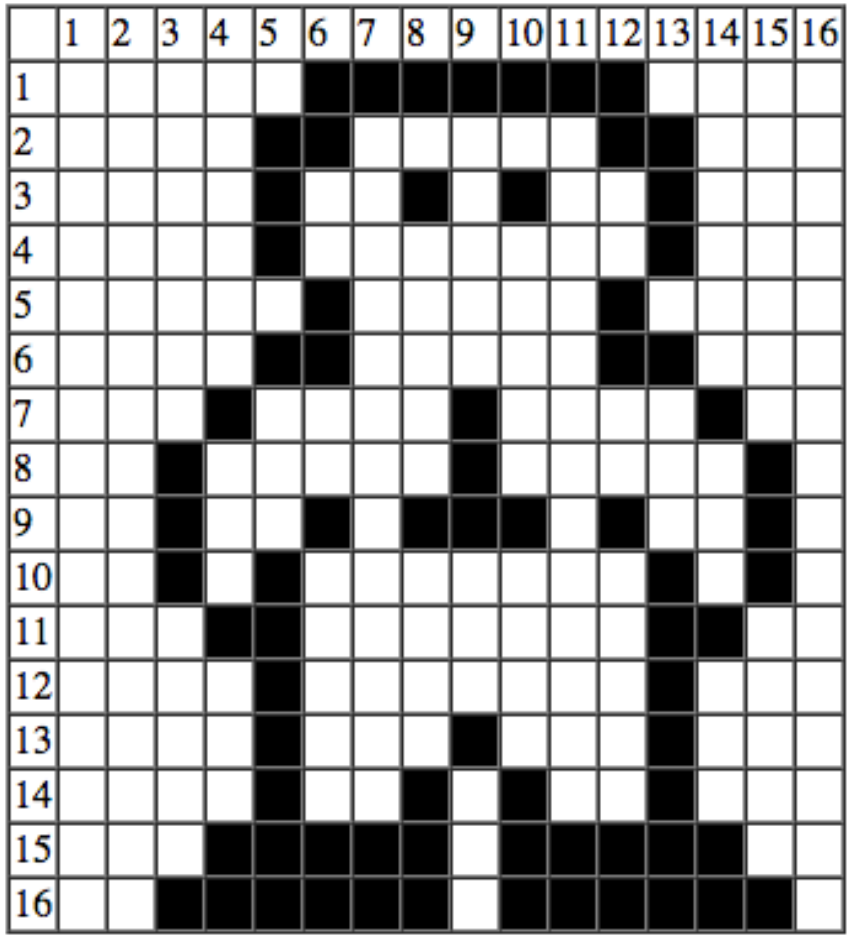
Efficiency:

75 pixel drawing operations

OS implementation of `drawPixel(`*x*`,`*y*`)`

// sets pixel (*x*,*y*) to black / white:
address = 32 * *y* + *x* / 16
value = `Memory.peek[`16384 + *address*`]`
set the (*x* % 16)*th* bit of *value* to 0 or 1
do `Memory.poke(`*address*`,`*value*`)`

Custom drawing



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

Custom drawing

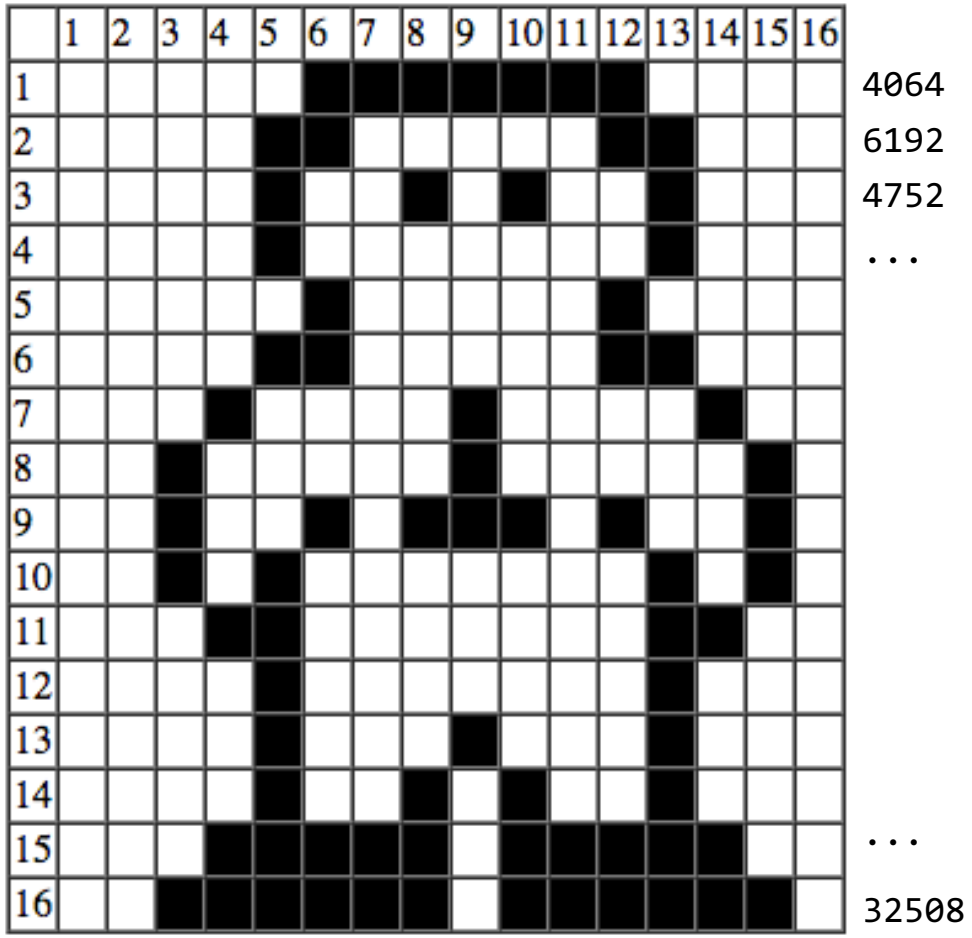


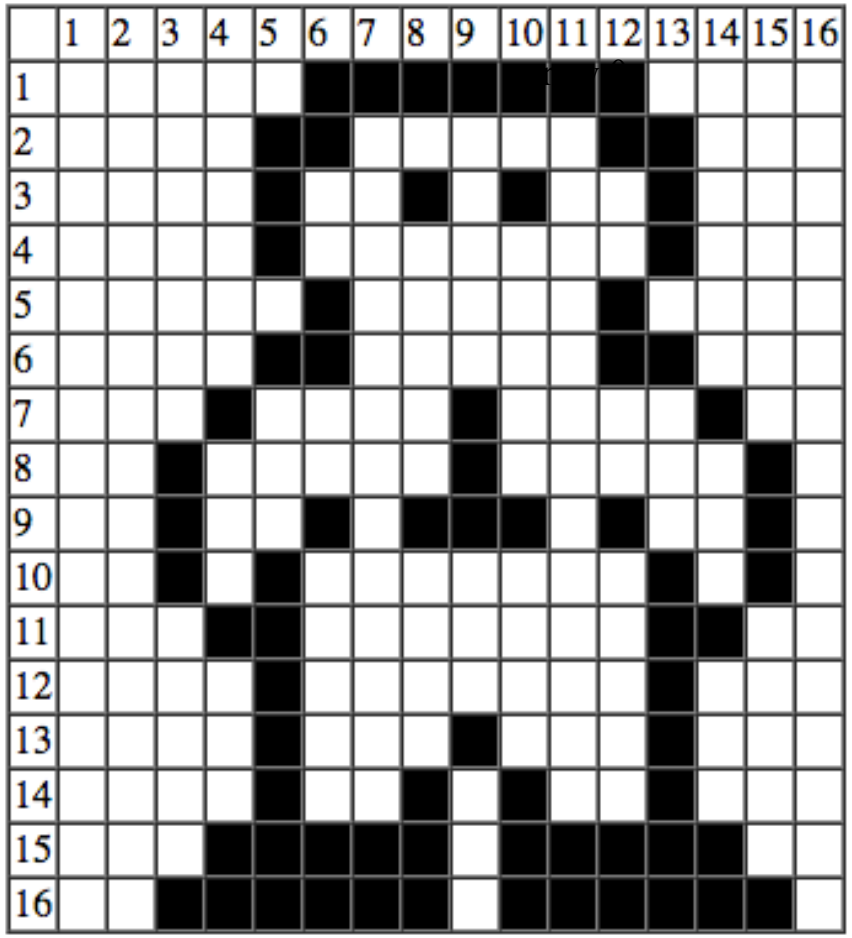
Image drawing code:

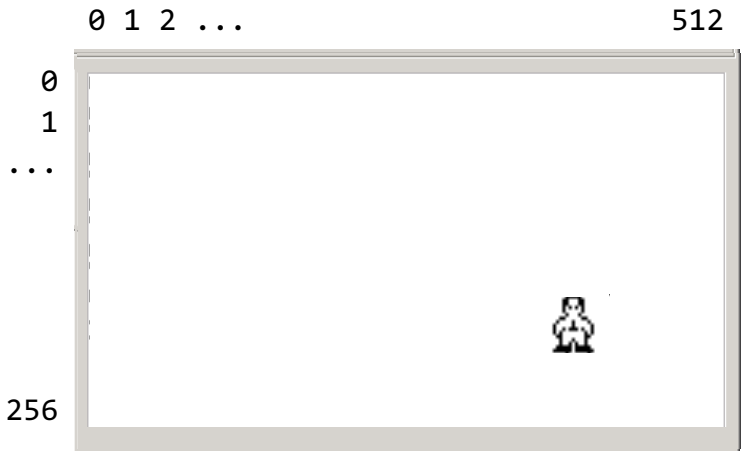
```
// Draws the sprite
do Memory.poke(addr0, 4064);
do Memory.poke(addr1, 6192);
do Memory.poke(addr2, 4752);
...
do Memory.poke(addr15,32508);
```

Efficiency:

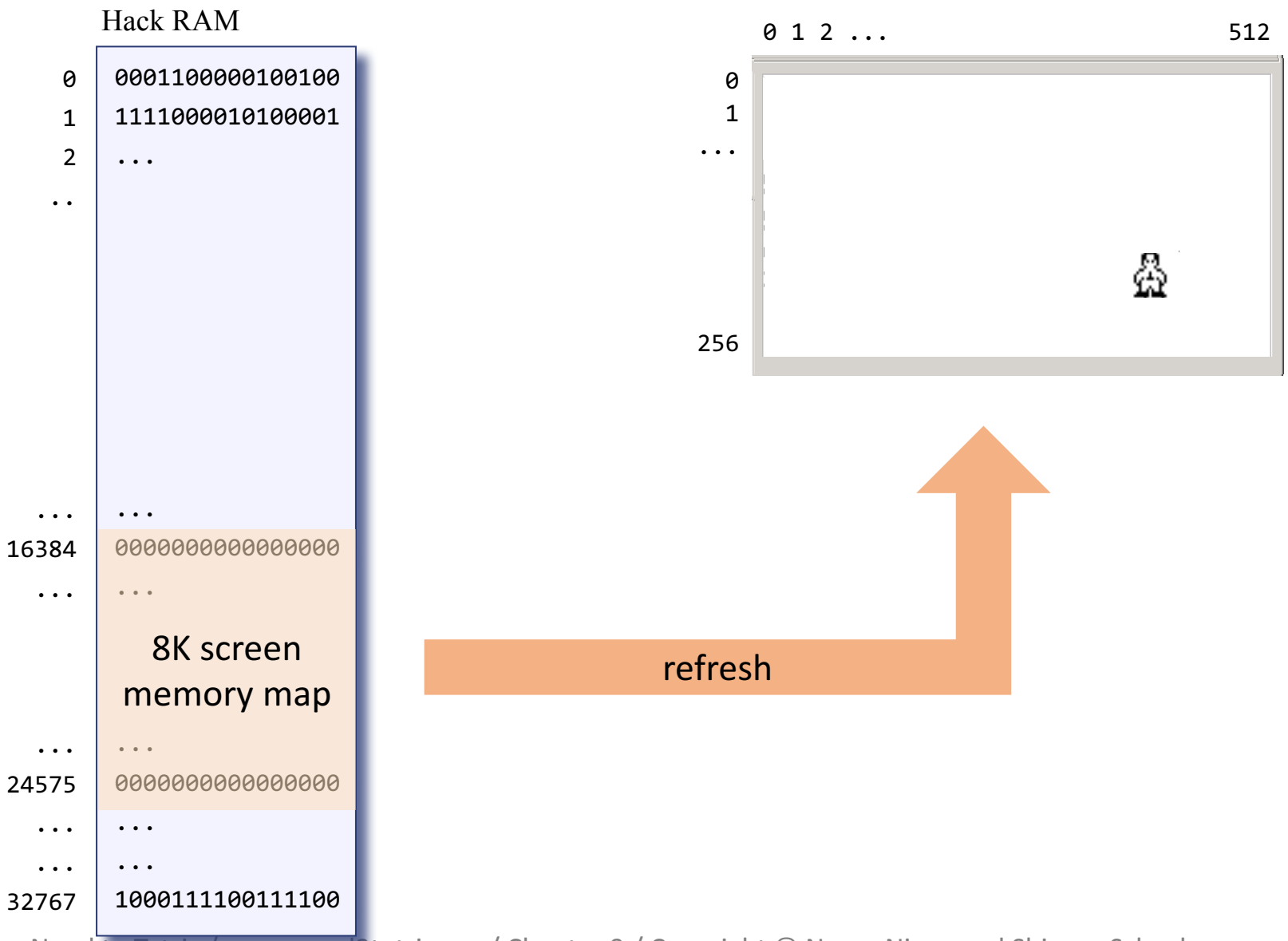
16 memory write operations

Custom drawing

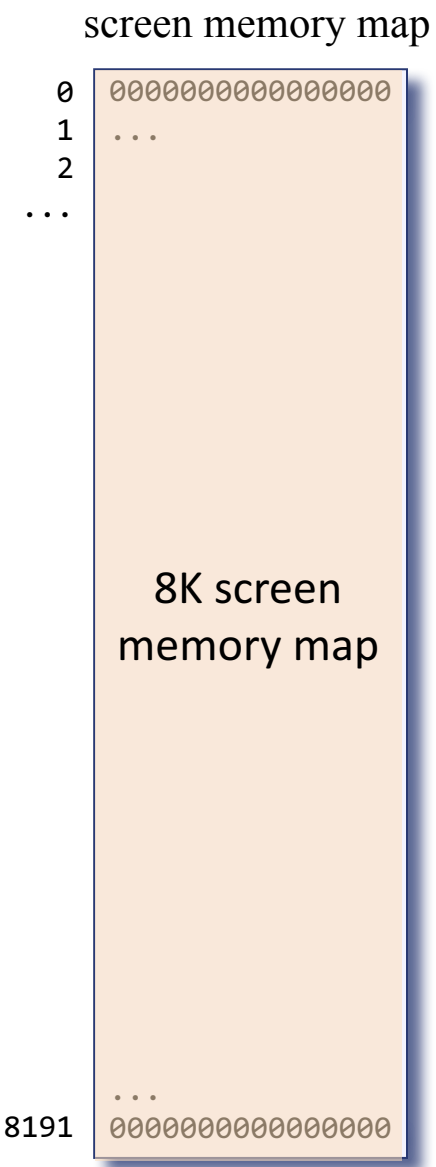


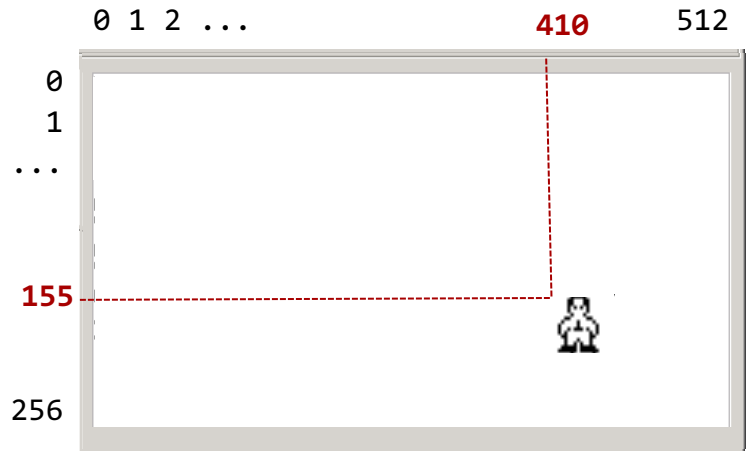


Custom drawing



Custom drawing





Custom drawing

screen memory map

0 .. 31	0000000000000000 ... 0000000000000000	row 0
32 .. 63	0000000000000000 ... 0000000000000000	row 1
..	...	
... addr ...	... 0000011111110000 ...	row 155
... addr + 32 ...	... 0000110000011000 ...	row 156
... addr + 64 ...	... 0000100101001000 ...	row 157
...	...	
... addr + 480 ...	... 0111111011111100 ...	row 170
...	...	
8191	0000000000000000 ... 0000000000000000	row 256

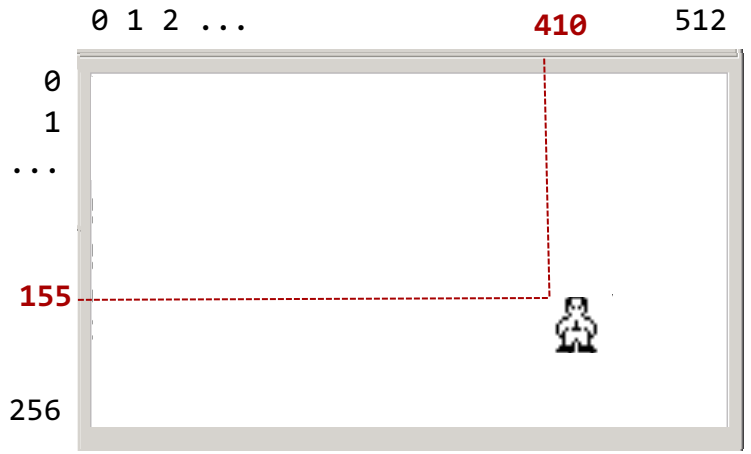


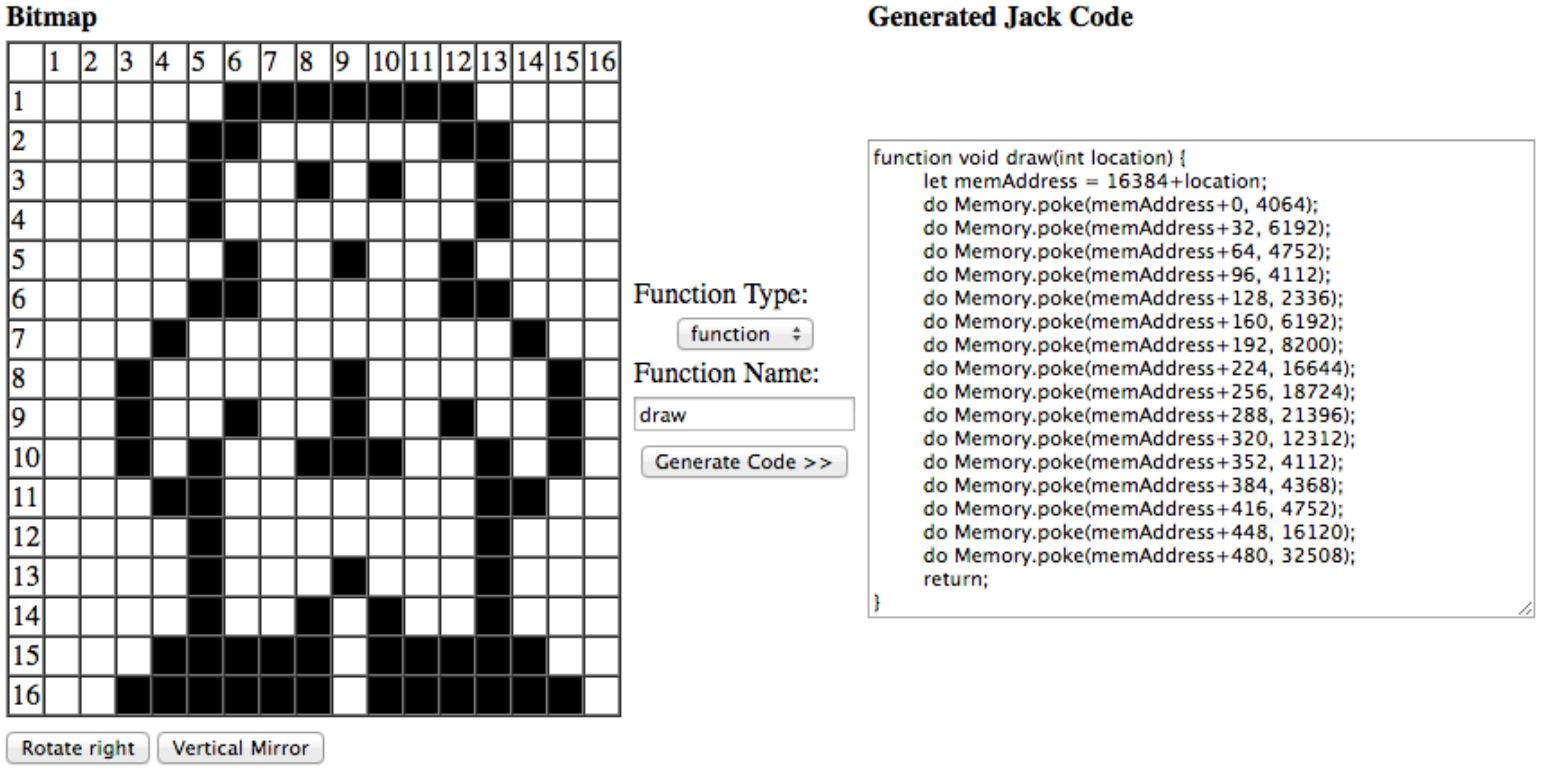
Image drawing code:

```
// Draws the sprite
do Memory.poke(addr +   0, 4064);
do Memory.poke(addr +  32, 6192);
do Memory.poke(addr +  64, 4752);
...
do Memory.poke(addr + 480, 32508);
```

Absolute addressing

Add (`16384` + `location`) to `addr`

Bitmap editor



- Developed by Golan Parashi
- Available in `nand2tetris/projects/09`

Best practice

- For simple graphics, use the OS services
- For high-performance graphics, use your own functions
- If you need high-performance sprites, use the bitmap editor:
  - Design the sprite
  - Generate the Jack code
  - Plant the generated Jack code in your own Jack code

Perspective

Jack is a nice little language,

Featuring most of the essentual elements of

- procedural
- OO programming

Limitations

- Few control structures
- Some peculiar syntax
- No inheritance

Motivation: a minimal language that can be implemented by a simple compiler

Data types

- Primitive type system
- Weakly typed

Motivation: to give the programmer full control, especially for writing the OS.