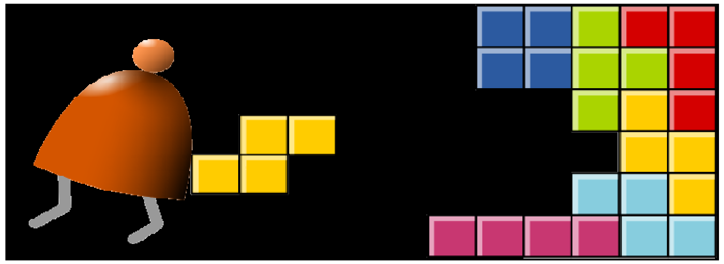
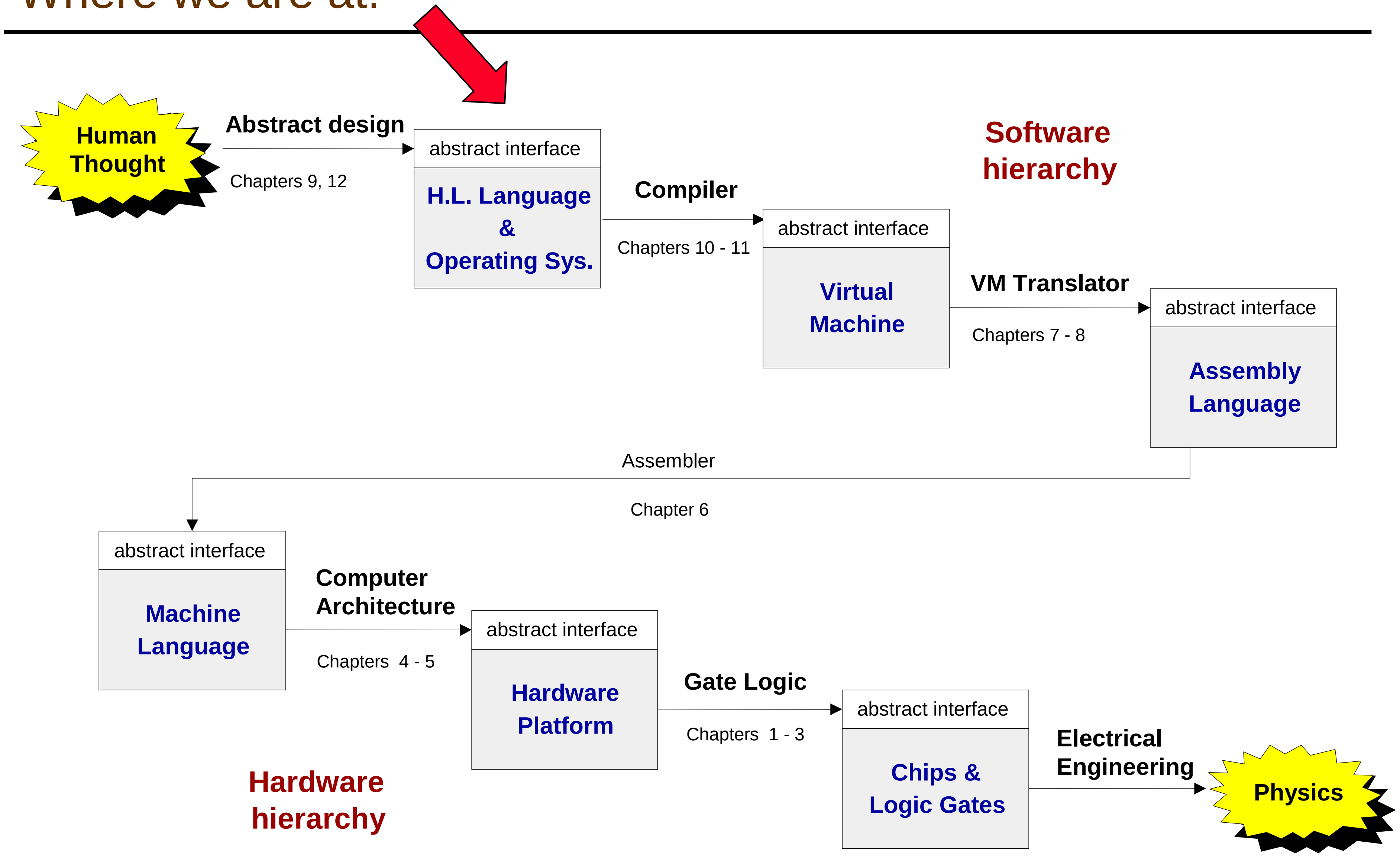
High-Level Language



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



A brief evolution of object-oriented languages

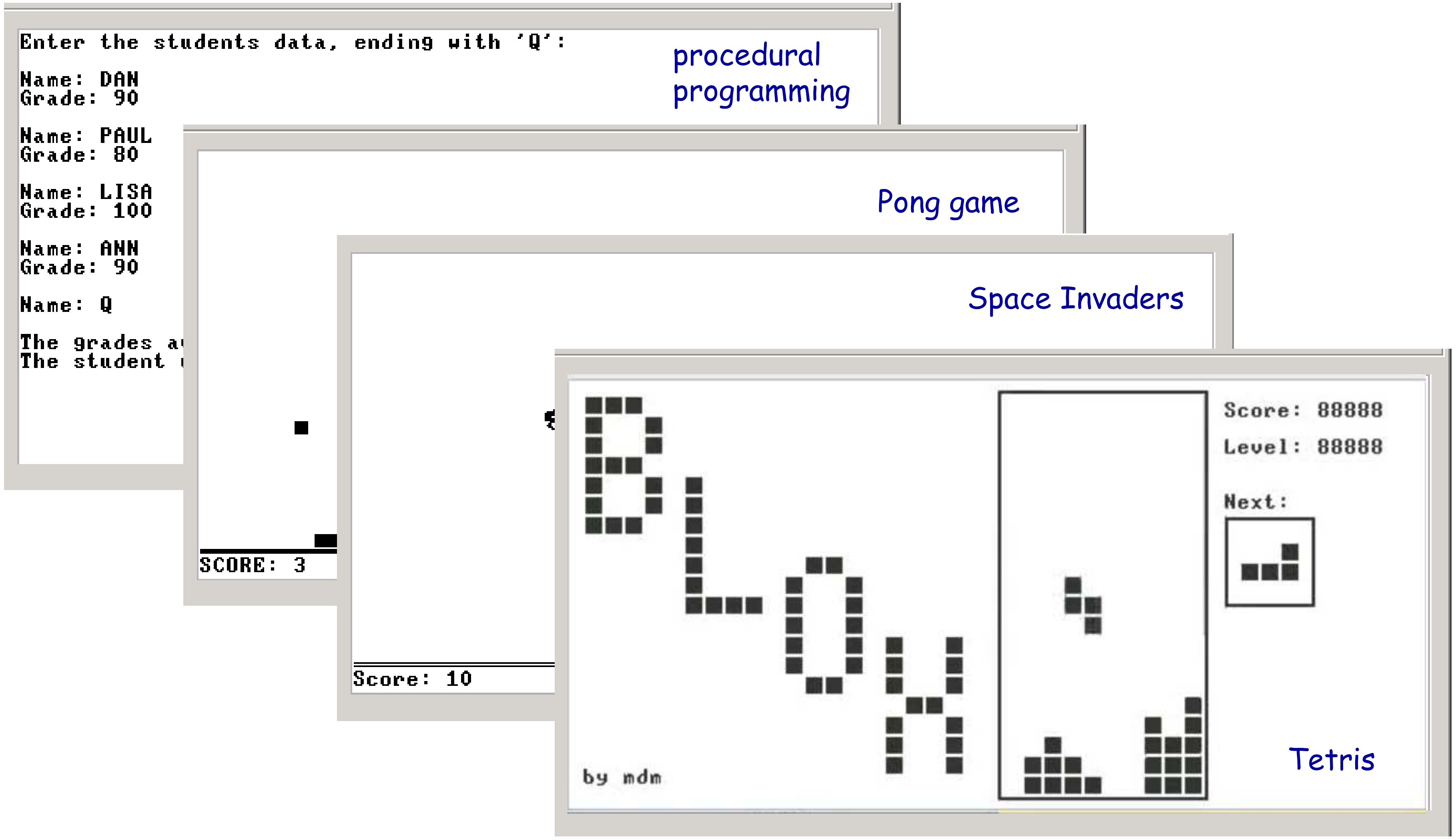
- Machine language (binary code)
- Assembly language (low-level symbolic programming)
- Simple procedural languages, e.g. `Fortran`, `Basic`, `Pascal`, `C`
- Simple object-based languages (without inheritance), e.g. early versions of `Visual Basic`, `JavaScript`

  Jack
- Fancy object-oriented languages (with inheritance): `C++`, `Java`, `C#`

The Jack programming language

Jack: a simple, object-based, high-level language with a Java-like syntax

Some sample applications written in Jack:



```
Enter the students data, ending with 'Q':

Name: DAN
Grade: 90

Name: PAUL
Grade: 80

Name: LISA
Grade: 100

Name: ANN
Grade: 90

Name: Q

The grades a
The student
```

Disclaimer

Although Jack is a real programming language, we don't view it as an *end*

Rather, we use Jack as a *means* for teaching:

- How to build a compiler
- How the compiler and the language interface with the operating system
- How the topmost piece in the software hierarchy fits into the big picture

Jack can be learned (and un-learned) in one hour.

Hello world

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

Some observations:

- Java-like syntax
- Typical comments format
- Standard library
- Language-specific peculiarities.

Typical programming tasks in Jack

Jack can be used to develop any app that comes to my mind, for example:

Procedural programming:	a program that computes 1 + 2 + ... + n
Object-oriented programming:	a class representing bank accounts
Abstract data type representation:	a class representing fractions (like 2/5)
Data structure representation:	a class representing linked lists
Etc.	

We will now discuss the above app examples

As we do so, we'll begin to unravel how the magic of a high-level object-based language is delivered by the compiler and by the VM

These insights will serve us in the next lectures, when we build the Jack compiler.

Procedural programming example

```
class Main {

  /** Sums up 1 + 2 + 3 + ... + n */
  function int sum (int n) {
    var int sum, i;
    let sum = 0;
    let i = 1;
    while (~(i > n)) {
      let sum = sum + i;
      let i = i + 1;
    }
    return sum;
  }

  function void main () {
    var int n;
    let n = Keyboard.readInt("Enter n: ");
    do Output.printString("The result is: ");
    do Output.printInt(sum(n));
    return;
  }
}
```

Jack program = a collection of one or more classes

Jack class = a collection of one or more subroutines

Execution order: when we execute a Jack program, `Main.main()` starts running.

Jack subroutine:

- `method`
- `constructor`
- `function` (static method)
- (the example on the left has functions only, as it is "object-less")

Standard library: a set of OS services (methods and functions) organized in 8 supplied classes: `Math`, `String`. `Array`, `Output`, `Keyboard`, `Screen`, `Memory`, `Sys` (OS API in the book).

Object-oriented programming example

The `BankAccount` class API (method sigantures)

```
/** Represents a bank account.
    A bank account has an owner, an id, and a balance.
    The id values start at 0 and increment by 1 each
    time a new account is created. */

class BankAccount {

    /** Constructs a new bank account with a 0 balance. */
    constructor BankAccount new(String owner)

    /** Deposits the given amount in this account. */
    method void deposit(int amount)

    /** Withdraws the given amount from this account. */
    method void withdraw(int amount)

    /** Prints the data of this account. */
    method void printInfo()

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

}
```

Object-oriented programming example (continues)

```
/** Represents a bank account. */

class BankAccount {
  // class-level variable
  static int newAcctId;

  // Private variables (aka fields / properties)
  field int id;
  field String owner;
  field int balance;

  /** Constructs a new bank account */
  constructor BankAccount new (String owner) {
      let id = newAcctId;
      let newAcctId = newAcctId + 1;
      let this.owner = owner;
      let balance = 0;
      return this;   2
  }

  // More BankAccount methods.

}
```

```
  // Code in any other class:
  var int x;
  var BankAccount b;   1
3 let b = BankAccount.new("joe");
```

Explain: `return this`

The constructor returns the RAM base address of the memory block that stores the data of the newly created BankAccount object

Explain: `b = BankAccount.new("joe")`

Calls the constructor (which creates a new BankAccount object), then stores a pointer to the object's base memory address in variable b

Behind the scene (following compilation):

```
 // b = BankAccount.new("joe")
push "joe"
call BankAccount.new
pop b
```

Explanation: the VM code calls the constructor; the constructor creates a new object, pushes its base address onto the stack, and returns;

The calling code then pops the base address into a variable that will now point to the new object.

Object-oriented programming example (continues)

```
class BankAccount {
  static int nAccounts;

  field int id;
  field String owner;
  field int balance;

  // Constructor ... (omitted)

  /** Handles deposits */
  method void deposit (int amount) {
      let balance = balance + amount;
      return;
  }

  /** Handles withdrawls */
  method void withdraw (int amount){
      if (~(amount > balance)) {
          let balance = balance - amount;
      }
      return;
  }

  // More BankAccount methods.

}
```

```
...
var BankAccount b1, b2;
...
let b1 = BankAccount.new("joe");
let b2 = BankAccount.new("jane");
do b1.deposit(5000);
do b1.withdraw(1000);
...
```

Explain: `do b1.deposit(5000)`

- In Jack, `void` methods are invoked using the keyword `do` (a compilation artifact)
- The object-oriented method invocation style `b1.deposit(5000)` is a fabcy way to expres the procedural semantics `deposit(b1,5000)`

Behind the scene (following compilation):

```
// do b1.deposit(5000)
push b1
push 5000
call BanAccount.deposit
```

Object-oriented programming example (continues)

```
class BankAccount {
  static int nAccounts;

  field int id;
  field String owner;
  field int balance;

  // Constructor ... (omitted)

  /** Prints information about this account. */
  method void printInfo () {
      do Output.printInt(id);
      do Output.printString(owner);
      do Output.printInt(balance);
      return;
  }

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

  // More BankAccount methods.

}
```

```
// Code in any other class:
...
var int x;
var BankAccount b;

let b = BankAccount.new("joe");
// Manipulates b...
do b.printInfo();
do b.dispose();
...
```

Explain `do b.dispose()`

Jack has no garbage collection; The programmer is responsible for explicitly recycling memory resources of objects that are no longer needed.

Abstract data type example

The `Fraction` class API (method sigantures)

```
/** Represents a fraction data type.
    A fraction consists of a numerator and a denominator, both int values */

class Fraction {

    /** Constructs a fraction from the given data */
    constructor Fraction new(int numerator, int denominator)

    /** Reduces this fraction, e.g. changes 20/100 to 1/5. */
    method void reduce()

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

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

    /** Returns the product of this fraction and the other one */
    method Fraction product(Fraction other)

    /** Prints this fraction */
    method void print()

    /** Disposes this fraction */
    method void dispose()
}
```

Abstract data type example (continues)

```
/** Represents a fraction data type.
    A fraction consists of a numerator and a denominator, both int values */
class Fraction {
    field int numerator, denominator;

    constructor Fraction new (int numerator, int denominator) {
        let this.numerator = numerator;
        let this.denominator = denominator;
        do reduce() // Reduces the new fraction
        return this
    }

    /** Reduces this fraction */
    method void reduce () {
        // Code omitted
    }

    // A static method that computes the greatest common denominator of a and b.
    function int gcd (int a, int b) {
        // Code omitted
    }

    method int getNumerator () {
        return numerator;
    }

    method int getDenominator () {
        return denominator;
    }

    // More Fraction methods follow.
```

```
// Code in any other class:
...
var Fraction a, b;
let a = Fraction.new(2,5);
let b = Fraction.new(70,210);
do b.print() // prints "1/3"
...
// (print method in next slide)
```

Abstract data type example (continues)

```
/** Represents a fraction data type.
    A fraction consists of a numerator and a denominator, both int values */
class Fraction {
    field int numerator, denominator;

    // Constructor and previously defined methods omitted

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

    // Similar fraction arithmetic methods follow, code omitted.

    /** Prints this fraction */
    method void print () {
        do Output.printInt(numerator);
        do Output.printString("/");
        do Output.printInt(denominator);
        return
    }

 }
```

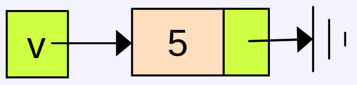
```
// Code in any other class:
var Fraction a, b, c;
let a = Fraction.new(2,3);
let b = Fraction.new(1,5);
// computes c = a + b
let c = a.plus(b);
do c.print(); // prints "13/15"
```

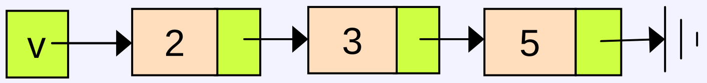
Data structure example

```
/** Represents a sequence of int values, implemented as a linked list.
    The list consists of an atom, which is an int value,
    and a tail, which is either a list or a null value.  */
class List {
    field int data;
    field List next;

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

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





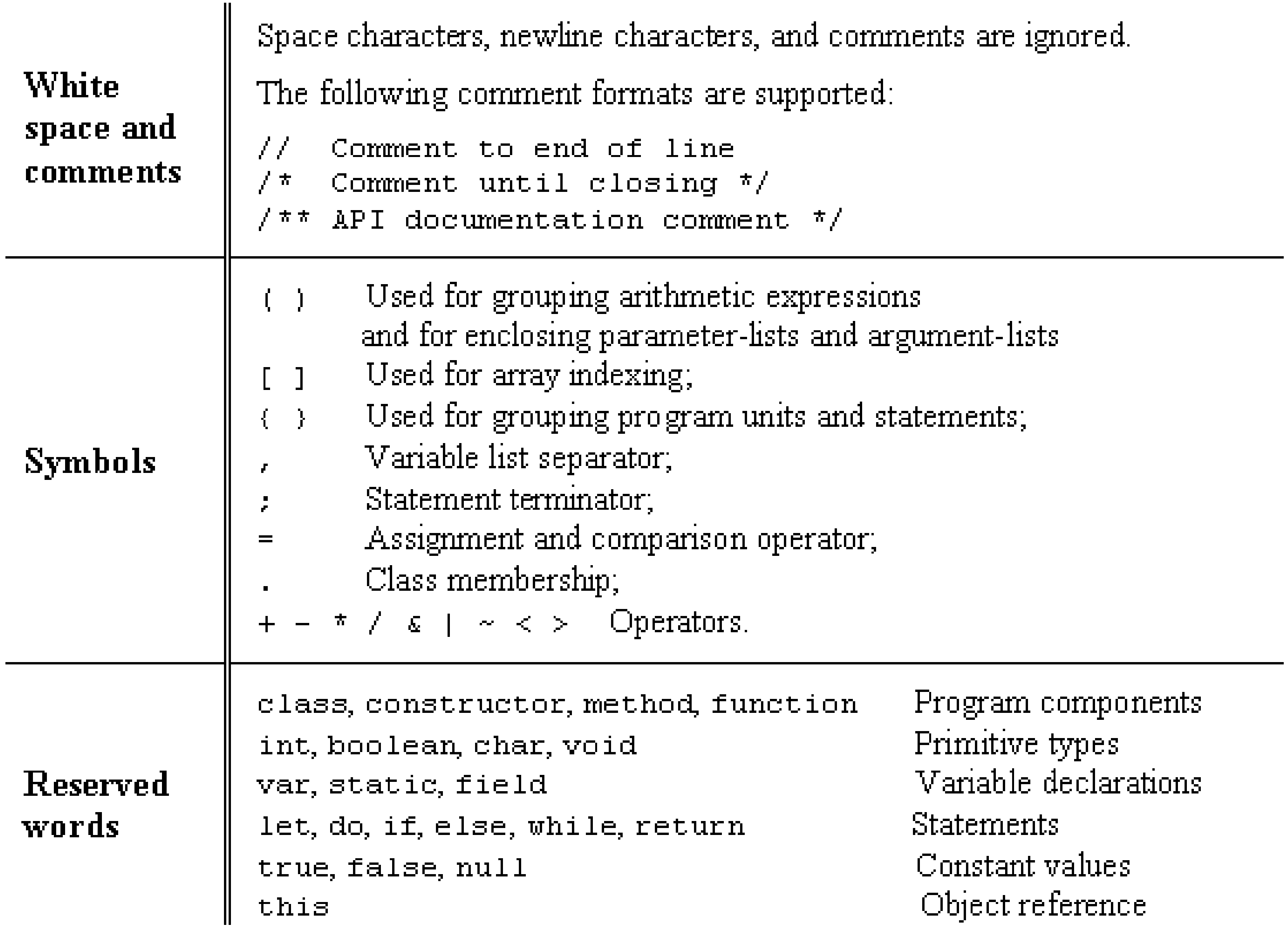
```
// Code in any other class:
...
// Creates a list holding the numbers 2,3, and 5:
var List v;
let v = List.new(5 , null);
let v = List.new(2 , List.new(3,v));
...
```

Jack language specification

- Syntax
- Data types
- Variable kinds
- Expressions
- Statements
- Subroutine calling
- Program structure
- Standard library

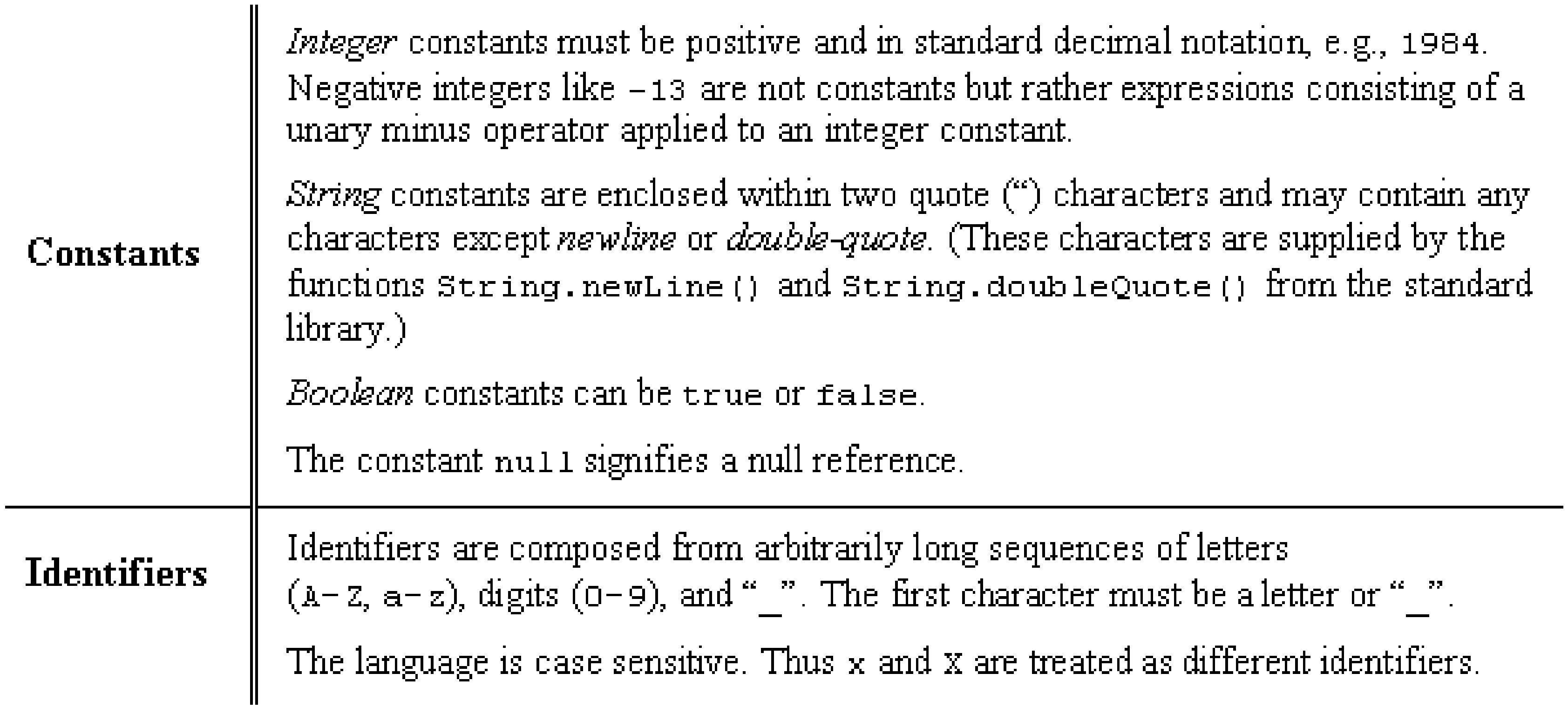
(for complete language specification, see the book).

Jack syntax



White space and comments	Space characters, newline characters, and comments are ignored.	
	The following comment formats are supported:	
	`//`	`Comment to end of line`
	`/*`	`Comment until closing */`
	`/**`	`API documentation comment */`
Symbols	`( )`	Used for grouping arithmetic expressions and for enclosing parameter-lists and argument-lists
	`[ ]`	Used for array indexing;
	`{ }`	Used for grouping program units and statements;
	`,`	Variable list separator;
	`;`	Statement terminator;
	`=`	Assignment and comparison operator;
	`.`	Class membership;
	`+ - * / & | ~ < >`	Operators.
Reserved words	`class, constructor, method, function`	Program components
	`int, boolean, char, void`	Primitive types
	`var, static, field`	Variable declarations
	`let, do, if, else, while, return`	Statements
	`true, false, null`	Constant values
	`this`	Object reference

Jack syntax (continues)



Constants	*Integer* constants must be positive and in standard decimal notation, e.g., `1984`. Negative integers like `-13` are not constants but rather expressions consisting of a unary minus operator applied to an integer constant. *String* constants are enclosed within two quote (") characters and may contain any characters except *newline* or *double-quote*. (These characters are supplied by the functions `String.newLine()` and `String.doubleQuote()` from the standard library.) *Boolean* constants can be `true` or `false`. The constant `null` signifies a null reference.
Identifiers	Identifiers are composed from arbitrarily long sequences of letters (`A-Z`, `a-z`), digits (`0-9`), and "_". The first character must be a letter or "_". The language is case sensitive. Thus `x` and `X` are treated as different identifiers.

Jack data types

<u>Primitive types</u> (Part of the language; Realized by the compiler):

- `int` 16-bit 2's complement (from -32768 to 32767)
- `boolean` `0` and `-1`, standing for `true` and `false`
- `char` unicode character ('a', 'x', '+', '%', ...)

<u>Abstract data types</u> (Standard language extensions; Realized by the OS / standard library):

- `String`
- `Array`

... (extensible)

<u>Application-specific types</u> (User-defined; Realized by user applications):

- `BankAccount`
- `Fraction`
- `List`
- `Bat / Ball`

. . . (as needed)

Jack variable kinds and scope

Variable kind	**Definition / Description**	**Declared in**	**Scope**
Static variables	**`static`** *type name1, name2, ... ;* Only one copy of each static variable exists, and this copy is shared by all the object instances of the class (like *private static variables* in Java)	Class declaration.	The class in which they are declared.
Field variables	**`field`** *type name1, name2, ... ;* Every object instance of the class has a private copy of the field variables (like *private object variables* in Java)	Class declaration.	The class in which they are declared, except for functions.
Local variables	**`var`** *type name1, name2, ... ;* Local variables are allocated on the stack when the subroutine is called and freed when it returns (like *local variables* in Java)	Subroutine declaration.	The subroutine in which they are declared.
Parameter variables	*type name1, name2, ...* Used to specify inputs of subroutines, for example: `function void drive (`**`Car c, int miles`**`)`	Appear in parameter lists as part of subroutine declarations.	The subroutine in which they are declared.

Jack expressions

A Jack *expression* is any one of the following:

- A constant
- A variable name in scope (the variable may be static, field, local, or a parameter)
- The keyword `this`, denoting the current object
- An array element using the syntax *arrayName*`[`*expression*`]`,
  where *arrayNname* is a variable name of type `Array` in scope
- A subroutine call that returns a non-void type
- An *expression* prefixed by one of the unary operators `-` or `~` :

-*expression*	(arithmetic negation)
~*expression*	(logical negation)

- An expression of the form *expression op expression* where *op* is one of the following:

+ - * /	(integer arithmetic operators)
& |	(boolean and and or operators, bit-wise)
< > =	(comparison operators)

- (*expression*) (an expression within parentheses)

Jack Statements

```
let varName = expression;
or
let varName[expression] = expression;
```

```
if (expression) {
    statements
}
else {
    statements
}
```

```
while (expression) {
     statements
}
```

```
do function-or-method-call;
```

```
return expression;
or
return;
```

Jack subroutine calls

General syntax: *subroutineName*(*arg0*, *arg1*, …)

where each argument is a valid Jack expression

Parameter passing is *by-value* (primitive types) or *by-reference* (object types)

Example 1:

Consider the function (static method): `function int sqrt(int n)`

This function can be invoked as follows:

```
sqrt(17)
sqrt(x)
sqrt((b * b) – (4 * a * c))
sqrt(a * sqrt(c - 17) + 3)
```

Etc. In all these examples the argument value is computed and passed by-value

Example 2:

Consider the method: `method Matrix plus (Matrix other);`

If `u` and `v` were variables of type `Matrix`, this method can be invoked using: `u.plus(v)`

The `v` variable is passed by-reference, since it refers to an object.

Noteworthy features of the Jack language

- The (cumbersome) `let` keyword, as in `let x = 0;`
- The (cumbersome) `do` keyword, as in `do reduce();`
- No operator priority:

  `1 + 2 * 3` yields `9`, since expressions are evaluated left-to-right;

  To effect the commonly expected result, use `1 + (2 * 3)`
- Only three primitive data types: `int`, `boolean`, `char`;
  In fact, each one of them is treated as a 16-bit value
- No casting; a value of any type can be assigned to a variable of any type
- Array declaration: `Array x;` followed by `x = Array.new();`
- Static methods are called `function`
- Constructor methods are called `constructor`;
  Invoking a constructor is done using the syntax *ClassName*`.new(`*argsList*`)`

Q: Why did we introduce these features into the Jack language?

A: To make the writing of the Jack compiler easy!

Any one of these language features can be modified, with a reasonable amount of work, to make them conform to a more typical Java-like syntax.

Jack program structure

```
class ClassName {

    field variable declarations;

    static variable declarations;

    constructor type { parameterList ) {
        local variable declarations;
        statements
    }

    method type { parameterList ) {
        local variable declarations;
        statements
    }

    function type { parameterList ) {
        local variable declarations;
        statements
    }
}
```

__About this spec:__

- Every part in this spec can apper 0 or more times
- The order of the `field` / `static` declarations is arbitrary
- The order of the subroutine declarations is arbitrary
- Each *type* is either `int`, `boolean`, `char`, or a class name.

__A Jack program:__

- Each class is written in a separate file (compilation unit)
- Jack program = collection of one or more classes, one of which must be named `Main`
- The `Main` class must contain at least one method, named `main()`

Jack standard library aka language extensions aka Jack OS

```
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)
}
```

Perspective

- Jack is an object-based language: no inheritance
- Primitive type system
- Standard library
- Our hidden agenda: gearing up to learn how to develop the ...
    - Compiler (projects 10 and 11)
    - OS (project 12).