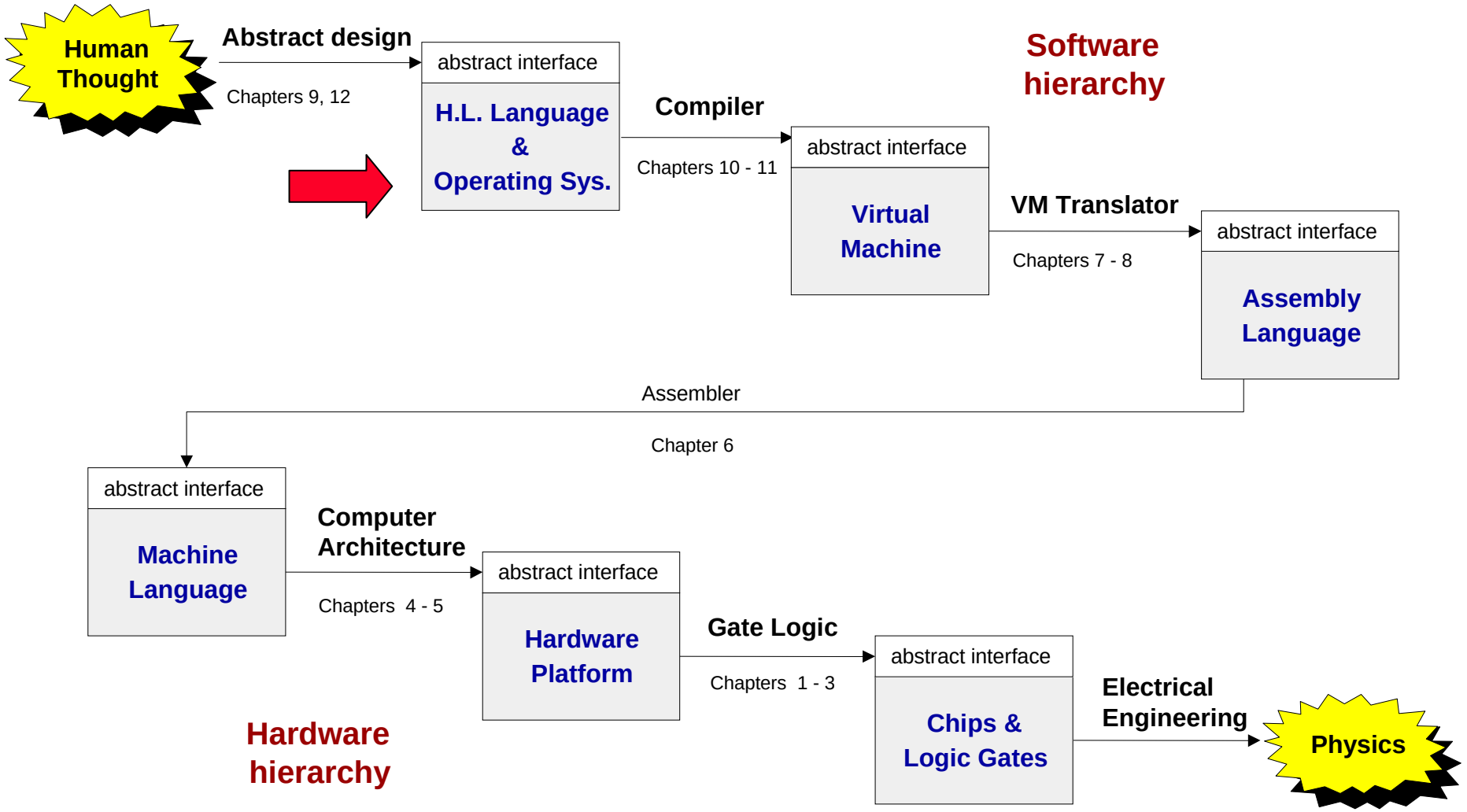
Chapter 12:
Operating System

Where we are at:



Jack revisited

```
/** Computes the average of a sequence of integers. */
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 the next number: ");
      let sum = sum + a[i];
      let i = i + 1;
    }

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

Jack revisited

```
/** Computes the average of a sequence of integers. */
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 the next number: ");
      let sum = sum + a[i];
      let i = i + 1;
    }

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

Typical OS functions

Language extensions / standard library

- Mathematical operations (**abs**, **sqrt**, ...)
- Abstract data types (**String**, **Date**, ...)
- Output functions (**printChar**, **printString** ...)
- Input functions (**readChar**, **readLine** ...)
- Graphics functions (**drawPixel**, **drawCircle**, ...)
- And more ...

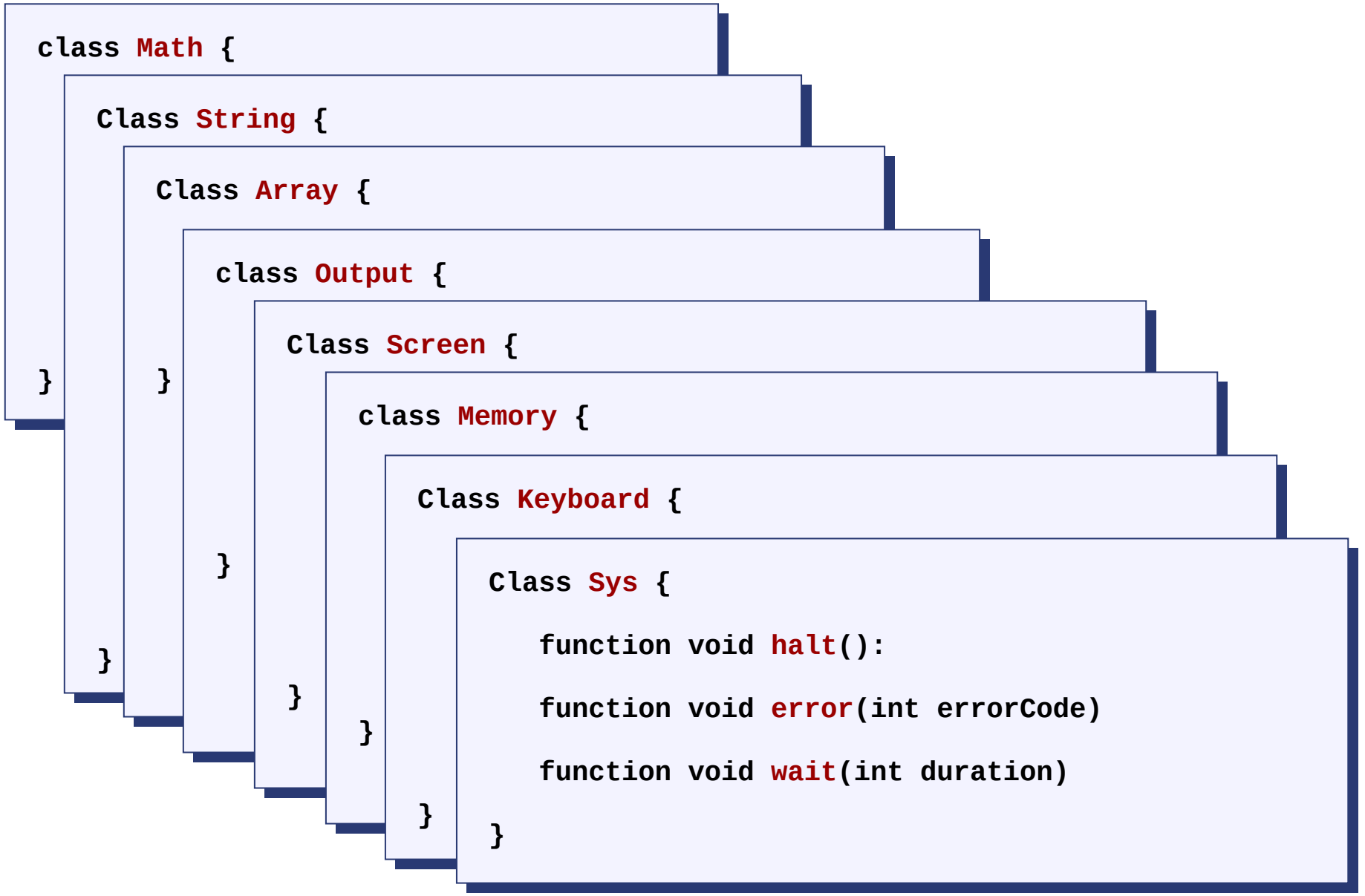
System-oriented services

- Memory management (objects, arrays, ...)
- I/O device drivers
- Mass storage
- File system
- Multi-tasking
- UI management (shell / windows)
- Security
- Communications
- And more ...

The Jack OS

- **`Math`:** Provides basic mathematical operations;
- **`String`:** Implements the **`String`** type and string-related operations;
- **`Array`:** Implements the **`Array`** type and array-related operations;
- **`Output`:** Handles text output to the screen;
- **`Screen`:** Handles graphic output to the screen;
- **`Keyboard`:** Handles user input from the keyboard;
- **`Memory`:** Handles memory operations;
- **`Sys`:** Provides some execution-related services.

Jack OS API


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

A typical OS:

- Is modular and scalable
- Empowers programmers (language extensions)
- Empowers users (file system, GUI, ...)
- Closes gaps between software and hardware
- Runs in "protected mode"
- Typically written in some high level language
- Typically grow gradually, assuming more and more functions
- Must be efficient.

Efficiency first

- We have to implement various operations on *n*-bit binary numbers (*n* = 16, 32, 64, ...). Example: multiplication
- Naïve algorithm: to multiply x*y: { for i = 1... y do sum = sum + x }

  Run-time is proportional to *y*

  In a 64-bit system, *y* can be as large as 2^{64}.

  The multiplication will take years to complete

  If the run-time were proportional to 64 instead, we are OK
- In general, algorithms that operate on *n*-bit inputs are either:
  - Naïve: run-time is prop. to the *value* of the *n*-bit inputs
  - Good: run-time is proportional to *n*.

Example I: multiplication

The "steps"

$$
\begin{array}{cccccc}
 & 1 & 0 & 1 & 1 & = 11 \\
 & & 1 & 0 & 1 & = 5 \\
\hline
 & 1 & 0 & 1 & 1 & \\
0 & 0 & 0 & 0 & & \\
1 & 0 & 1 & 1 & & \\
\hline
1 & 1 & 0 & 1 & 1 & 1 = 55
\end{array}
$$

The algorithm explained (first 4 of 16 iteration)

x:	0	0	0	1	0	1	1	
y:	0	0	0	0	1	0	1	j'th bit of y
	0	0	0	**1**	**0**	**1**	**1**	1
	0	0	**1**	**0**	**1**	**1**	0	0
	0	**1**	**0**	**1**	**1**	0	0	1
	1	**0**	**1**	**1**	0	0	0	0
$x \cdot y$:	0	1	1	0	1	1	1	sum

multiply(x, y):

```
// Where x, y ≥ 0
sum = 0
shiftedX = x
for j = 0...(n-1) do
    if (j-th bit of y) = 1 then
        sum = sum + shiftedX
    shiftedX = shiftedX * 2
```

- Run-time: proportional to *n*
- Can be implemented in SW or HW
- Division: similar idea.

Example II: square root

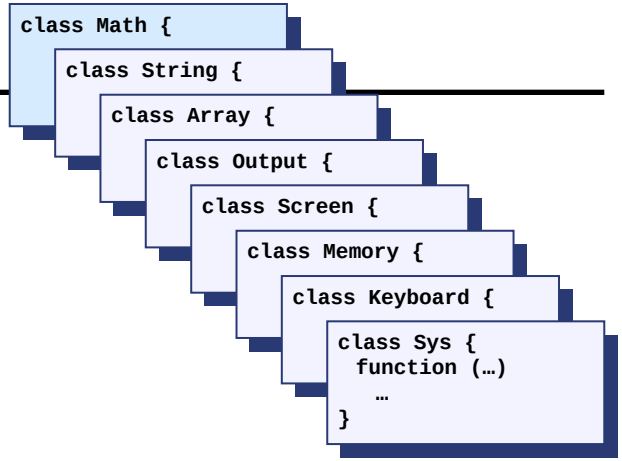
- The square root function has two useful properties:
  - An inverse function that we know how to compute
  - Monotonically increasing
- Ergo, square root can be computed via binary search:

sqrt(x):

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

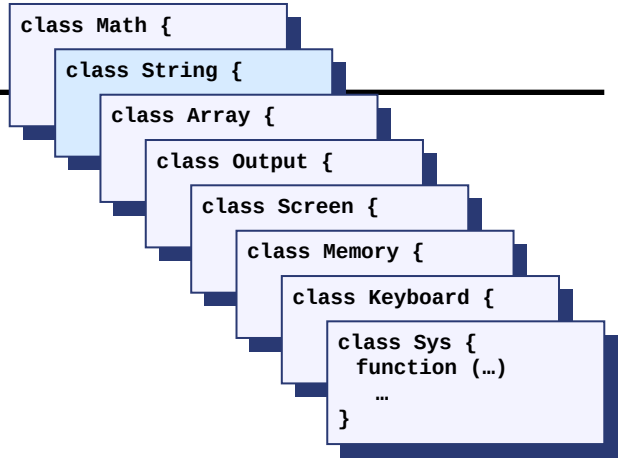
- Number of loop iterations is bound by n/2, thus the run-time is O(n).

Math operations (in 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)

}
```

String processing (in the Jack OS))




```
Class String {

   constructor String new(int maxLength)

   method void   dispose()

   method int    length()

   method char   charAt(int j)

   method void   setCharAt(int j, char c)

   method String appendChar(char c)

   method void   eraseLastChar()

   method int    intValue()

   method void   setInt(int j)

   function char backSpace()

   function char doubleQuote()

   function char newLine()

}
```

Converting a single digit to its ASCII code

Character:	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII code:	48	49	50	51	52	53	54	55	56	57

- ASCIICode(digit) == 48 + digit
- Reverse conversion: easy.

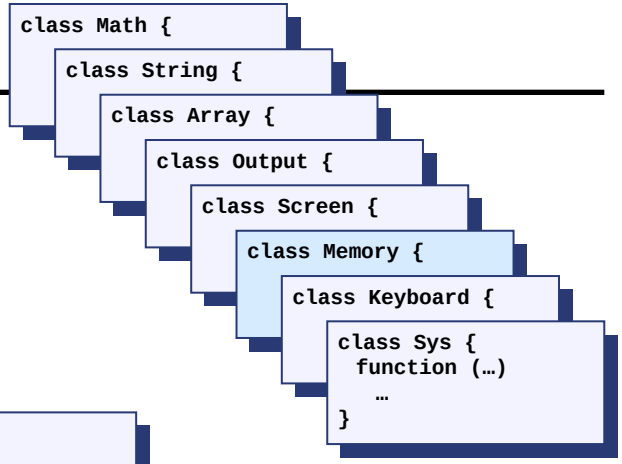
Converting a number to a string

- SingleDigit-to-character conversions: done
- Number-to-string conversions:

```
// Convert a non-negative number to a string
int2String(n):
    lastDigit = n % 10
    c = character representing lastDigit
    if n < 10
       return c (as a string)
    else
       return int2String(n / 10).append(c)
```

```
// Convert a string to a non-negative number
string2Int(s):
    v = 0
    for i = 1... length of s do
        d = integer value of the digit s[i]
        v = v * 10 + d
    return v
    // (Assuming that s[1] is the most
    //  significant digit character of s.)
```

Memory management (in the Jack OS)




```
class Memory {

   function int peek(int address)

   function void poke(int address, int value)

   function Array alloc(int size)

   function void deAlloc(Array o)

}
```

Memory management (simple)

- When a program constructs (destructs) an object, the OS has to allocate (de-allocate) a RAM block on the heap:
  - **`alloc(size)`:** returns a reference to a free RAM block of size **`size`**
  - **`deAlloc(object)`:** recycles the RAM block that **`object`** points at

Initialization: *free* = *heapBase*

// Allocate a memory block of *size* words.
alloc(*size*):
  pointer = *free*
  free = *free* + *size*
  return *pointer*

// De-allocate the memory space of a given *object*.
deAlloc(*object*):
  do nothing

- The data structure that this algorithm manages is a single pointer: free.

Memory management (improved)

Initialization:

freeList = *heapBase*

freeList.length = *heapLength*

freeList.next = null

// Allocate a memory space of *size* words.

alloc(*size*):

Search *freeList* using best-fit or first-fit heuristics
to obtain a segment with *segment.length* > *size*

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

block = needed part of the found segment
(or all of it, if the segment remainder is too small)

Update *freeList* to reflect the allocation

block[-1] = *size* + 1 // Remember block size, for de-allocation

Return *block*

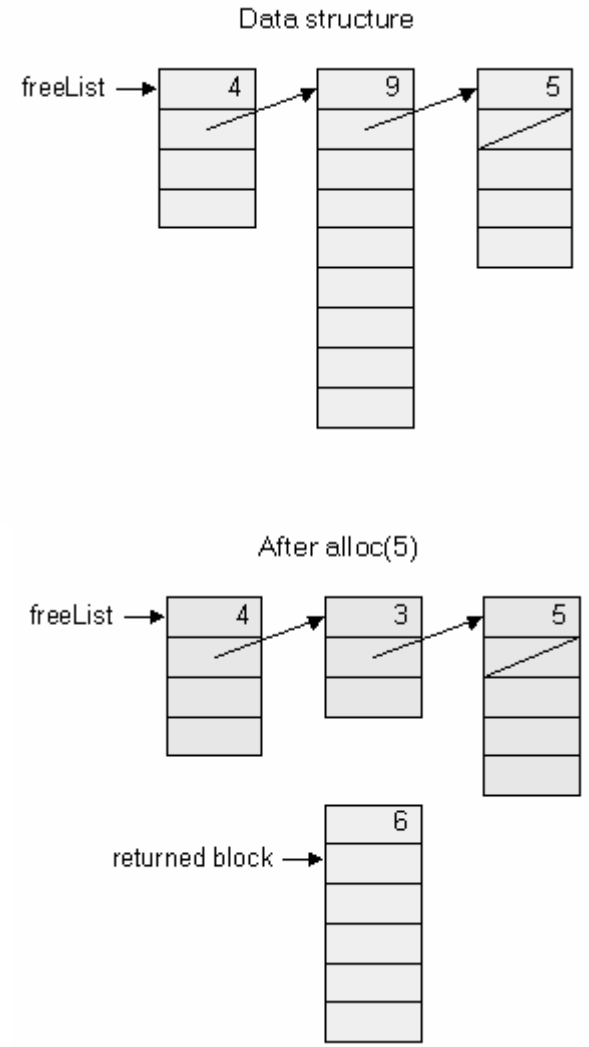
// Deallocate a decommissioned *object*.

deAlloc(*object*):

segment = *object* - 1

segment.length = *object*[-1]

Insert *segment* into the *freeList*



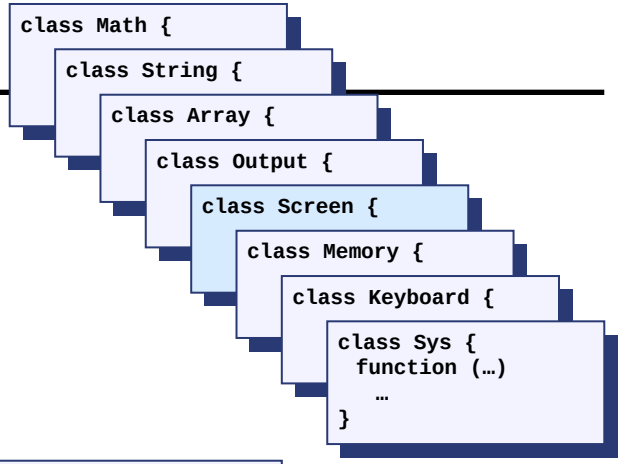
Peek and poke

```
class Memory {

   function int peek(int address)

   function void poke(int address, int value)

   function Array alloc(int size)

   function void deAlloc(Array o)

}
```

- Implementation: exploiting exotic casting in Jack:

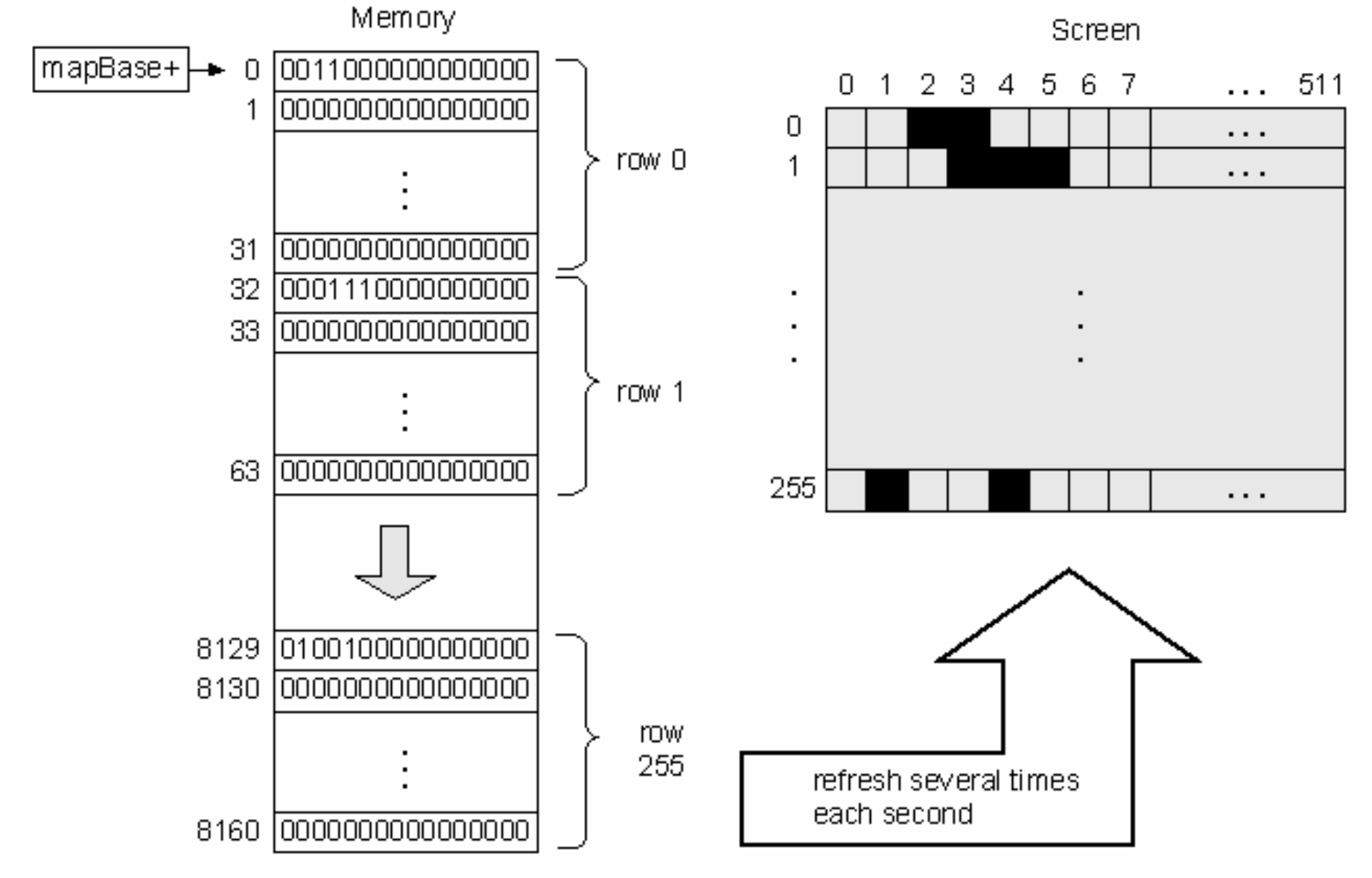
```
// To create a Jack-level "proxy" of the RAM:
var Array memory;
let memory = 0;
// From this point on we can use code like:
let x = memory[j]  // Where j is any RAM address
let memory[j] = y  // Where j is any RAM address
```

Graphics primitives (in the Jack OS)




```
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)

}
```

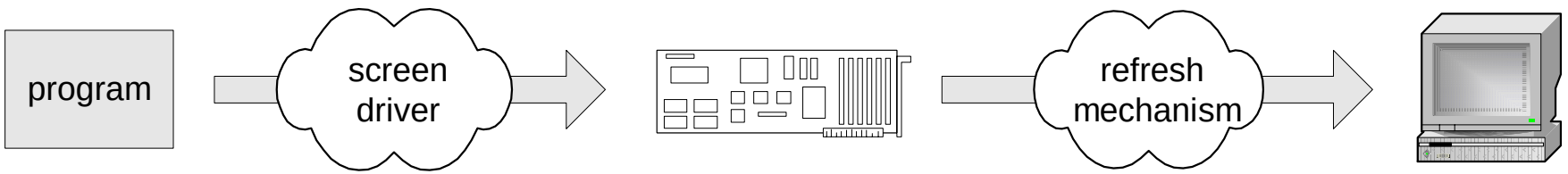
Memory-mapped screen



Pixel drawing

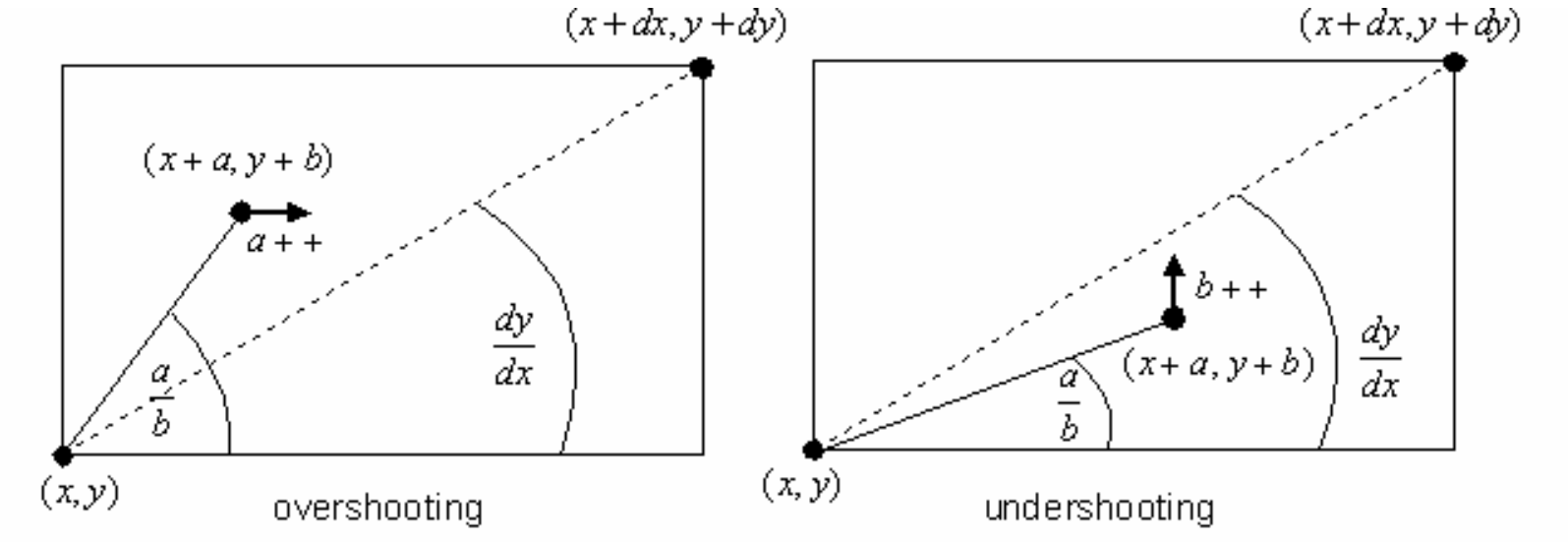
drawPixel (x, y):
// Hardware-specific.
// Assuming a memory mapped screen:
Write a predetermined value in the RAM
location corresponding to screen location (x, y).

- Implementation: using poke(address,value)



application | part of the operating system | screen memory map | part of the hardware | physical screen

Line drawing



drawLine$(x, y, x+dx, y+dy)$:

// Assuming $dx, dy > 0$

initialize $(a, b) = (0,0)$

while $a \leq dx$ and $b \leq dy$ do

drawPixel $(x+a, y+b)$

if $a/dx < b/dy$ then a++ else b++

- dx=0 and dy=0 are not handled
- Must also handle (dx,dy<0), (dx>0,dy<0), (dx<0,dy>0)

Line drawing

```
drawLine(x, y, x+dx, y+dy):
   // Assuming dx, dy > 0
   initialize (a, b) = (0,0)
   while a ≤ dx and b ≤ dy do
         drawPixel (x + a, y + b)
         if a / dx < b / dy then a++ else b++
```

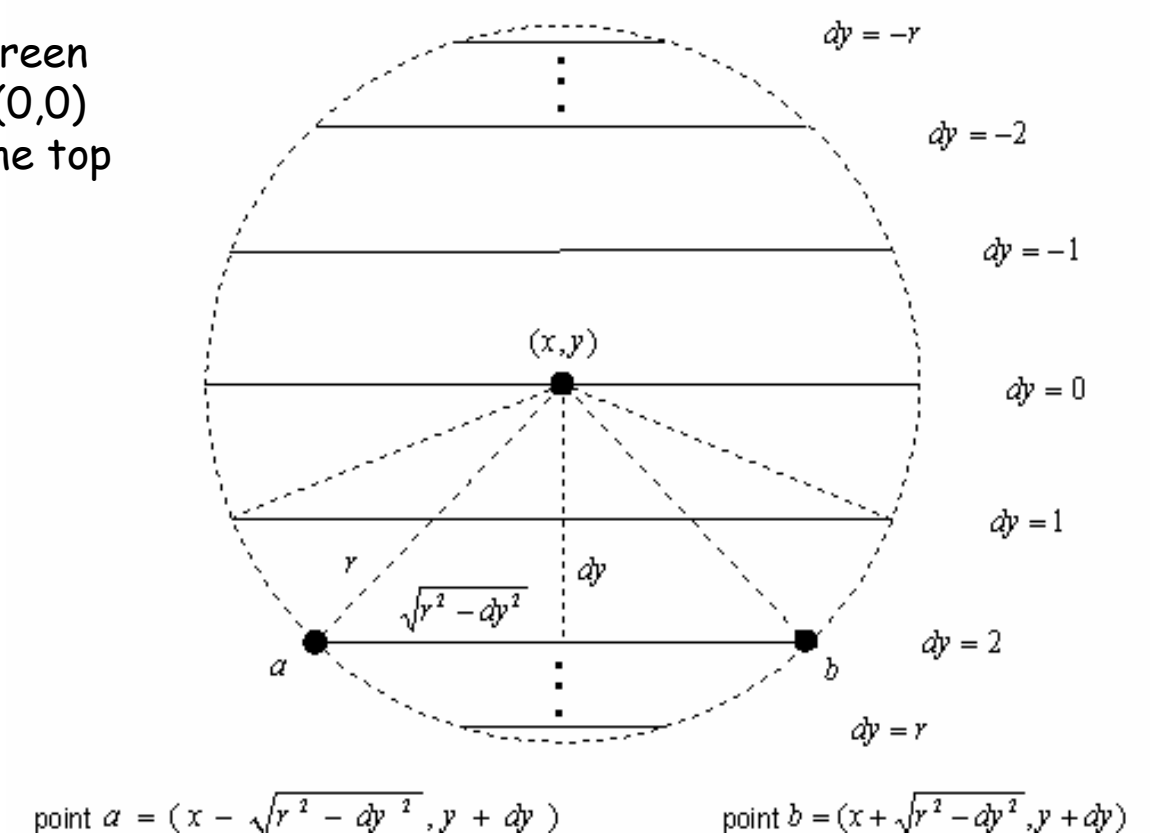
a/dx < b/dy is the same as a*dy < b*dx

```
// To test whether a / dx < b / dy, maintain a variable adyMinusbdx,
// and test if it becomes negative.
   Initialization:                set adyMinusbdx = 0
   When a++ is performed:         set adyMinusbdx = adyMinusbdx + dy
   When b++ is performed:         set adyMinusbdx = adyMinusbdx - dx
```

And so we are back to addition …

Circle drawing

The screen origin (0,0) is at the top left.

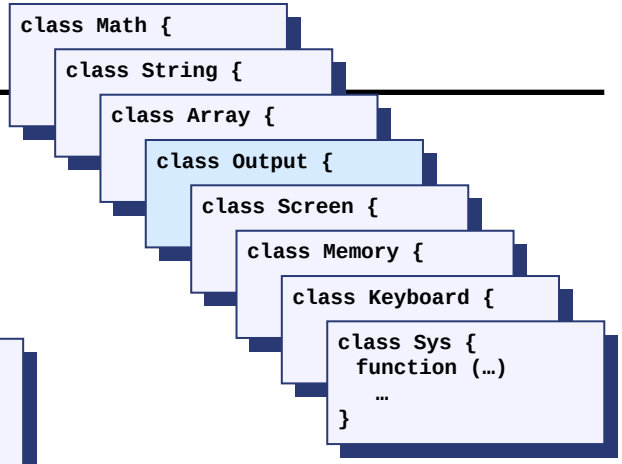


drawCircle(x, y, r):

for each $dy \in -r \ldots r$ do

drawLine from $(x-\sqrt{r^2-dy^2}, y+dy)$ to $(x+\sqrt{r^2-dy^2}, y+dy)$

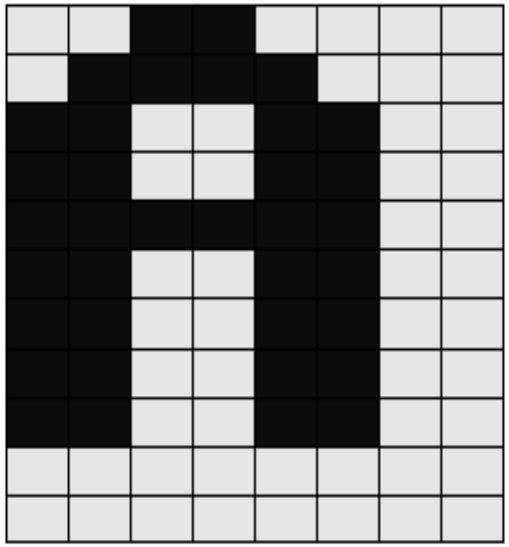
Character output primitives (in the Jack OS)




```
class Output {

   function void moveCursor(int i, int j)

   function void printChar(char c)

   function void printString(String s)

   function void printInt(int i)

   function void println()

   function void backSpace()

}
```

Character output

- Given: a physical screen, say 256 rows by 512 columns
- We can allocate an 11 by 8 grid for each character
- Hence, our output package should manage a 23 lines by 64 characters screen
- Each displayable character must have a bitmap
- In addition, we have to manage a "cursor".

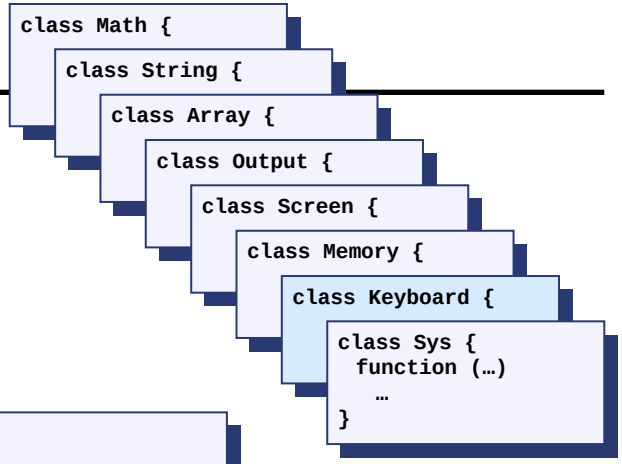


A font implementation (in the Jack OS)

```
class Output {
   static Array charMaps;
   function void initMap() {
      let charMaps = Array.new(127);
      // Assign a bitmap for each character
      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);  // #
      ...
      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(65,0,0,0,0,0,0,0,0,0,0,0);          // A ** TO BE FILLED **
      do Output.create(66,31,51,51,51,31,51,51,51,31,0,0); // B
      do Output.create(67,28,54,35,3,3,3,35,54,28,0,0);    // C
      . . .
      return;
 }
```

```
// Creates a character map array
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[10] = k;
    return; }
```

Keyboard primitives (in the Jack OS)




```
Class Keyboard {

   function char keyPressed()

   function char readChar()

   function String readLine(String message)

   function int readInt(String message)

}
```

Keyboard input

```
keyPressed():
   // Depends on the specifics of the keyboard interface
   if a key is presently pressed on the keyboard
      return the ASCII value of the key
   else
      return 0
```

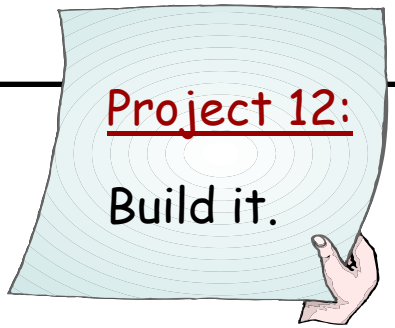
- If the RAM address of the keyboard's memory map is known, can be implemented using a peek function
- Problem I: the elapsed time between a "key press" and key release" events is unpredictable
- Problem II: when pressing a key, the user should get some visible feedback (cursor, echo, ...).

Keyboard input (cont.)

```
readChar():
    // Read and echo a single character
    display the cursor
    while no key is pressed on the keyboard
        do nothing  // wait till the user presses a key
    c = code of currently pressed key
    while a key is pressed
        do nothing // wait for the user to let go
    print c at the current cursor location
    move the cursor one position to the right
    return c
```

```
readLine():
    // Read and echo a "line" (until newline)
    s = empty string
    repeat
        c = readChar()
        if c = newline character
            print newline
            return s
        else if c = backspace character
                remove last character from s
                move the cursor 1 position back
            else
                s = s.append(c)
    return s
```

Jack OS recap




```
class Math {
    Class String {
        Class Array {
            class Output {

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

- Implementation: similar to how GNU Unix and Linux were built:
- Start with an existing system,
  and gradually replace it with a new system,
  one library at a time.

Perspective

- What we presented can be described as a:
  - Mini OS
  - Standard library
- Many classical OS functions are missing
- No separation between user mode and OS mode
- Some algorithms (e.g. multiplication and division) are standard
- Other algorithms (e.g. line- and circle-drawing) can be accelerated with special hardware
- And, by the way, we've just finished building the computer.

The End