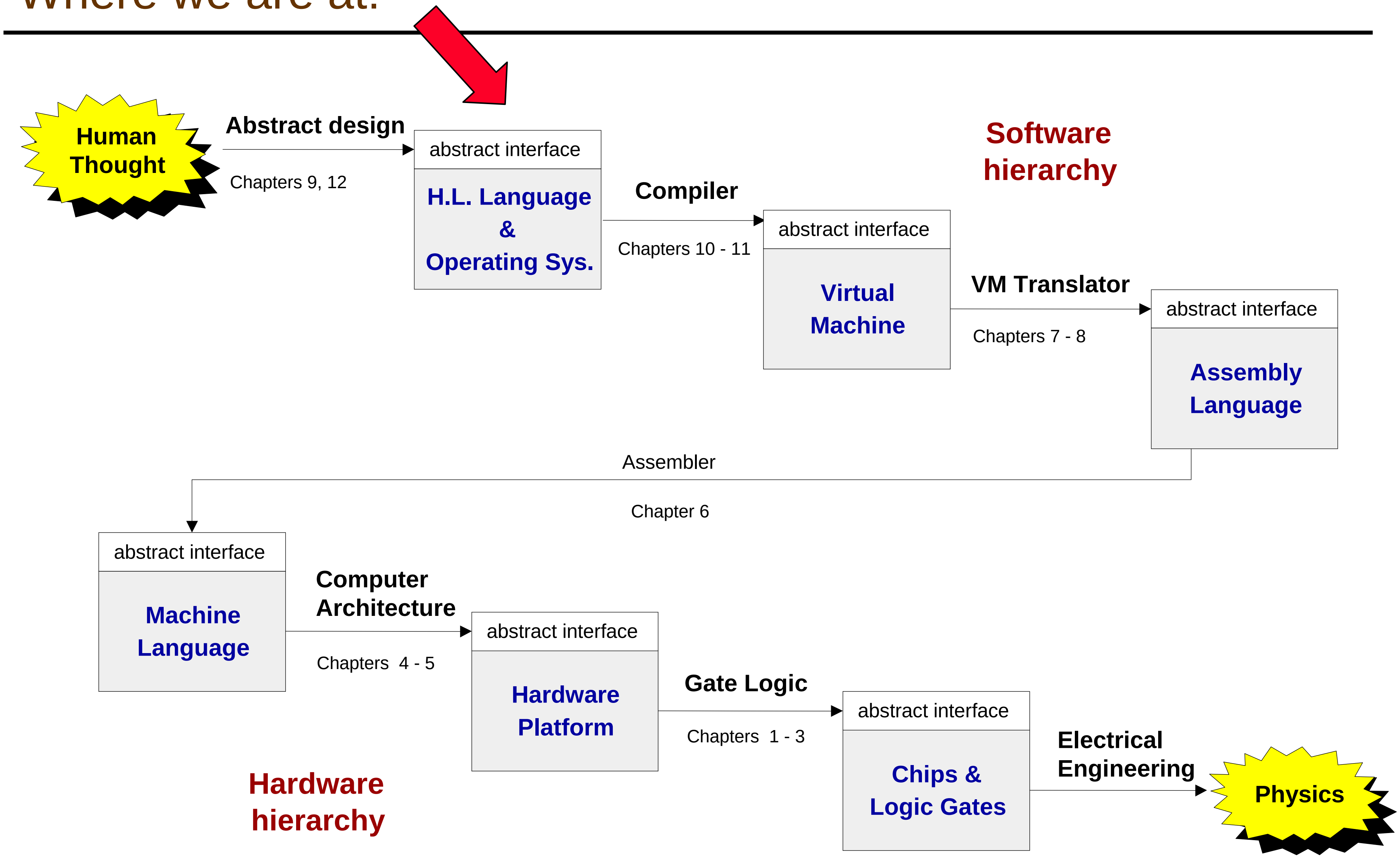
Chapter 9:
High Level Language

Where we are at:



Brief history of programming languages (some milestones)

- Machine language
- Assembler: symbolic programming
- Fortran: formula translation
- Algol: structured programming, dynamic memory
- Pascal, C: industrial strength compilers
- C++: OO
- Java, C#: OO done well
- Other paradigms.

The OO approach to programming

- Object = entity associated with properties (fields)
  and operations (methods)
- Objects are instances of *classes*
  E.g. bank account, employee, transaction, window, gameSession, …
- OO programming: identifying, designing and implementing classes
- Each class is typically:
  - A template for generating and manipulating objects

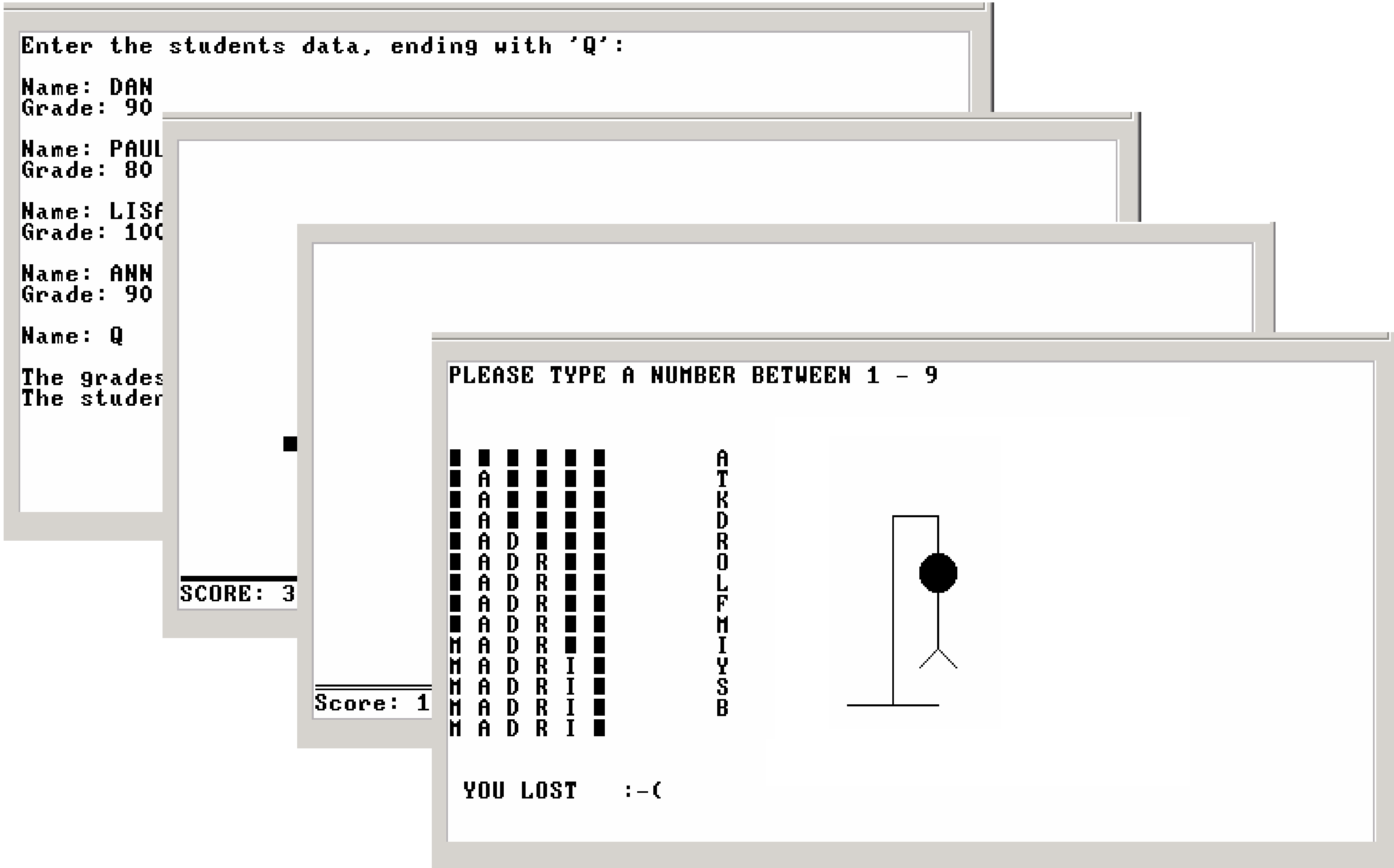
    and/or

  - A collection of related subroutines.

An OO programming language can be used for …

- Procedural programming
- Abstract data types
- Concrete objects
- Abstract objects
- Graphical objects
- Software libraries
- And more.

Jack: a typical OO language -- sample applications



Disclaimer

- Although Jack is a real programming language, we don't view it as an *end*
- Rather, we view Jack as a *means* through which we explain:
  - How to build a compiler
  - How the compiler and the language interface with an OS
  - How the topmost piece in the software hierarchy fits into the picture
- Jack's most important virtue: it can be learned (and un-learned) it in 1 hour.

Example 0: 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;
  }
}
```

- Java-like syntax
- Comments
- Standard library.

Example 1: procedural programming

```
class Main {

  /* Sums up 1+2+3+...+n */
  function int sum(int n) {
    var int i, sum;
    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, x;
    let n = Keyboard.readInt(”Enter n: ”);
    let x = Main.sum(n);
    do Output.printString("The result is: ");
    do Output.printInt(sum);
    do Output.println();
    return;
  }
} // Main
```

- Jack program = collection of one or more classes
- Jack class = collection of one or more subroutines
- Jack subroutine:
  - Function
  - Method
  - Constructor

  (the example on the left has functions only, as it is “object-less”)
- There must be one **`Main`** class, and one of its methods must be **`main`**.

Example 2: OO programming

```
class BankAccount {
  static int nAccounts;

  // account properties
  field int id;
  field String owner;
  field int balance;

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

} // BankAccount
```

```
...
var int sum;
var BankAccount b, c;

let b=BankAccount.new("Joe");
...
```

Example 2: typical OO programming (cont.)

```
class BankAccount {
  static int nAccounts;

  // account properties
  field int id;
  field String owner;
  field int balance;

  // Constructor ... (omitted)

  /* Deposits money in this account. */
  method void deposit(int amount) {
    let balance = balance + amount;
    return;
  }

  /* Withdraws money from this account. */
  method void withdraw(int amount){
    if (balance > amount) {
      let balance = balance - amount;
    }
    return;
  }

  // ... More BankAccount methods.
} // BankAccount
```

```
...
var int sum;
var BankAccount b, c;

let b=BankAccount.new(”Joe”);
do b.deposit(5000);

let c=BankAccount.new(”jane”);
let sum = 1000;
do b.withdraw(sum);
...
```

Example 2: typical OO programming (cont.)

```
class BankAccount {
  static int nAccounts;

  // account properties
  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;
  }

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

  // ... More BankAccount methods.
} // BankAccount
```

```
...
var int sum;
var BankAccount b, c;

// Construct and manipulate
// b and c ...

do b.printInfo();
do b.dispose();
...
```

Example 3: abstract data types (API + usage)

- Motivation: Jack has only 3 primitive data type: **int**, **char**, **boolean**

```
// An object representation of n/m where n and m are integers (e.g. 17/253).
field int numerator, denominator        // Fraction object properties
constructor Fraction new(int a, int b)  // Returns a new Fraction object
method int getNumerator()               // Returns the numerator of this fraction
method int getDenominator()             // Returns the denominator of this fraction
method Fraction plus(Fraction other)    // Returns the sum of this fraction and
                                        // another fraction, as a fraction
method void print()                     // Prints this fraction in the format
                                        // "numerator/denominator"
// Additional fraction-related services are specified here, as needed.
```

Fraction API

```
// Computes the sum of 2/3 and 1/5.
class Main {
  function void main() {
    var Fraction a, b, c;
    let a = Fraction.new(2,3);
    let b = Fraction.new(1,5);
    let c = a.plus(b);  // Compute c = a + b
    do c.print();  // Should print the text "13/15"
    return;
  }
}
```

Using the Fraction API (example)

- API = public contract
- Interface / implementation.

Example 3: abstract data types (implementation)

```
/** Provides the Fraction type and related services. */
class Fraction {

  field int numerator, denominator;

  constructor Fraction new(int a, int b) {
    let numerator = a;  let denominator = b;
    do reduce();  // If a/b is not reduced, reduce it
    return this;
  }

  method void reduce() {
    // Reduces the fraction - see the book.
  }

  function int gcd(int a, int b){
    // Computes the greatest common denominator of a and b. See the book.
  }

  method int getNumerator() {
     return numerator;
  }

  method int getDenominator() {
     return denominator;
  }

  // More methods follow.

}
```

Example 3: abstract data types (implementation cont.)

```
/** Provides the Fraction type and related services. */
class Fraction {

  // Fields, constructor, and methods from previous slide come here ...

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

  // More fraction-related methods come here: minus, times, div, etc.

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

Example 4: linked list

```
/** Provides a linked list abstraction. */
class List {
  field int data;
  field List next;

  /* Creates a new List object. */
  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.
```

```
class Foo {
  ...
  // Creates a list holding the numbers (2,3,5).
  function void create235() {
    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 coverage, see chapter 9 in 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 (cont.)

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

- Primitive:
  - **`Int`** 16-bit 2's complement (15, -2, 3, ...)
  - **`Boolean`** 0 and –1, standing for true and false
  - **`Char`** unicode character ('a', 'x', '+', '%', ...)
- Abstract data types (supplied by the OS or by the user):
  - **`String`**
  - **`Fraction`**
  - **`List`**
  - . . .
- Application-specific objects:
  - **`BankAccount`**
  - **`Bat / Ball`**
  - . . .

Jack data types: memory allocation

```
// This code assumes the existence of Car and Employee classes.
// Car objects have model and licensePlate fields.
// Employee objects have name and Car fields.
var Employee e, f;    // Creates variables e, f that contain null references
var Car c;            // Creates a variable c that contains a null reference
...
let c = Car.new("Jaguar","007")     // Constructs a new Car object
let e = Employee.new("Bond",c)      // Constructs a new Employee object
// At this point c and e hold the base addresses of the memory segments
// allocated to the two objects.
let f = e;  // Only the reference is copied - no new object is constructed.
```

- Object types are represented by a class name and implemented as a reference, i.e. a memory address
- Memory allocation:
  - Primitive variables are allocated memory space when they are declared
  - Object variables are allocated memory space when they are created via a constructor.

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 one of the following:

- A *constant;*
- A *variable name* in scope (the variable may be *static*, *field*, *local*, or *parameter*);
- The `this` keyword, denoting the current object (cannot be used in functions);
- An *array element* using the syntax *name*[*expression*], where *name* is a variable name of type `Array` in scope;
- A *subroutine call* that returns a non-void type;
- An expression prefixed by one of the unary operators – or ~:
  - *expression*: arithmetic negation;
  ~ *expression*: boolean negation (bit-wise for integers);
- An expression of the form *expression operator expression* where *operator* is one of the following binary operators:

+ – * /	Integer arithmetic operators;
& |	Boolean And and Boolean Or (bit-wise for integers) operators;
< > =	Comparison operators;

- **(***expression***) :** An expression in parenthesis.

- No operator priority!

Jack Statements

```
let variable = expression;
or
let variable [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(arg1, arg2, …)*
- Each argument is a valid Jack expression
- Parameter passing is *by value*

Example: suppose we have `function int sqrt(int n)`

This function can be invoked as follows:

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

Etc.

Jack subroutine calls (cont.)

```
class Foo {
  // Some subroutine declarations - code omitted
  ...
  method void f() {
    var Bar b;      // Declares a local variable of class type Bar
    var int i;      // Declares a local variable of primitive type int
    ...
    do g(5,7);          // Calls method g of class Foo (on this object)
    do Foo.p(2);        // Calls function p of class Foo
    do Bar.h(3);        // Calls function h of class Bar
    let b = Bar.r(4); // Calls constructor or function r of class Bar
    do b.q();           // Calls method q of class Bar (on object b)
    let i = w(b.s(3), Foo.t());  // Calls method w on this object,
                                 // method s on object b and function
                                 // or constructor t of class Foo
    ...
  }
}
```

Jack program structure

Class declarations have the following format:

```
class name {
    field and static variable declarations
    subroutine declarations
    // (a sequence of constructor, method,
    //   and function declarations)
}
```

Subroutine declarations have the following formats:

```
constructor type name (parameter-list) {
    declarations
    statements
}

method type name (parameter-list) {
    declarations
    statements
}

function type name (parameter-list) {
    declarations
    statements
}
```

- Each class in a separate file (compilation unit)
- Jack program = collection of classes, containing a **`Main.main()`**

Jack standard library = language extensions = 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 understand how to develop a:
  - Compiler (chapters 10-11)
  - OS (chapter 12).