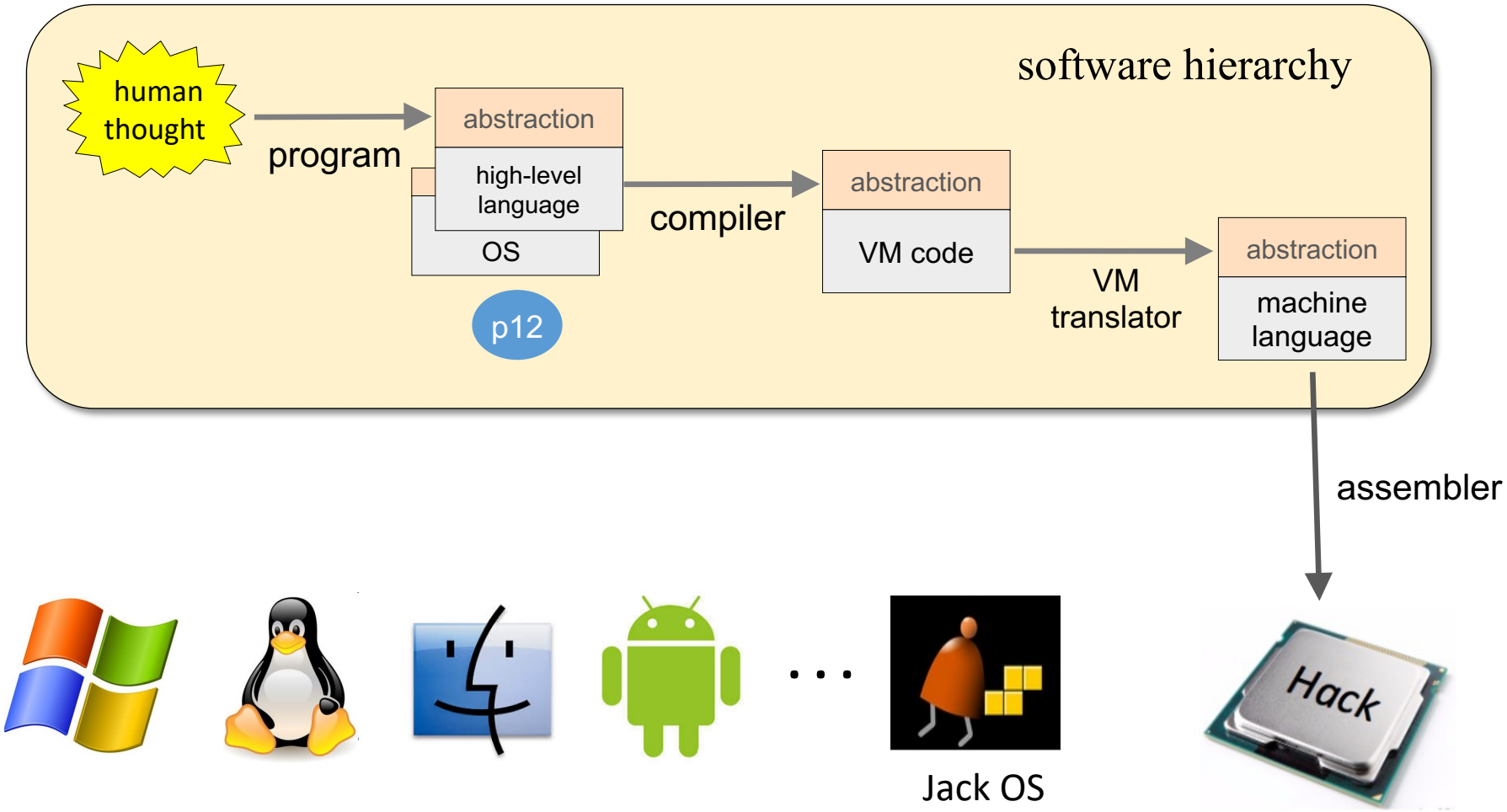
Chapter 12

Operating System

These slides support chapter 12 of the book
The Elements of Computing Systems
By Noam Nisan and Shimon Schocken
MIT Press

Nand to Tetris Roadmap: Part II



High-level programming

Typical program

```
/* Computes the length of the hypotenuse in a right triangle */
class Main {
  function void main() {
     var int a;
     var int b;
     var int c;
     let a = Keyboard.readInt("Enter the length of side 1: ");
     let b = Keyboard.readInt("Enter the length of side 2: ");
     do Output.printString("The hypotenuse length is: ");
     do Output.printInt(Main.hypotenuseLength(a,b));
     return;
  }

  function int hypotenuseLength(int x, int y) {
     return Math.sqrt((x*x) + (y*y));
  }
}
```

OS services highlighted

Typical OS services

Language extensions

- ✓ Mathematical operations
  (`abs`, `sqrt`, ...)
- ✓ Abstract data types
  (`String`, `Array`, ...)
- ✓ Input functions
  (`readChar`, `readLine` ...)
- ✓ Textual output
  (`printChar`, `printString` ...)
- ✓ Graphics output
  (`drawLine`, `drawCircle`, ...)

. . .

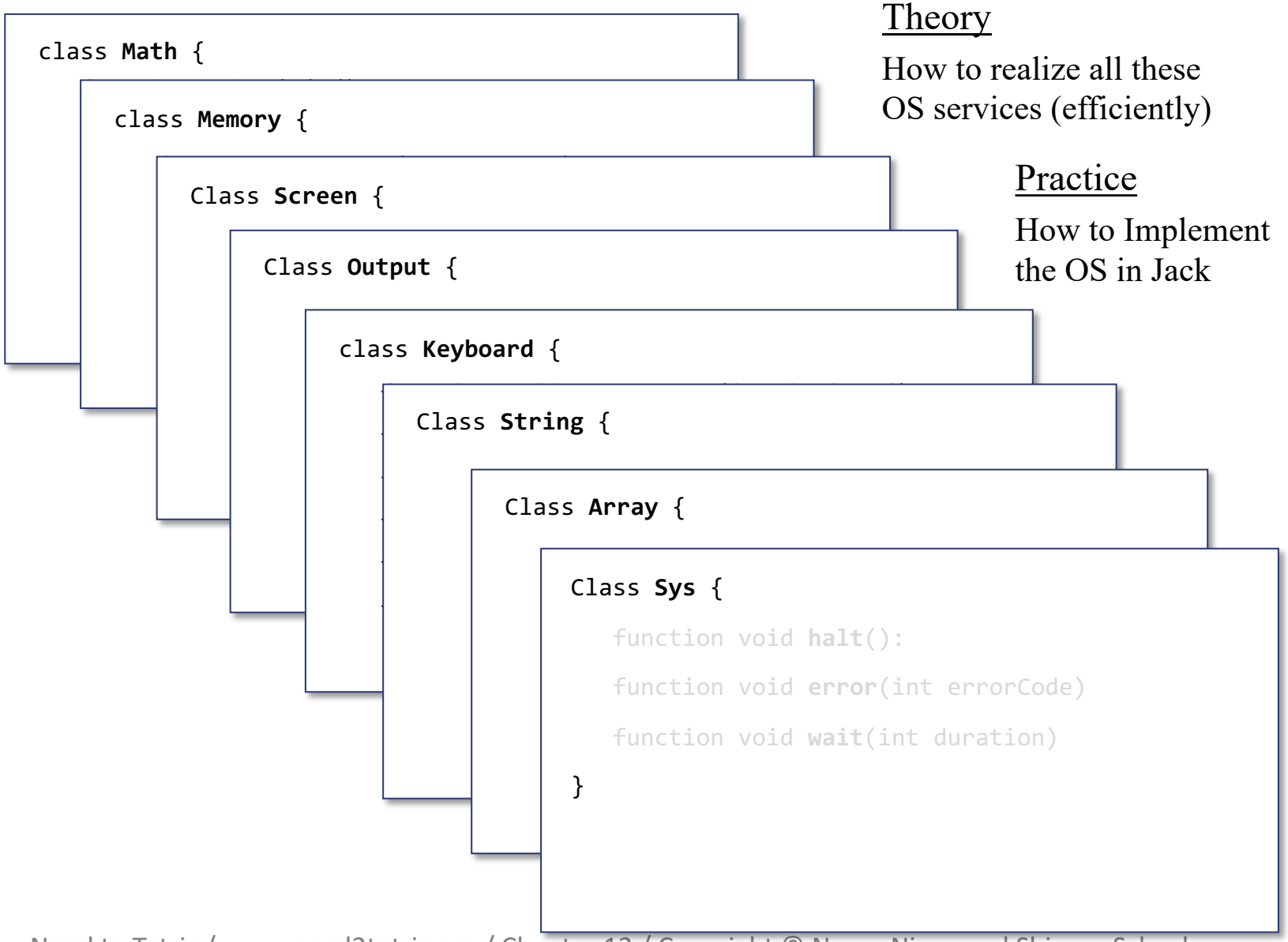
System services

- ✓ Memory management
  (objects, arrays, ...)
- ✓ I/O device drivers
- File system
- UI management
  (shell / windows)
- Multi-tasking
- Networking
- Security

. . .

✓ : implemented in the Jack OS

The Jack OS



Theory

How to realize all these OS services (efficiently)

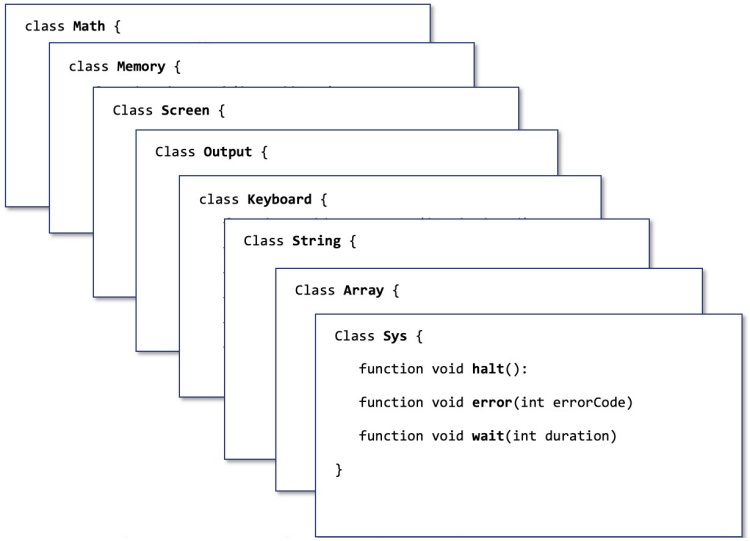
Practice

How to Implement the OS in Jack

Take home lessons

- Time complexity
- Memory management
- Input handling
- Output handling:
  - Vector graphics
  - Textual outputs
- Type conversions
- String processing

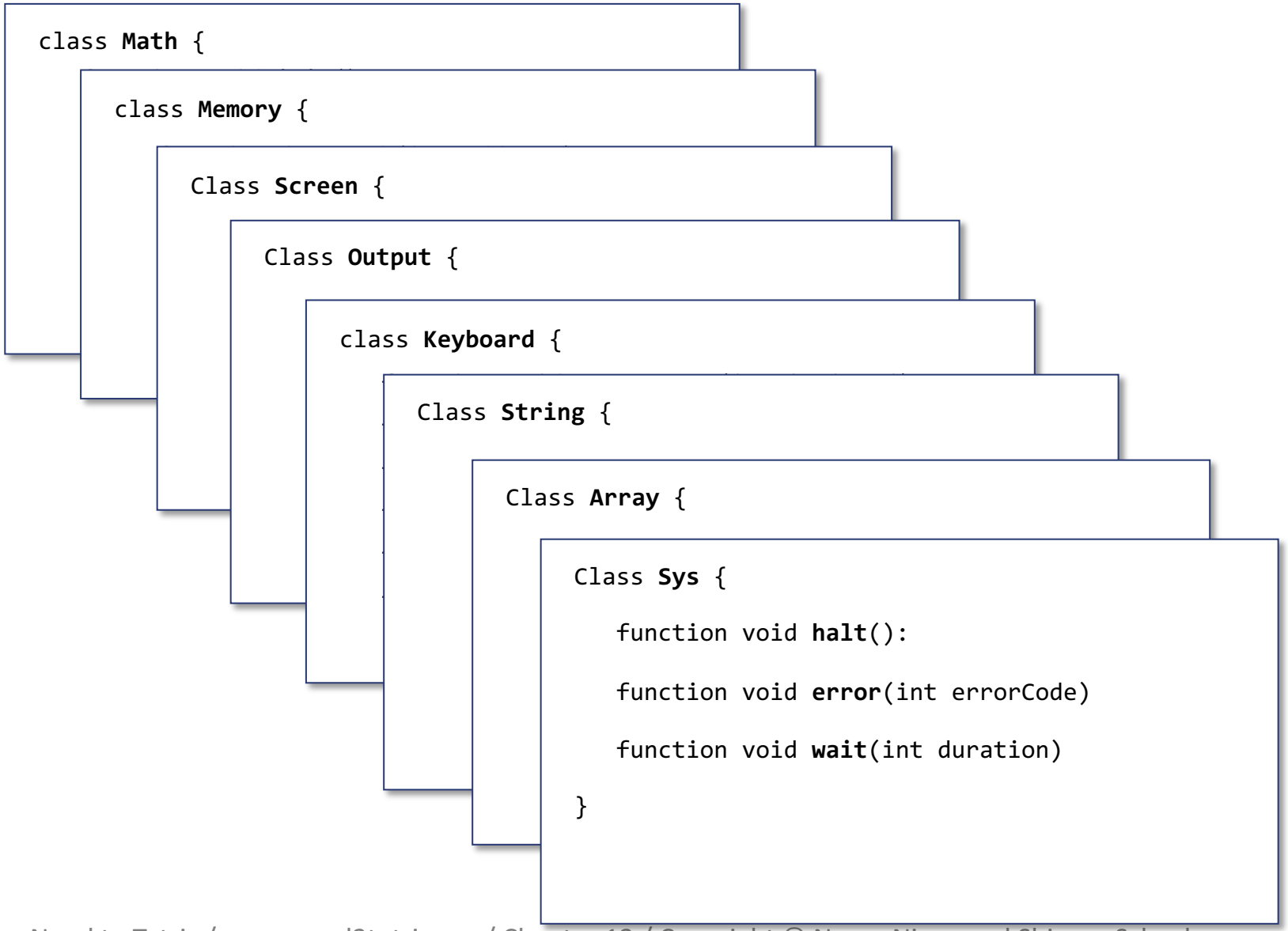
…



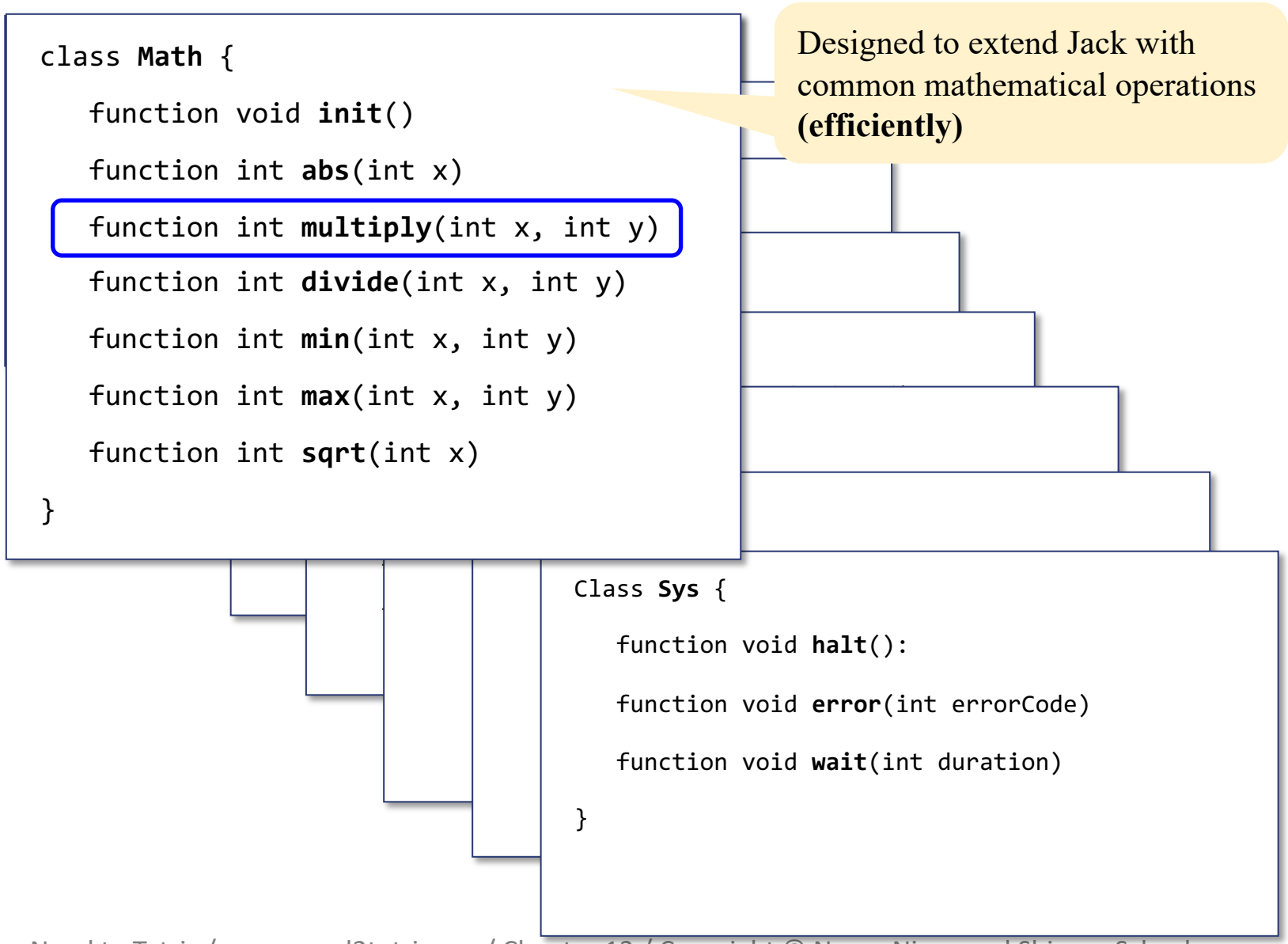
Methodology:

- Classical algorithms
- Cool hacks
- Typical trade-offs
- Implementation issues.

The Jack OS


class Math {
class Memory {
Class Screen {
Class Output {
class Keyboard {
Class String {
Class Array {
Class Sys {
function void halt():
function void error(int errorCode)
function void wait(int duration)
}

The Jack OS


class Math {
function void init()
function int abs(int x)
function int multiply(int x, int y)
function int divide(int x, int y)
function int min(int x, int y)
function int max(int x, int y)
function int sqrt(int x)
}
Designed to extend Jack with common mathematical operations **(efficiently)**
Class Sys {
function void halt():
function void error(int errorCode)
function void wait(int duration)
}

Multiplication

In some class:

```
...
let a = 27 * 9;
// Same as:
let a = Math.multiply(27,9);
...
```

OS `Math` class

```
...
/* Returns x*y, where x, y ≥ 0. */
// Naïve strategy: repetitive addition
multiply(x, y):
    sum = 0
    for i = 0 ... y – 1   do
        sum = sum + x
    return sum
...
```

Algorithm's run-time:

Proportional to y

- What happens when y = 3773112?
- Number of iterations: 3773112

In this lecture we present algorithms using block indentation, Python-style

Multiplication

		1	1	0	1	1	= 27
			1	0	0	1	= 9
		1	1	0	1	1	
	0	0	0	0	0		
0	0	0	0	0			
1	1	0	1	1			
1	1	1	1	0	0	1	1	= 243

x:	...	0	0	0	**1**	**1**	**0**	**1**	**1**	
y:	...	0	0	0	0	**1**	**0**	**0**	**1**	i'th bit of y
	...	0	0	0	**1**	**1**	**0**	**1**	**1**	1
	...	0	0	1	1	0	1	1	0	0
	...	0	1	1	0	1	1	0	0	0
	...	**1**	**1**	**0**	**1**	**1**	0	0	0	1
$x*y$:	...	1	1	1	1	0	0	1	1	

Multiplication

		1	1	0	1	1	= 27
		1	0	0	1		= 9
		1	1	0	1	1	
	0	0	0	0	0		
0	0	0	0	0			
1	1	0	1	1			
1	1	1	1	0	0	1	1	= 243

x:	...	0	0	0	**1**	**1**	**0**	**1**	**1**	
y:	...	0	0	0	0	**1**	**0**	**0**	**1**	i'th bit of y
	...	0	0	0	**1**	**1**	**0**	**1**	**1**	1
	...	0	0	**1**	**1**	**0**	**1**	**1**	0	0
	...	0	**1**	**1**	**0**	**1**	**1**	0	0	0
	...	**1**	**1**	**0**	**1**	**1**	0	0	0	1
$x*y$:	...	1	1	1	1	0	0	1	1	

OS `Math` class

```
/* Returns x*y, where x, y ≥ 0. */
// Bit-wise strategy for
// multiplying n-bit numbers:
multiply(x, y):
  sum = 0
  shiftedX = x
  for i = 0 ... n – 1  do
    if ((i'th bit of y) == 1)
      sum = sum + shiftedX
    shiftedX = shiftedX * 2
  return sum
```

Run-time: Proportional not to y, but rather to the *number of bits* in y: $n = log_2 y$

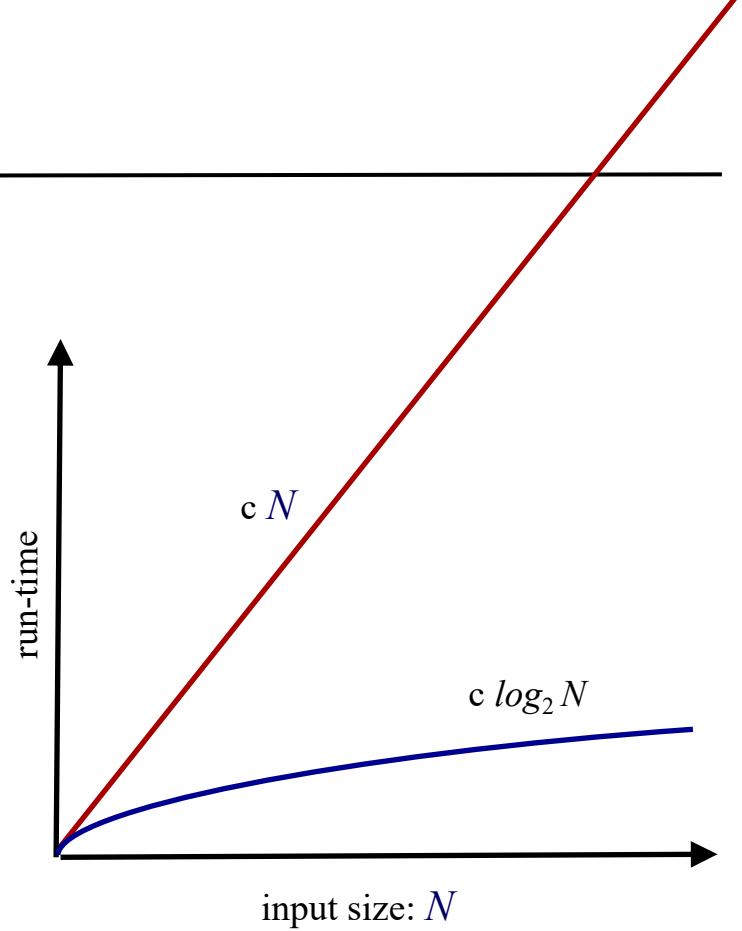
- What happens when y = 3773112?
- Number of iterations: log_2 3773112 = 22

- The algorithm involves only addition operations
- Can be implemented efficiently in either software or hardware

Run-time

input size: N	linear run-time: c N	log. run-time: c $log_2 N$
8	(say c = 10) 80	(c = 10) 30
16	160	40
32	320	50
64	640	60
100	1000	70
1,000	10,000	100
1,000,000	10,000,000	200
1,000,000,000	10,000,000,000	300

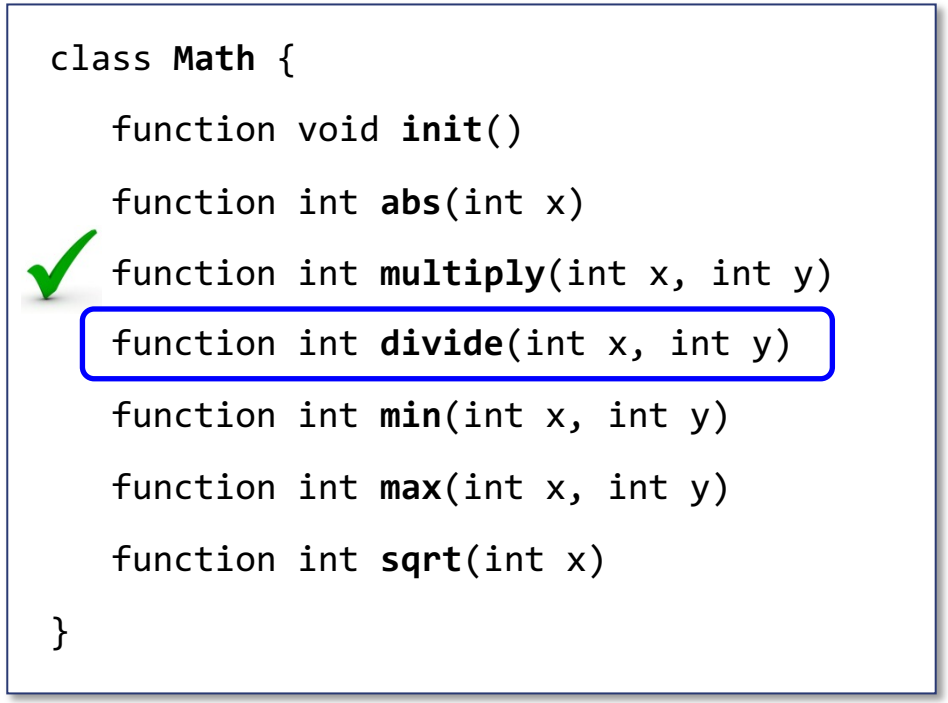
c: the number of machine operations in each iteration



Why is $log_2 N$ attractive?

- Because $log_2(2N) = log_2 N + 1$
- As the size of the input doubles, an algorithm with logarithmic run-time requires 1 additional step.

Recap



```
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)
}
```

Division

In some class:

```
...
let v = 175 / 3;
// Same as:
let a = Math.divide(175,3);
...
```

OS `Math` class

```
/* Returns the integer part of x / y,
   where x ≥ 0 and y > 0. */
// Naïve strategy: repetitive subtraction
divide (x,y):
  div = 0
  rem = x
  while rem ≤ x
      rem = rem – y
      div = div + 1
  return div
```

Run-time:

Proportional to x.

Division

```
      5   8
    ------------
1   7   5 | 3
1   5   0
-------------
    2   5
    2   4
-------------
        1
```

Huh?

Division

```
      5   8
    ------------
1   7   5 | 3
1   5   0
-------------
    2   5
    2   4
    -------
        1
```

Run-time (binary version): proportional to the number of digits in N, which is $log_2 N$

What is the largest number x = (90, 80, 70, 60, 50, 40, 30, 20, 10), so that $3*x \leq 175$?

Answer: *div* = at least 50,
and we still have to divide the remainder 25 by 3

What is the largest number x = (9, 8, 7, 6, 5, 4, 3, 2, 1), so that $3*x \leq 25$?

Answer: *div* = at least 50 + 8,
and we still have to divide the remainder 1 by 3

1 is less than 3, so the answer is: 58 with a remainder of 1.

Run-time

Division by repetitive subtraction:

```
divide (x, y):
  let div = 0
  let rem = x
  while rem ≤ x
     let rem = rem – y
     let div = div + 1
  return div
```

Run-time:

Proportional to the *input's size*

Division by the "long division" method:

```
      5   8
    ------
1   7   5 | 3
1   5   0
-------
    2   5
    2   4
    -----
        1
```

Run-time:

Proportional to the *number of digits*

Run-time

Division by repetitive subtraction:

```
divide (x, y):
  let div = 0
  let rem = x
  while rem ≤ x
     let rem = rem – y
     let div = div + 1
  return div
```

Example:

Compute 72347234723484938492340493 4 / 3

Run-time:

Proportional to the *input's size* / 3
~241157000000000000000000000

Division by the "long division" method:

```
      5   8
   ---------
1  7   5  | 3
1  5   0
---------
      2   5
      2   4
      ---
          1
```

Run-time:

Proportional to the *number of digits*: **27**

Division

Another efficient division algorithm:

```
/* Returns the integer part of x / y,
   where x ≥ 0 and y > 0  */
divide (x, y):
    if  (y > x) return 0
    q = divide (x, 2 * y)
    if ((x – 2 * q * y) < y)
        return 2 * q
    else
        return 2 * q + 1
```

Run-time:

Proportional to $log_2 N$

- The algorithm involves only addition operations
- Can be implemented efficiently in either software or hardware

Recap

```
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)
}
```

Square root

The square root function $\sqrt{x}$ has two appealing properties:

- Its inverse function (x^2) can be easily computed
- It is a monotonically increasing function

Therefore:

- Square roots can be computed using *binary search*
- And the run-time of binary search is logarithmic.

```
/* Computes the integer part of y = √x */
// Strategy: finds an integer y such that y^2 ≤ x < (y+1)^2  (for 0 ≤ x < 2^n)
// by performing a binary search in the range 0 ... 2^(n/2) – 1
sqrt (x):
    y = 0
    for j = n/2 – 1 ... 0 do
        if (y + 2^j)^2 ≤ x then y = y + 2^j
    return y
```

Implementation notes: Multiplication

```
/* Returns x*y, where x, y ≥ 0. */
// Bit-wise strategy for
// multiplying n-bit numbers:
multiply(x, y):
  sum = 0
  shiftedX = x
  for  i = 0 ... n – 1   do
      if  ((i'th bit of y) == 1)
          sum = sum + shiftedX
      shiftedX = shiftedX * 2
  return sum
```

How to handle negative numbers?

If the inputs are two's complement values, the algorithm works fine as-is

How to handle overflow?

The algorithm always returns the correct answer modulo 2^{16}

Implementation notes: Multiplication

```
/* Returns x*y, where x, y ≥ 0. */
// Bit-wise strategy for
// multiplying n-bit numbers:
multiply(x, y):
  sum = 0
  shiftedX = x
  for i = 0 ... n – 1  do
    if ((i'th bit of y) == 1)
      sum = sum + shiftedX
    shiftedX = shiftedX * 2
  return sum
```

How to implement *i*'th bit of *y* ?

We suggest encapsulating this operation as follows:

```
// Returns true if the i-th bit of x is 1, false otherwise
function boolean bit(int x, int i)
```

In principle, `bit(x,i)` can be implemented using bit shifting operations

Instead, we suggest using a fixed array that holds the 16 values 2^i, $i = 0,...,15$

- Declare a static array, say `twoToThe[i]`, as a class variable of the OS `Math` class
- Construct the array using a `Math.init` function
- This array can support the implementation of `bit(x,i)`

Implementation notes: Division

```
/* Returns the integer part of x / y,
   where x ≥ 0 and y > 0. */
divide (x,y):
    if  (y > x) return 0
    q = divide (x, 2 * y)
    if ((x – 2 * q * y) < y)
        return 2 * q
    else
        return 2 * q + 1
```

Issues:

- Handling negative numbers

  Solution: divide |*x*| by |*y*|,
  then set the result's sign

- Handling overflow (of *y*)

  Solution: the overflow can be detected when *y* becomes negative

  Change the function's first statement to:

  if (*y* > *x*) or (*y* < 0) return 0

Implementation notes: Square root

/* Computes the integer part of $y = \sqrt{x}$ */
// Strategy: finds an integer y such that $y^2 \le x < (y+1)^2$ (for $0 \le x < 2^n$)
// by performing a binary search in the range $0 \ldots 2^{n/2} - 1$
sqrt (x):
  $y = 0$
  for $j = n/2 - 1 \ldots 0$ do
  if $(y + 2^j)^2 \le x$ then $y = y + 2^j$
  return y

Issue:

The calculation of $(y + 2^j)^2$ can overflow

Solution:

Change the condition $(y + 2^j)^2 \le x$ to:

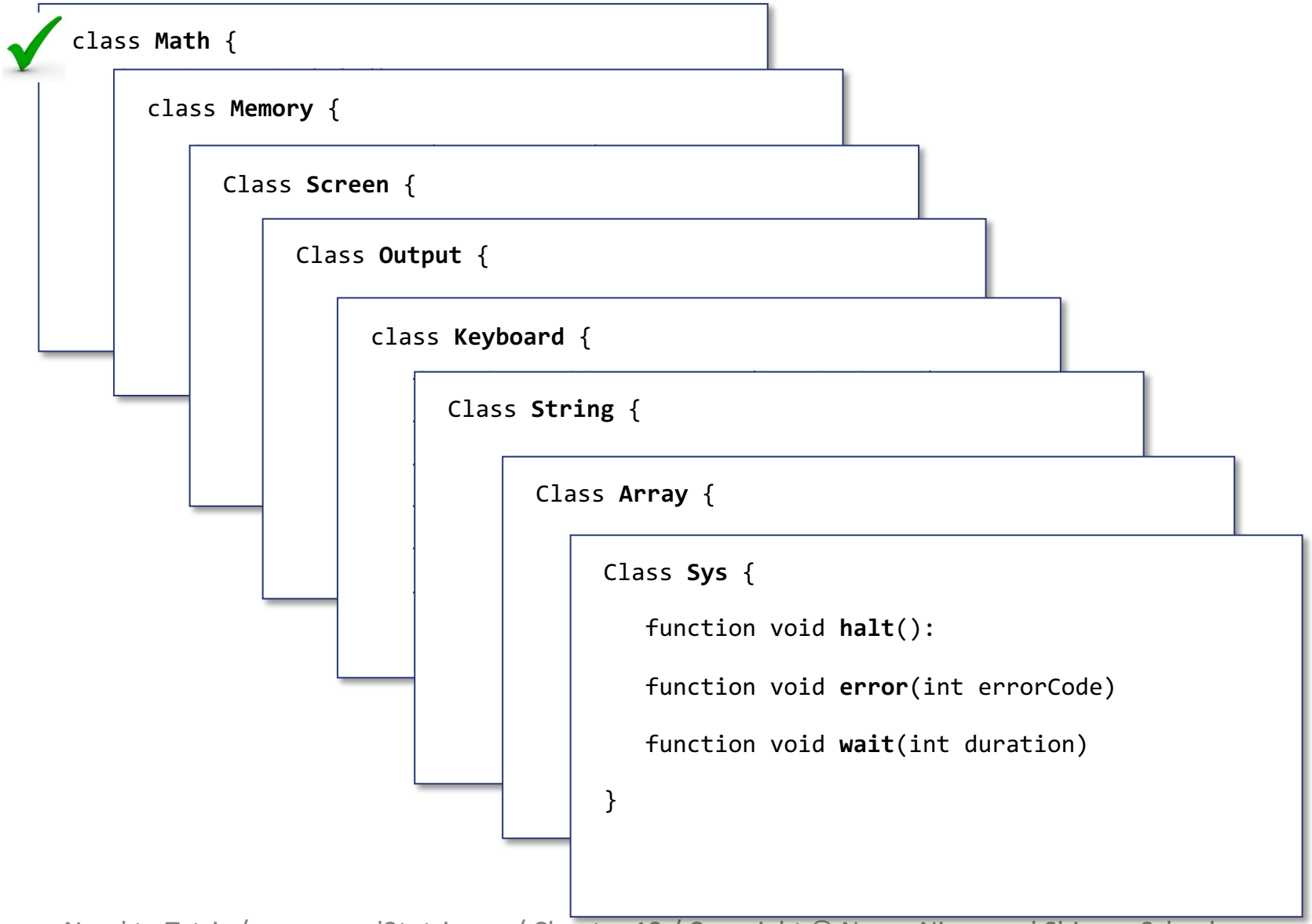
$$(y + 2^j)^2 \le x \text{ and } (y + 2^j)^2 > 0$$

Recap

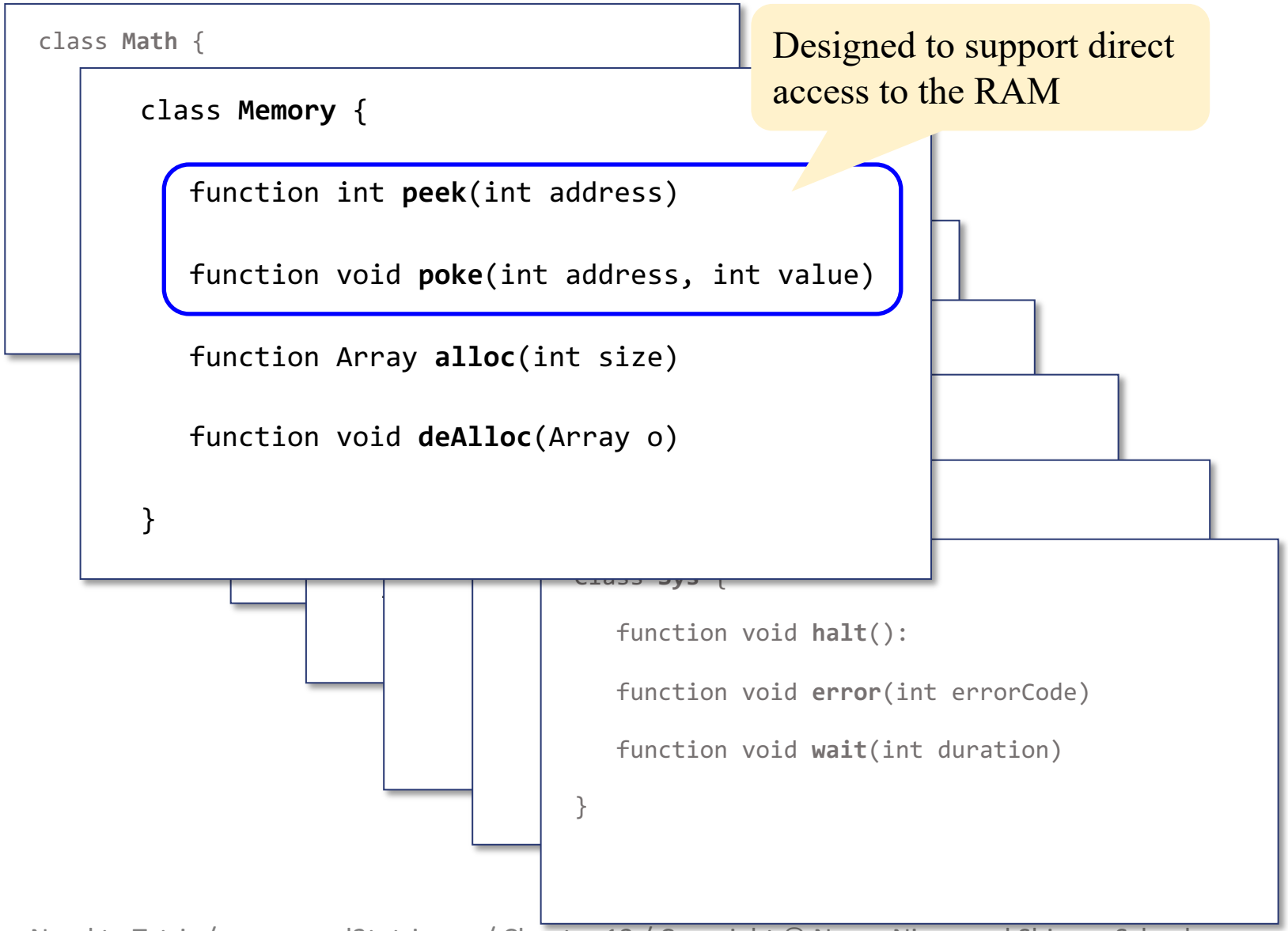
```
class Math {
   function void init()
   function int abs(int x)
   function int multiply(int x, int y)
   function int divide(int x, int y)
   function int min(int x, int y)
   function int max(int x, int y)
   function int sqrt(int x)
}
```

The implementation of the remaining `Math` functions is simple.

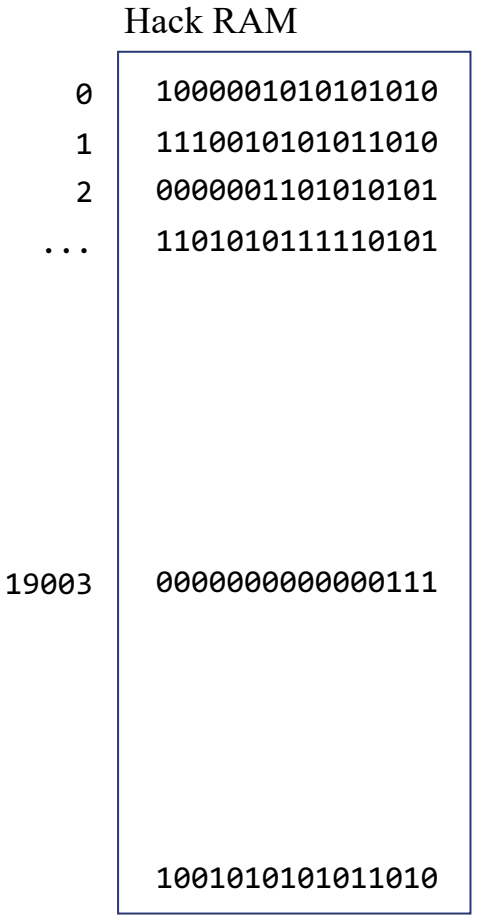
Recap


class Math {
class Memory {
Class Screen {
Class Output {
class Keyboard {
Class String {
Class Array {
Class Sys {
function void halt():
function void error(int errorCode)
function void wait(int duration)
}

Recap


class Math {
class Memory {
function int peek(int address)
function void poke(int address, int value)
function Array alloc(int size)
function void deAlloc(Array o)
}
function void halt():
function void error(int errorCode)
function void wait(int duration)
}
Designed to support direct access to the RAM

OS class `Memory`



The need

High-level programs need direct access to individual words in the RAM

Solution

Have the OS provide low-level services that facilitate direct RAM access (read / write).

OS class Memory

Hack RAM

0	1000001010101010
1	1110010101011010
2	0000001101010101
...	1101010111110101
19003	0000000000000111
	1001010101011010

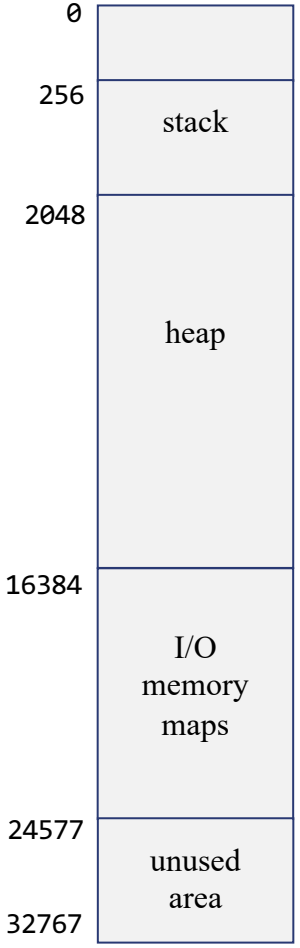
OS Memory class API

```
/* OS Memory operations  */
class Memory {

    /* Returns the value of the given RAM address */
    function int peek(int address) {}

    /* Sets the value of the given RAM address to the given value */
    function void poke(int address, int value) {}

    ...
}
```

Usage:

```
...
let x = Memory.peek(19003)   // x becomes 7
...
do Memory.poke(19003,-1)     // RAM[19003] becomes 11…1
...
```

Implementation notes



Accessing the RAM from Jack:

```
/* OS Memory operations  */
class Memory {
   ...
   static Array ram;
   ...
   function void init() {
      let ram = 0;
      ...
   }

   /* Returns the value of the given RAM address */
   function int peek(int address) {}

   /* Sets the value of the given RAM address to the given value */
   function void poke(int address, int value) {}

   ...
}
```

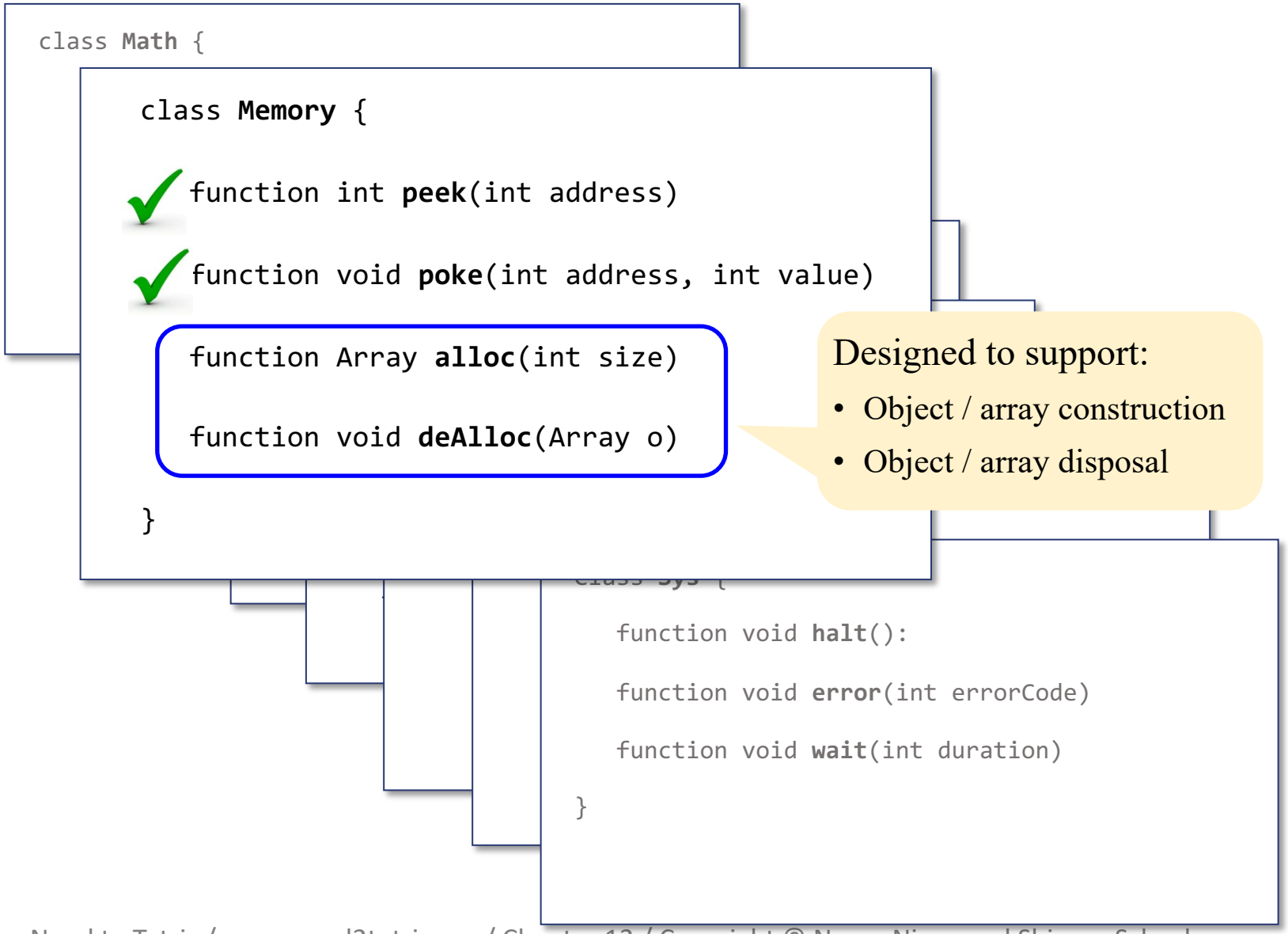
The trick works because...

- Jack is weakly typed:
- `let ram = 0` does not cause the compiler to complain
- From this point onward:
- To set `RAM[`*addr*`]` to *val* :
- `let ram[`*addr*`] =` *val*`;`
- (the compiler generates code that puts *val* in address 0 + *addr*)

Implementing `peek` and `poke`:

Manipulate the `ram` array as shown above.

Recap



Designed to support:

- Object / array construction
- Object / array disposal

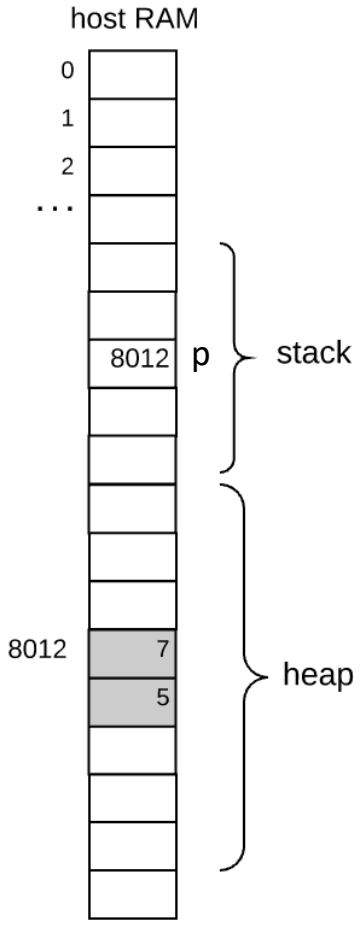
The need: object construction

```
// In some class:
var Point p;
...
let p = Point.new(7,5);
```

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

When the compiler translates the constructor, it injects low-level (VM) code affecting:

`this = Memory.alloc(2)`



OS `Memory` class API

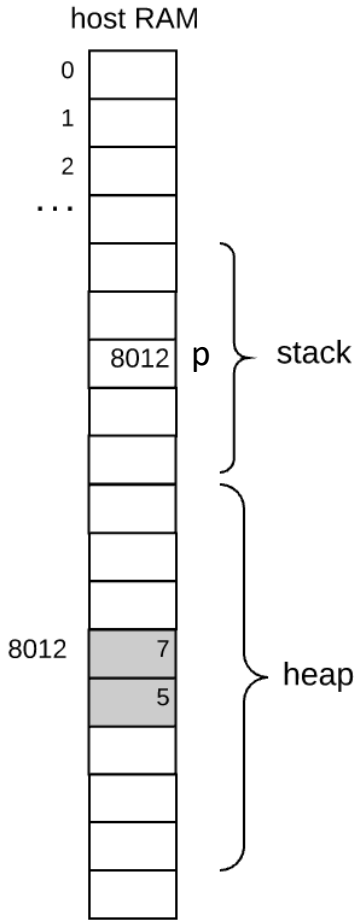
```
class Memory {
   ...
   /* Finds and allocates from the heap a memory block of the
      specified size and returns a reference to its base address */
   function int alloc(int size) {}
   /* De-allocates the given object and frees its space */
   function void deAlloc(int object) {}
}
```

The need: object destruction

```
// In some class:
...
do p.dispose();
```

```
class Point {
   ...
   /** Disposes this point */
   method void dispose() {
      do Memory.deAlloc(this);
      return;
   }
...
```

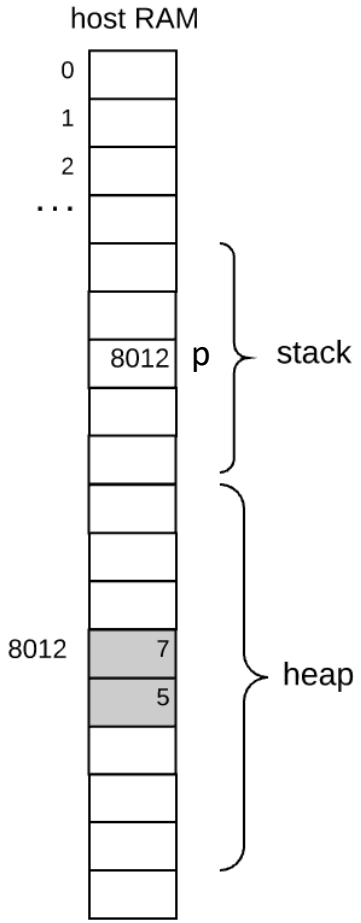
In languages that have *garbage collection*, there is no need to dispose objects explicitly



OS `Memory` class API

```
class Memory {
   ...
   /* Finds and allocates from the heap a memory block of the
      specified size and returns a reference to its base address */
   function int alloc(int size) {}
   /* De-allocates the given object and frees its space */
   function void deAlloc(int object) {}
}
```

Implementing `alloc` and `deAlloc`



Implementation:

Heap Management

OS `Memory` class API

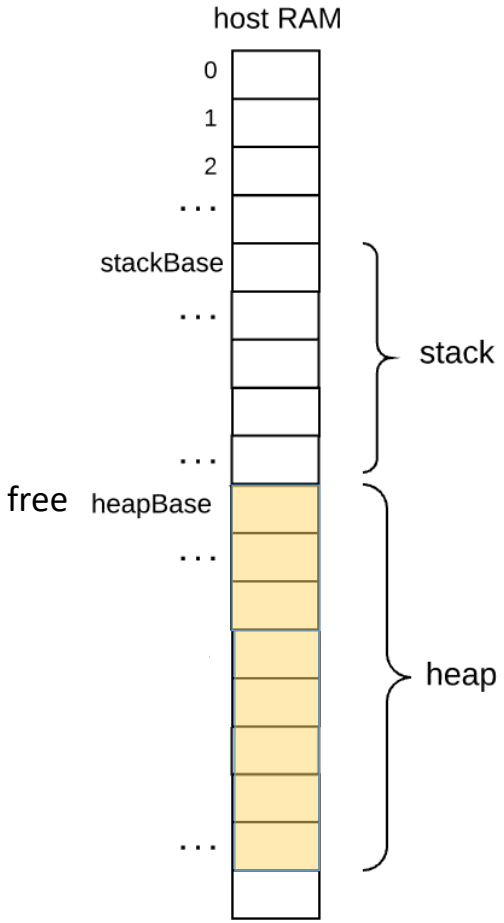
```
class Memory {
    ...
    /* Finds and allocates from the heap a memory block of the
       specified size and returns a reference to its base address */
    function int alloc(int size) {}
    /* De-allocates the given object and frees its space */
    function void deAlloc(int object) {}
}
```

Heap management (simple)

Heap management

init:

free = heapBase



Heap management (simple)

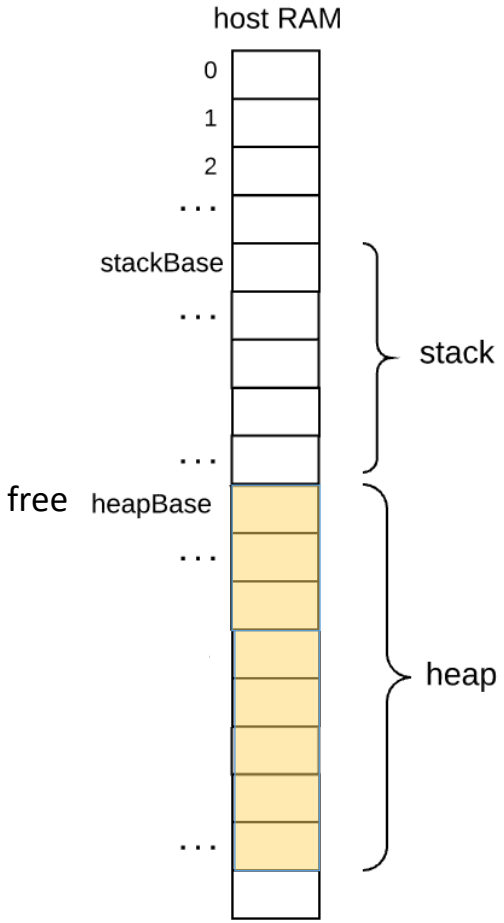
Heap management

init:

free = heapBase

// allocates a memory block of *size* words

alloc (*size*):



`alloc(3)`:

Heap management (simple)

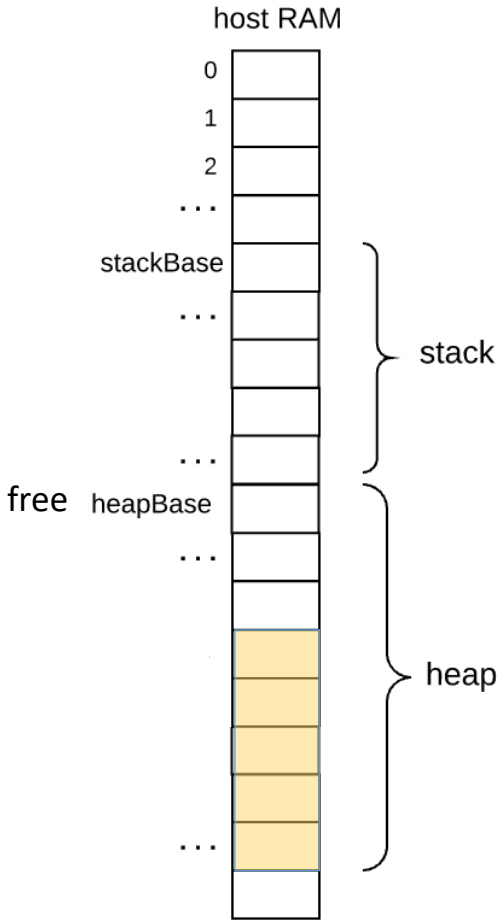
Heap management

init:

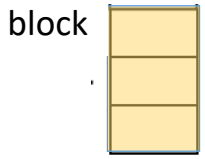
free = *heapBase*

// allocates a memory block of *size* words

alloc (*size*):



`alloc(3)`:



Heap management (simple)

Heap management

init:

free = heapBase

// allocates a memory block of *size* words

alloc (*size*):

block = free

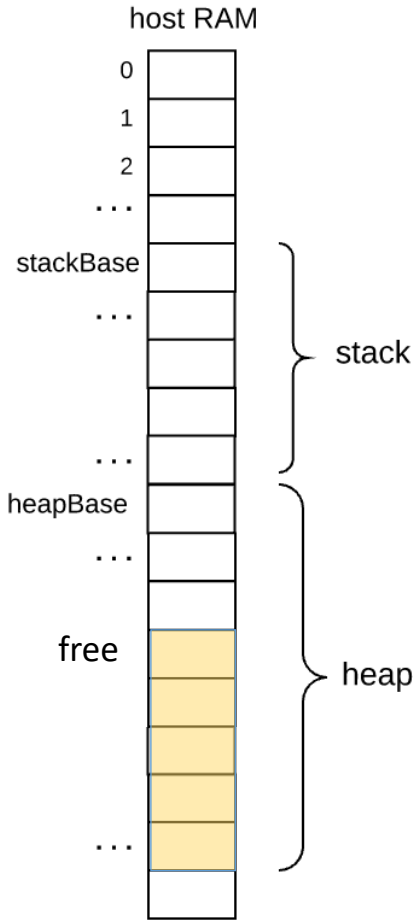
free = free + size

return *block*

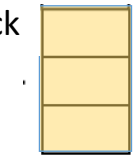
// de-allocates the memory space of the given object

deAlloc (*object*):

do nothing

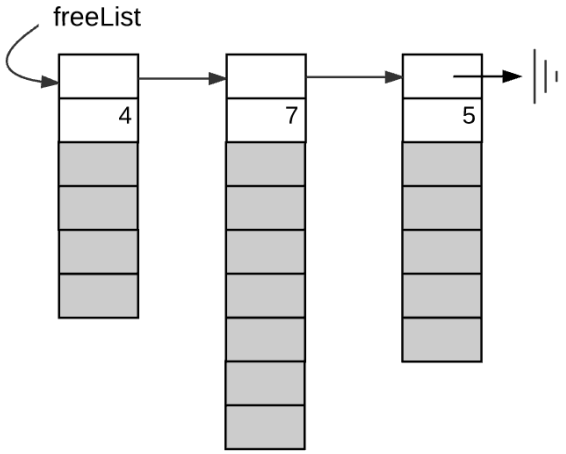


`deAlloc(block):`

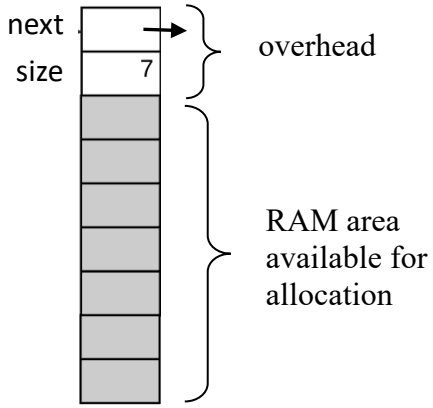


Heap management

Use a linked list to keep track of available memory segments

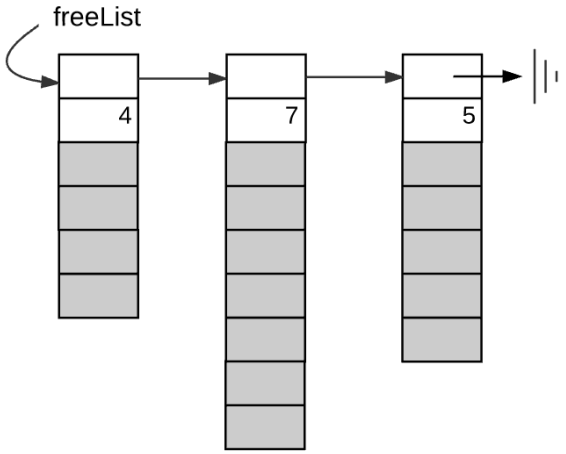


typical segment



Heap management

Use a linked list to keep track of available memory segments



alloc(*size*):

finds a block of size *size* in one of the segments, removes it from the segment, and gives it to the client

deAlloc(*object*):

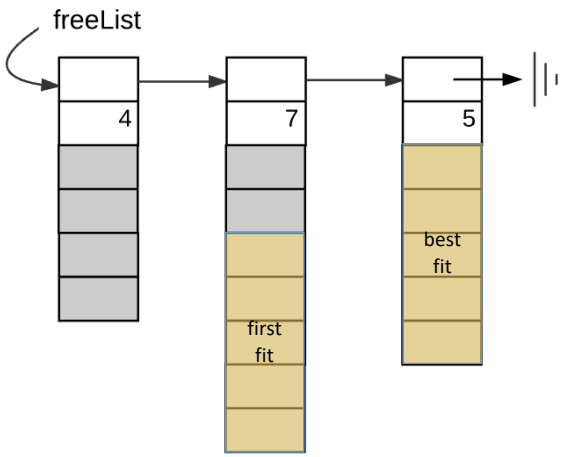
appends the object/block to the *freeList*

Heap management (detailed)

alloc(*size*):

Terminology: if *segment.size* $\geq$ *size*+2
we say that the segment is *possible*

- search the *freeList* for:
  - the first possible segment (*first fit*) , or
  - the smallest possible segments (*best fit*)

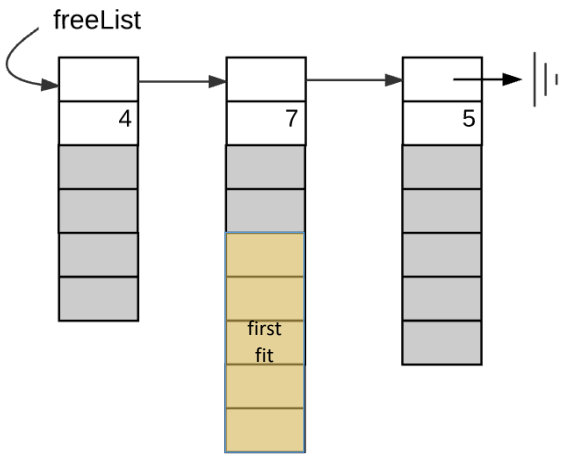


Heap management (detailed)

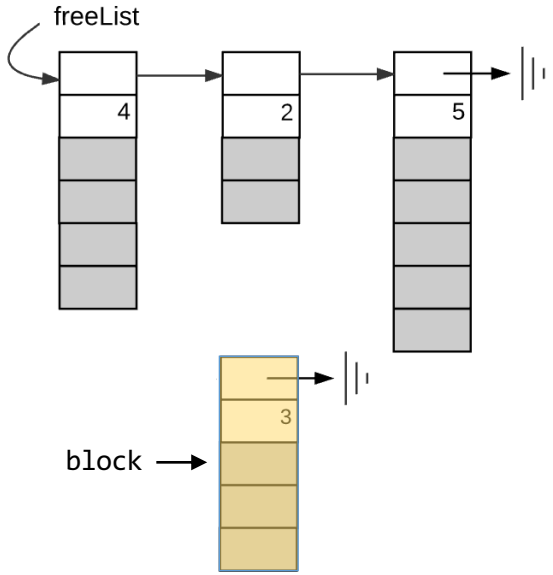
alloc(*size*):

Terminology: if *segment.size* ≥ *size*+2
we say that the segment is *possible*

- search the *freeList* for:
  - the first possible segment (*first fit*) , or
  - the smallest possible segments (*best fit*)
- carve a block of size *size*+2 from this segment
- return the base address of the block's data part



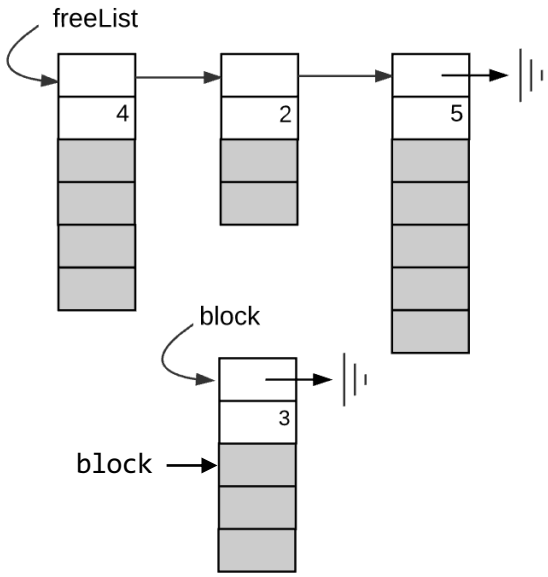
After `alloc(3)`:



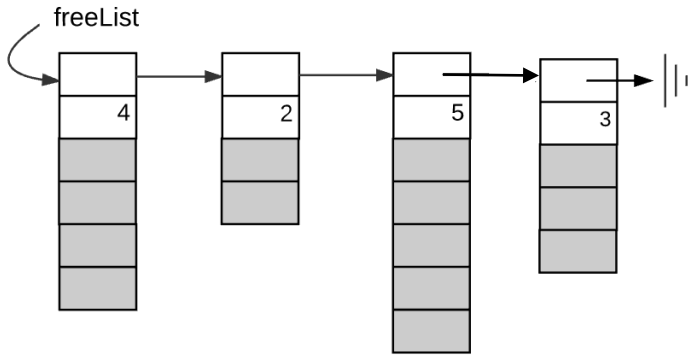
Heap management (detailed)

deAlloc(*object*):

append *object* to the end of the *freeList*



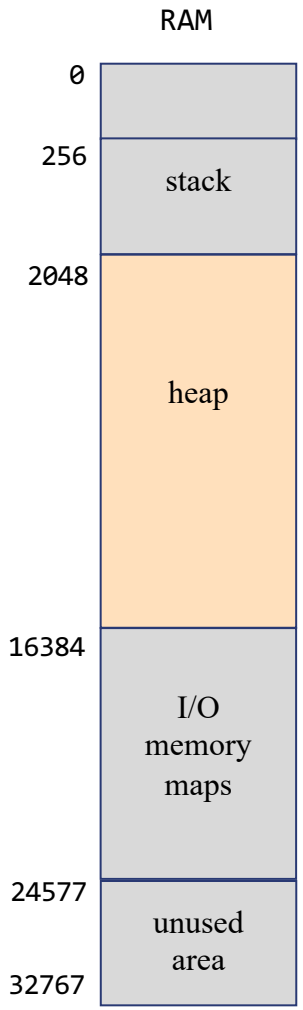
After `deAlloc(block)`:



The more we recycle (`deAlloc`), the more the `freeeList` becomes fragmented

Solution: Periodical defragmentation (nice extension to the Jack OS).

Proposed implementation



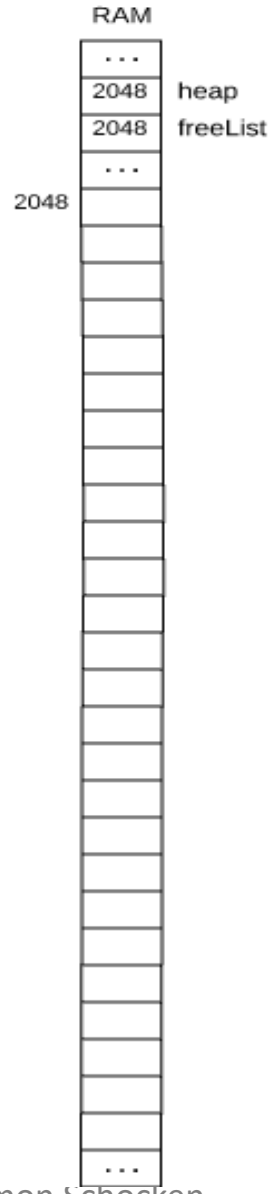
Proposed implementation

```
class Memory {
   ...
   static Array heap;
   ...
   function Memory.init(…)
      let heap = 2048; // heapBase
      let freeList = heap;
      let heap[0] = 0;      // next
      let heap[1] = 14334; // length
      ...
   }
   ...
}
```

Initial state:

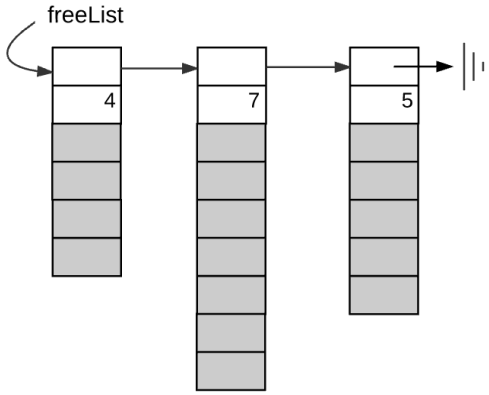
The entire heap is available

(*freeList* points at one long segment)

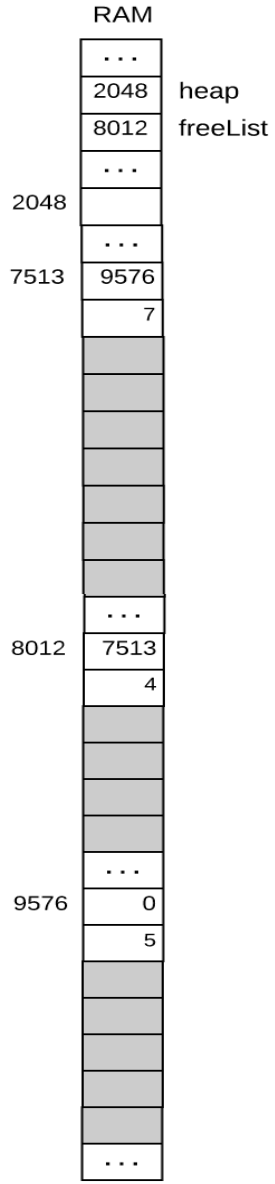


Proposed implementation

```
class Memory {
   ...
   static Array heap;
   ...
   function Memory.init(...)
      let heap = 2048; // heapBase
      ...
   }
   function Memory.alloc(...)
   ...
   function Memory.deAlloc(...)
   ...
}
```



Heap state after several `alloc` and `deAlloc` operations



- The `freeList` can be realized using the `heap` array
- The `next` and `size` properties of the memory segment beginning at address `addr` can be realized by `heap[addr-1]` and `heap[addr-2]`
- `alloc`, `deAlloc`, and `deFrag` can be realized as operations on the `heap` array.

Proposed implementation (recap)

init:

freeList = *heapBase*

freeList.*size* = *heapSize*

freeList.*next* = `0`

/* Allocates a memory block of *size* words */

alloc(*size*):

search *freeList* using best-fit or first-fit heuristics to obtain a segment with *segment.size* ≥ *size* + 2

if no such segment is found, return failure (or attempt defragmentation)

block = base address of the found space

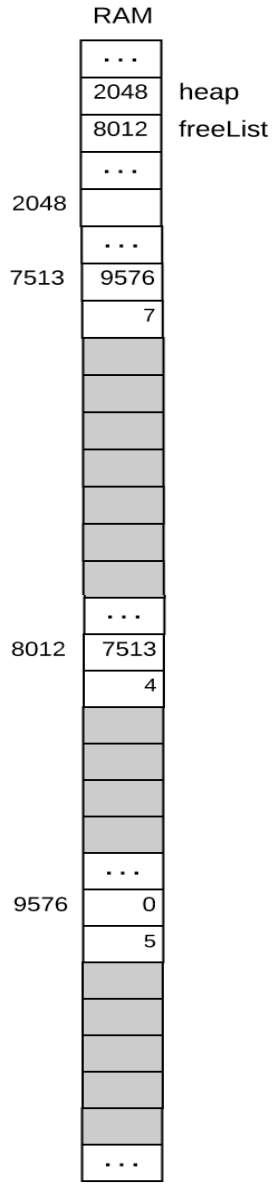
update the *freeList* and the fields of *block* to account for the allocation

return *block*

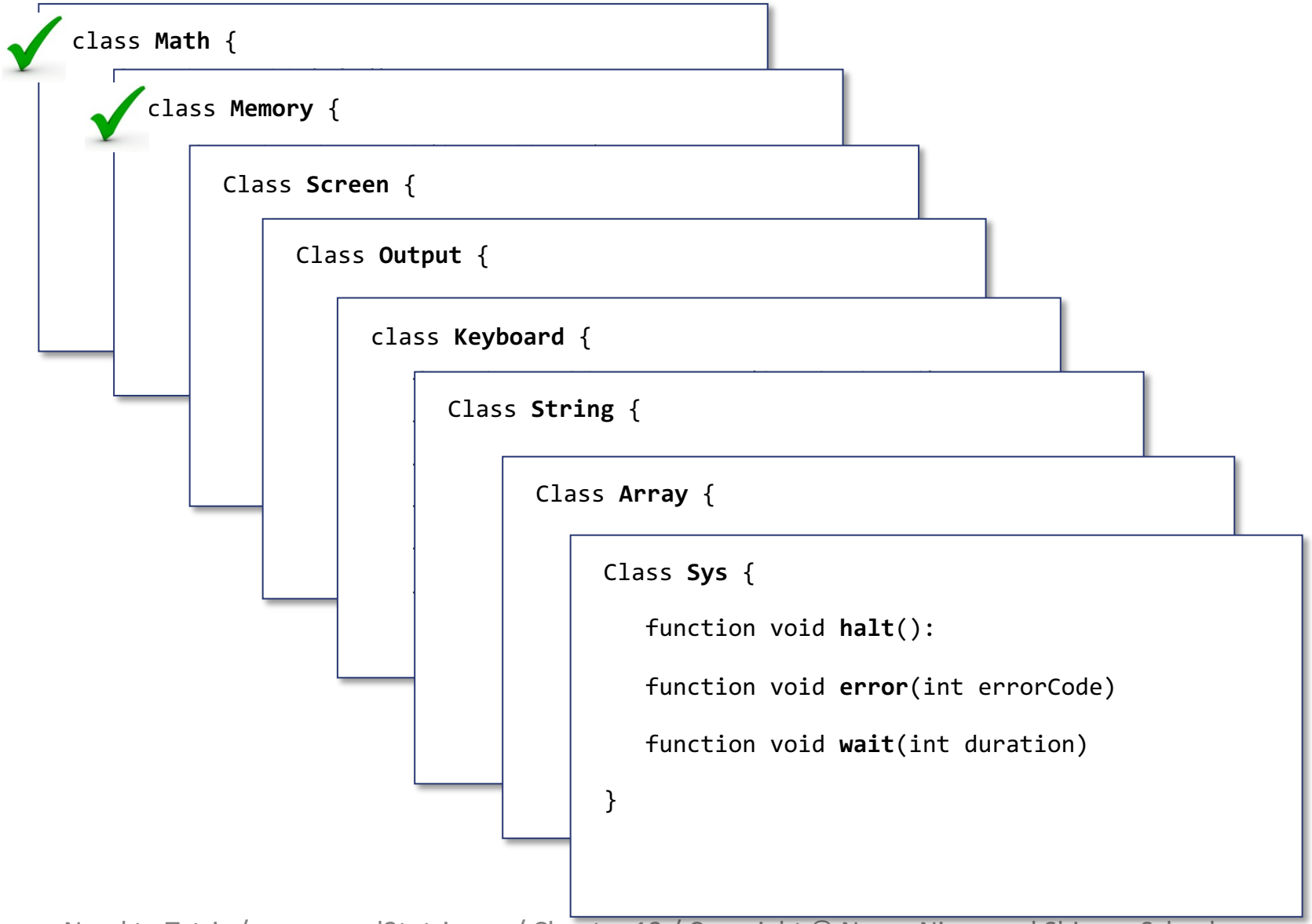
/* de-allocates the memory space of the given object */

deAlloc(*object*):

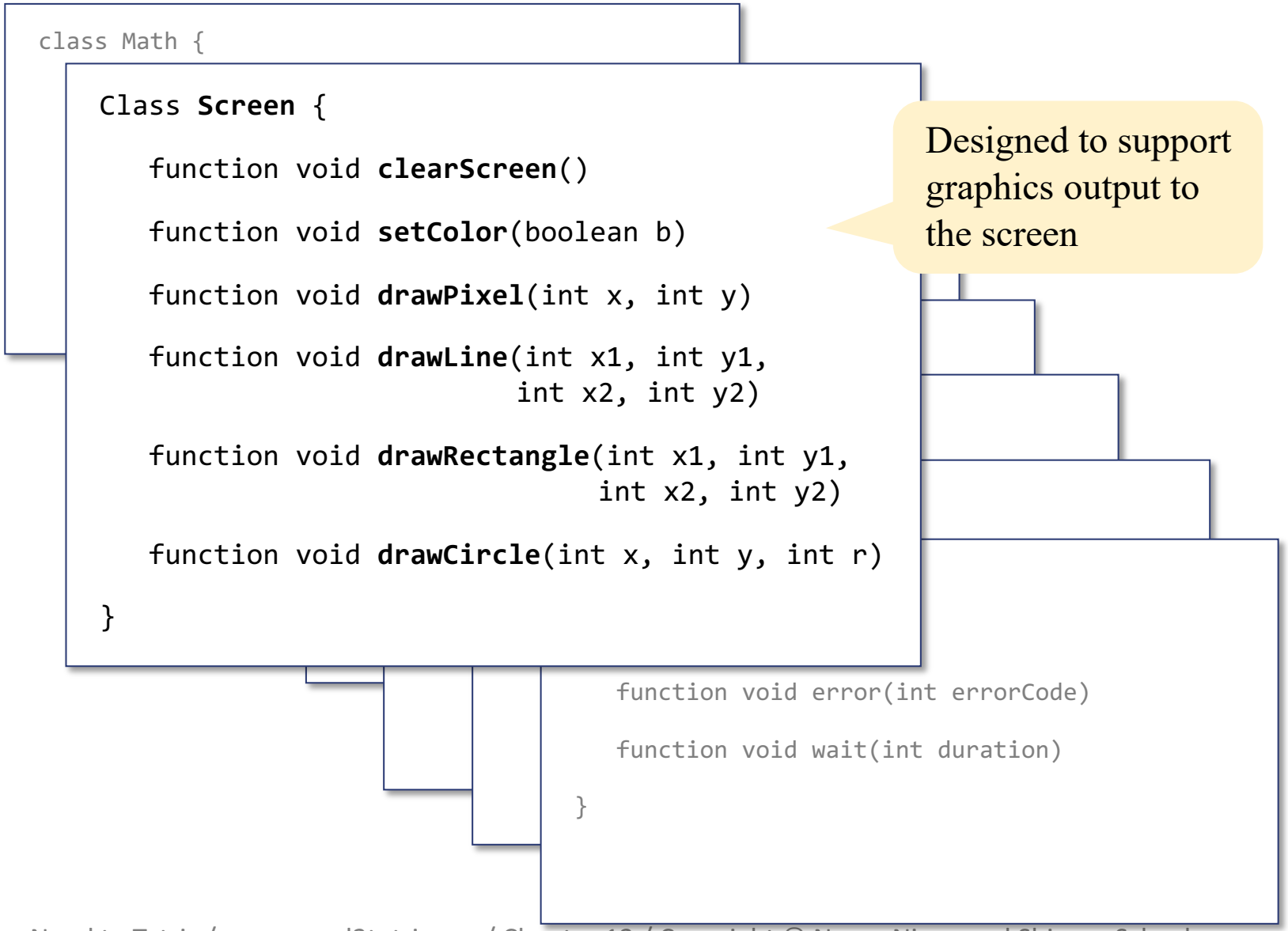
append *object* to the end of the *freeList*



The Jack OS

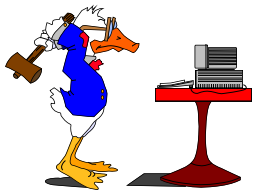

class Math {
class Memory {
Class Screen {
Class Output {
class Keyboard {
Class String {
Class Array {
Class Sys {
function void halt():
function void error(int errorCode)
function void wait(int duration)
}

The Jack OS

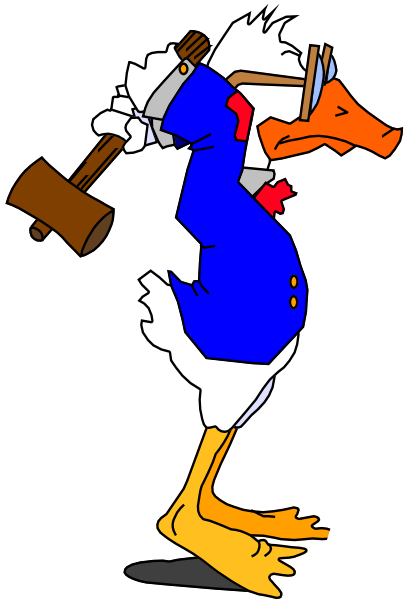


Designed to support graphics output to the screen

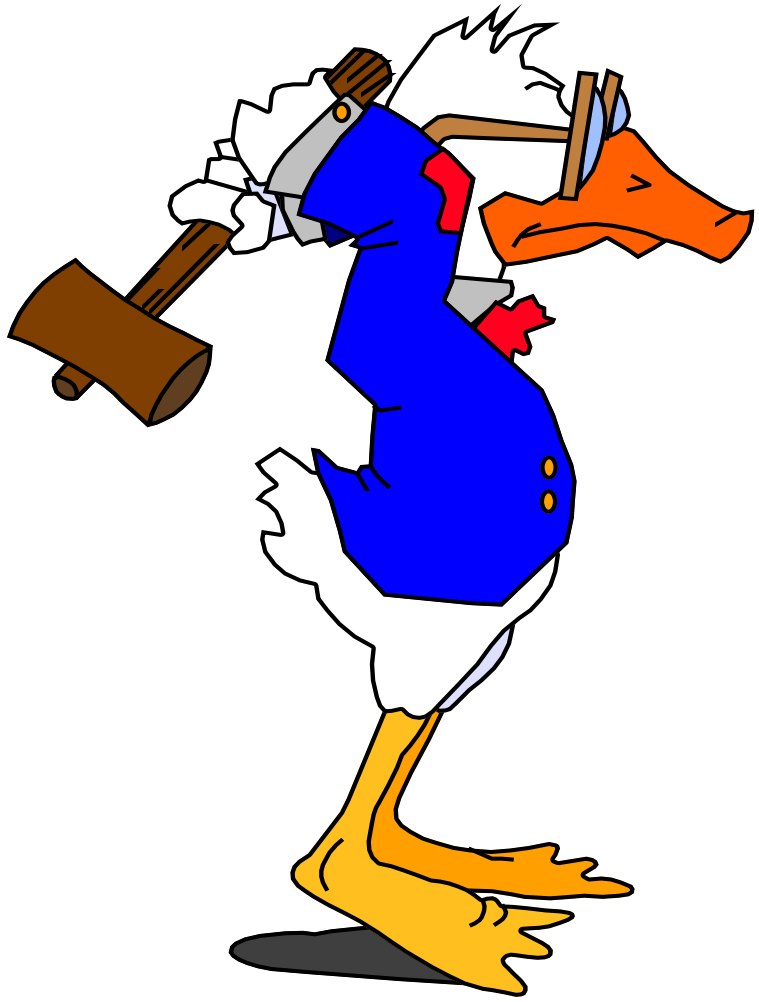
Graphics



Graphics



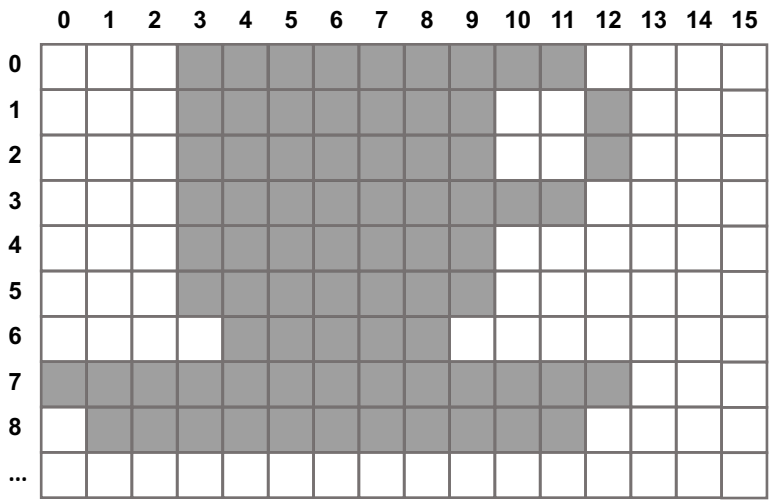
Graphics



vector graphics

bitmap graphics

Graphics



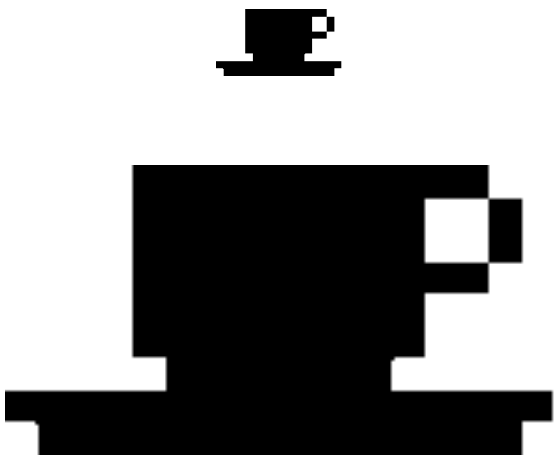
vector graphics file

```
drawLine(3,0,11,0);
drawRectangle(3,1,9,5);
drawLine(12,1,12,2);
drawLine(10,3,11,3);
drawLine(4,6,8,6);
drawLine(0,7,12,7);
drawLine(1,8,11,8);
```

bitmap graphics file

```
0001111111100000
0001111111010000
0001111111010000
0001111111100000
0001111111000000
0001111111000000
...
```

Graphics



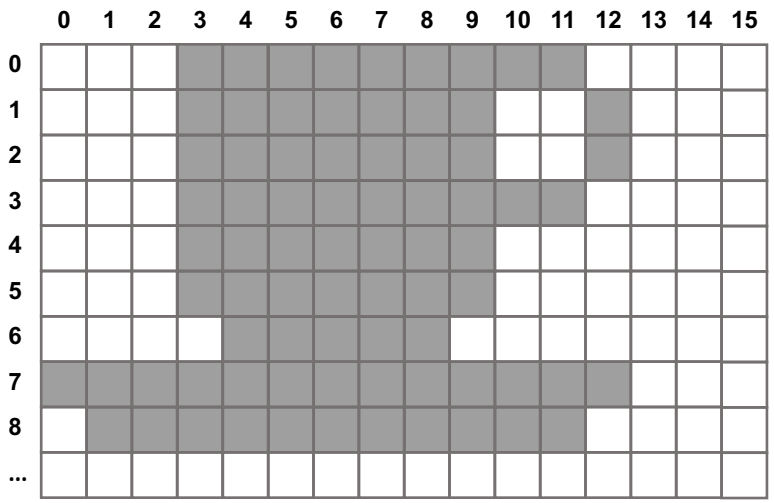
vector graphics file

```
drawLine(3,0,11,0);
drawRectangle(3,1,9,5);
drawLine(12,1,12,2);
drawLine(10,3,11,3);
drawLine(4,6,8,6);
drawLine(0,7,12,7);
drawLine(1,8,11,8);
```

bitmap graphics file

```
0001111111100000
0001111111010000
0001111111010000
0001111111100000
0001111111000000
0001111111000000
...
```

Graphics



vector graphics file

```
drawLine(3,0,11,0);
drawRectangle(3,1,9,5);
drawLine(12,1,12,2);
drawLine(10,3,11,3);
drawLine(4,6,8,6);
drawLine(0,7,12,7);
drawLine(1,8,11,8);
```

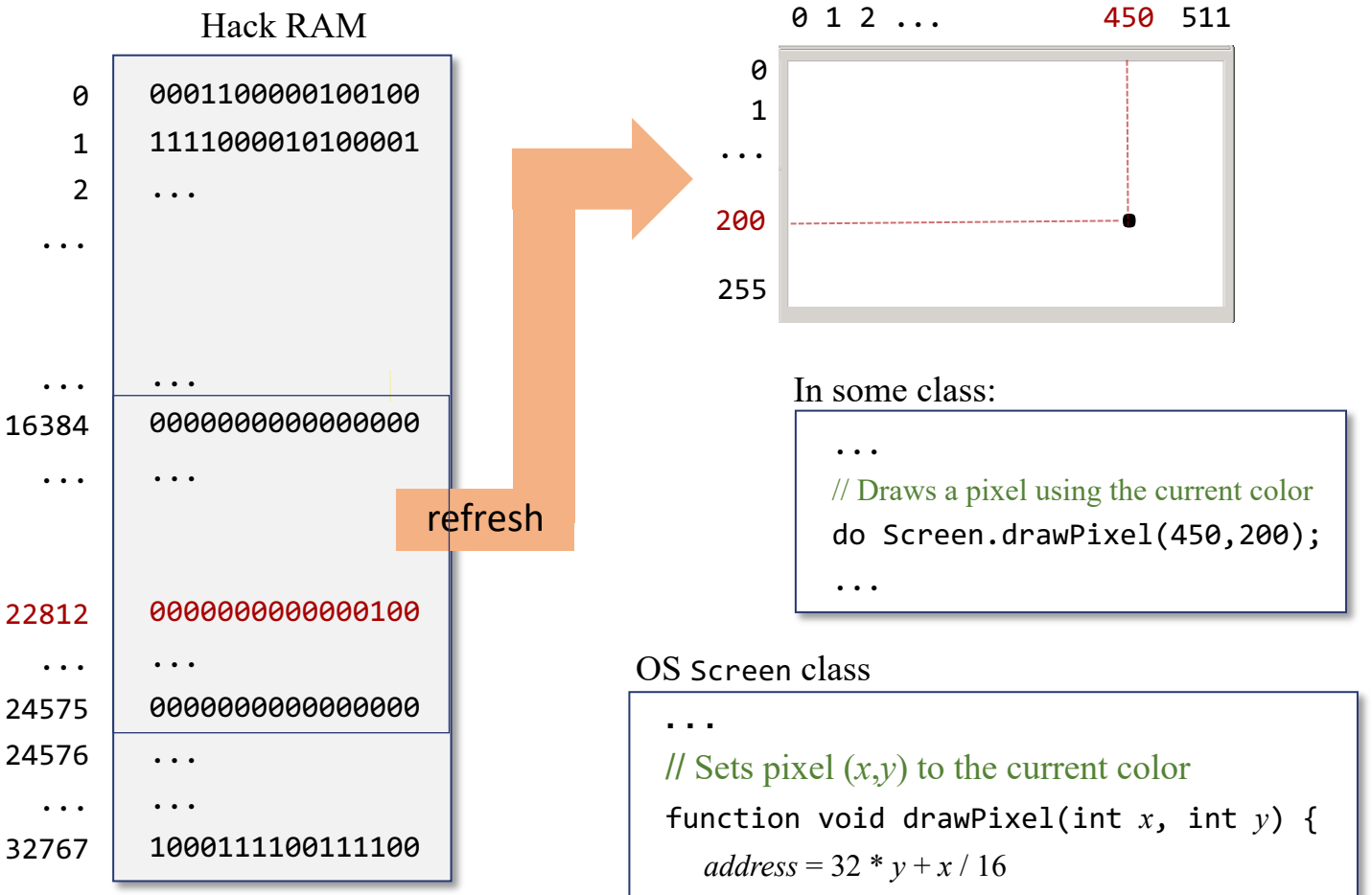
Vector graphics images can be easily:

- Stored
- Transmitted
- Scaled
- Turned into bitmap

Vector graphics primitives

- `drawPixel` (x, y)
- `drawLine` (x_1, y_1, x_2, y_2)
- `drawCircle` (x, y, r)

Pixel drawing



In some class:

```
...
// Draws a pixel using the current color
do Screen.drawPixel(450,200);
...
```

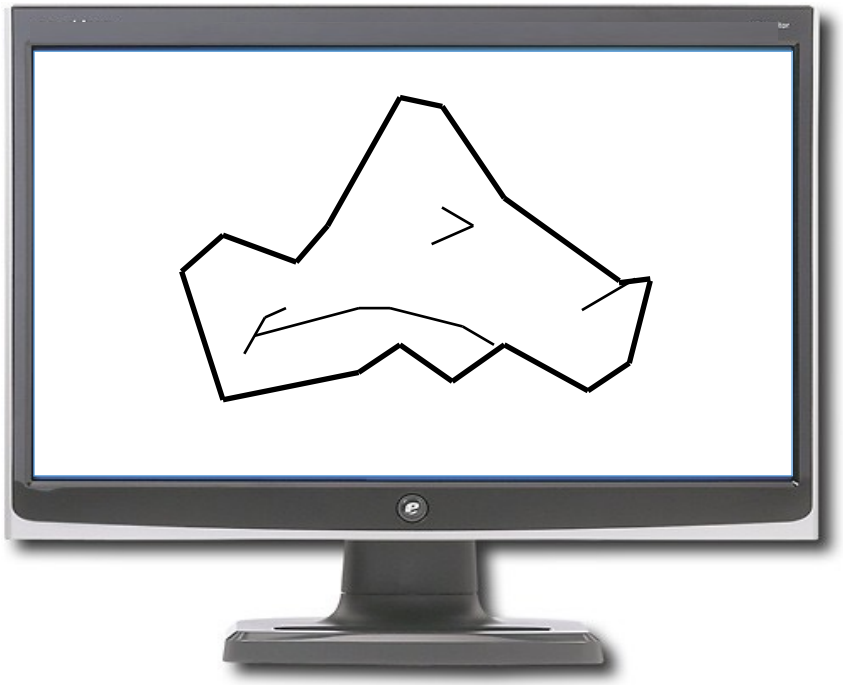
OS `Screen` class

```
...
// Sets pixel (x,y) to the current color
function void drawPixel(int x, int y) {
   address = 32 * y + x / 16
   value = Memory.peek[16384 + address]
   set the (x % 16)th bit of value to the current color
   do Memory.poke(address,value)
}
```

Recap

```
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)

}
```

Line drawing



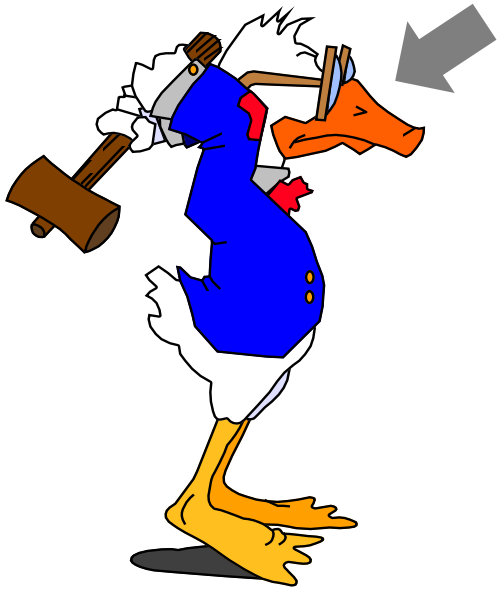
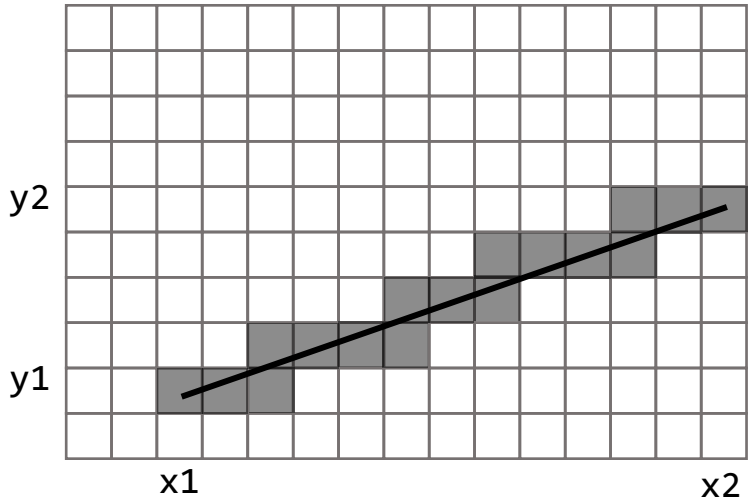


Image drawing can be done by a sequence of `drawLine` operations.

Challenge: Draw the lines *fast*.

Line drawing

`drawLine(x1,y1,x2,y2)`



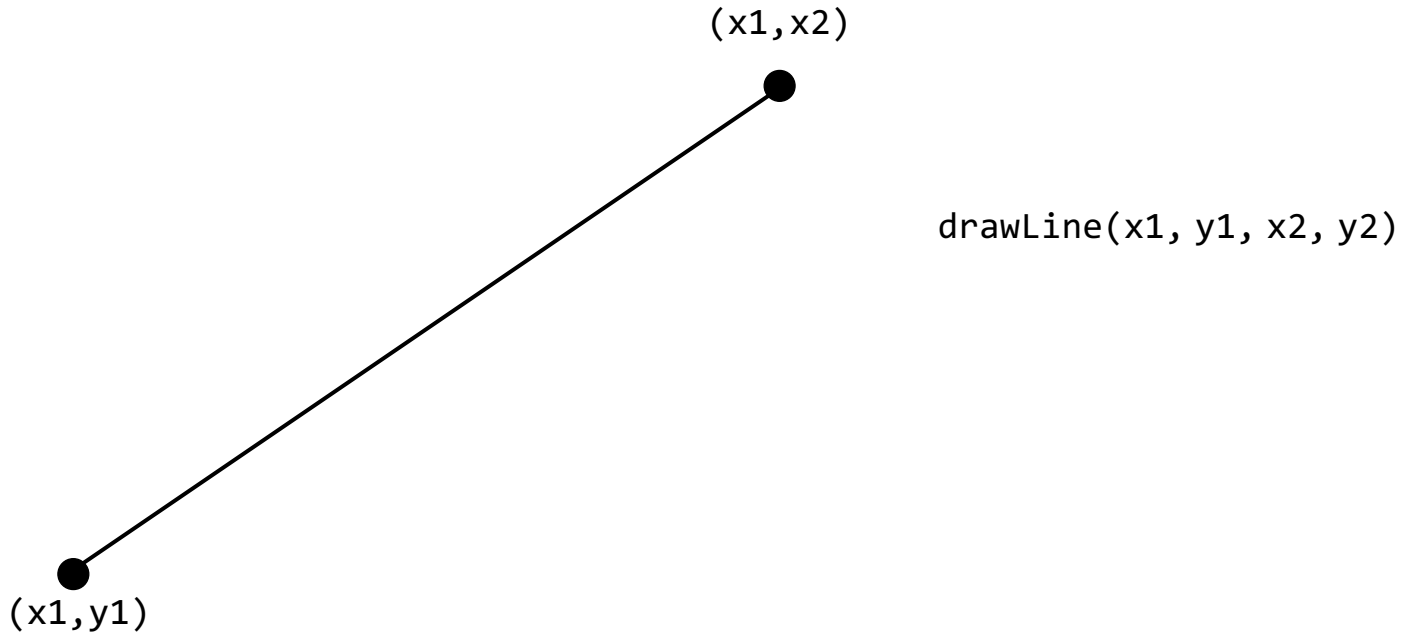
Simplifying assumption:

Let us focus on lines that go *north-east*

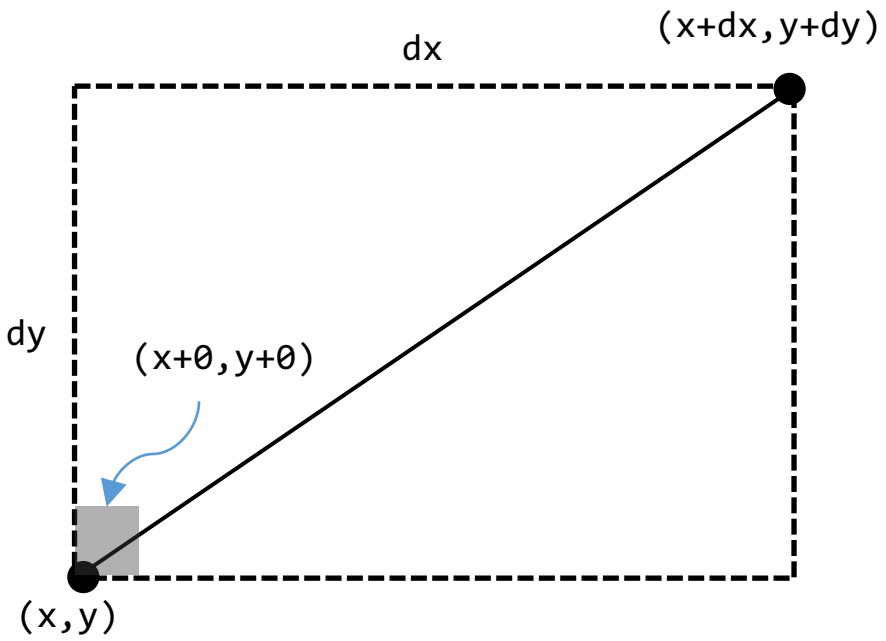
- `drawLine` is implemented through a sequence of `drawPixel` operations;
- In each stage we have to decide if we have to go *right*, or *up*

Challenge: how can we make these decisions ***fast*** ?

Line drawing



Line drawing



drawLine(x1, y1, x2, y2)

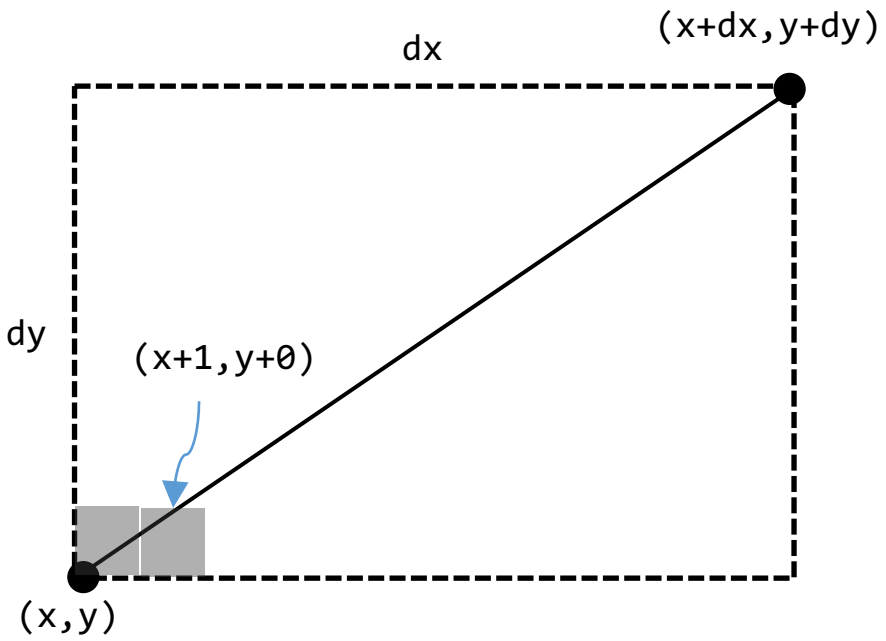
let:

```
x = x1
y = y1
dx = x2 – x1
dy = y2 – y1
```

```
a = 0, b = 0
while ... There's more work to do:
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if going right: a++
   else            b++
```

a and b record how many times we went up, and right

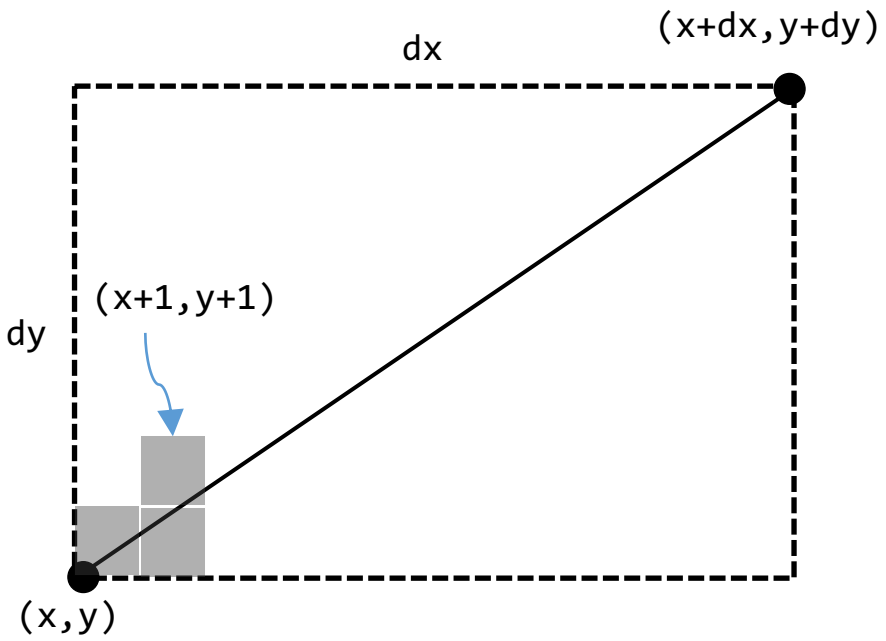
Line drawing




```
a = 0, b = 0
while ... There's more work to do:
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if going right: a++
   else            b++
```

a and b record how many times we went up, and right

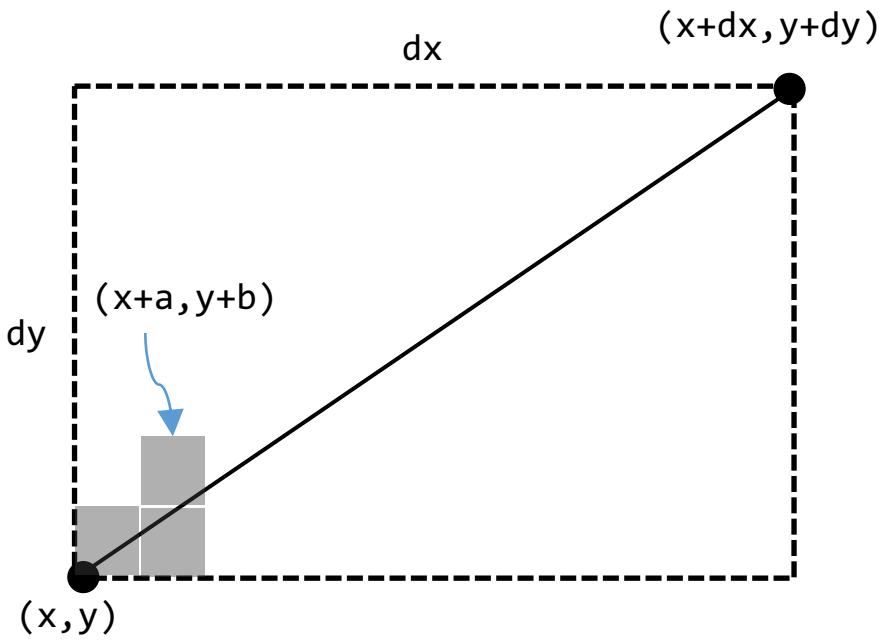
Line drawing




```
a = 0, b = 0
while  ... There's more work to do:
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if going right: a++
   else            b++
```

a and b record how many times we went up, and right

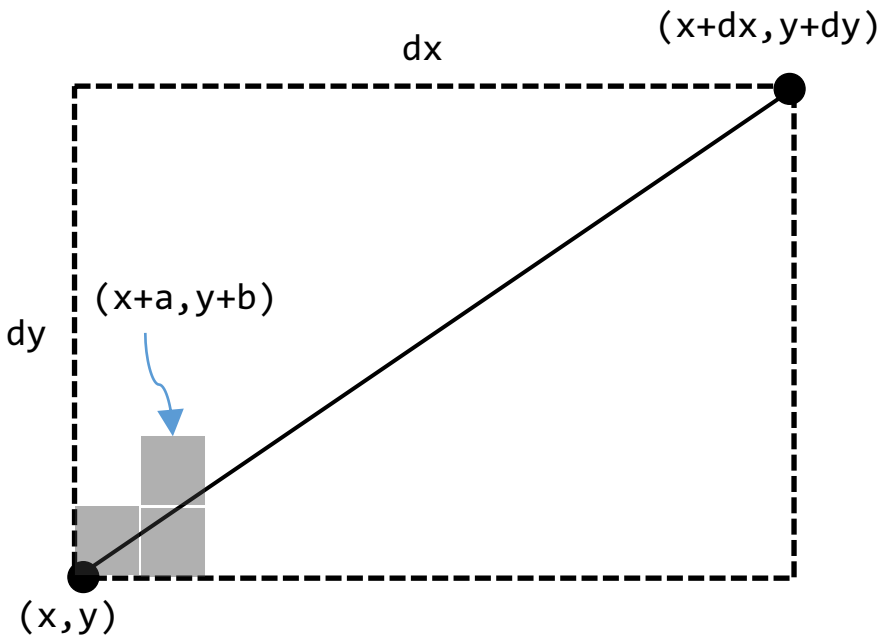
Line drawing




```
a = 0, b = 0
while ... There's more work to do:
    drawPixel(x+a, y+b)
    // decides if to go right, or up:
    if going right: a++
    else            b++
```

a and b record how many times we went up, and right

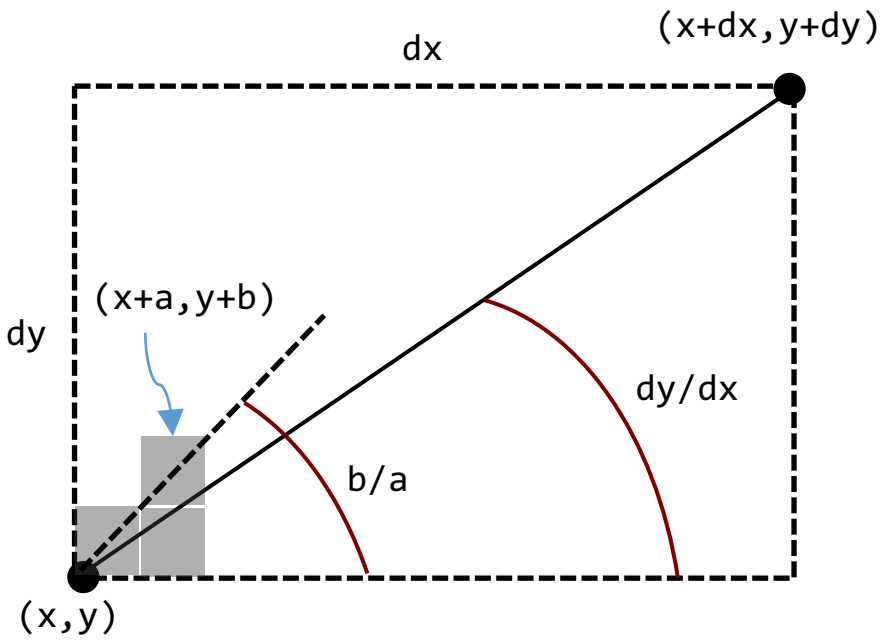
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
    drawPixel(x+a, y+b)
    // decides if to go right, or up:
    if going right: a++
    else            b++
```

a and b record how many times we went up, and right

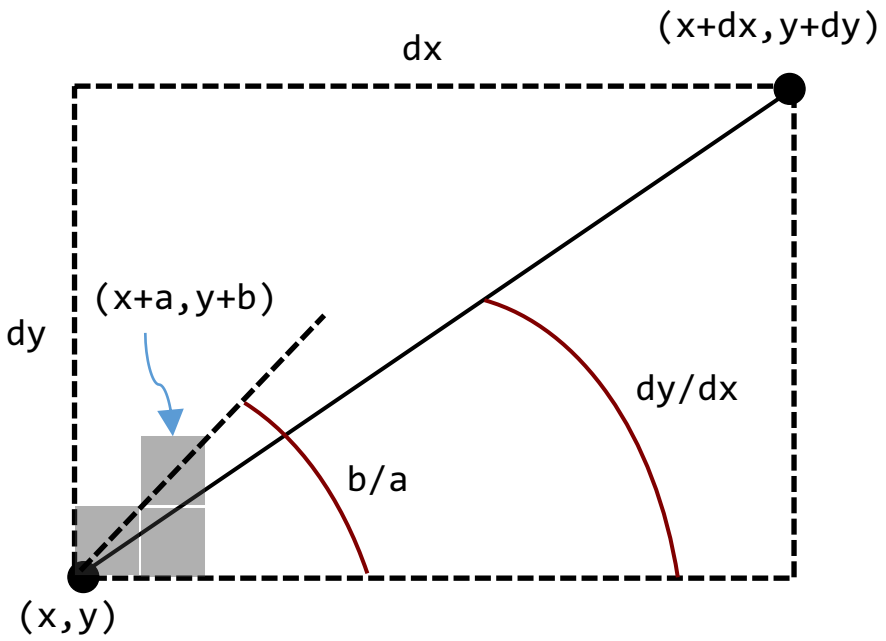
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if going right: a++
   else            b++
```

a and b record how many times we went up, and right

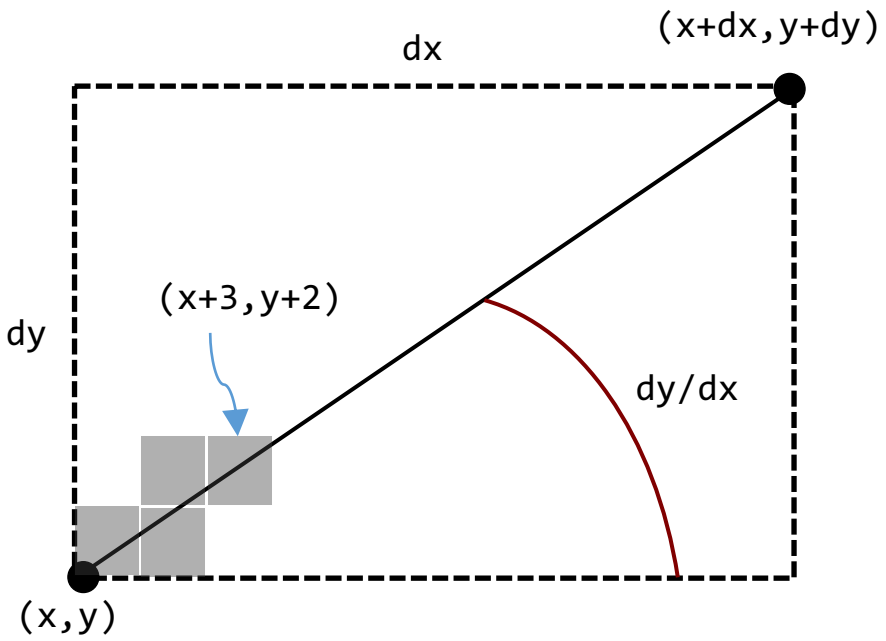
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

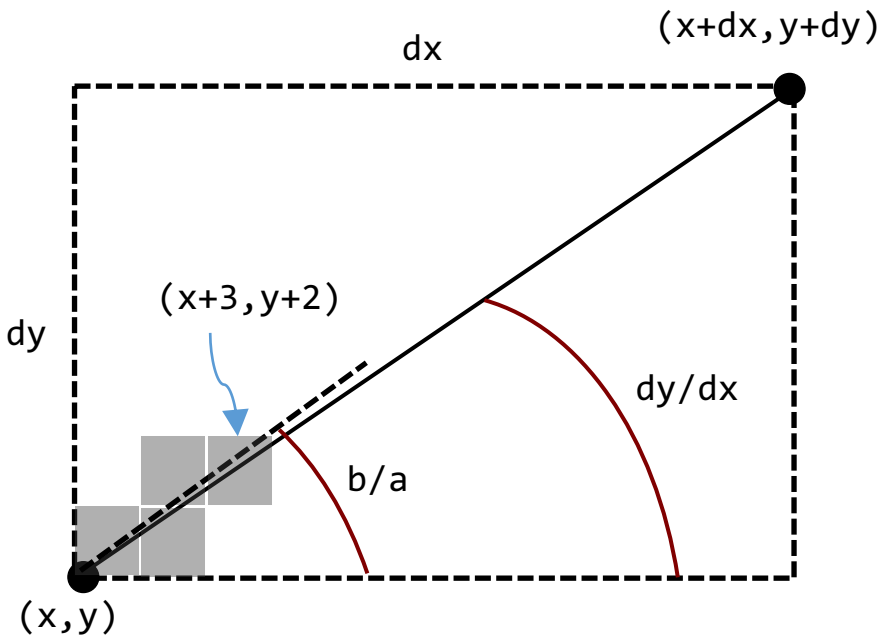
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

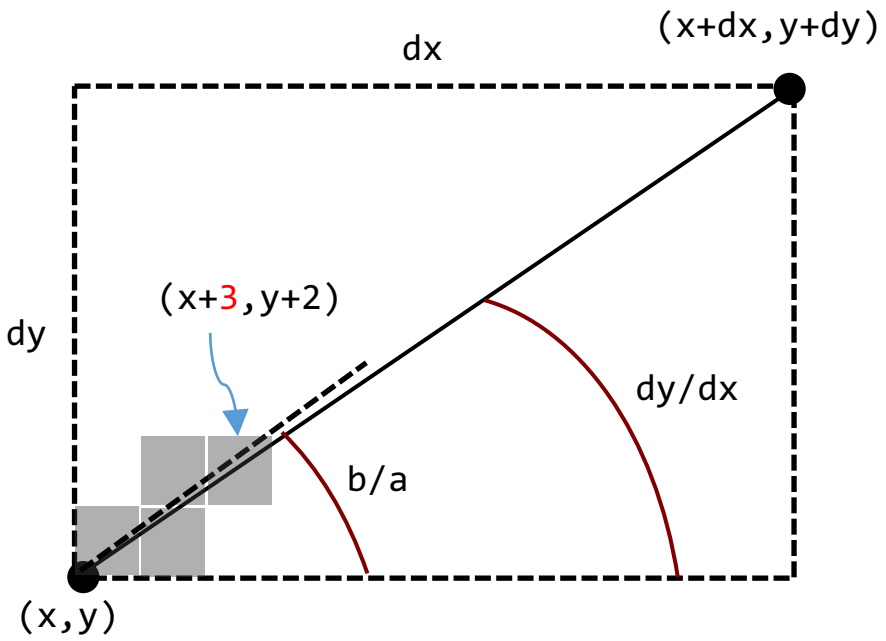
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

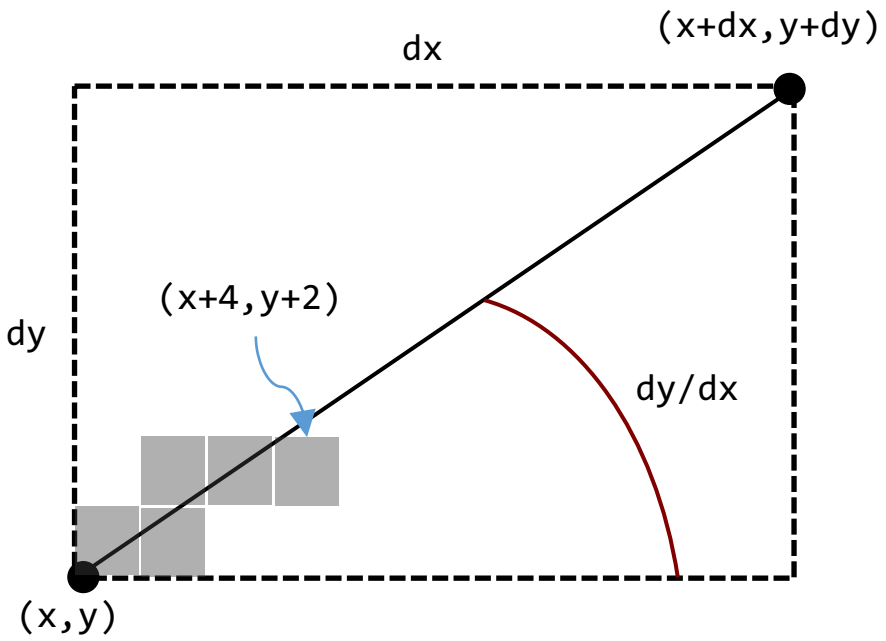
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

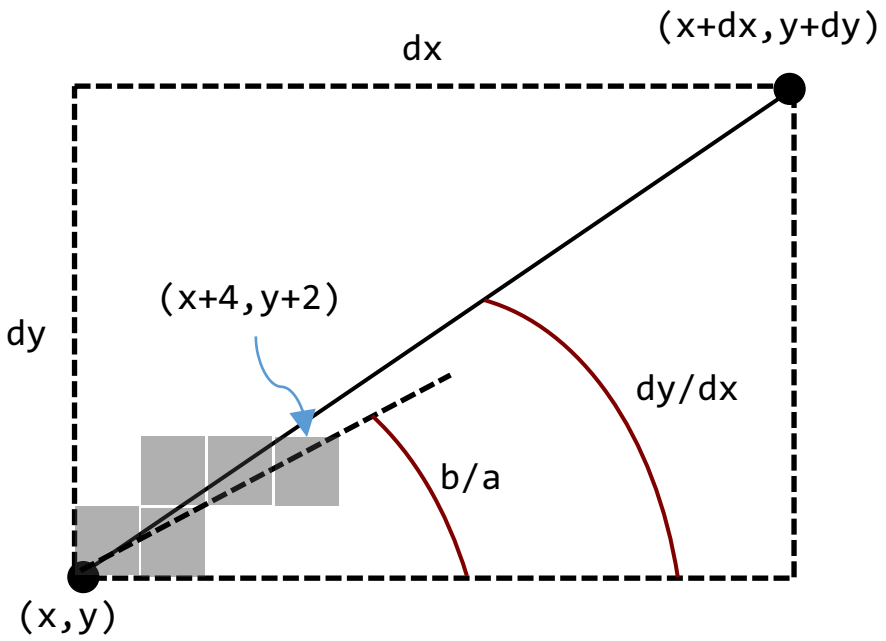
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

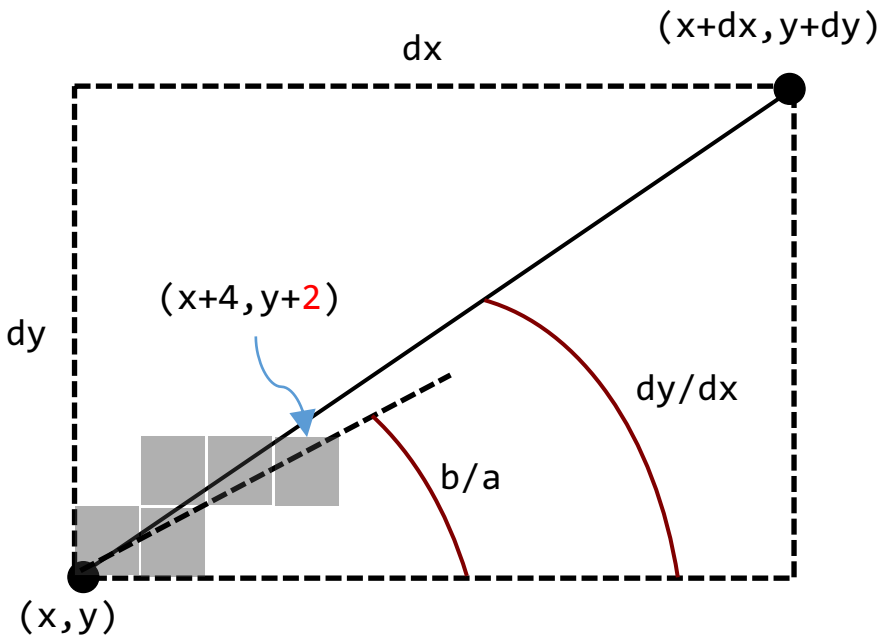
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

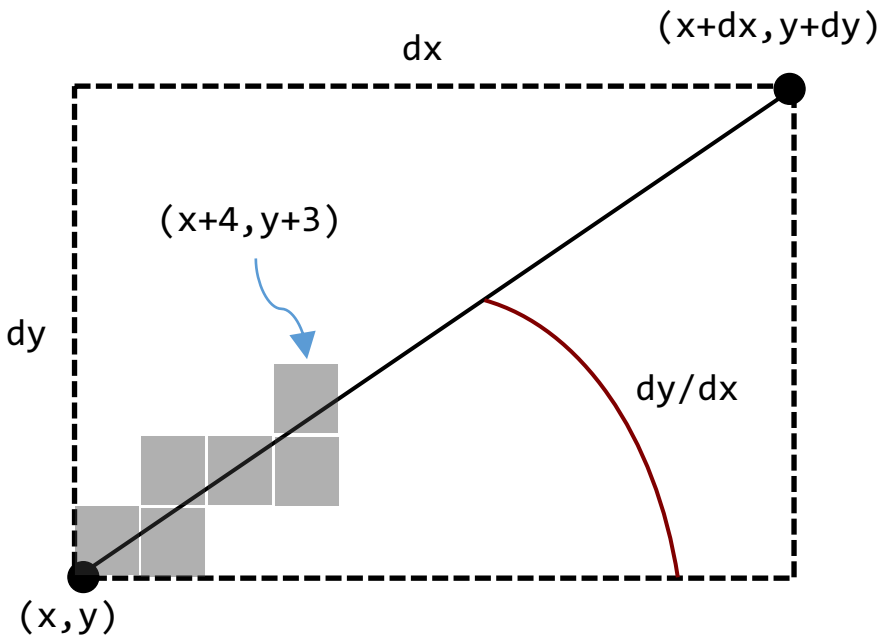
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

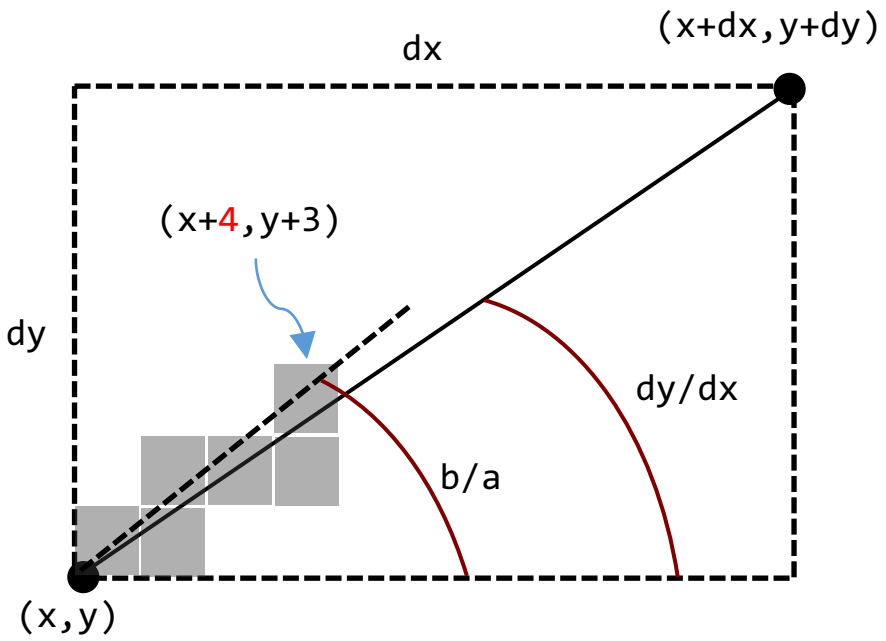
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

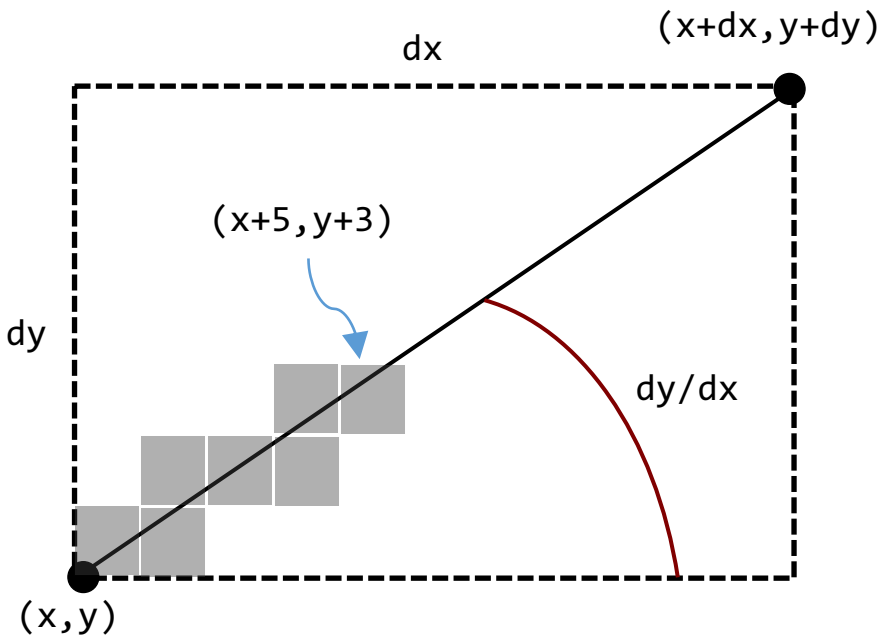
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

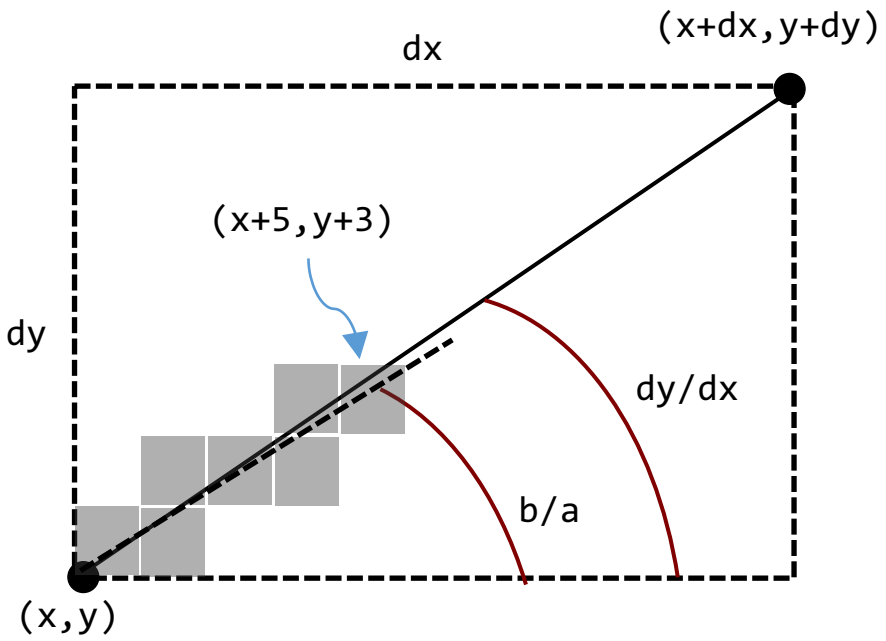
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

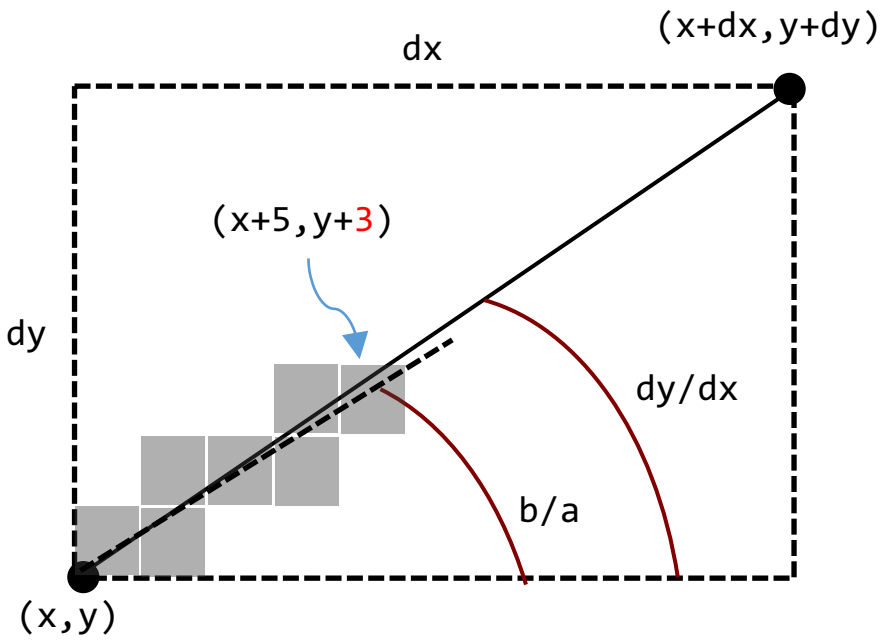
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

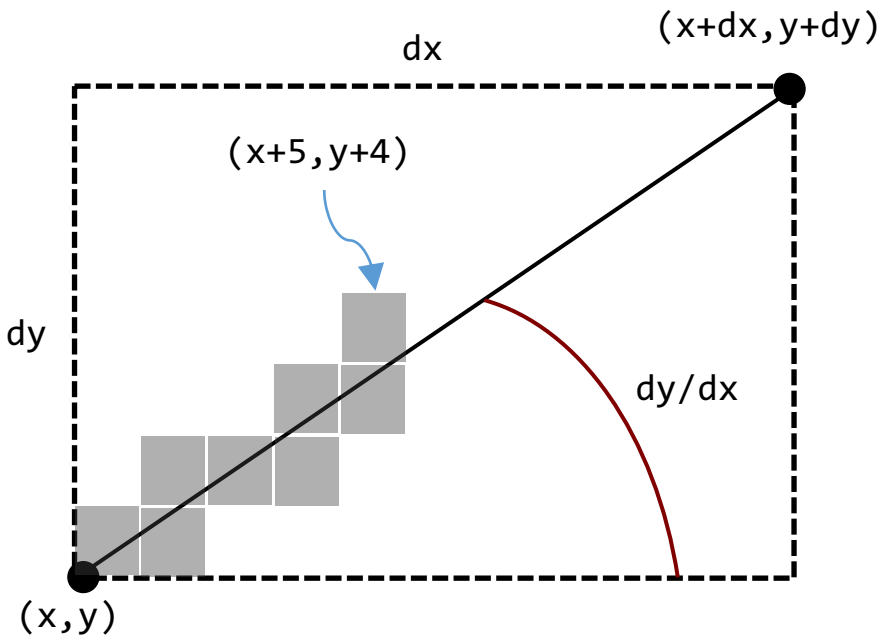
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

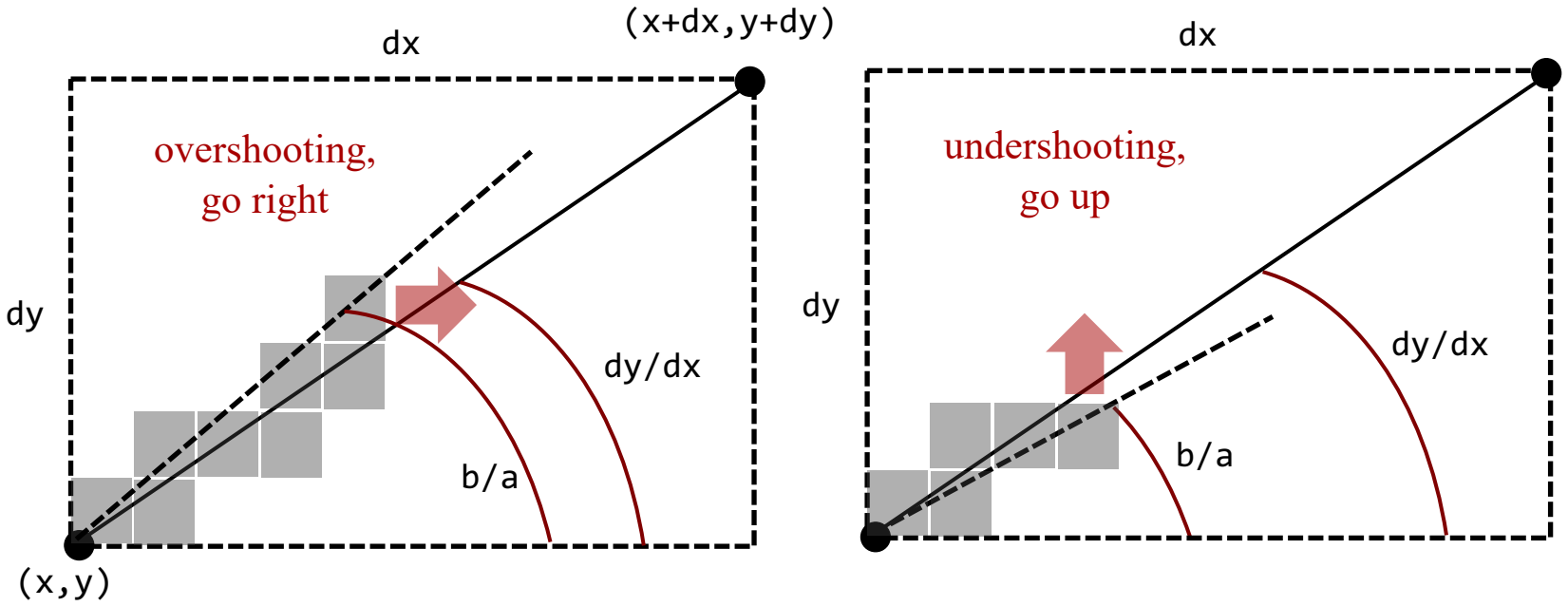
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (b/a > dy/dx) a++
   else             b++
```

a and b record how many times we went up, and right

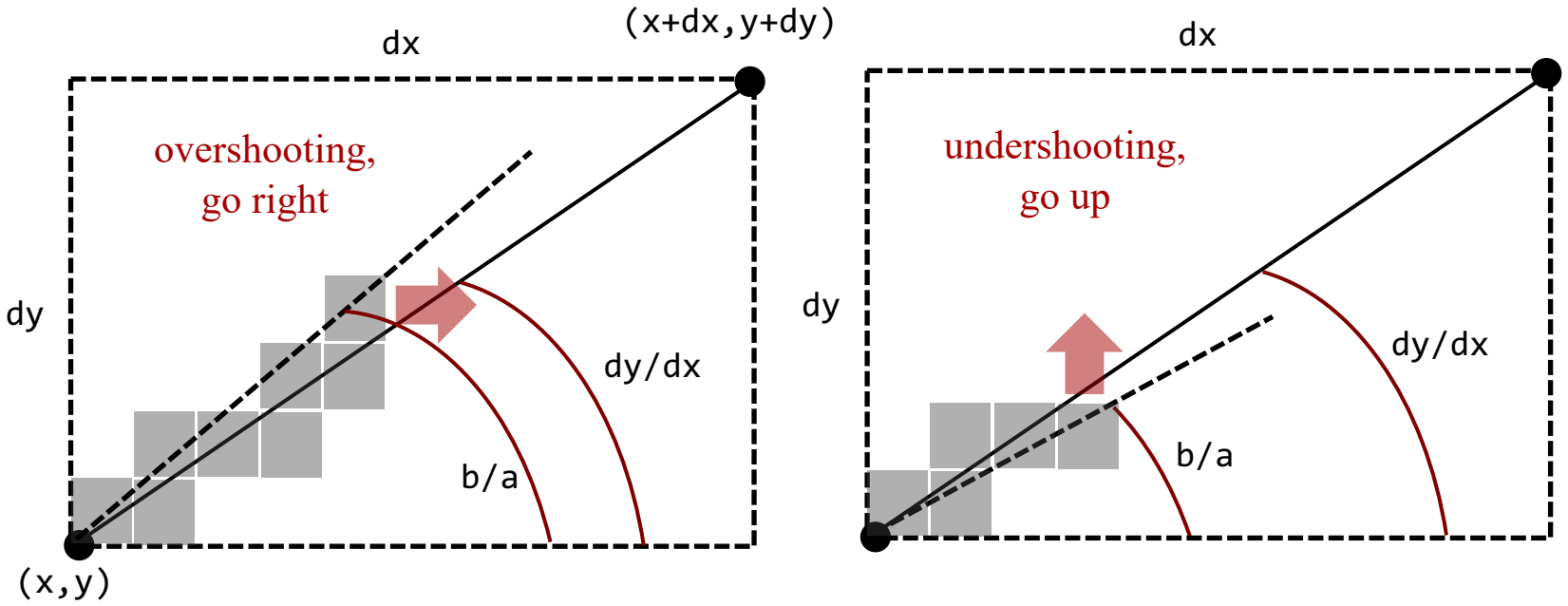
Line drawing




```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
  // decides if to go right, or up:
  if (b/a > dy/dx) a++
   else            b++
```

Let's simplify this calculation

Line drawing



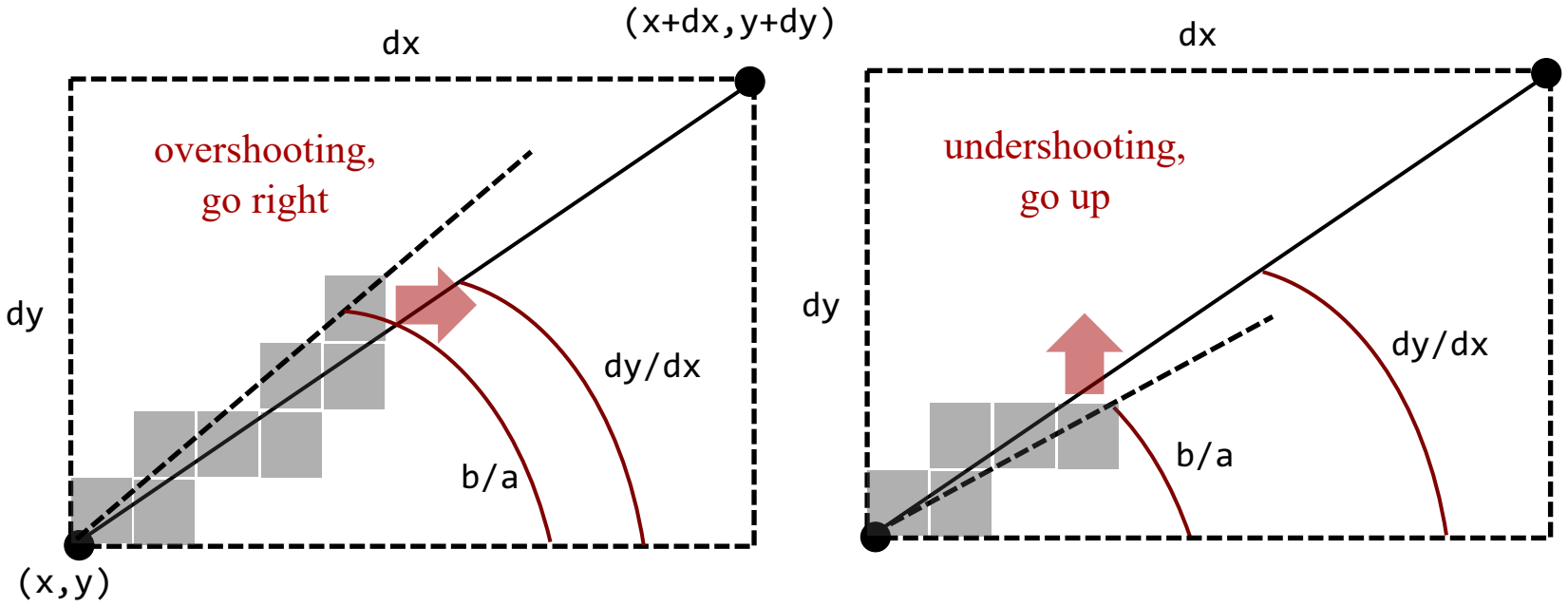

```
a = 0, b = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
  // decides if to go right, or up:
  if (b/a > dy/dx) a++
   else            b++
```

- `b/a > dy/dx` has the same value as `a*dy < b*dx`
- let `diff = a*dy – b*dx`

How do `a++` and `b++` impact `diff`?

- when we do `a++`, `diff` goes up by `dy`
- when we do `b++`, `diff` goes down by `dx`

Line drawing



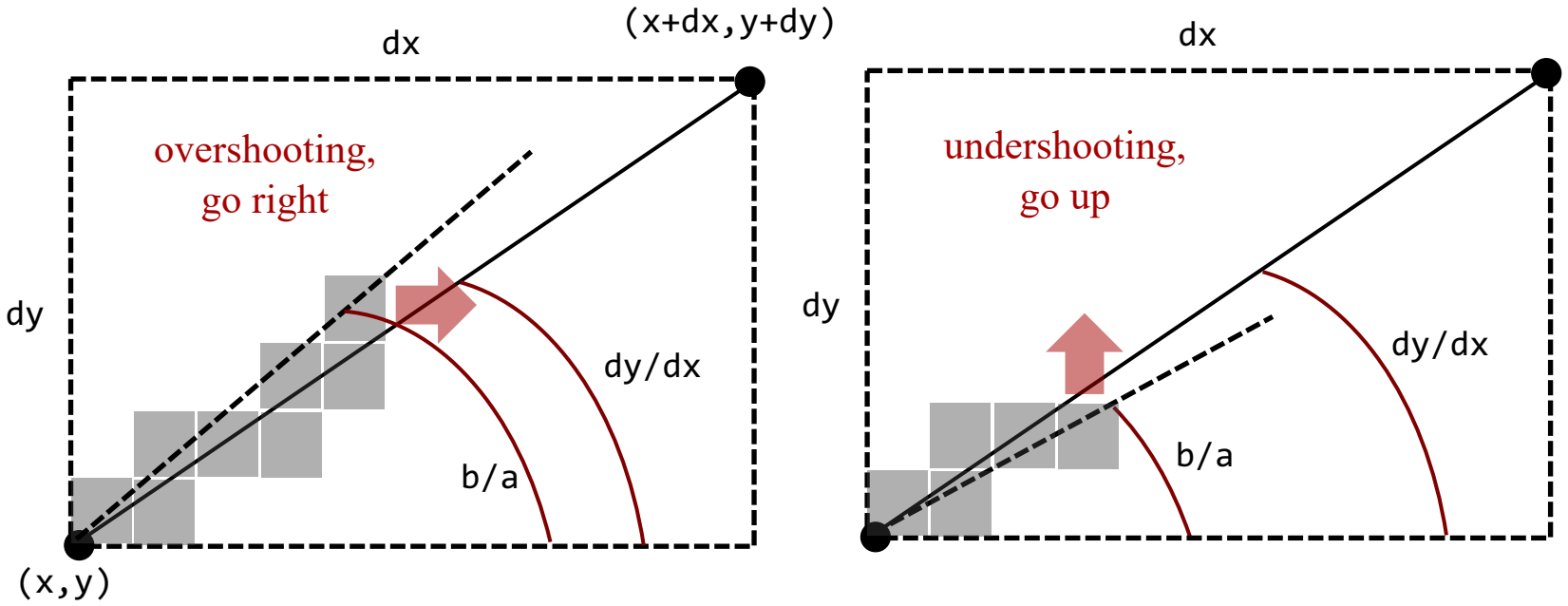

```
a = 0, b = 0, diff = 0
while ((a ≤ dx) and (b ≤ dy))
    drawPixel(x+a, y+b)
    // decides if to go right, or up:
    if (diff < 0) {a++, diff+=dy}
    else          {b++, diff-=dx}
```

- `b/a > dy/dx` has the same value as `a*dy < b*dx`
- let `diff = a*dy – b*dx`

How do `a++` and `b++` impact `diff`?

- when we do `a++`, `diff` goes up by `dy`
- when we do `b++`, `diff` goes down by `dx`

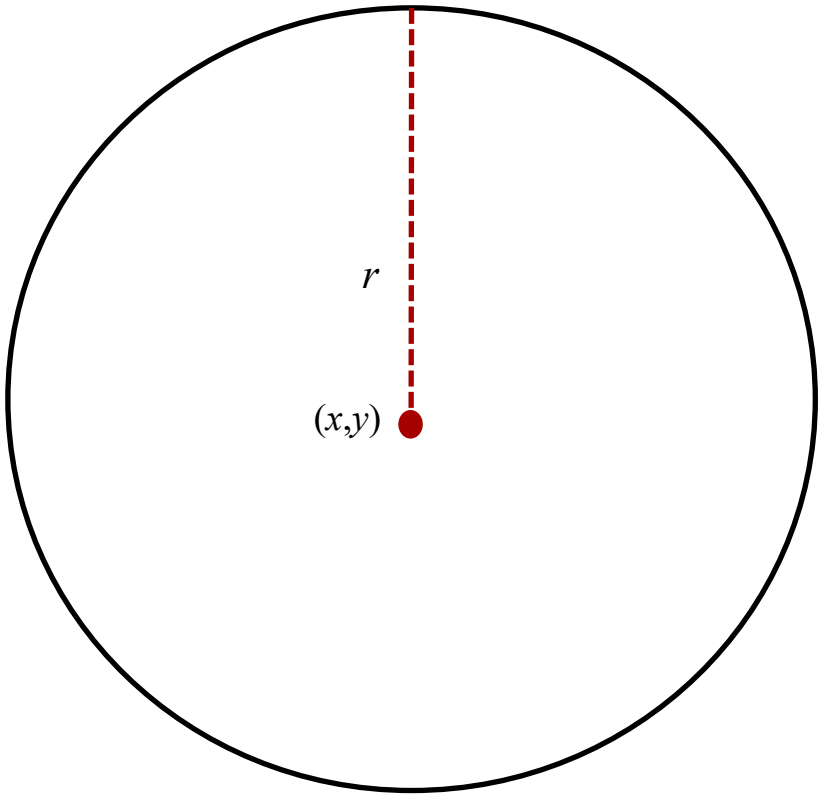
Line drawing



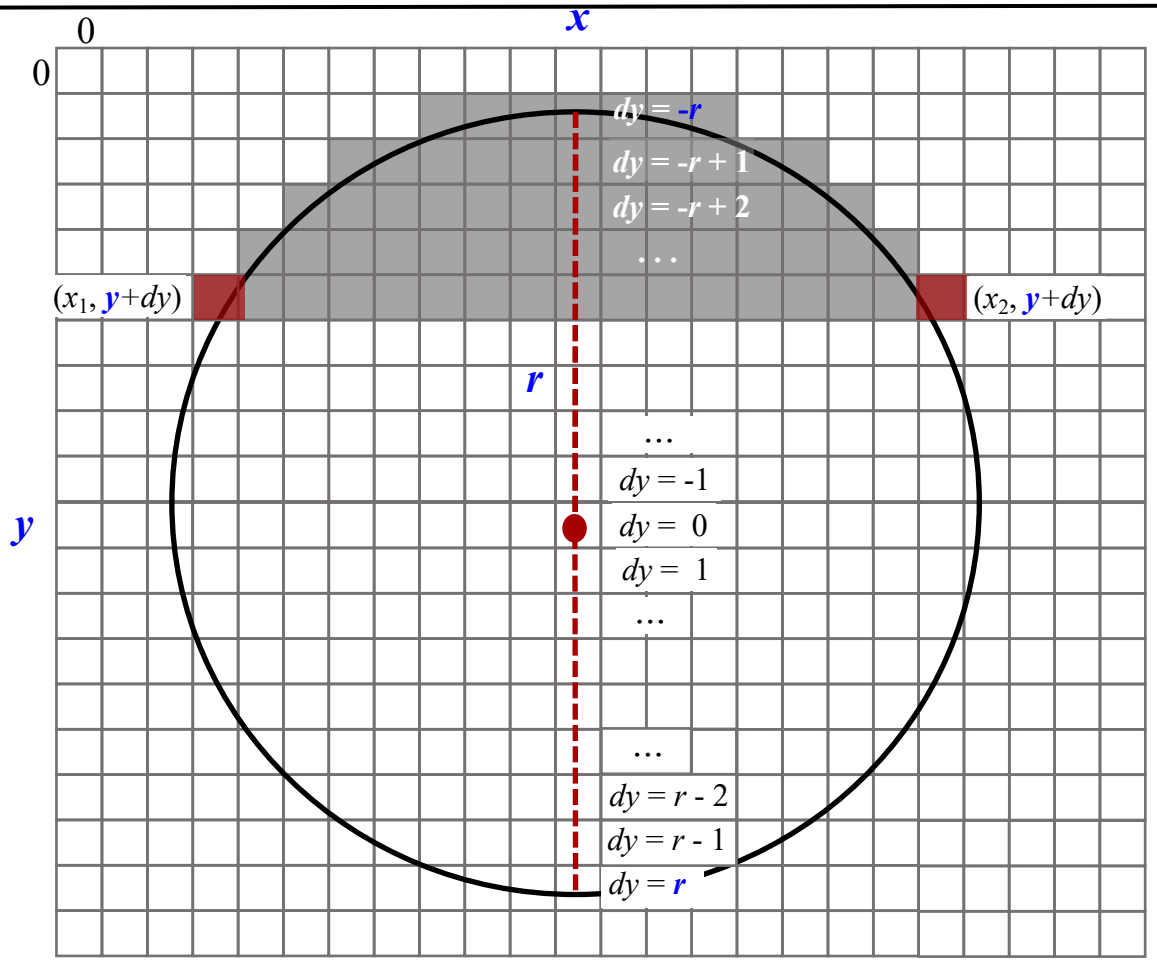

```
a = 0, b = 0, diff = 0
while ((a ≤ dx) and (b ≤ dy))
   drawPixel(x+a, y+b)
   // decides if to go right, or up:
   if (diff < 0) {a++, diff+=dy}
   else          {b++, diff-=dx}
```

- involves only addition and subtraction operations
- can be implemented either in software or hardware.

Circle drawing

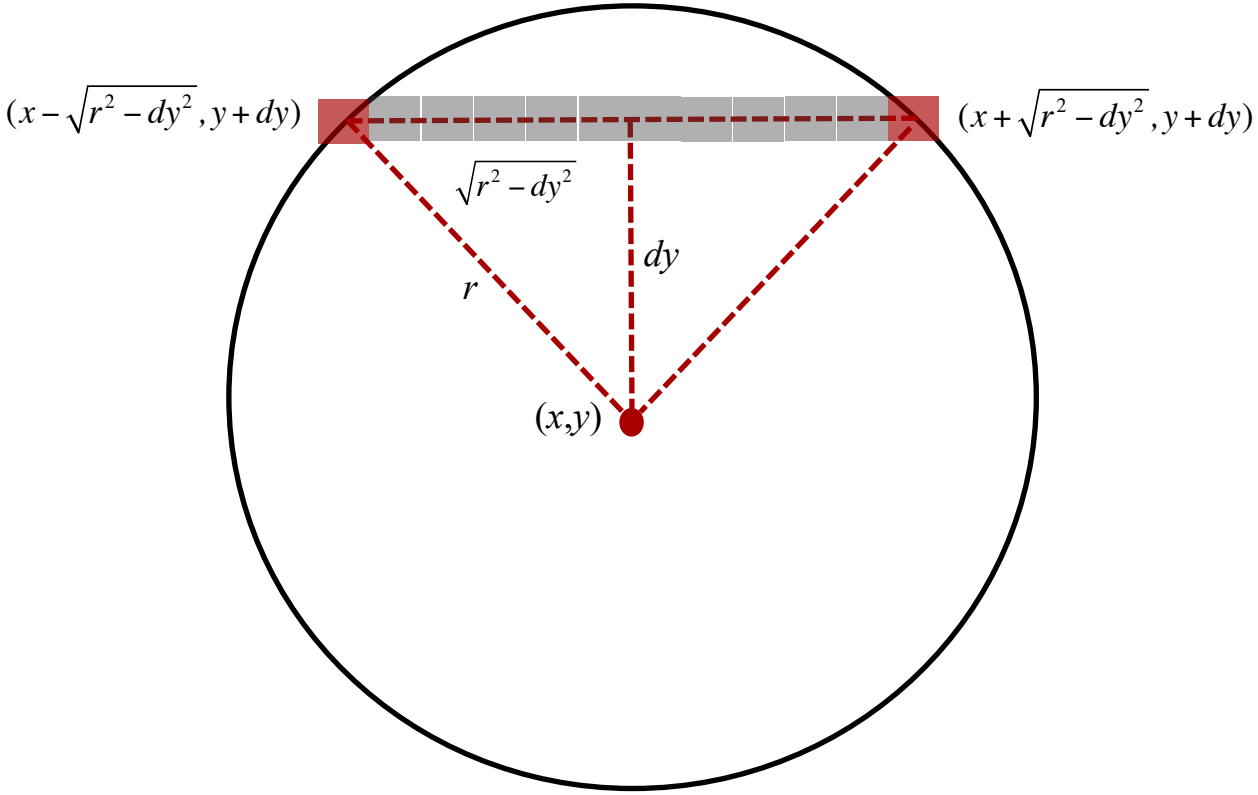


Circle drawing



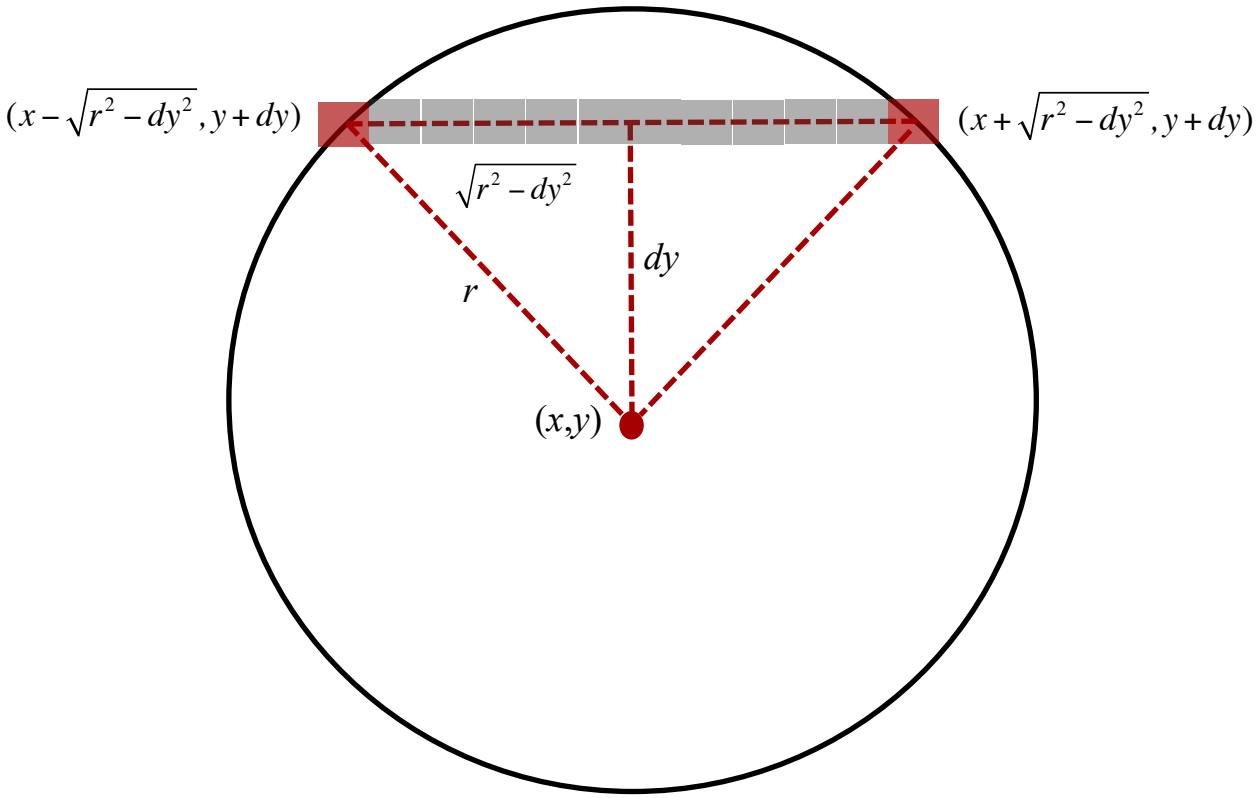

```
drawCircle (x, y, r)
    for each dy = –r  to  r  do:
        drawLine ( x1, y+dy,   x2, y+dy )
```

Circle drawing




```
drawCircle (x, y, r)
    for each dy = –r  to  r  do:
        drawLine ( x1, y+dy,   x2, y+dy )
```

Circle drawing




```
drawCircle (x, y, r)
    for each dy = –r  to  r  do:
        drawLine ( x − √(r² − dy²), y + dy  ,  x + √(r² − dy²), y + dy )
```

Implementation notes: `drawPixel`

```
/* Sets pixel (x,y) to the current color */
function void drawPixel(int x, int y) {
    address = 32 * y + x / 16
    value = Memory.peek[16384 + address]
    set the (x % 16)th bit of value to the current color
    do Memory.poke(address,value)
}
```

- Uses the services of the OS Memory class:
  - `Memory.peek`
  - `Memory.poke`
- Setting a single bit in a 16-bit value can be done using logical 16-bit operations

Implementation notes: `drawLine`

```
/* Draws a line from (x1,y1) to (x2,y2)  */
function void drawLine(int x1, int y1, int x2, int y2)
   dx = x2 - x1; dy = y2 – y1;
   a = 0; b = 0; diff = 0;
   while ((a <= dx) and (b <= dy))
      drawPixel(x + a, y + b);
      // decides which way to go (up, or right):
      if (diff < 0) { a++; diff += dy; }
      else          { b++; diff -= dx; }
```

- Make sure that the screen's top-left corner is pixel (0,0)
- Make sure that the algorithm draws lines that go in any direction
- Handle horizontal and vertical lines as special cases.

Implementation notes: `drawCircle`

/* Draws a filled circle of radius *r* around (*cx*, *cy*) */

function void drawCircle(int *cx*, int *cy*, int *r*) {

for each $dy = -r$ to r do:

drawLine ($cx - \sqrt{r^2 - dy^2}, cy + dy$, $cx + \sqrt{r^2 - dy^2}, cy + dy$)

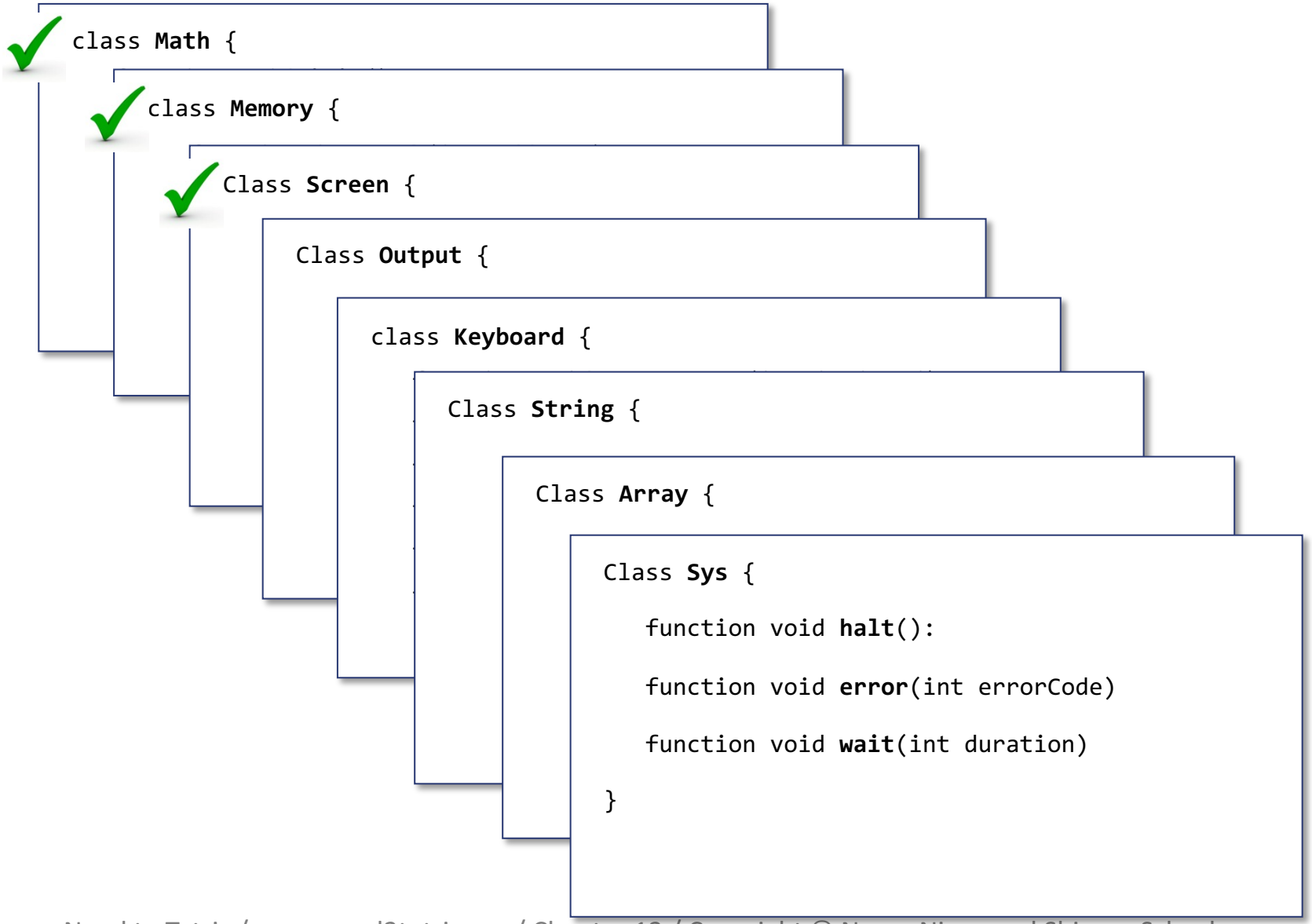
- Can potentially lead to overflow
- To handle, limit *r* to be no greater than 181.

Recap

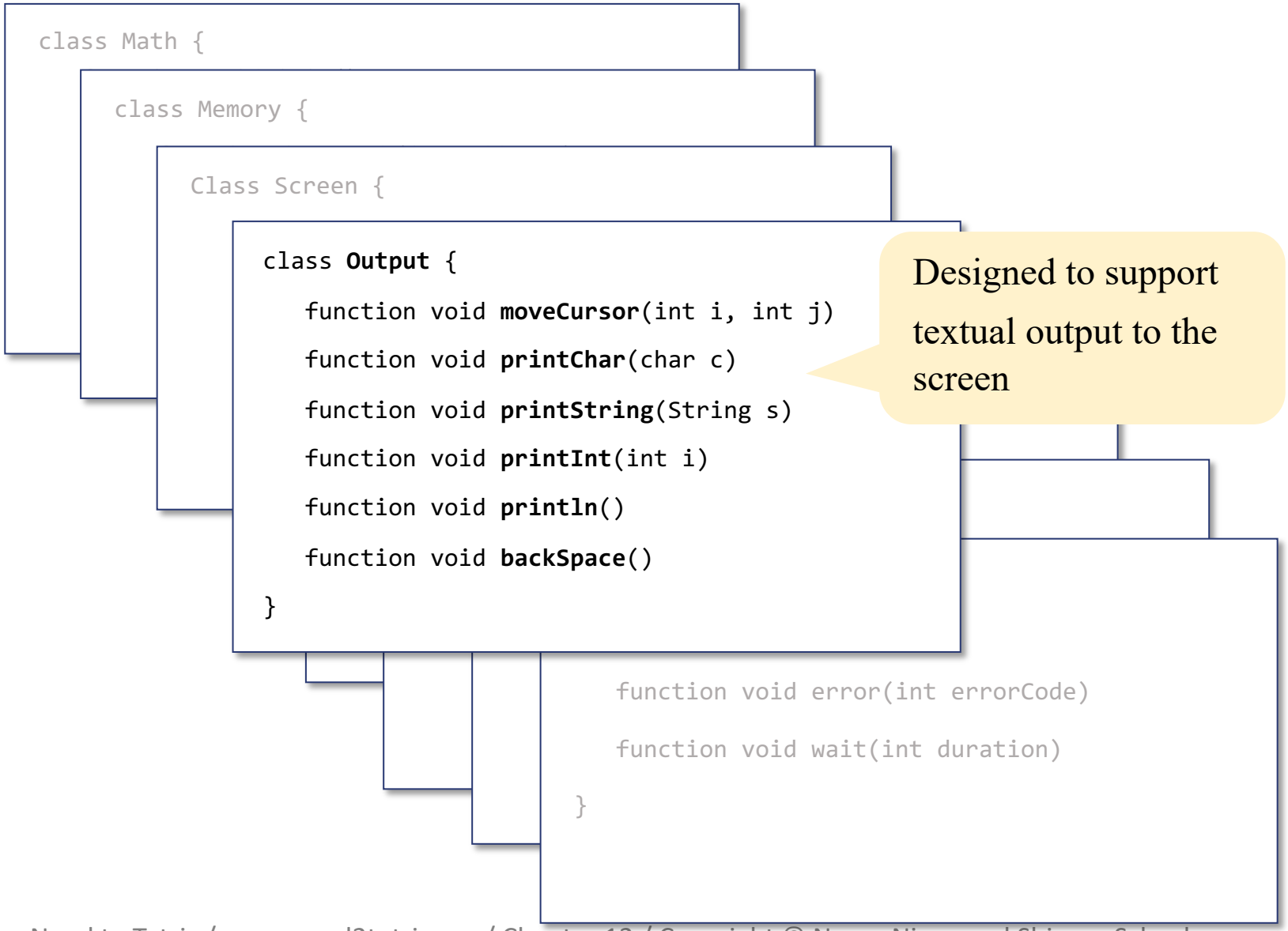
```
Class Screen {
    function void clearScreen()
    function void setColor(boolean b)
    function void drawPixel(int x, int y)
    function void drawLine(int x1, int y1, int x2, int y2)
    function void drawRectangle(int x1, int y1, int x2, int y2)
    function void drawCircle(int x, int y, int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

The implementation of the remaining `Screen` functions is simple.

The Jack OS


class Math {
class Memory {
Class Screen {
Class Output {
class Keyboard {
Class String {
Class Array {
Class Sys {
function void halt():
function void error(int errorCode)
function void wait(int duration)
}

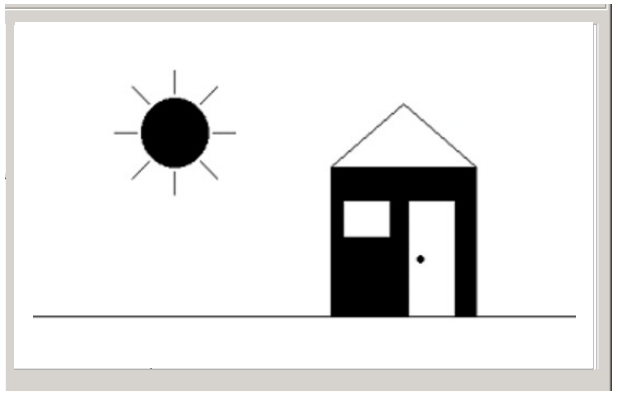
The Jack OS


class Math {
class Memory {
Class Screen {
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()
}
function void error(int errorCode)
function void wait(int duration)
}
Designed to support textual output to the screen

Output modes

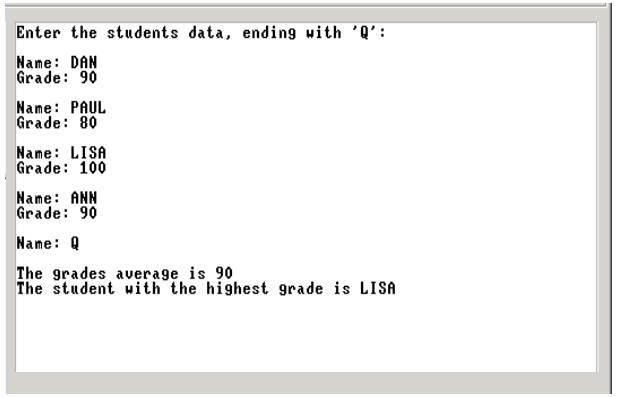
Graphical output

- Screen abstraction:
  256 rows of 512 pixels, b&w
- Driven by the Jack OS class `Screen`



Textual output:

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



The Hack character set

printable characters

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

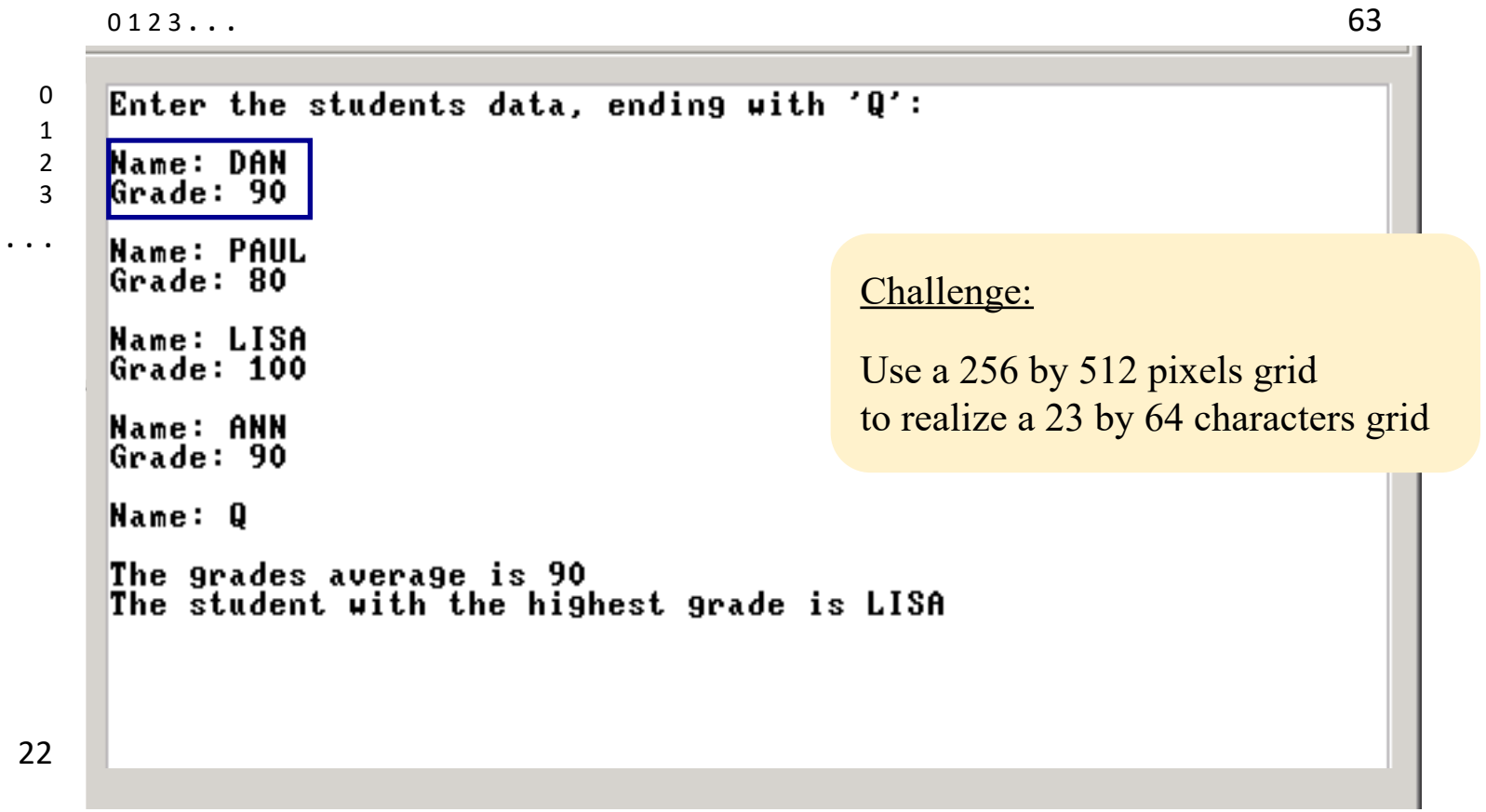
key	code
a	97
b	98
c	99
…	…
z	122

{	123
|	124
}	125
~	126

non-printables

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

Textual output

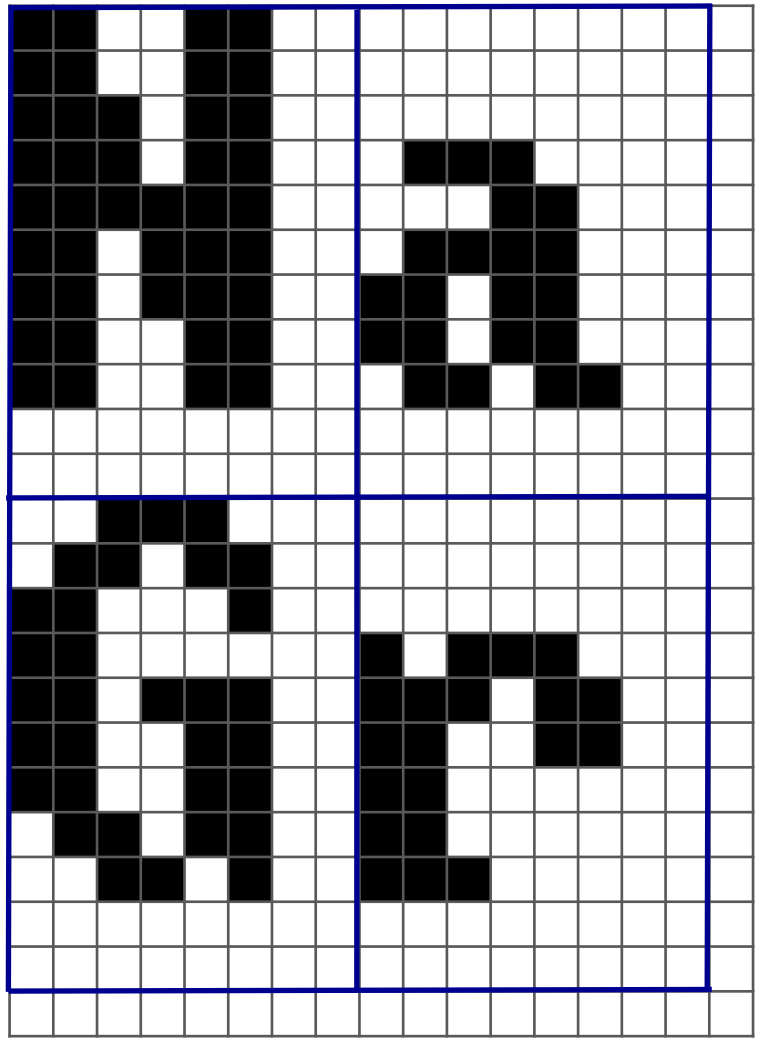


Challenge:

Use a 256 by 512 pixels grid
to realize a 23 by 64 characters grid

Font

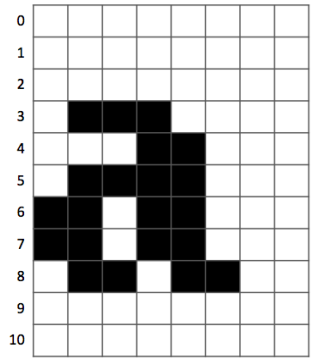
```
Name: DAN
Grade: 90
```



Hack font

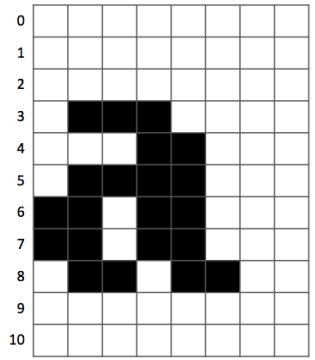
- Each character occupies a fixed 11-pixel high and 8-pixel wide frame
- The frame includes 2 empty right columns and 1 empty bottom row for character spacing

Font implementation



```
class Output {
    static Array charMaps;  // character maps for displaying characters
    ...
    // Builds the character map array
    function void initMap() {
        let charMaps = Array.new(127);

        // Uses a helper function to assign a bitmap for each character in the character set.
        do Output.create(97, 0,0,0,14,24,30,27,27,54,0,0);     // a
        do Output.create(98, 3,3,3,15,27,51,51,51,30,0,0);     // b
        do Output.create(99, 0,0,0,30,51,3,3,51,30,0,0);       // c
        ...
        do Output.create(48, 12,30,51,51,51,51,51,30,12,0,0); // 0
        do Output.create(49, 12,14,15,12,12,12,12,12,63,0,0); // 1
        do Output.create(50, 30,51,48,24,12,6,3,51,63,0,0);    // 2
        do Output.create(51, 30,51,48,48,28,48,48,51,30,0,0); // 3
        ...
        do Output.create(32, 0,0,0,0,0,0,0,0,0,0,0);           // (space)
        do Output.create(33, 12,30,30,30,12,12,0,12,12,0,0);   // !
        do Output.create(34, 54,54,20,0,0,0,0,0,0,0,0);        // "
        do Output.create(35, 0,18,18,63,18,18,63,18,18,0,0);   // #
        ...
        // black square (used for non printable characters)
        do Output.create(0,  63,63,63,63,63,63,63,63,63,0,0);
        return
}
```

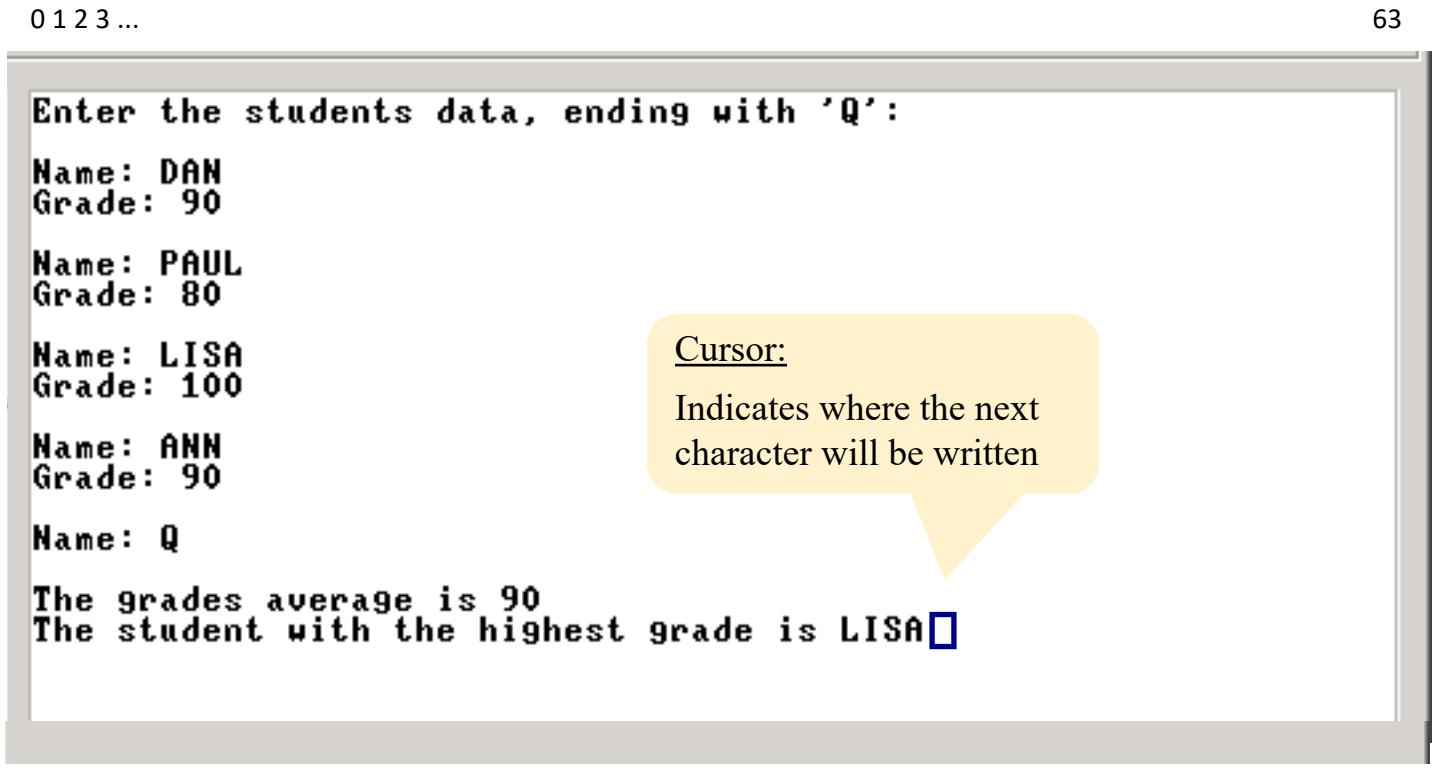
Font implementation



```
class Output {
    static Array charMaps;  // character maps for displaying characters
    ...
    // Builds the character map array
    function void initMap() {
        let charMaps = Array.new(127);

        // Uses a helper function to assign a bitmap for each character in the character set.
        do Output.create(97, 0,0,0,14,24,30,27,27,54,0,0);     // a
        do Output.create(98, 3,3,3,15,27,51,51,51,30,0,0);     // b
        do Output.create(99, 0,0,0,30,51,3,3,51,30,0,0);       // c
        ...
        do Output.create(48, 12,30,51,51,51,51,51,30,12,0,0); // 0
        do Output.create(49, 12,14,15,12,12,12,12,12,63,0,0); // 1
        do Output.create(50, 30,51,48,24,12,6,3,51,63,0,0);    // 2
        do Output.create(51, 30,51,48,48,28,48,48,51,30,0,0); // 3
        ...
        do Out
        do Out
        do Out
        do Out
        ...
        // black
        do Out
        return
}
```

```
// Creates a bitmap for the given char index, using the given values.
function void create(int index, int a, int b, int c, int d, int e,
                                 int f, int g, int h, int i, int j, int k) {
    var Array map;
    let map = Array.new(11);
    let charMaps[index] = map;
    let map[0] = a;  let map[1] = b;  let map[2] = c;
    let map[3] = d;  let map[4] = e;  let map[5] = f;
    let map[6] = g;  let map[7] = h;  let map[8] = i;
    let map[9] = j;  let map[10] = k;
    return;
}
```

Cursor



Cursor management:

- If asked to display `newLine`: Moves the cursor to the beginning of the next line
- If asked to display `backspace`: Moves the cursor one column left
- If asked to display any other character: Displays the character, and moves the cursor one column to the right.

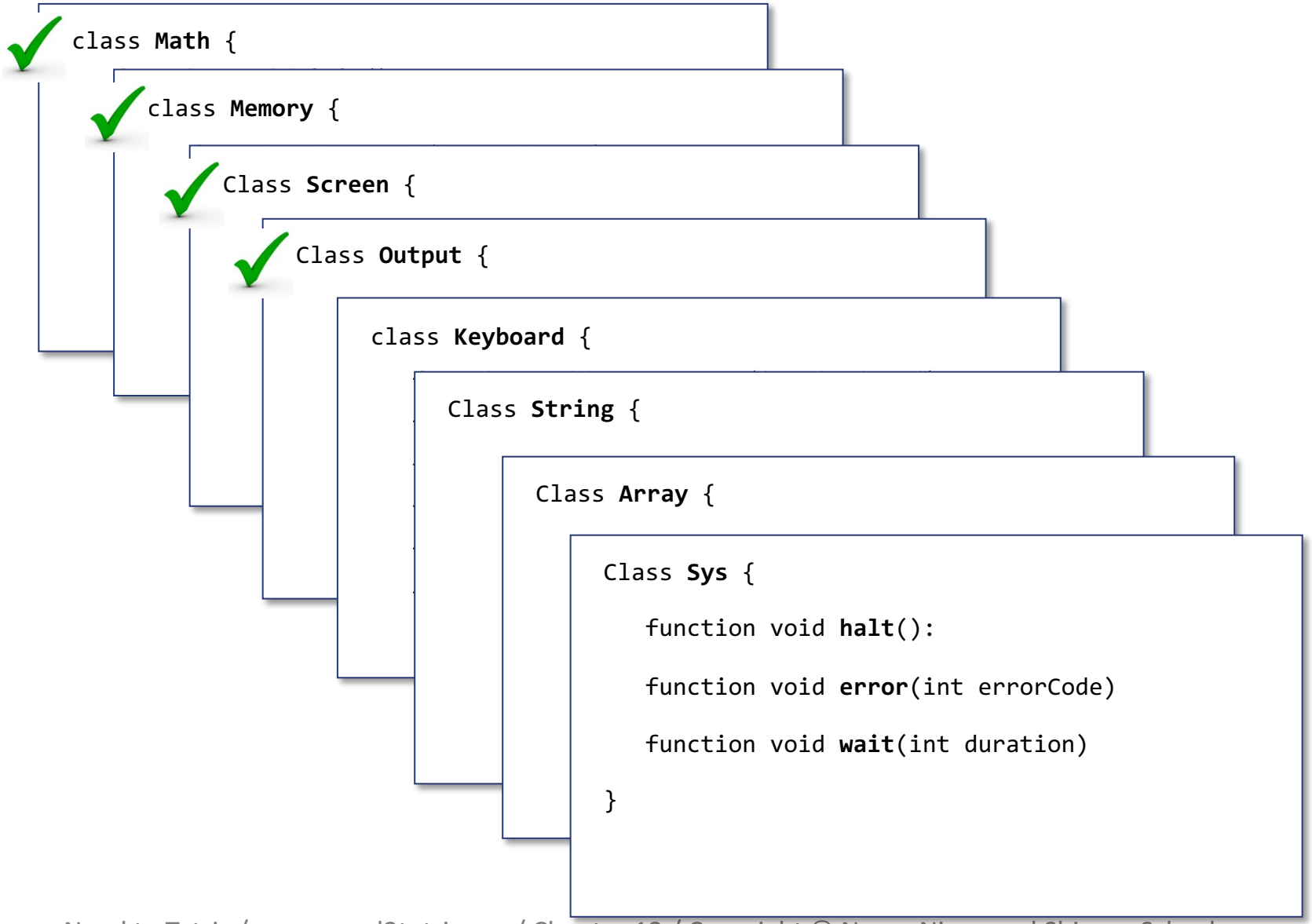
Implementation notes

`Output` class API

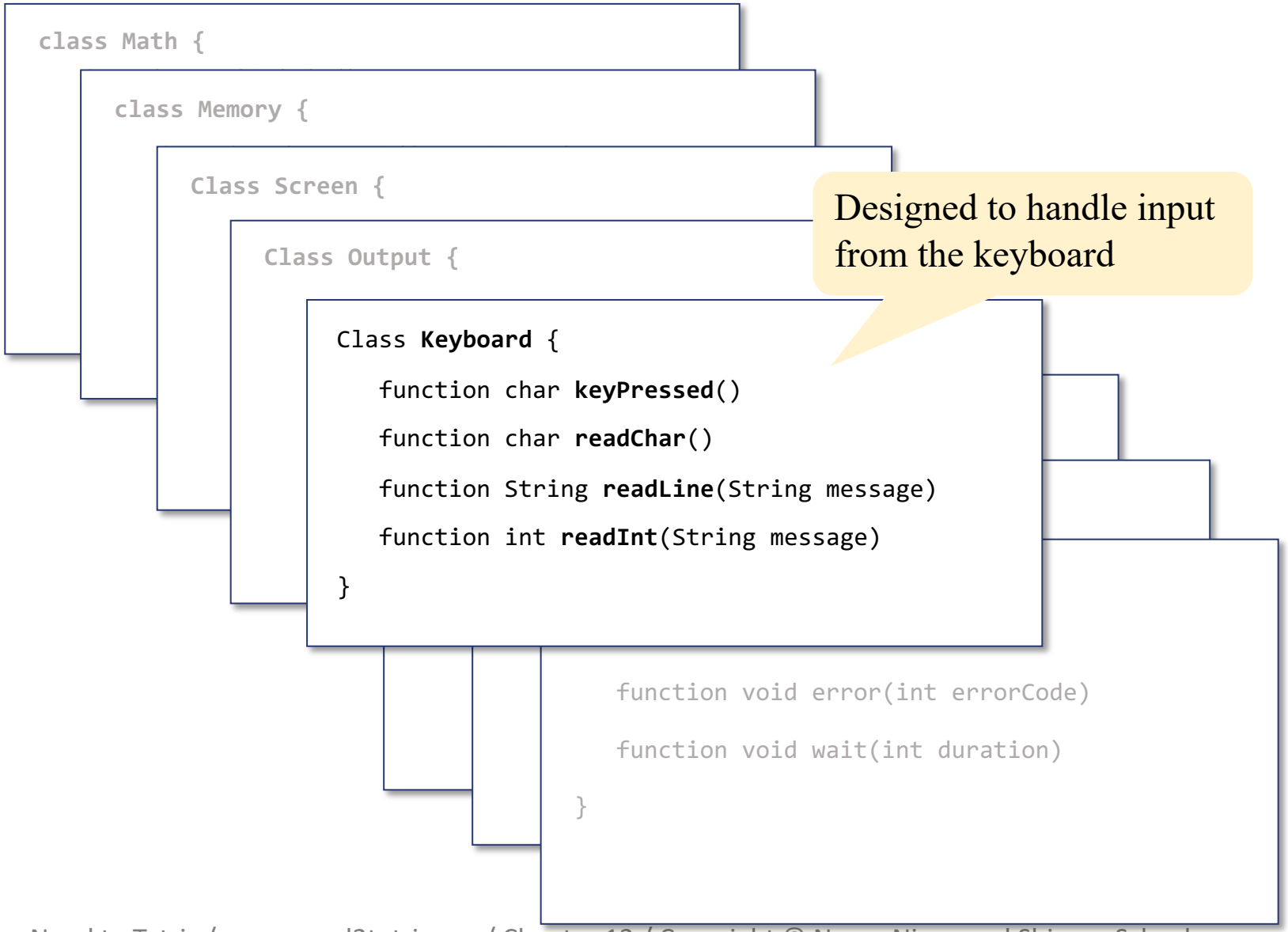
```
class Output {

  function void init() {}

  /* Moves the cursor to the j'th column of the i'th row, erasing the character that was there. */
  function void moveCursor(int i, int j) {}

  /* Displays c at the cursor location, and advances the cursor one column right. */
  function void printChar(char c) {}

  /* Displays str starting at the cursor location, and advances the cursor appropriately. */
  function void printString(String str) {}

  /* Displays i starting at the cursor location, and advances the cursor appropriately. */
  function void printInt(int i) {}

  /* Advances the cursor to the beginning of the next line. */
  function void println() {}

  /* Erases the character that was last written and moves the cursor one column back. */
  function void backSpace() {}

}
```

- Implementing the Hack font: Described / given in previous slides
- Given that the font is taken care of, the rest of the implementation is simple.

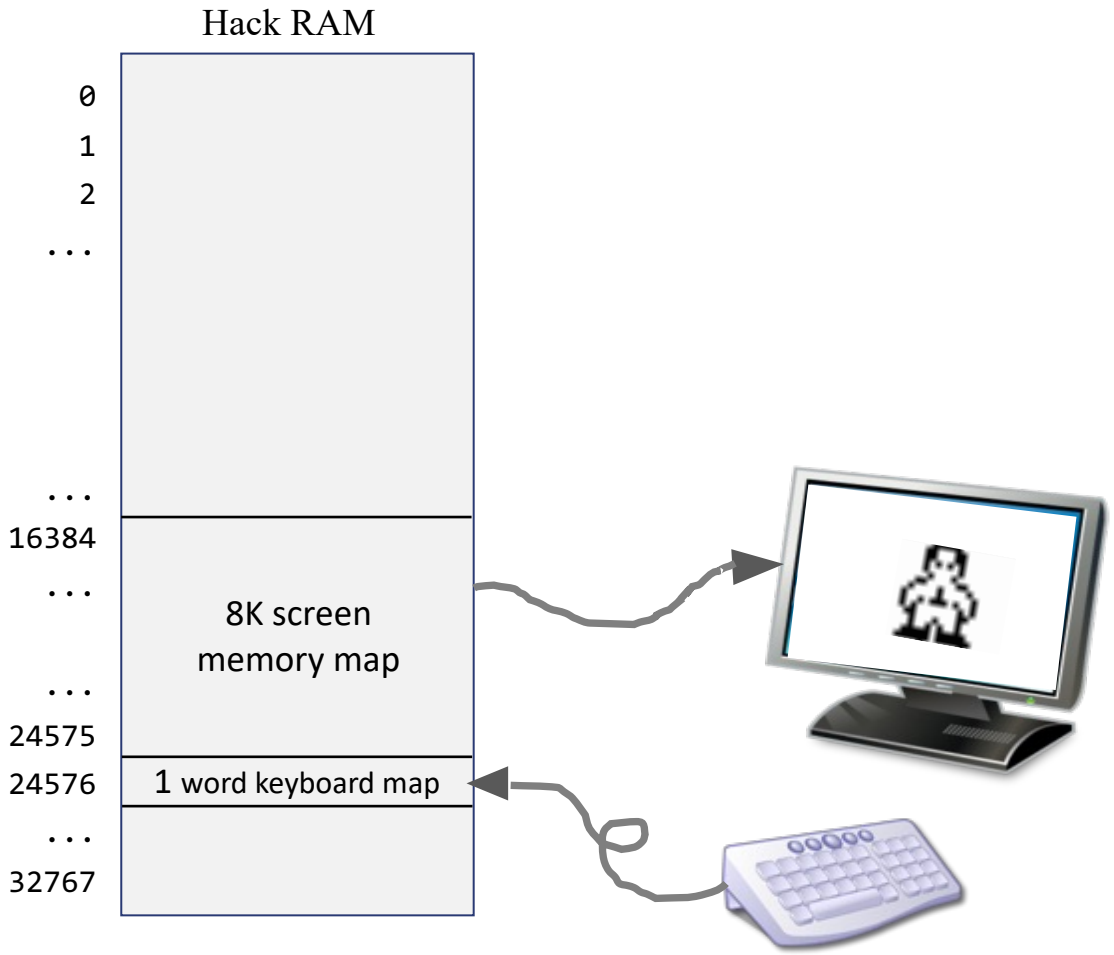
The Jack OS


class Math {
class Memory {
Class Screen {
Class Output {
class Keyboard {
Class String {
Class Array {
Class Sys {
function void halt():
function void error(int errorCode)
function void wait(int duration)
}

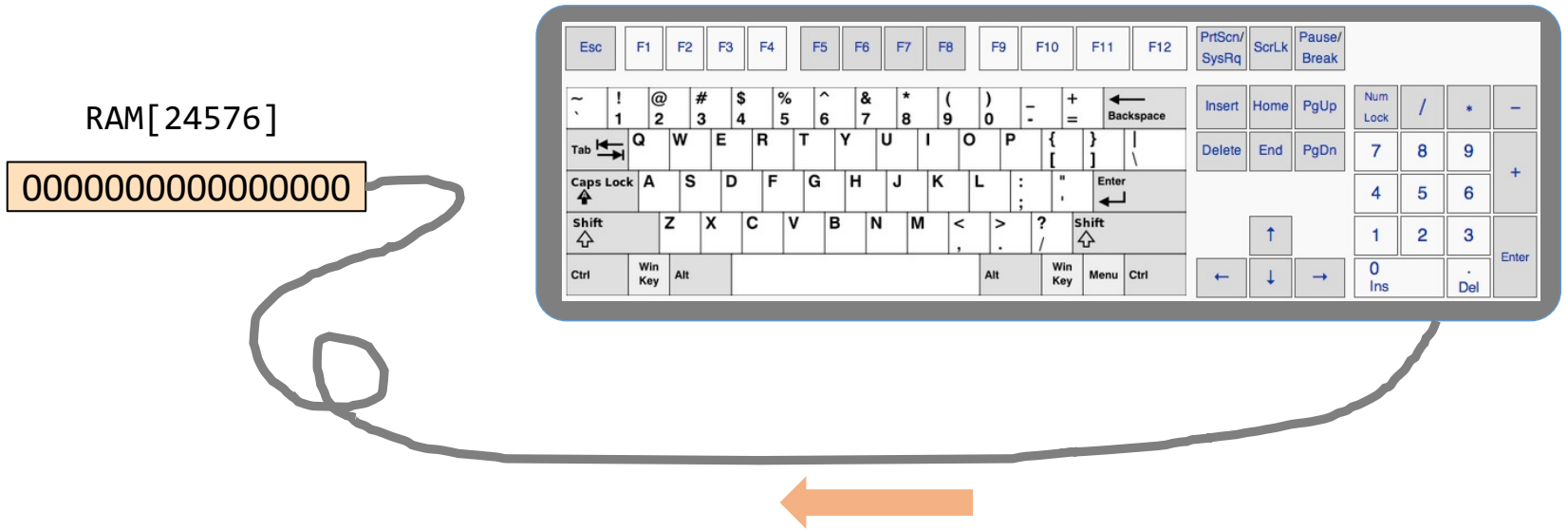
The Jack OS


class Math {
class Memory {
Class Screen {
Class Output {
Class Keyboard {
function char keyPressed()
function char readChar()
function String readLine(String message)
function int readInt(String message)
}
function void error(int errorCode)
function void wait(int duration)
}
Designed to handle input from the keyboard

Hack I/O


Hack RAM
0
1
2
...
...
16384
...
...
24575
24576
...
32767
8K screen
memory map
1 word keyboard map

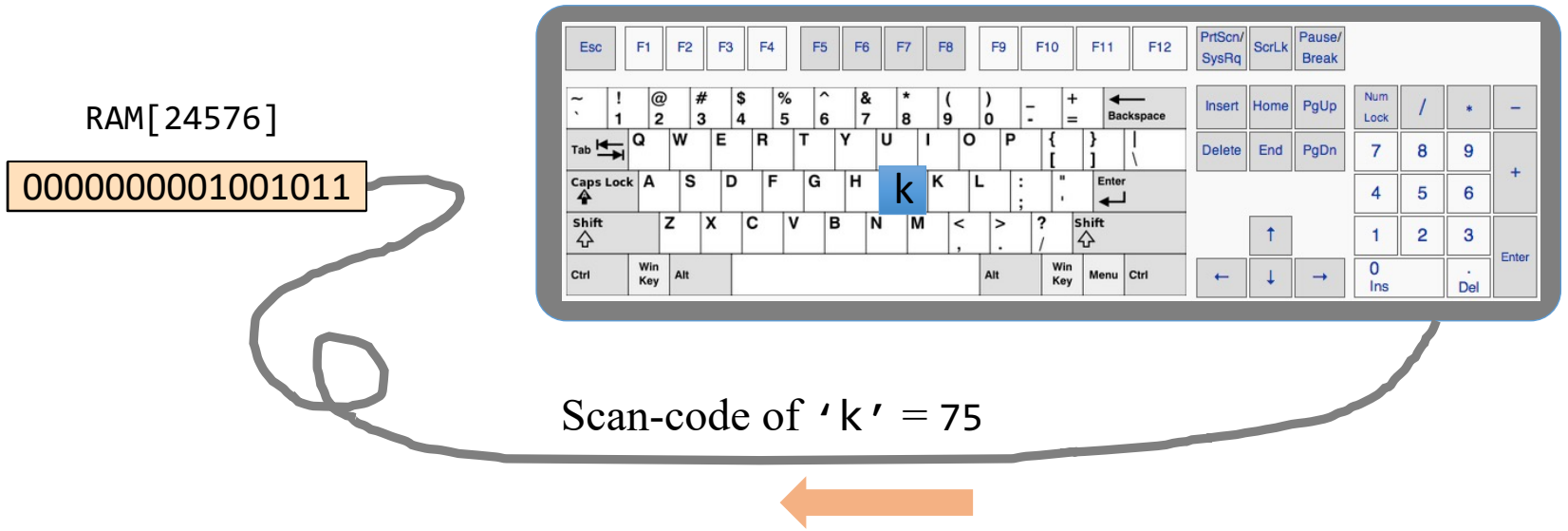
Keyboard memory map



Keyboard memory map

- When a key is pressed on the keyboard,
  the keyboard register is set to the key's *character code*
- When no key is pressed, the keyboard register is set to `0`

Keyboard memory map



Keyboard memory map

- When a key is pressed on the keyboard,
  the keyboard register is set to the key's *character code*
- When no key is pressed, the keyboard register is set to `0`

The Hack character set

printable characters

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

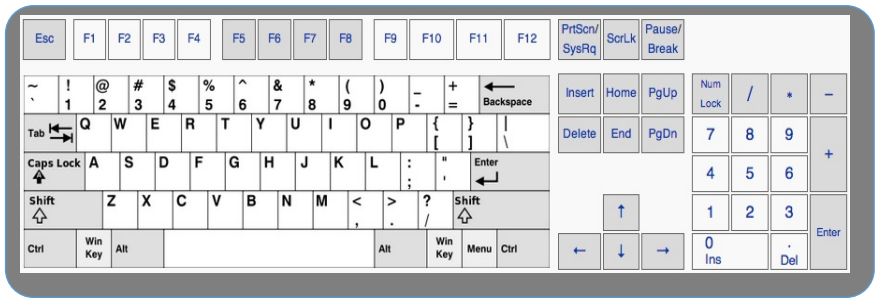
non-printables

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

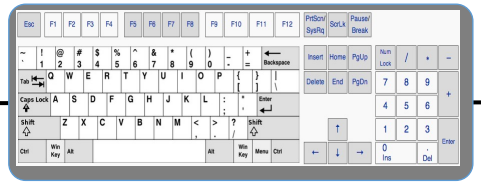
OS Keyboard class

Keyboard class API

```
class Keyboard {

    /* Returns the character code of the currently pressed key,
       or 0 if no key is currently pressed.*/
    function char keyPressed() {}

    /* Reads the next character from the keyboard:
       waits until a key is pressed and then released, then echoes
       the key to the screen, and returns the code of the pressed key.*/
    function char readChar() {}

    /* Prints the message on the screen, reads the next line (until a newLine
       character) from the keyboard, and returns its value. */
    function String readLine(String message) {}

    /* Prints the message on the screen, reads the next line
       (until a newline character) from the keyboard, and returns its
       integer value (until the first non numeric character). */
    function int readInt(String message) {}
}
```



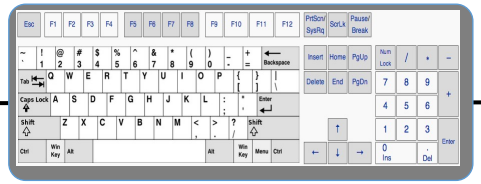
keyPressed



listens to the keyboard

```
keyPressed():
   if a keyboard key is pressed:
        return the key's character code
   else
        return 0
```

keyPressed / readChar



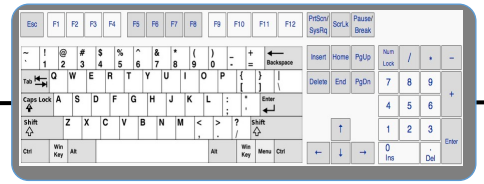
listens to the keyboard

```
keyPressed():
  if a keyboard key is pressed:
      return the key's character code
  else
      return 0
```

gets a character

```
/* Waits until a key is pressed and released;
echoes the key on the screen, advances the cursor,
and returns the key's character. */
readChar():
  display the cursor
  // waits until a key is pressed
  while (keyPressed() == 0):
      do nothing
  c = code of the currently pressed key
  // waits until the key is released
  while (keyPressed() ≠ 0):
      do nothing
  display c at the current cursor location
  advance the cursor
  return c
```

keyPressed / readChar / readLine



listens to the keyboard

```
keyPressed():
  if a keyboard key is pressed:
      return the key's character code
  else
      return 0
```

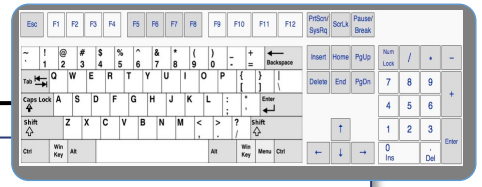
gets a character

```
/* Waits until a key is pressed and released;
echoes the key on the screen, advances the cursor,
and returns the key's character. */
readChar():
  display the cursor
  // waits until a key is pressed
  while (keyPressed() == 0):
      do nothing
  c = code of the currently pressed key
  // waits until the key is released
  while (keyPressed() ≠ 0):
      do nothing
  display c at the current cursor location
  advance the cursor
  return c
```

gets a string

```
/* Displays the message on the screen, gets the next
line (until a newLine character) from the keyboard,
and returns its value, as a string. */
readLine():
  str = empty string
  repeat
      c = readChar()
      if (c == newLine):
          display newLine
          return str
      else if (c = backSpace):
              remove the last character from str
              do Output.backspace()
          else
              str = str.append(c)
  return str
```

Implementation notes



```
class Keyboard {

    /* Returns the character of the currently pressed key,
        or 0 if no key is currently pressed.*/
    function char keyPressed() {}

    /* Reads the next character from the keyboard:
       waits until a key is pressed and then released, then echoes
       the key to the screen, and returns the value of the pressed key.*/
    function char readChar() {}

    /* Prints the message on the screen, reads the next line (until a newLine
       character) from the keyboard, and returns its value. */
    function String readLine(String message) {}

    /* Prints the message on the screen, reads the next line
       (until a newline character) from the keyboard, and returns its
       integer value (until the first non numeric character). */
    function int readInt(String message) {}
}
```

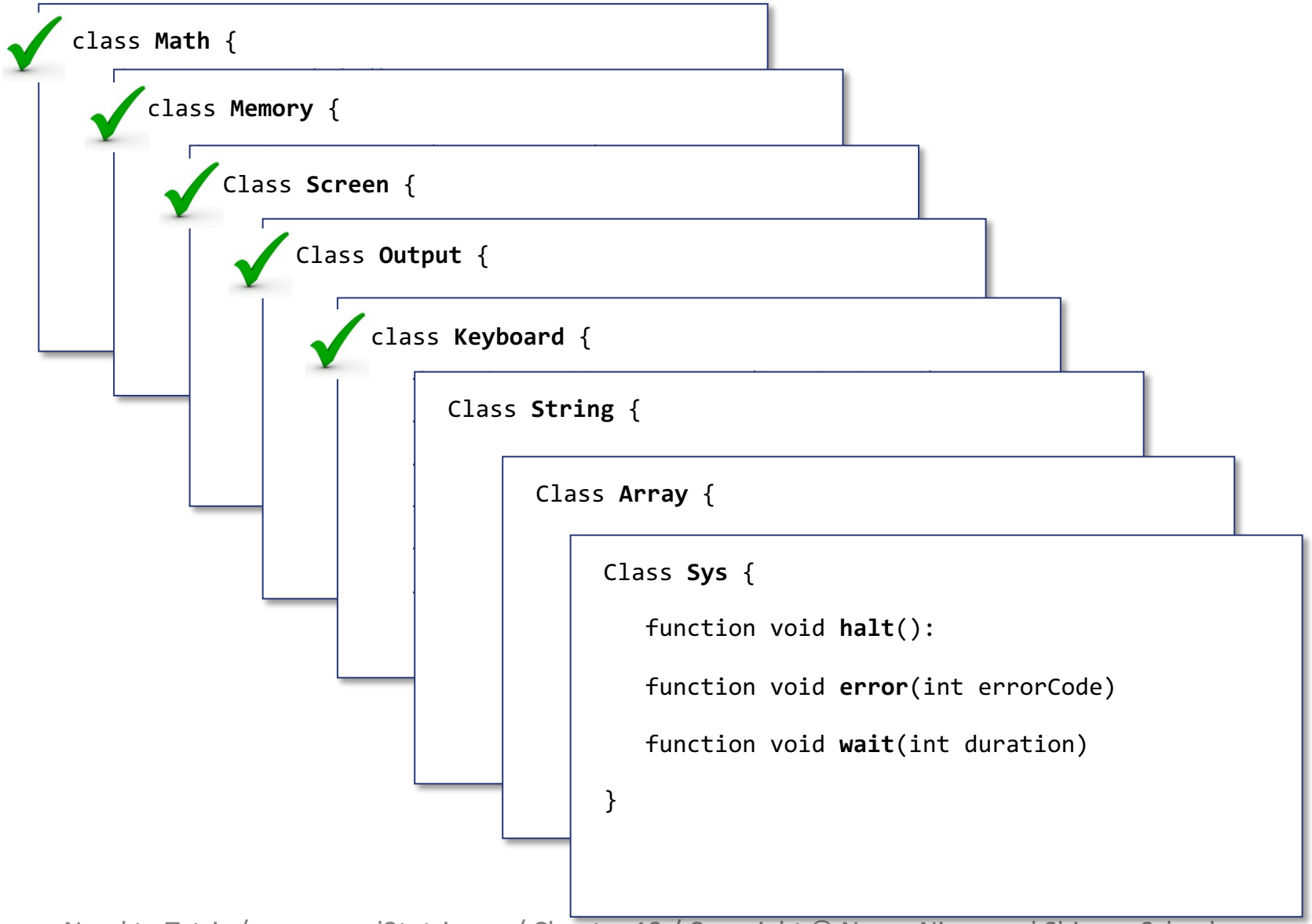
Use `Memory.peek`

Implement the algorithm

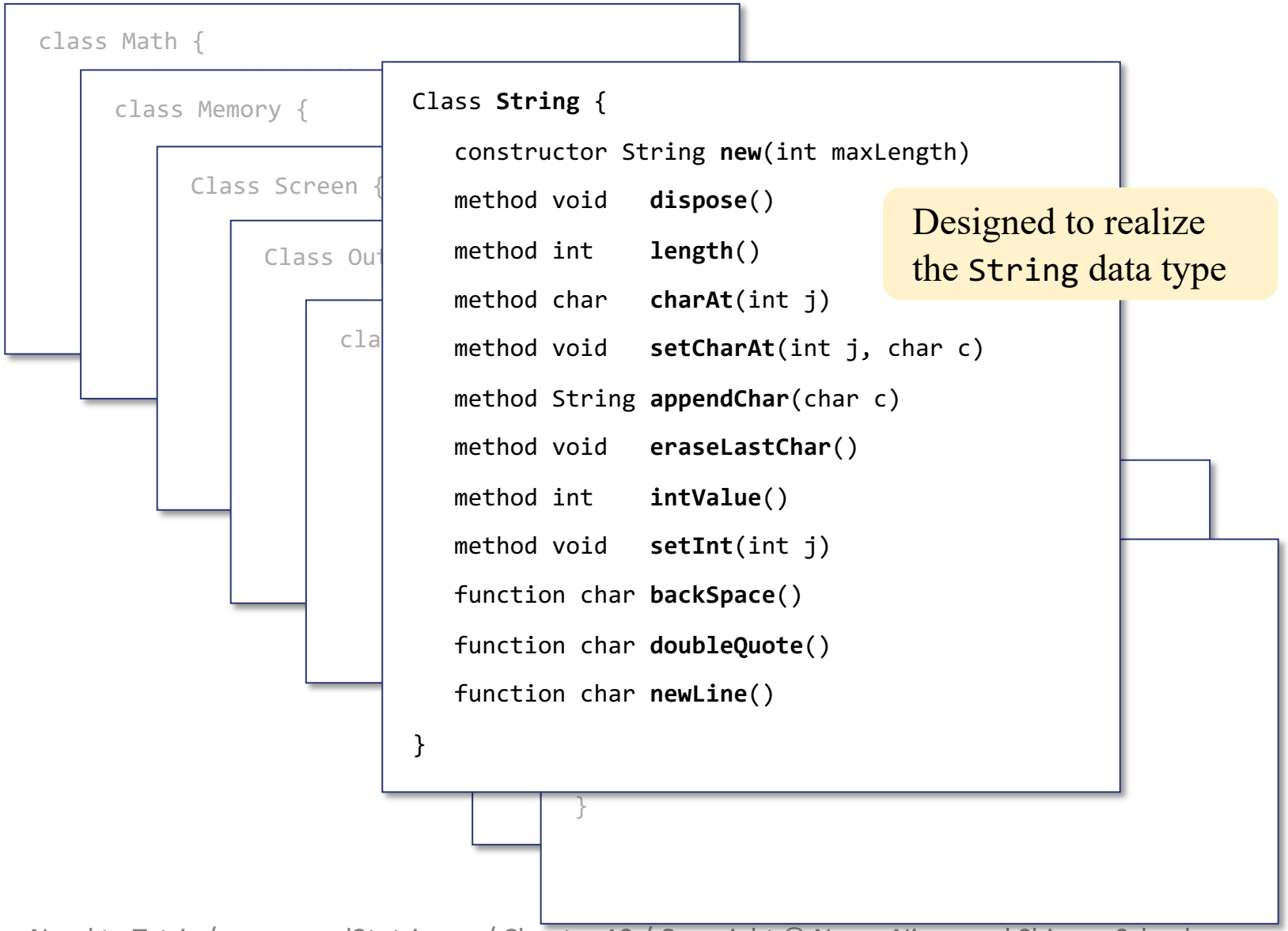
Implement the algorithm

Read characters (digits) and build the integer value

The Jack OS


class Math {
class Memory {
Class Screen {
Class Output {
class Keyboard {
Class String {
Class Array {
Class Sys {
function void halt():
function void error(int errorCode)
function void wait(int duration)
}

The Jack OS


class Math {
class Memory {
Class Screen {
Class Out
cla
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()
}
}
Designed to realize the String data type

`String` class API

```
/* Implements the String type. */
class String {

    /* Constructs a new empty string with a maximum length. */
    constructor String new(int maxLength) {}

    /* De-allocates the string and frees its memory space. */
    method void dispose() {}

    /* Returns the length of this string. */
    method int length() {}

    /* Returns the character value at the specified index */
    method char charAt(int j) {}

    /* Sets the j'th character of this string to the given character. */
    method void setCharAt(int j, char c) {}

    /* Appends the given character to the end of this string, and returns the string. */
    method String appendChar(char c) {}

    /* Erases the last character from this string. */
    method void eraseLastChar() {}

    /* Returns the integer value of this string until the first non-numeric character. */
    method int intValue() {}

    /* Sets this String to the representation of the given number. */
    method void setInt(int number) {}

    /* Returns the new line character. */
    function char newLine() {}

    /* Returns the backspace character. */
    function char backSpace() {}

    /* Returns the double quote (") character. */
    function char doubleQuote() {}
}
```

The client's view

Typical string usage (in some Jack class):

```
...
var String s;                // declares a String variable
var int x;
...
let s = String.new(6);       // constructs a string with a
                             // maximum capacity of 6 characters
...
let s = s.appendChar(97);    // s = "a"
let s = s.appendChar(98);    // s = "ab"
let s = s.appendChar(99);    // s = "abc"
let s = s.appendChar(100);   // s = "abcd"
let x = s.length();          // x = 4 (actual length)
...
do s.setInt(314);            // s = "314"
...
let x = 2 * s.intValue();    // x = 628
...
```

int to string

string to int

int ⟷ string algorithms

int to string:

```
/* Returns the string representation of
   a non-negative integer */
int2String (val) :
    lastDigit = val % 10
    c = character representing lastDigit
    if (val < 10)
        return  c  (as a string)
    else
        return  int2String (val / 10).append(c)
```

string to int:

```
/* Returns the integer value of a string
   of digit characters, assuming that str[0]
   represents the most significant digit. */
string2Int (str) :
    val = 0
    for ( i = 0 ... str.length ) do
        d = integer value of str[i]
        val = val * 10 + d
    return  val
```

These algorithms can help implement the `String` class.

`String` class implementation

```
/* Implements the String type. */
class String {

    /* Constructs a new empty string with a maximum length. */
    constructor String new(int maxLength) {}

    /* De-allocates the string and frees its memory space. */
    method void dispose() {}

    /* Returns the length of this string. */
    method int length() {}

    /* Returns the character value at the specified index */
    method char charAt(int j) {}

    /* Sets the j'th character of this string to the given character. */
    method void setCharAt(int j, char c) {}

    /* Appends the given character to the end of this string, and returns the string. */
    method String appendChar(char c) {}

    /* Erases the last character from this string. */
    method void eraseLastChar() {}

    /* Returns the integer value of this string until the first non-numeric character. */
    method int intValue() {}

    /* Sets this String to the representation of the given number. */
    method void setInt(int number) {}

    /* Returns the new line character. */
    function char newLine() {}

    /* Returns the backspace character. */
    function char backSpace() {}

    /* Returns the double quote (") character. */
    function char doubleQuote() {}
}
```

Implementation notes:

- `length`, `charAt`, `setCharAt`, `appendChar`, `eraseLastChar`:

Implement using array manipulations

- `intValue`:

Implement the *string2int* algorithm

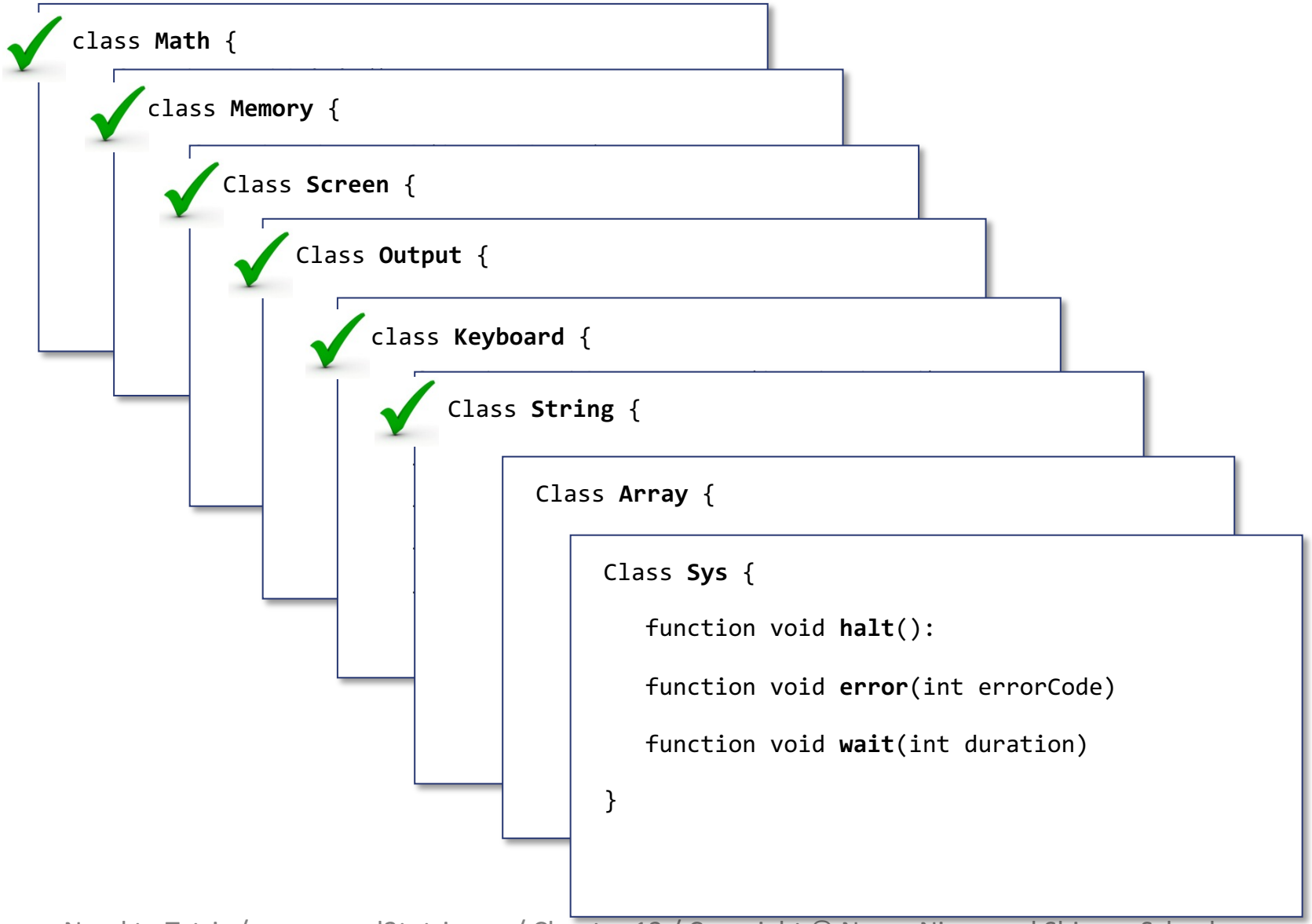
- `setInt`:

Implement the *int2String* algorithm

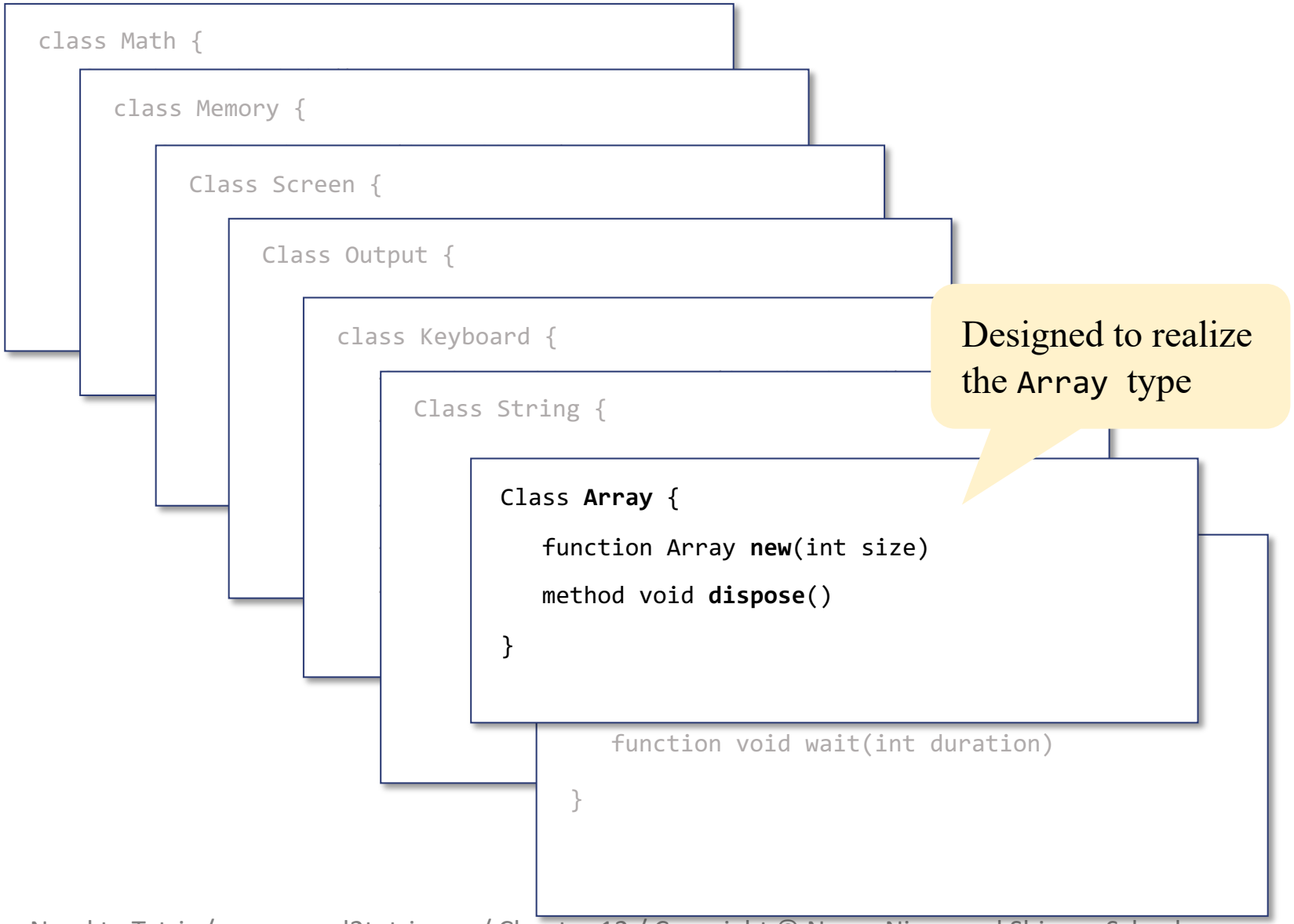
- `newLine`, `backSpace`, `doubleQuote`:

Implement by returning the `int` values `128`, `129`, `34`.

The Jack OS


class Math {
class Memory {
Class Screen {
Class Output {
class Keyboard {
Class String {
Class Array {
Class Sys {
function void halt():
function void error(int errorCode)
function void wait(int duration)
}

The Jack OS: `Array`



The client's view

OS `Array` class

```
Class Array {

   function Array new(int size)

   method void dispose()

}
```

Typical client view:

```
...
var Array a, b;
...
let a = Array.new(3);
...
let a[2] = 222;
...
let b = Array.new(50);
...
let b[1] = a[2] - 100;
...
do a.dispose();
...
```

Implementation notes

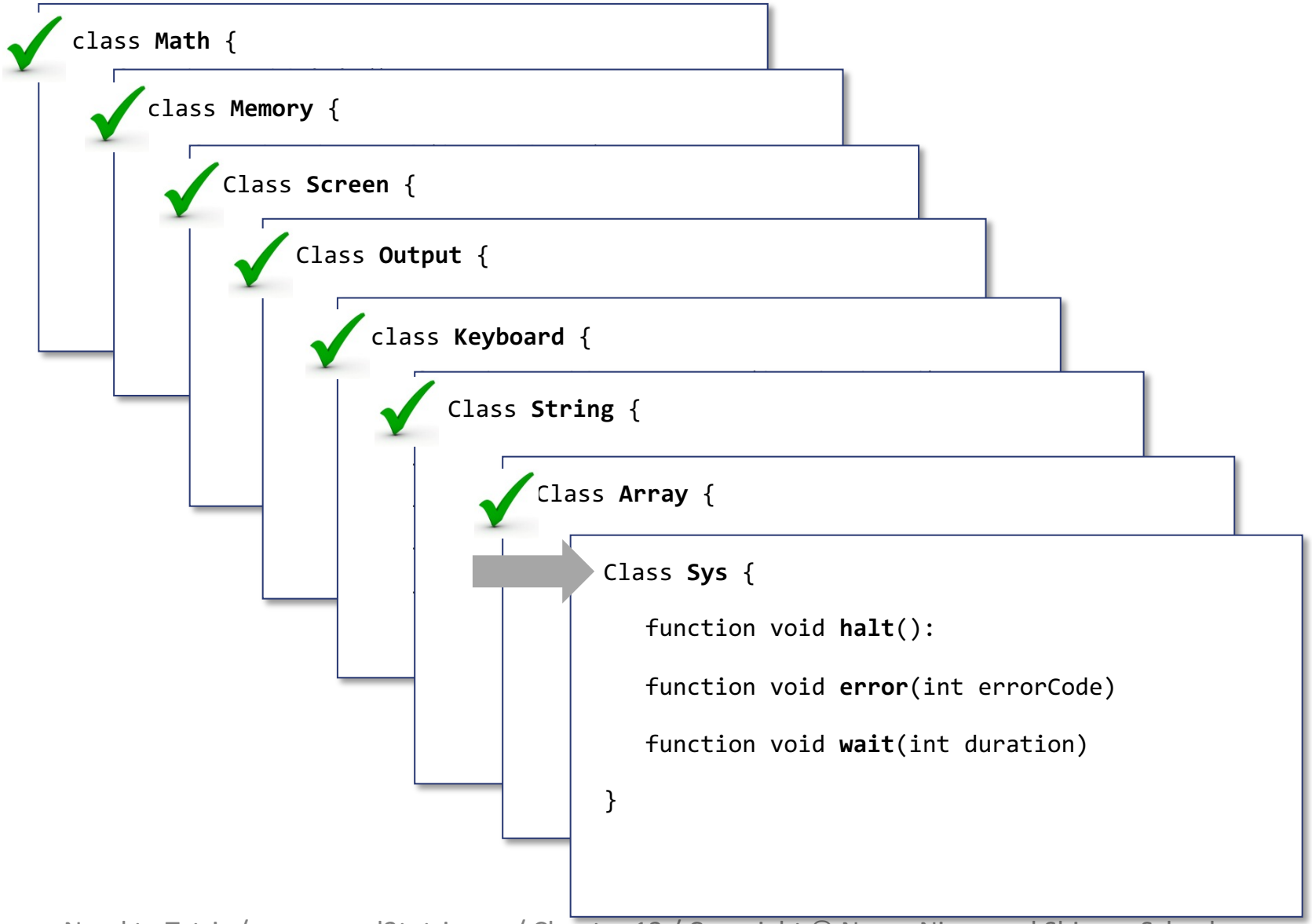
- implemented as a *function*, not as a constructor
- The implementation must call `Memory.alloc`

OS `Array` class

```
Class Array {

   function Array new(int size)

   method void dispose()

}
```

The implementation calls `Memory.deAlloc`

The Jack OS


class Math {
class Memory {
Class Screen {
Class Output {
class Keyboard {
Class String {
Class Array {
Class Sys {
function void halt():
function void error(int errorCode)
function void wait(int duration)
}

Bootstrapping

Bootstrapping (booting): the process of loading the basic software into the memory of a computer after power-on or general reset, especially the operating system, which will then take care of loading other software as needed (Wikipedia)

Hardware contract

When the computer is reset, execution starts with the instruction in `ROM[0]`

VM contract

The following code should be placed at the top of the `ROM`, beginning in `ROM[0]`:

```
sp = 256
call Sys.init
```

Jack contract

Program execution starts with the function `Main.main()`

OS contract

`Sys.init` should initialize the OS, and then call `Main.main()`

Implementation notes

```
/* A library of basic system services  */
class Sys {

    // Performs all the OS initializations, and calls ”main”
    function void init() {
       do Math.init();
       do Memory.init();
       do Screen.init();
       ...
       do Main.main();
   }
```

Implementation notes

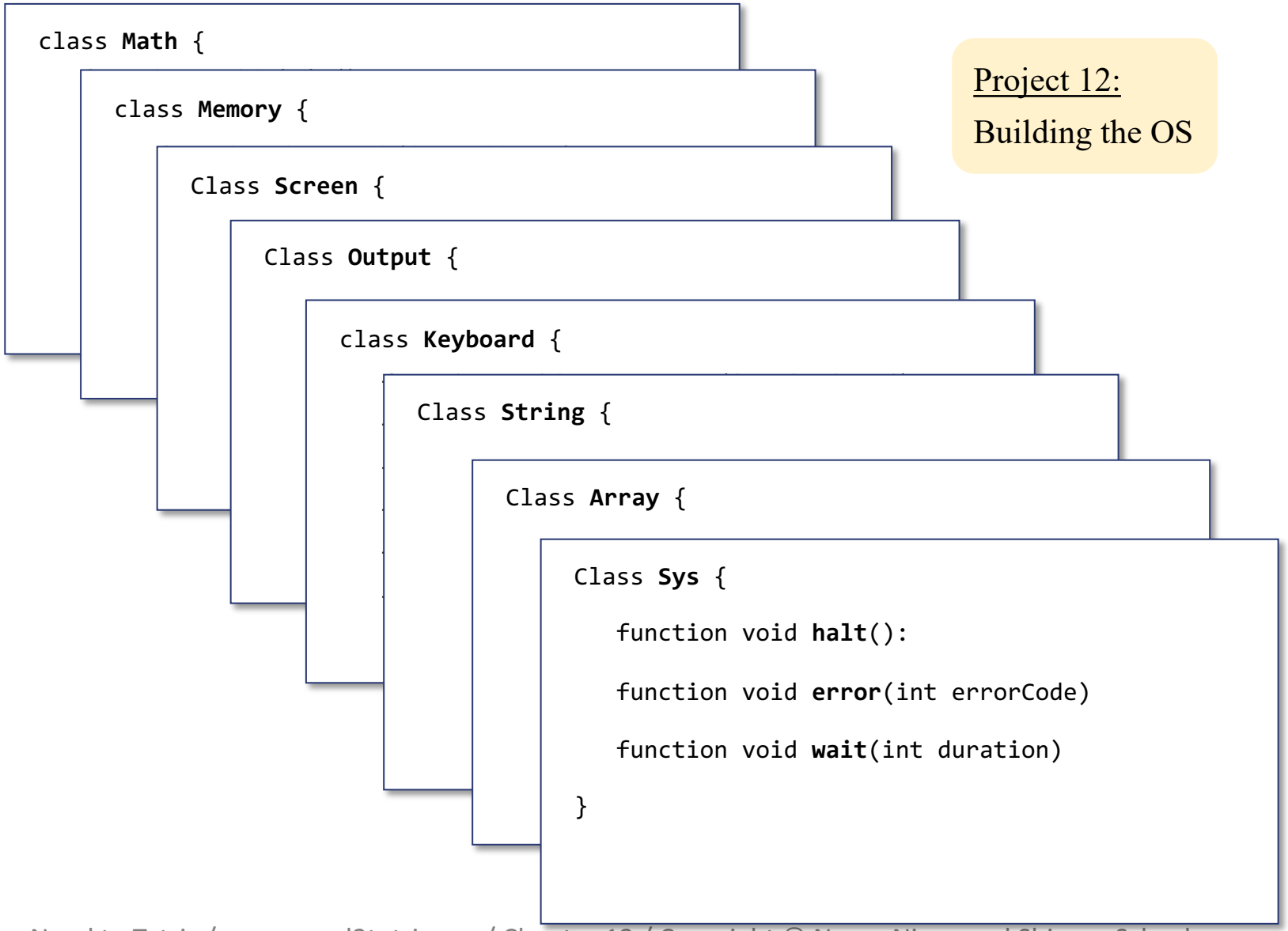
```
/* A library of basic system services  */
class Sys {

    // Performs all the OS initializations, and calls ”main”
    function void init() {}

    /* Halts execution */
    function void halt() {}

    /* Waits approximately duration milliseconds, and returns */
    function void wait(int duration) {}

    /* Prints the given error code in the form "ERR<errorCode>", and halts. */
    function void error(int errorCode) {}

}
```

Use an infinite loop

Use a loop, hardware specific

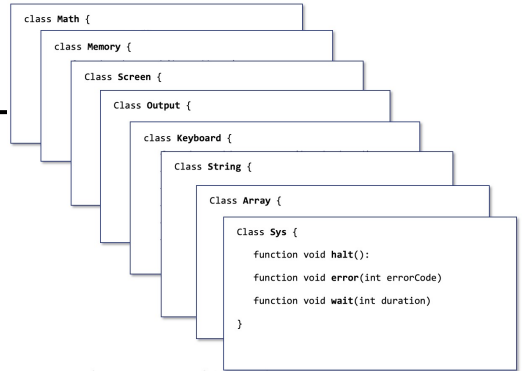
Simple

The Jack OS



Project 12:
Building the OS

The Jack OS



OS abstraction: specified by the Jack OS API

OS implementations

- **VM emulator:** features a Java-based (built-in) OS implementation

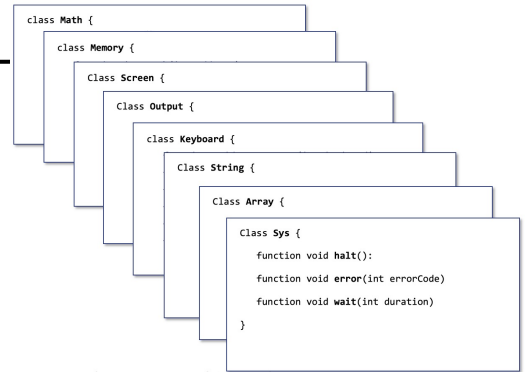
- **nand2tetris/tools/os:** features an executable Jack-based OS implementation (VM code):

  `Math.vm`, `Memory.vm`, `Screen.vm`, `Output.vm`, `Keyboard.vm`,

  `String.vm`, `Array.vm`, `Sys.vm`

- **Project 12:** write your own Jack-based OS implementation.

Reverse engineering



Suppose you wish to implement an existing OS, consisting of *n* modules, with high inter-dependency. You have access only to *executable versions* of the *n* modules.

Strategy: For each module in the OS, implement the module separately, using the remaining *n* – 1 executable modules to service it

Example:

Suppose you wish to develop the OS class `Screen`, and test it using some class `Main.jack`

<u>If using the supplied VM emulator:</u>

- Put your `Screen.jack` implementation and `Main.jack` in a folder
- Compile the folder
- Load the folder into the VM emulator, and execute the code
- (every call to a `Screen.`*function* will be serviced by your `Screen` implementation; every call to any other OS function will be serviced by the built-in implementations of the remaining 7 OS classes)

<u>If using the supplied CPU emulator:</u>

- Put the files `Screen.jack` and `Main.jack` in a folder, as well as all the `OS.vm` files except for `Screen.vm`
- Compile the folder
- Use a VM translator to translate the folder into the file *folderName*`.asm`
- Load *folderName*`.asm` into the supplied CPU emulator, and execute the code.

Example: Implementing / testing the OS `Screen` class

"stub file", supplied by us

```
/* A library of OS functions for displaying graphics on the screen */
class Screen {

    // Initializes the Screen
    function void init() {}

    /* Erases the entire screen */
    function void clearScreen() {}

    /* Sets the current color, to be used for all subsequent drawXXX commands.
       Black is represented by true, white by false */
    function void setColor(boolean b) {}

    /* Draws the (x,y) pixel, using the current color */
    function void drawPixel(int x, int y) {}

    /* Draws a line from pixel (x1,y1) to pixel (x2,y2), using the current color */
    function void drawLine(int x1, int y1, int x2, int y2) {}

    /* Draws a filled rectangle whose top left corner is (x1, y1)
       and bottom right corner is (x2,y2), using the current color */
    function void drawRectangle(int x1, int y1, int x2, int y2) {}

    /* Draws a filled circle of radius r<=181 around (x,y), using the current color */
    function void drawCircle(int x, int y, int r) {}
}
```

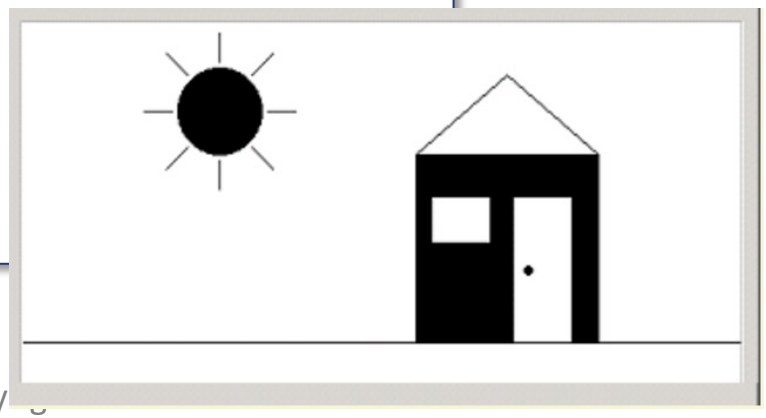
Project 12:

Implement the class (calling subroutines from the other 7 OS classes, as needed)

Example: Implementing / testing the OS `Screen` class

test code, supplied by us

```
/* Test program for the OS Screen class. */
class Main {

    /* Draws a sample picture on the screen using lines and circles */
    function void main() {
        do Screen.drawLine(0,220,511,220);        // base line
        do Screen.drawRectangle(280,90,410,220);  // house

        do Screen.setColor(false);
        do Screen.drawRectangle(350,120,390,219); // door
        do Screen.drawRectangle(292,120,332,150); // window

        do Screen.setColor(true);
        do Screen.drawCircle(360,170,3);          // door handle
        do Screen.drawLine(280,90,345,35);        // roof
        do Screen.drawLine(345,35,410,90);        // roof

        do Screen.drawCircle(140,60,30);          // sun
        do Screen.drawLine(140,26, 140, 6);
        ...

        return;
    }
}
```



Project 12

Step-wise development:

- `Screen`, `Output`, `String`, `Keyboard`, `Sys`: develop/test separately, using the supplied `Main.jack` test programs
- `Memory`, `Array`, `Math`: develop/test separately, using the supplied `.jack`, `.tst` and `.cmp` files
- Can be developed in any order

Final test:

1. Make a copy of the given folder `nand2tetris/projects/11/Pong`
2. Copy the 8 compiled `.vm` files of your OS into this folder
3. Execute this folder in the VM emulator.

Project 12

Perspective

Missing elements in our OS

- Multi-threading
- Multi-processing
- File system
- Shell / windowing
- Security
- Communications
- Many more missing services.

Perspective

Our OS...

- Closes significant gaps between the underlying hardware and application programs
- Hides many gory low-level details from the application programmer
- Performs its job elegantly and efficiently
- provides a feeling of what it takes to design and implement a simple OS, or an OS module.

Perspective

OS code is typically considered *privileged*:

- Accessing OS services requires a permission mechanism that is more elaborate than a simple function call
- OS functions execute in a special protected mode
- OS functions receive more hardware resources
- (in the Hack system, user-level code and OS code are treated the same way).

Perspective

Efficiency: Mathematical operations

- In most computers, core mathematical operations are implemented in hardware
- The hardware and software implementations of these operations are often based on the same algorithmic ideas
- The running time of the multiplication and division algorithms that we presented is $O(N)$ addition operations, N being the number of bits
- Since our addition implementation also requires $O(N)$ operations, the overall running time of our multiplication and division algorithms is $O(N^2)$
- There exist multiplication and division algorithms whose running time is asymptotically much faster than $O(N^2)$
- However, these algorithms are useful only when the number of bits that we have to process is very large.

Perspective

Efficiency: Graphical operations

- The line drawing algorithms that we presented are efficient
- In most systems these graphics primitives are implemented by a combination of software and special graphics-acceleration hardware
- Today, most computers are equipped with a *Graphical Processing Unit*
- The GPU off-loads the CPU from handling high-performance graphics tasks, like drawing 3D images and rendering smooth surfaces
- In the Hack-Jack platform, there is no such separation of responsibilities between the CPU and other dedicated processors.

Chapter 12

Operating System

These slides support chapter 12 of the book
The Elements of Computing Systems
By Noam Nisan and Shimon Schocken
MIT Press