TODO: insert a spiffy title page

(e.g. author of notes, course title, course instructor, date)

01 course overview

Object-oriented programming
  - Object
    - repository of data
  - Class
    - type of object
  - Method
    - procedure/function that operates on particular type of object (or class)
  - Inheritance
    - classes can inherit properties from more general classes
    - e.g. ShoppingList inherits from List the property of storing a sequence of items
  - Polymorphism
    - one method call can work on several different classes of objects
      even if the classes need different implementations
    - e.g. "addItem() on every kind of List, even though adding
      item to ShoppingList is different from adding item to ShoppingCart
  - Object-oriented
    - each object knows its own class & methods
    - e.g. each ShoppingList and ShoppingCart knows which addItem() it uses

Java
  - fetching classes
    - use one defined by someone else (lots exist in Java)
    - create one yourself
  - must declare any variable before using it
  - e.g.

```
    $ String myString; // myString[] <- variable in memory (nothing there initially)
    $ myString = new String(); // constructor that creates a new String object
                               // assignment operator causes myString to reference the new
String object
                               // myString[-]---->[  ] (a string object)
```

  - differences between Java & Scheme
    1 everything in Java has a type, and that type must be declared
    2 things must be compiled

```
          Scheme:                         Java:
      scheme program (.scm)          java program (.java)
            |                              |
            | eval                         | javac
            |                              |
            V                              V
         answer                      .class files
                                           |
                                           | java (JVM)
                                           |
                                           V
                                        answer
```

  - primitive types
    - variables with a primitive type are not references to objects

02 using objects

Objects and Methods

```
  $ String s1 = “Yow!”; // s1[-]--->[Yow!]
                        // (syntactic sugar: combine declaration and assignment)

  $ String s2 = s1;     // s1[-]--->[Yow!]<---[-]s2
                        // s1 and s2 reference the same object

  $ s2 = new String(s1); // s1[-]--->[Yow!] s2[-]--->[Yow!]
                         // copy constructor enables s1 and s2 to ref. dif. objs w/ same data

  - Strings are immutable

  - constructors

    - new String() constructs empty string (contains no characters)
    - “whatever” constructs string containing the characters in the quote
    - new String(str) takes an argument, str, and copies the chars contained in str

    - constructors always have the same name as their class
      (except for data enclosed in quotes, where Java Strings those)
    - constructors (except for quote data) always require the ‘new’ keyword

  $ s2 = s1.toUppercase(); // s2[-]--->[YOW!]

  $ s3 = s2.concat(“!!”);  // s3[-]--->[YOW!!!]
  $ s3 = s2 + “!!”;        // java implicitly calls concat()

  $ s4 = “*”.concat(s2).concat(“*”);
```

Java I/O Classes

```
  - Objects in System class interact with a user

  - System.out is a PrintStream object that outputs to the screen
  - System.in is an InputStream object that reads from the keyboard
    - These statements are actually shorthand for
      “System.blah is a variable that references an xStream object”

  - readLine method is defined for BufferedReader objects


  - to construct a BufferedReader, construct an InputStreamReader
  - to construct an InputStreamReader, construct an InputStream
  - to construct an InputStream, System.in is one!

  - InputStream reads raw bitstreams
  - InputStreamReader composes those bitstreams into characters (typ. 2 bytes long)
  - BufferedReader composes those characters into entire lines of text

  - this composure is based on modularity
    (separate objects for each logical step, so they are flexible to internal/external change)
```

Sample program:

```
import java.io.*;

class SimpleIO {
  public static void main(String[] arg) throws IOException {
    BufferedReader keyboard = new BufferedReader(
      new InputStreamReader(System.in));
    System.out.println(keyboard.readLine());
  }
}

//end program
```

- to use Java libraries other than java.lang, you must import them
- a Java program always begins at a method called main

03 defining classes

Classes
- contain fields (or, "instance variables")
  - variables stored in objects
- declaration
  - signified by 'class' keyword
    - e.g. "class Human { public int age; public String name; }"
- constructors, in declaration, don't have return values
  because they return the object they construct
- once a constructor is defined, regardless of its parameters,
  it negates the java-provided default constructor
- 'this'
  - invoking an object's method implicitly passes an extra parameter: this
  - 'this' is the object whose method is invoked
  - parameters and local variables take precedence in methods, so 'this' is used to fetch fields
  - its value cannot be changed (would trigger a compiler error)

Static
- keyword
- it denotes the variable it applies to is shared by the entire class of objects
- i.e. 1 static variable per class, not per object
- aka "class variable" (as opposed to objects' "instance variable" titling)
- can be referred to outside of the class (e.g. Humans.numberOfHumans)
- can apply to methods
  - does not implicitly take an object as a parameter
  - (there is no 'this')
  - useful when methods have no use referring to a particular object
  - e.g. printHumanCount()
- main method is always static since at initial run, prog has no objects to reference
- in static methods, THERE IS NO 'THIS'!!!!!

Lifetimes of variables
- local variables
  - (those declared inside methods)
  - they die when the method concludes
- instance variables
  - (non-static fields)
  - last as long as their object exists
    (as long as there exists a reference to reach that object via)
- class variables
  - (static fields)
  - last as long as the program is running

04 types and conditionals

Primitive types

- byte 8-bit integer [-128 ... 127]
- short 16-bit integer [-32768 ... 32767]
- int   32-bit integer [-2147483648 ... 2147483647] i.e. 2 billion
- long 64-bit integer [......]
  - should be the default go-to, rather than int, since
- float   32-bit floating point number
- double 64-bit floating point number
- boolean [“true”, “false”]
- char   [a character]

- explicit primitive type specification
  - overrides default
  - defaults: int and double
  - long x = 124L;
  - float f = 18.9f;

Object vs Primitive types

- contains... what?
  - obj: ref
  - prim: value

- how defined?
  - obj: class definition
  - prim: built into Java

- how created?
  - obj: ‘new’ keyword
  - prim: built into Java; e.g.: “6”, “3.4”, “true”

- how initialized?
  - obj: constructor
  - prim: default (usually zero)

- how used?
  - ojb: methods
  - prim: operators; e.g.: “+”, “*”

Operators for primitive numbers

- negation	-x
- addition	x+y
- subtraction	x-y
- multiplication	x*y
- division	x/y (dividing int by int -> fractional part dropped)
- remainder	x%y (integer only, plz)

- java.lang’s Math class

- absolute value	Math.abs(x);
- square root	Math.sqrt(x);
- sine	...

- Integer class

- parse int	Integer.parseInt(“1984”);

- Double class

- parse double	Double.parseDouble(“3.14”);

Type issues

- ints can be assigned to variables of longer types
  - e.g.

```
int i = 43;
long l = 43L;
l = i;           // fine
```

- assigning to shorter types freaks Java out
  - e.g.

```
i = l            // compiler error
```

- casting

```
i = (int) l;

double d = 23.7;
float f  = (float) d;

d = f;           // fine
```

Boolean value truth table

a	b	a && b	a || b	!a
f	f	false	false	t
f	t	false	true	t
t	f	false	true	f
t	t	true	true	f

Comparison operators

==	equivalence
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
!=	non-equivalence

Conditionals

- if-then-else clauses can be...
  - nested   (e.g. if statement inside an if statement)
  - daisy-chained (e.g. else immediately followed by another if)

- switch statements e.g.

```
switch (month) {
case 2:
  days = 28;
  break;          // jump to end of the switch clause
case 4:
case 6:
case 9:
case 11:
  days = 30;      // adjoining cases
  break;
default:          // executed in the event no case was realized
  days = 31;
  break;
}
```

```
Return
    - keyword causing a method to end immediately (control returns to the calling method)
    - provides means of delivering a specific value from the function
    - "function" in Java: a method whose return value is not void (it returns something)
```

05 iteration and arrays

```
While loop
  - the body is executed repeatedly, until the condition is no longer satisfied
  - the condition is more formally called the "loop condition" (& body "loop body")


For loop
  - syntax: for(initialize; test; next) { statements; }
  - the structure helps clarify the loop's logic


Arrays
  - every array is an object consisting of an indexed list of variables
  - each numbered variable can be a primitive type or a reference
  - in memory e.g.
             0 1 2 3
    c[-]--->[b|l|u|e]

  - syntax
    char[] c;           //reference to array (any length) of chars
    c = new char[4];    //woo!
    c[0] = 'b';
    c[3] = 'e';
    c[4] = 's';         //Java's going to be angry when it sees what you've done
                        // runtime error - array index out of bounds exception

    * char c[]; is also a valid declaration. those brackets can go both ways

  - c.length: field indicating length of array (read-only; writing to it -> compile-time error)


Multi-dimensional arrays

  - 2 dimensional: an array of references to arrays
```

06 iteration and arrays

```
Automatic array construction

    int[][] table = new int[x][y];      // creates x arrays of y ints

  - Initializers

    Human[] b = {kayla, rishi, new Human("Paolo")};
    int[][] c = {{7,3,2},
                 {x},
                 {8, 5, 0, 0},
                 {y + z, 3}};

    int[] a, b, c;                      // all variables reference arrays
    int a[], b, c[][];                  // c-style declaration, where each varaible is mapped
                                        // to its specific dimension of array
    int[] a, b[];                       // you're sick! but it works; a is 1-D, b is 2-D
```

- array of objects
  when declaring an array of objects,
  Java will not create each object (must construct them in a loop)

Do loop

- while loop, but with one guaranteed loop body execution
- useful to make entry point more natural

- e.g.

```
do {
  s = keybd.readLine();
  process(s);
} while (s.lenth() > 0);

//           read
// enter --> ^  |
//  exit <-- |  V
//          process
```

- trying to keep the loop's entry and exit points at the same location can be tough
  e.g. with there, process is called with that empty string that denotes loop termination

- break

  break exits the innermost “switch” or loop enclosing the break

  break DOES NOT break if bodies

```
//           read
// enter --> ^  | --> exit
//           |  V
//          process
```

  (a “time-and-a-half” loop)

  with no break statement:

```
s = keybd.readLine();
while (s.length() > 0) {
  process(s);
  s = keybd.readLine();
}
```

  with break statement:

```
while (true) {
  s = keybd.readLine();
  if (s.length() == 0) {
    break;                  // break statement exits loop in middle of loop body
  }
  process(s);
}
```

```
    - pros/cons

      - w/ no break
        - duplicated code
          - s = keybd.readLine() is duplicated,
            and logically, could be a much larger segment of code
          - maintainability = worse (DRY violated)
        - if delegating the "readLine()" notion to subroutine for DRY abidance,
          procedure loses sight of local variables

      - w/ break
        - obfuscated (loop isn't aligned with natural endpoint)
```

More subtleties of flow control statements

- labels can be applied to conditions to denote where they are logically

- this can be intricate, so e.g.

```
    test:
      if (x == 0) {
      loop:
      while (i < 9) {
        stuff: {
          switch (z[i]) {
          case 0: break;
          case 1: break stuff;
          case 2: break loop;
          case 3: break test;
          case 4: continue;
          default: continue loop;
          }
          statement1();
        }
        statement2();
        i++;
        // location 3
      }
      statement4();
    }
    statement5();
```

- continue
  - only applies to loops
  - does not exit loop
  - another iteration may commence (if condition of while/do/for is true)

Constants

- 'final' keyword: value that can never be changed
- bad form: using magic numbers
- good form e.g.
  public final static int FEBRUARY = 2;

- for an array x, x.lenght is a final field, hence its read-only nature

07 linked lists

Lists

- raw arrays ain't no good

  - inserting item at beginning or middle -> time proportional to length of array
  - fixed length

- Linked List (recursive data type)

  - comprised of nodes

  - Node
    - stores an item
    - stores a reference to the next node

  - advantages over array lists

    - inserting item into middle of list -> constant time
      *** IF you have a ref to the previous node

    - no size limitation, aside from what the memory will fit

  - disadvantages over array lists

    - acquiring references to nth item -> time proportional to n
      (vs. constant time with arrays)

- List class motivation

    1 inserting new item at beginning of list

      - no concept of the head of a list, so references to the list
        must keep track of its composure on their own basis

        - e.g. x, y both ref. a list,
          and x adds an item to the front,
          y don't know 'bout that

    2 representing an empty list??

      - can't just do null, as method calls will upset the java overlord

08 linked lists

Privacy

- private method or field: invisible and inaccessible to other classes

- privacy motivation
  1 to prevent data from being corrupted by other classes
  2 to enhance impl's flexibility by preventing other classes from depending on its details

Interface

- prototypes for public methods (occasionally, public fields)
- descriptions of those public methods' behaviors (in plain English)

Abstract Data Type (ADT)

- a class (or multiple classes) with a well-defined interface
- ... whose implementation details are hidden from other classes
- ADTs allow enforcement of invariants
- not all classes are ADTs (some just store data with no invariants)

Invariant

- fact about a data structure that is always true (sans bugs!)
- e.g. a Date object always representing a valid date

The SList ADT

- another advantage of the SList class: invariant enforcement

  1 size field is always correct
    (size == cardinality of items reachable by beginning at head and following next's)

  2 list is never circularly linked

- this enforcement exists ONLY BECAUSE only SList methods operate on the lists

  - the fields of SList (head, size) cannot be tampered with because they are private
  - individual nodes cannot be tampered with because no SList method returns an SNode

Doubly-Linked Lists

- insertion/deletion at front of an SList is easy,
  but @ end of an SList, things are tougher

- I mean, don't you want traversal in either direction?

- I won't continue to elaborate on the motivation of Doubly-Linked lists

- They're just great. They beat out SLists 95% of the time

- insert/delete items at both ends -> time proportional to constant

- implementation may make use of a sentinel

  - a special node that does not represent an item

```
Invariants of Doubly-Linked List via sentinel

  1 for any DList d,      d.head != null
  2 for any DListNode x,  x.next != null
  3 for any DListNode x,  x.prev != null
  4 for any DListNode x,  if    x.next == y,
                          then  y.prev = x
  5 for any DListNode x,  if    x.prev == y,
                          then  y.next = x
  6 for any DList d,      d.size == the # of DListNodes in the list, discounting the sentinel

  7 for an empty DList d, sentinel node’s prev & next point to itself
```

09 stack frames

Heap

- stores all objects
  - arrays
  - class variables

Stack

- stores all local variables
  - parameters

Method invocation

- when method is called, Java creates a stack frame (aka activation record)
- stack frame stores the parameters and local variables
- JVM maintains a stack of stack frames
  - typ. capacity for a few thousands of stack frames
  - need to watch recursive growth rate & consider iterative alternatives
- when method finishes, its stack frame is erased permanently

Debug help

- in java.lang, Thread.dumpstack() method can be useful
  (gives all the stack frames currently on the stack)

Parameter passing

- all parameters are passed by value (copied)
- therefore, passing references to objects (e.g. variables of non-primitive types) is the means of achieving other languages' "pass by reference" paradigm

Binary search

- search a sorted array for value "findMe"
- if we find "findMe", return its array index;
- otherwise, return FAILURE
- recursion base cases

  1 findMe = middle element: return its index (terminates)
  2 subarray has length zero: return FAILURE

- runtime
  - assuming n elements, each recurse divides n by 2
    -> log(base=2, val=n)

Logarithm review

- Data Structures and Algorithms in Java
  - Goodrich, Tamassia
  - Section 4.1.2 (logarithm overview)
  - Section 4.1.7 (exponentiation overview)

Scope & recursion

- scope of a variable: portion of program where the variable is accessible

- class variables

  - in scope everywhere in class
    (except when overwritten (shadowed) by local variables / parameters)

  - can be fully qualified to access despite shadowing (disambiguates reference)
    (un-shadowable)
    - e.g. System.out

  - if public, then they're in scope everywhere

- instance variables

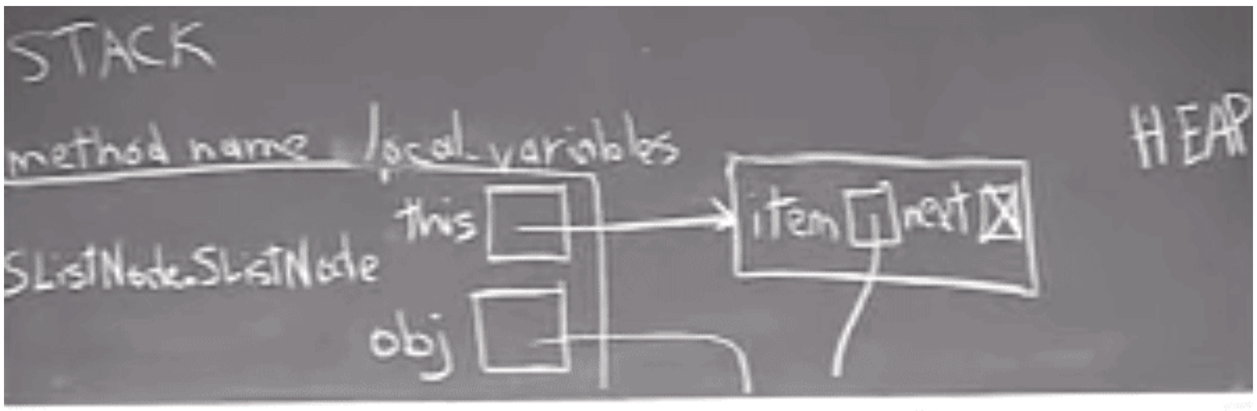
  - in scope in non-static methods of the class
    (except when shadowed)

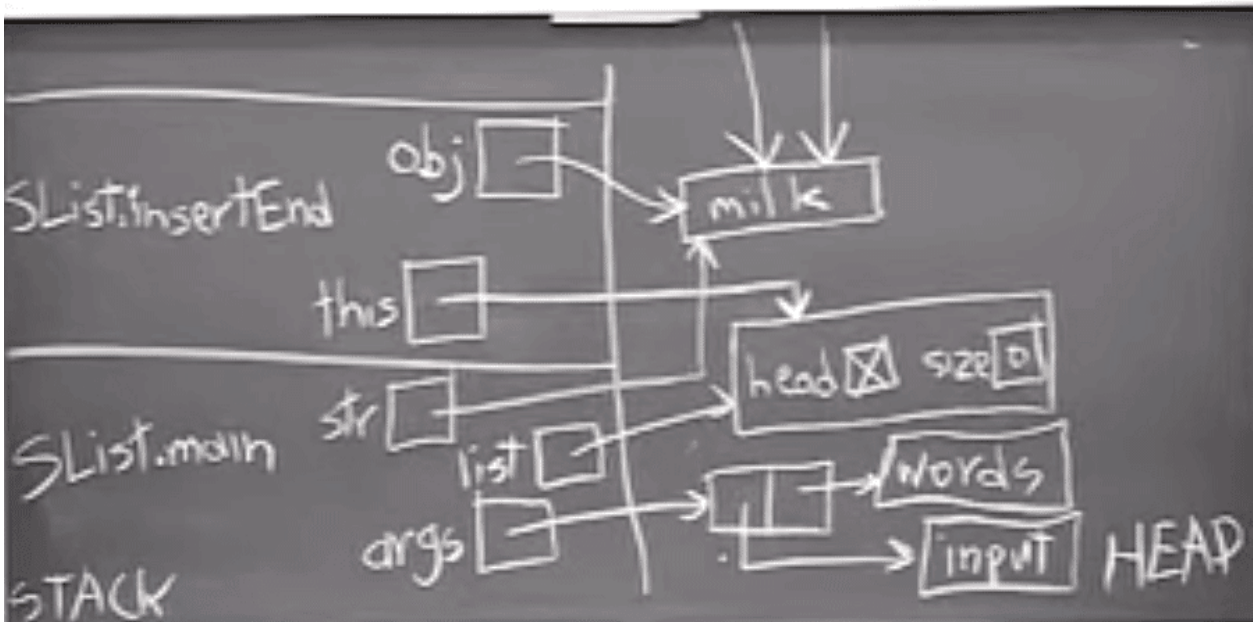
  - can also be fully qualified
    - e.g. kayla.name, this.item

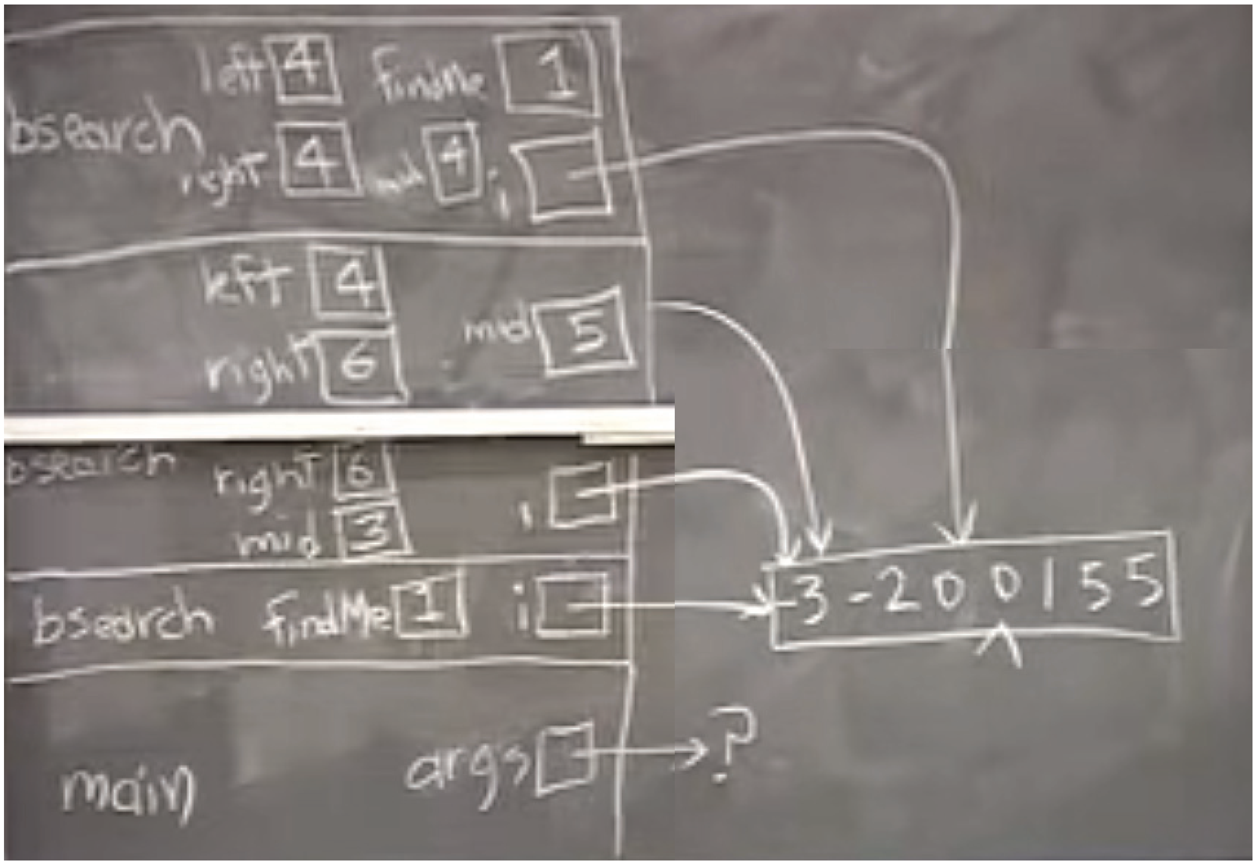
  - if public, then they're in scope everywhere

- local variables & parameters

    - only in scope inside the method (and more specifically, the block) that defines them
      (only for topmost stack frame)


STACK
method name
local variables
HEAP
SListNode.SListNode
this
obj
item
next


SList.insertEnd
obj
this
milk
head
size 0
SList.main
str
list
args
words
input
HEAP
STACK


bsearch
left 4
findMe 1
right 4
mid 4
i
left 4
right 6
mid 5
bsearch
right 6
mid 3
i
bsearch
findMe 1
i
-3 -2 0 0 1 5 5
main
args
?

10 testing

Equals

- equals(): a method that every class has (even if not explicitly defined)

- default behavior checks if 2 references point to the same obj.
  e.g. r1.equals(r2); // true if r1 and r2 point to same obj., false if otherwise

- typically overriden to compare equality of content, not of object instance
  e.g. String: s1 - “Hi” -> s1.equals(“Hi”) -> true
  * s1 == “Hi” -> false

Equality levels

1 Reference equality
  equals() does exactly what == does (checks if ref’d obj’s are the same obj.)

2 Shallow structural equality
  fields are ==

3 Deep structural equality
  recursive check of fields’ equality

4 Logical equality
  semantic perspective, looking into nature of fields
  e.g. Fractions 1/3 and 2/6 are logically equal
  e.g. Sets {a, b} and {b, a} are logically equal

Testing

1 Modular testing

  - test drivers

    - they call the code & check the results
    - location
      - often put in main() method,
        since a class’s main() is rarely the entry point of the program
      - if class *is* entry point for program, need an explicit test driver

      e.g.

```
public class MyProgram {
  // test code resides here for access to private attributes
  public static void TestDriver() { ... //test code ... }
}

// separate class for invoking the test driver method
public class TestMyProgram {
  // runnable via ‘java TestMyProgram’
  public static void main(String[] args) {
    MyProgram.TestDriver();
  }
}
```

```
    - stubs

      - bits of code called by the code being tested
      - typ. short

      - motivation

        1 a tested method must call another, non-existing method
          so stub fills in for that missing method

        2 a bug is out there, but its location is unknown
          replace the callee with a stub to find if
          the bug lies in the calling method or the called method

        3 good hook for simulating test-case data
          (especially regarding exceptional edge-cases)
          and hey, it's repeatable


  2 Integration testing

    - testing interactions between modules

    - best way to head off the need for this
      - define interfaces well
      - ensure no ambiguity exists in the descriptions of the interfaces' behaviors

    - learn to use the Java debugger


  3 Result verification

    - data structure integrity checkers
      - inspect the data structure to determine upholdance of invariants

    - algorithm result checkers
      e.g. after a sorting algorithm, checking that each number in a list is <= the next


  4 Regression testing

    - rest suite coded up that can be re-run every time changes are made to the code

    - checks that the changes have not modified the desired behaviors of the program

    - good because it localizes where bugs were introduced (the diff between prev and cur)

    - good because it notifies early the programmer who caused the bug
      (code diff still fresh in their mind)


Assertion

  - code that tests an invariant or result

  e.g. assert(x == 3); // checking at a point where it's assumed x == 3 that indeed, x == 3
  e.g. assert(list.size = list.countLength()): "wrong SList size: " + list.size;

    - will print out a colon-preceded string as the error message
```

- assertions can be switched off and on

```
$ java -ea SList # assertions enabled
$ java -da SList # assertions disabled
```

- assertion invocations should never contain logic the program needs to run normally (since their execution is not guaranteed)

11 inheritance

Inheritance

- subclasses (key to oop)
  - means of modifying the superclass it inherits from
    1 declare new fields
    2 declare new methods
    3 override old methods with new implementations
  - cannot override constructors
  - triggered via ‘extends’ keyword
- constructors
  - e.g. with subclass TailList, superclass SList

    ```
    TailList(); // first, java executes SList() constructor
    ```
- example creation

  ```
  SList s = new TailList(); // great!

  TailList t = new SList(); // no, you jerk! We wanted a tail.
                            // any anticipation of a TailList will be >:(
                            // if you give it an SList only
  ```

Protected

- privacy keyword
- private actually hides the field/method it applies to from subclasses
- demotes privacy of fields/methods s.t. subclasses can access them

Class hierarchies

- everything in Java inherits from Object (it is inexorable)
- subclassing is transitive (if A sub’s B, B sub’s C, then A sub’s C)
- tree typology is enforced; no cycles in inheritance scheme

Dynamic method lookup

- every X subclassing Y is, in fact, a Y (liskov sub. principle)

- static type
  - the type of a variable given to it at its declaration
- dynamic type
  - the type of a variable pulled from the class it references
  - can change dynamically

- example differentiating static and dynamic type

  ```
  SList s = new TailList();
  ```

  - s's static type: SList (this is what we told Java s is)
  - s's dynamic type: TailList (this is what s is pointing to)

- when an overridden method is invoked,
  Java calls the method for the object's dynamic type
  - Java doesn't heed the static type
  - called dynamic method lookup
  - e.g.

    ```
    // dynamic method lookup leveraging subclass's overridden method
    SList s = new TailList();
    s.insertEnd(obj);             // invokes TailList.insertEnd()

    // dynamic method lookup just going the general route, as static type == dynamic type
    s = new SList();
    s.insertEnd(obj);             // invokes SList.insertEnd()
    ```

  - motivation
    - e.g. w/ a method written to sort an SList, via only method calls (no field manip.)
      -> TailList can leverage that written method
    - Liskov sub. principle can be leveraged

12 abstract classes

```
Subtleties of inheritance

  - given TailList subclassing SList, some example scenarios

    1 new method in TailList called eatList()

      TailList t = new TailList();
      t.eatList();                      // fine

      SList s = new TailList();
      s.eatList();                      // compile-time error

      - s is static-type'd as an SList
      - java doesn't consider the dynamic type here
        - the static type doesn't inherently contain an "eatList()" method
        - therefore, dynamic method lookup is not leveraged to discover the proper method


    2 assignment between variables

      SList s;
      TailList t = new TailList();

      s = t;                            // fine
      t = s;                            // compile-time error

      - java doesn't consider the dynamic type here
        - it just knows you're trying to give a more specific variable
          a more general object value
        - it gets afraid, much like when it fears precision loss,
          that the good subclass bells 'n whistles will be lost
        - casting makes all the difference

      t = (TailList) s;                 // fine

      s = new SList();
      t = (TailList) s;                 // fine by Java, but of course, screwy (run-time error)

      int x = t.nth(1).intValue();      // nth returns Object, whom intValue()
                                        // is not defined for (compile-time error)

      int x = ((Integer) t.nth(1)).intValue(); // fine (needs parentheses to cause precedence
                                               // to deliver the valid syntax)

      - CASTS CHANGE THE STATIC TYPE of an expression
        (not the static type of a variable)

      - calling a method causes Java to invoke dynamic method lookup,
        which only considers the dynamic type


    3 figuring out the type of an object

      - instanceof operator returns the dynamic type of the invoking object
        ** the operator has a lower-case 'o' in 'of'

      if (s instanceof TailList) {      // false if f is null or doesn't reference a TailList
                                        // or subclass of TailList
        t = (TailList) s;
      }
```

Abstract classes

- denoted via keyword ‘abstract’ in class definition statement

- a class whose sole purpose is to be extended

- e.g. an ADT with no/an incomplete implementation

e.g.

```
// note: while impl is incomplete, this remains an ADT
public abstract class List {

  protected int size;

  public int length() {
    return size;
  }

  // ‘abstract’ keyword mandates subclasses implement this method
  // provides visibility of this method for objects typed as Lists
  public abstract void insertFront(Object item);

}
```

- cannot be used to instantiate a List object

- can be used to declare a List variable
  (e.g. a reference that will point to a subclass of List)

e.g.

```
List myList;                        // fine
myList = new List();                // compile-time error
```

e.g.

```
public class SList extends List {

  // inherits ‘size’ field

  protected SListNode head;

  // inherits ‘length()’ method

  // note lack of ‘abstract’ keyword, as it’s a true implementation
  public void insertFront(Object item) {
    head = new SListNode(item, head);
    size++;
  }

}
```

- non-abstract (sometimes ‘concrete’) classes’ restrictions

  - can’t contain an abstract method
  - can’t inherit an abstract method without providing an implementation

- abstract methods leverage polymorphism

  - callers needn’t be considered from a dynamic-type perspective
  - as long as they’re concrete, their appropriate dynamic method impl. will be used

  - e.g. a List sorter could, if abiding by List’s interface, sort any type of List

Subclasses of List

- SList, DList, TailList

- TimedList (records time spent doing operations)

- TransactionList (logs all changes on a disk in event of hard failures)

- module layering e.g.

```
[Application]
      |
      V
[ListSorter]
      |
      V
 [List ADT]
```

  - the application, not the list sorter, chooses what kind of list is used

Java interfaces

- ‘interface’ keyword, ‘implements’ keyword

- different from the c.s. concept of “interfaces”
  (public method prototypes & corresponding behaviors)

- much like an abstract class, with 2 differences

  1 while classes can only inherit from 1 class,
    multiple interfaces can be implemented

  2 Java interfaces lack features that abstract classes have
    - can’t implement any methods (must all be abstract)
    - can’t include any fields (except ‘static final’ constants)

- therefore, interfaces contain only method prototypes and constants

- motivation: a sort of means to achieve multiple inheritance (a weak version)

e.g.

```
// must be in Nukeable.java, as the case with a Java class
public interface Nukeable {
  public void nuke();
}

// defined in java.lang (typ. used for general sorting algo’s)
public interface Comparable {
  // the method one must define to meet the Comparable interface’s definition
  public int compareTo(Object o);
}
```

```
  public class SList extends List implements Nukeable, Comparable {

    // methods achieving your SList's contractual obligations
    // e.g.
    public void nuke() {
      head = null;
      size = 0;
    }
  }
```

- interfaces can be the static type of a variable

  e.g.

```
    Nukeable n = new SList();

    Comparable c = (Comparable) n;    // cast needed as not every Nukeable is a Comparable
```

- sub-interfaces can have multiple super-interfaces

  e.g.

```
    public interface NukeAndCompare extends Nukeable, Comparable { ... }
```

- syntactic diff. btwn subclassing/extension/sub-interfacing/etc.....

```
    - class inheriting interface:       'implements'

    - class inheriting another class:   'extends'

    - interface inheriting interface:   'extends'
```

13 java packages

```
Source code file behavior

  e.g.

    given SList.java: {SList class, Nukeable interface}
    javac SList.java -> {SList.class, Nukeable.class}

    - though the 1 java file contains both constructs,
      compilation distributes them into separate .class files

  - public classes must be declared in files named after their class


Java packages

  - package: collection of classes, Java interfaces, & sub-packages

  - typ. cooperate to provide 1 or whatnot ADT,
    with a willingness to interdepend because author tends to be same

  - advantages

    1 packages can contain hidden classes not visible outside package

    2 classes can have package-only visibility for fields & methods

    3 dif. packages can have classes with same name


Package use

  - as fully qualified name: java.lang.System.out.println("big 'ole name right there");

  - as via 'import' keyword: import java.io.File;
                             // File now visible (needn't qualify it further than that)

    - can import entire package: import java.io.*;
                                 // everything in java.io package now visible

  - every program implicitly imports java.lang.*

  - name collision req's qualifying at a high-enough level to disambiguate references

    e.g.

      java.awt.Frame.add(photo.Frame.canvas);

  - Java must be aware of where to locate package contents

    e.g.

      - must know for x.y.z.Class where x/y/z/Class.class is located

      - under Unix, set environmental variable named CLASSPATH
        to include path to relevant directories

        - colons separate directories

        - printenv CLASSPATH # reveals value of CLASSPATH
```

Building packages

- e.g.

```
// in list/SList.java

package list;

public class SList {
  SListNode head;
  int size;
}


// in list/SListNode.java

package list;

class SListNode {
  Object item;
  SListNode next;
}
```

- ‘package’ protection = default for package-rooted class attributes

    - between private and protected
      (more private than protected, less private than private itself)
    - visible to any class in same package
    - invisible to anything outside package (i.e. not in the package’s directory)

    - files outside only see public classes/methods/fields
        - except if they’re a subclass of a package class
        - in that case, they have visibility for protected entities

- ‘package’ classes can go in any *.java file

  i.e. they needn’t exist in their own separate file named after their class

Compilation & running

e.g.

```
$ javac -g list/SList.java      # must be outside the package dir; -g sets debug option
$ java list.SList               # runs the test code

$ javac -g list/*.java heap/*.java *.java # multiple compilation args allowed for javac
                                          # (which det’s interdependencies among args)
```

Protection level summary

	term	visibility - in same package	in a subclass	in anywhere
1	‘public’	yes	yes	yes
2	‘protected’	yes	yes	---
3	package	yes	---	---
4	‘private’	---	---	---

14 exceptions

```
Exceptions

  - run-time error: Java “throws an exception” ...

    - Exception is a class, so throwing one is throwing an object
    - catching an Exception “prevents” the error,
      as it can be examined and handled to see how to proceed

  - motivation

    1 surviving errors
      e.g.

        a prog. that reads a file so it can respond to the contents
        but oh no, file does not exist (or is otherwise inaccessible)

        solution: print error message and proceed

      - solution impl

        - place code capable of causing the exception into a try block

        e.g.

          try {
            f = new FileInputStream(“xyz.fakefile”);
            i = f.read();
          }
          catch (FileNotFoundException e1) {          // actually a variable declaration
            whine(“Foiled!!!”);                       // (of a static variable ‘e1’, which
                                                      // ref’s the exception object thrown)
          }
          catch (IOException e2) {
            f.close();
          }
          catch (Exception e3) {                      // global catcher (though there is 1
                                                      // Exception more generic)
          }

      - execution: the ‘try’ block is executed
        - if no exception is thrown, skip ‘catch’ clauses
        - if an exception is thrown, seek first ‘catch’ clause that
          matches the type of exception thrown
          - ‘matching’ -> exception is same class or subclass of the
            specified variable’s type in the catch clause
          - code executed inside ‘catch’ block called “exception handler”
          - after exception handler concludes, skip over any remaining exception handlers


    2 escaping a sinking ship

      - throw your own exception

      - must define own exception
      e.g.

        // albeit lacking new logic, does distinguish self by virtue of its class
        public class ParserException extends Exception { }

        public ParseTree parseExpression() throws ParserException {
          ...
          if (somethingWrong) {
            throw new ParserExpression();
          }
          ...
        }

      - differences from return

        1 needn't return anything (which helpful when logically, nothing exists to return)

        2 exceptions can fly multiple methods down the stack until caught


    - methods can be able to throw multiple exceptions


e.g.

  public ParseTree parse() throws ParserException, BadCodeException {

  }

  public void compile() {
    ParseTree p;
    try {
      p = parse();
      p.toByteCode();
    }
    catch (ParserException e1) { }
  }


  - if a method on the top of the stack throws an exception,
    java burns down the stack frames until it finds someone willing to catch the exception


Checked v. unchecked

  - Java distinguishes exceptions upon the basis of checked/unchecked

  - Checked exceptions must be explicitly noted as thrown

  - all exceptions inherit from the class Throwable

  - Throwable has 2 direct children

    1 Exceptions

      e.g. RunTimeException (children: NullPointer, ArrayIndex...)
           ParserException

    2 Error

      e.g. running out of memory
      - typ. more serious than Exceptions
```

- unchecked exceptions are either RunTimeExceptions or Errors

  - per ‘unchecked’ categorization, don’t require “throws” declaration
  - this is because they are rather globally applicable

- options for when a method calls a method that throws a checked exception

  1 catch the exception and handle it

  2 allow it to propagate through the caller,
    which must in turn throw that exception, itself

  - one of these must be chosen for the compiler to give your code the go-ahead

15 shadowing and types

Finally

- ‘finally’ keyword

- optional part of try clause

e.g.

```
try {
  statementX;
  return 1;
}
catch (SomeException e) {
  e.printStackTrace();
  return 2;
}
finally {
  // executed no matter the outcome of the try block
  file.close();
  return 3;
}
```

- if ‘try’ statement begins executing

  - ‘finally’ clause will be executed at the end, no matter what
  - so, in above example, finally’s return statement makes try and catch’s returns pointless

  - again, based on e.g. above, possible flows

    1 statementX causes SomeException
      a) ‘catch’   clase executes
      b) ‘finally’ clause executes

    2 statementX causes OtherException
      a) ‘finally’ clause executes
      b) exception continues down the stack in search of a catcher’s mitt

Exceptions on top of Exceptions

- if a catch clause throws an Exception, the program behaves as you’d expect
  - ‘finally’ does still get executed before the Exception begins its journey

- also, any nesting of try/catch/finally is permitted

```
    - if finally clause throws an Exception
      - if no catch block nested in there, it replaces the older Exception
      - this does interrupt (permanently) the execution of the finally clause


Exception constructors

  - most Throwables have at least 2 constructors
    1 one that takes no parameters
    2 one that takes a String

  e.g.

    class MyException extends Exception {

      public MyException() { super(); }

      // String -> opportunity to represent error message
      // printed if Exception propagates out of main() or
      // if caught and manually printed earlier
      public MyException(String s) { super(s); }
    }


Field shadowing

  - a danger given the nature of inheritance

  - super and sub classes can have a field of the same name

    - not nice like method overriding
      - dynamic method lookup makes use of dynamic type to select appropriate method
      - but with shadowed field, *static* type used to select field

  e.g.

    class Super {
      int x = 2;
      int f() { return 2; }
    }

    class Sub extends Super {
      int x = 4;
      int f() { return 4; }
      void g() {
        int i;
        i = this.x;             // i == 4
        i = ((Super) this).x;   // i == 2
        i = super.x             // i == 2 - shorthand for ((Super) this)
    }


    Sub sub = new Sub();
    Super supe = sub;       // sub[-]---> [ sub.x[4] super.x[2] ] <---[-]supe

    int i;

    i = supe.x;             // i == 2
    i = sub.x;              // i == 4

    i = ((Supe) sub.x);     // i == 2
    i = ((Sub) supe.x);     // i == 4
    // as per dynamic method lookup, dynamic type of Sub yields that class’s method

    i = supe.f();            // i == 4
    i = sub.f();             // i == 4

    i = ((Supe) sub).f();    // i == 4
    i = ((Sub) supe).f();    // i == 4
```

Static methods

  - follow same shadowing rules as fields
  - they use the static type to determine what method to invoke
  - this is because there is no referenced object whose dynamic type can be observed

Final methods and classes

  - ‘final’ keyword
  - final methods can’t be overridden
  - final classes can’t be extended

  - attempting those -> compile-time error
  - final methods can be invoked more quickly, because they won’t depend on dynamic method lookup

  - final methods can be used to override other methods, themselves

Simplified for

  - “for each” construct

  e.g.

```
    int[] array = {7, 12, 3, 8, 4, 9};
    for (int i : array) {
      System.out.print(i + “ “);
    }
```

  - requires the object looped through implement the Iterable interface

17 encapsulation

Kluge

- a workaround. quick-and-dirty solution
- difficult to extend and maintain
- kluges often violate encapsulation

Module

- set of methods that work together to perform some task
- modules and encapsulation
  - modules are “encapsulated” if...

    1 their implementation is hidden
    2 they’re accessible only through a documented interface

    - data can ONLY go in and out through the module’s interface
- ADTs are types of modules

Encapsulation pros

1 the implementation of the module is independent of the functionality
  - armed only with interface documentation, a new implementation can be developed

2 prevention of corrupting the module’s internal data
  - dramatically reduces debugging time

3 ADTs have the power to ensure their invariants are enforced

4 once interfaces are defined, programmers can work in teams
  - parts can be developed independently

5 documentation and maintainability
  - its behavior is much more clearly defined,
    so it is understandable by those who didn’t write it

Module interfaces

- serve as contracts between module writers
- specify how they will communicate

Enforcing encapsulation

- most languages reqire self-discipline
- Java’s packages & privacy levels
  - though note: packages can often contain multiple modules
    - e.g. when multiple modules operate on same class, convenient to store in 1 package

Module conceptual size

```
  - module  may contain  few or many methods
  - module  may comprise several     classes
  - class   may comprise several     modules
  - package may contain  1 or many   modules

  - so the motivation of module size: module is defined by clean interfaces
    - whatever it takes to make its interface clean, that's what it should comprise
```

Module interface documentation

```
  - list the prototypes of methods visible to external callers
  - behavior comment: describe every parameter and return value
  - complete & unambiguous
  - unusual & erroneous input / circumstances
```

19 asymptotic analysis

Big-Oh notation

```
  - O(f(n)) is the set of ALL functions T(n) that satisfy:

    there exist positive constants c and N s.t.
    for all n >= N, T(n) <= cf(n)

  - almost never includes positive constant factors of n

  - serves as an upper bound to a function
    - can indicate a function is speedy
    - cannot indicate a function is slow (but hey, that's what Big-THETA is for)
      e.g. n is a member of O(n^3), but that doesn't mean it's slow

  - if function under consideration is a combination of other functions,
    only the fastest-growing function need be considered

    it highlights the dominating term
```

Common Big-Oh sets

```
  (sorted from smallest to largest)

    function        common name
    ----------      -----------
    O(1)            constant
  c O(log n)        logarithmic
  c O(log^2 n)      log-squared
  c O(n^.5)         root-n
  c O(n)            linear
  c O(nlog n)       n-log-n
  c O(n^2)          quadratic
  c O(n^3)          cubic
  c O(2^n)          exponential
  c O(e^n)          exponential (but far worse)
  c O(n!)           factorial
  c O(n^n)          satan
```

Relative acceptabilities

- O(nlog n) and faster considered efficient
- n^7 and slower considered garbage

- what functions lie between might be ok
  - can be dependent on n

Warnings

1 keep c a constant
  if you let c = n, you'd prove n^2 elOf O(n)

2 Big-Oh notation does not say what the functions *are*
  it only explains a relatinship between 2 functions

  - e.g. 47 + 18log n - 3/n elOf O(the worst-case running time)
    that is fine

3 constant factors can matter depending on their location

  - e.g. e^3n not elOf O(e^n)
    (it's bigger by a factor of e^2n)

4 Big-Oh notation's simplifications can hide important realities

Variations based on n

e.g. binary search on an array

- worst-case running time elOf O(log n)
- best-case running time elOf O(1)
- memory use elOf O(n)

21 hash tables

Dictionaries

- map key to value
  i.e. map any set of discrete objects to any other such set

e.g. 2-letter words & their definitions

- 26x26 = 676 words
  - store word references as array
  - will contain gaps for non-defined words
- inserting a definition into the dictionary
  - requires mapping a word to an index
  - can generate unique index per word
    - 'hash code'
    - essentially treat it as a base 26, 2-digit number
    - and mathematically convert that to base 10
    - works great for 2-letter words
    - array size grows exponentially vs word-length, however

Hash tables

- n: number of keys (words) stored
- table of N buckets (N perhaps a bit larger than n)
- motivation: mapping a huge set of possible keys to a much smaller number of buckets
  - must apply a compression function to each hash code
    - hash: the term for that compression function
  - simple way: h(hashCode) = hashCode mod N
    - pretty random, though, so nearly guaranteed to have collisions
- collision: several keys hash to same bucket
- typ. sol. for collision: chaining
  - each bucket references a linked list of entries mapped
  - now that bucket becomes ambiguous, must store key in table with its associated value

  e.g.

```
public Entry insert(key, value) {
  compute the key's hash code ('hash' the key)
  compress the hash code to determine its bucket
  insert the entry into the bucket's chain
}

public Entry find(key) {
  compute the key's hash code ('hash' the key)
  compress the hash code to determine its bucket
  search the bucket's chain for an entry containing the given key
    if found
      return the entry
    else
      does not exist
      return null, perhaps
}
```

```
    public Entry remove(key) {
      Entry found = find(key)
      if found != null
        remove found from bucket's chain
      return found
    }


  - given a possible attempt to add 2 different entries with the same key

    1 can insert both
      1a can return one arbitrarily, when sought
      1b can return all matching, when sought (may be separate method)

    2 can overwrite old value with new

    - choice is dependent on application


- another solution is probing

  - hop to either side of initial (& discovered full) bucket
  - hop lengths increase until empty bucket found
  - more sensitive to loosing its O(1), so load factor must be capped at more like 0.75


- beware the possibility of references to objects in table where
  the references aren't managed by the table
  - musn't get crazy and modify it (at least not its key)


Hash table performace

- "load factor" : n/N
  - 0.8 to 1.2 are decent ranges for load factor with regard to hash table's performance

- much ado about the length of the buckets' chains
  - short chains -> ops take O(1) time
  - load factor increases chains' lengths
    - asymptotically, with n outpacing N's growth, takes THETA(n) time

  - tends to be easy to go with a really big N
    - does cost time proportional to N to initialize
    - does consume more memory


- hash code & compression must also be 'good'

  - key --------> hashcode -------------> [0, N-1]
          hash              compression
                             function


  - ideal: map each key to a random bucket
    - evenly distribute entries amongst buckets
```

```
Compression function

  - bad e.g.

    - keys: ints
    - each int's hash code is just itself: hashCode(i) == i

    - compression fn h(hashCode) = hashCode mod N
    - N = 10,000 buckets

    - issue: apps tend to generate data with strong patterns

      e.g. keys being very typically divisible by 4
        - h() ->  a table index also divisible by 4
        - now 3/4 of the buckets are never used
          N -> N/4
          load factor just quadrupled

  - that same compression fn is better if N is prime
    - even if hashCodes typ. divisible by a particular number,
      N being prime -> hashCode mod N not divisible by any particular number

  - even better (and for even more obscure reasons):

    - h(hashCode) = ((a * hashCode + b) mod p) mod N
      where a, b, p are positive integers,
            p is a large prime
            p >> N

      - the mod p operation scrambles the bits crafted by (a * hashCode + b) really well
      - mod N gets it into your table
      - N needn't be prime any longer,
        so that responsibility is relegated to an internal impl. detail


A good String hash

  static final int SMALL_PRIME = 127;
  static final int LARGE_PRIME = 16908799;

  private int hashCode(String key) {
    int hashVal = 0;

    for (char c : key.toCharArray()) {
      hashVal = ( SMALL_PRIME * hashVal + c) % LARGE_PRIME;
    }

    return hashVal;
  }


Resizing hash tables

  - if load factor exceeds some constant, O(1) time lost

  - typ. at least double the bucket count

  - walk through old array and rehash all found entries
    - hash function is now abbreviated by a larger #
      therefore those bucket indices won't be valid for the keys they contain,
      leading to table's inconsistency
```

Hash tables rule because

- keys can be virtually anything, and values anything meaningfully associated with their key

  e.g. tic-tac-toe game grid tree

  - if paths converge (i.e. transpose), and each grid's game val is hashed by the grid, then the transposition can be recognized and the value needn't be recomputed

  - so minimax algo checks if grid is in table
    - if yes, return score
    - if no, recurse to determine score & store that in the table

22 stacks

Stacks

- operations

  1 push an item onto its top   'push'

  2 pop an item off of its top 'pop'   (returns that item)

  3 examine its top item   'top'   (like pop, but still on stack)

  - all should be O(1) time

- impl simply via singly-linked list

- stacks' simplicity can play in their favor
  - leading to easier debugging
  - leading to better understanding of the algorithm they're being used in
    e.g. "I have an algo that checks if parentheses match..."
    - "and it uses a list" -> ???
    - "and it uses a stack" -> impl much clearer

23 trees and traversals

Rooted trees

- tree: set of nodes and edges connecting them
  - no cycles; exactly 1 path btwn any 2 nodes

- 'rooted' tree: a particular note is designated the 'root'

- every node c, except root, has 1 parent p
  - p is the first node on the path leading from c to the root

- root has no parent

- leaf has no children

- general rooted trees -> node can have any # of children

- siblings have same parent

- ancestors of a node d - all nodes on path from d to root

- if a is an ancestor of d, d is a descendant of a

- length of path   - number of edges in path (0 for path of 1 node)

- depth of node n - length of path from n to root (root @ depth 0)

- height of node n - length of path from n to its deepest descendent

- height of a tree - height of the root

- subtree rooted at a node n - the tree formed by n and all its descendents

- binary tree
  - no node has children > 2
  - every child is either 'left' or 'right' (even if no siblings)

Representing rooted trees

1 each node has 3 references
  - item
  - parent
  - children (a list structure)

2 "" but
  - siblings are directly linked

  - data structure:

```
class SibTreeNode {
  Object item;
  SibTreeNode parent;
  SibTreeNode firstChild;
  SibTreeNode nextSibling;
}

public class SibTree {
  SibTreeNode root;
  int size;                // number of nodes
  int height;              // sometimes
}
```

Tree traversals

- manner of visiting each node in a tree once

- types

  - preorder traversal
    1 visit node (initially, root)
    2 recursively visit node's children

    - more convenient to define on SibTreeNode class, as it visits nodes

```
      class SibTreeNode {
        public void preorder() {
          this.visit();             // per application basis
                                    // (Hey, what about a strategy pattern, eh???)
          if (firstChild != null) {
            firstChild.preorder();
          }
          if (nextSibling != null) {
            nextSibling.preorder();
          }
        }
      }
```

  - order:

```
       1
     /   \
    2     6
   /|\   / \
  3 4 5 7   8
```

  - time

    - each visited once -> O(1)
    - all n nodes visited -> O(n)

  - natural way to print out, say, a directory (incl. indent per depth)

- postorder traversal
  1 recursively visit node's children (left-to-right)
  2 visit node (initially, root)

  - again, SibTreeNode class ideal

```
      class SibTreeNode {
        public void postOrder() {
          if (firstChild != null) {
            firstChild.postOrder();
          }
          this.visit();
          if (nextSibling != null) {
            nextSibling.postOrder();
          }
        }
      }
```

  - order:

```
       8
     /   \
    4     7
   /|\   / \
  1 2 3 5   6
```

  - time as preorder

    - natural way to sum total diskspace of directories
      - only leaves can be summed, as they contain summable file sizes

    - natural way to sum total diskspace of directories
      - only leaves can be summed, as they contain summable file sizes
      - after summing leaves, their parents can get summed, and so forth

  - inorder traversal (binary tree only)
    1 recursively visit left child
    2 visit node
    3 recursively visit right child

  - level-order traversal
    1 visit root
    2 visit all depth-1 nodes (left-right)
    3 visit all depth-2 nodes (left-right)
    4 and so forth

    - not naturally recursive

    - queue
      - list
      - enqueue: add items to   end
      - dequeue: take items from start

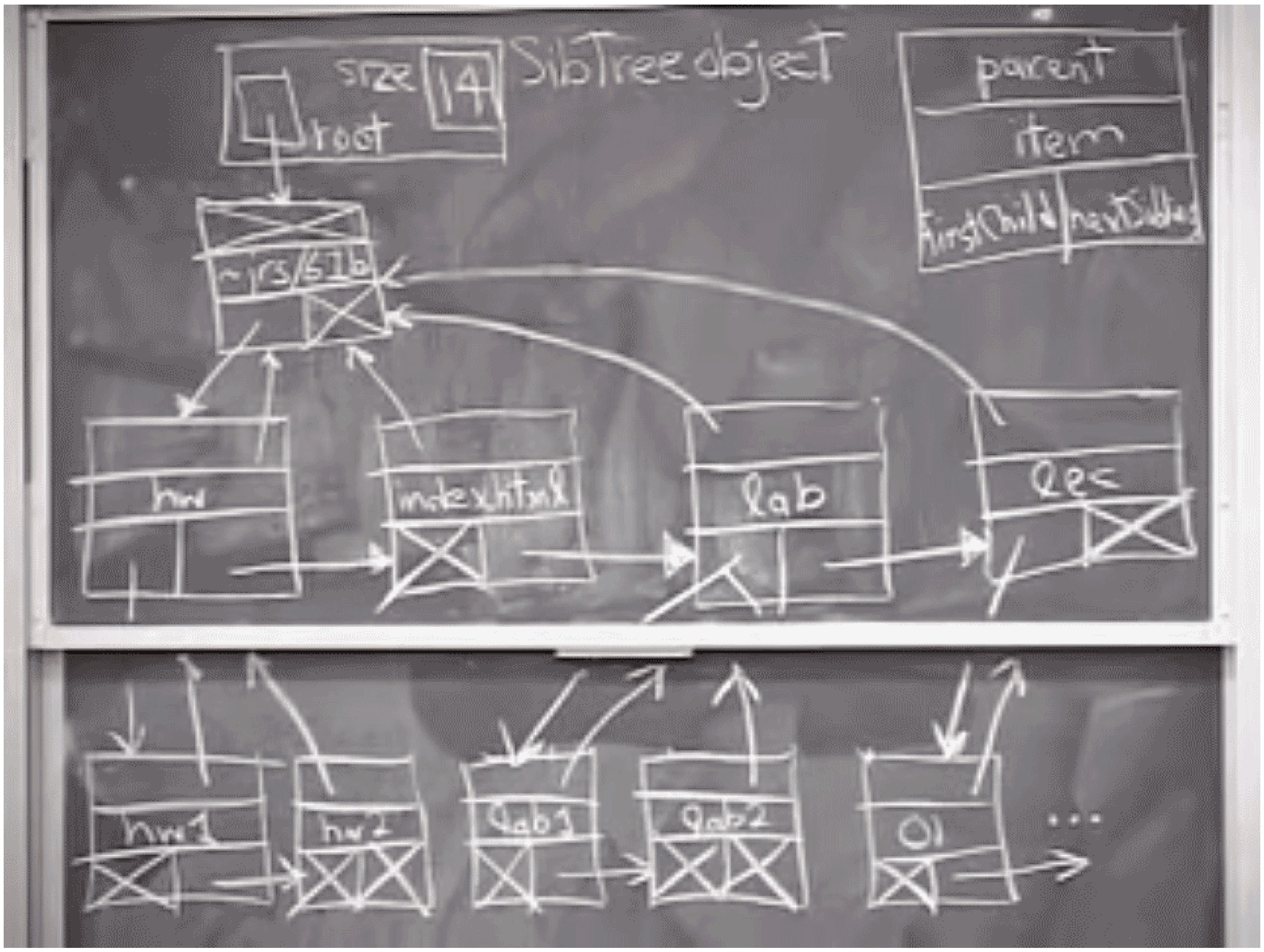
    - method
      1 use queue, first containing root only
      2 dequeue a node
      3 visit dequeued node
      4 enqueue its children left-right (if it has any)
      5 loop until queue is empty

- summing expression tree
  e.g.

```
       +
     /   \
    *     ^
   /\     /\
  3  7   4  2
```

  - inorder: 3 * 7 + 4 ^ 2 (human readable)
  - preorder: + * 3 7 ^ 4 2 (mmm, lisp. aka more computer legible)
  - postorder: 3 7 * 4 2 ^ + (again, more computer legible)

  - computer readability
    - both pre and post only have 1 way to parse such expressions
    - inorder requires precedent rules us humans memorize to disambiguate ordering



24 priority queues

Priority queue

- ADT
- entries consist of key and value, like dictionaries
- however, they use keys for total ordering, not value lookup

- three main operations
  1 remove entry with smallest key
    - typically the only one you remove
    - at least, it’s the only one guaranteed to have quick identification / removal
  2 any key may be inserted at any time
  3 peek at entry with smallest key

- commonly used as event queues in simulations
  - key is the time the event takes place
  - value is the event description
  - thereby allowing the
    - time-based processing of events
    - creation of future events resultant of past events

Binary heap

- an implementation of the ADT, priority queue

- a complete binary tree
  - a tree in which every level of the tree is full, except possibly bottom level
  - bottom level is filled in left-right

e.g.
```
                                 0 1 2 3 4 5  6 7  8  9 10
            2                   [X,2,5,3,9,6,11,4,17,10, 8]
         /         \
        5           3
     /     \      /   \
    9       6   11     4
   / \     /
17   10  8
```

- “heap order property” satisfied by entries
  - no child has a key less than its parent’s key

- “cool other property” satisfied by entries
  - all subtrees of binary heap are also binary heaps

- often stored as arrays of entries

  - order is by level-order traversal
  - called “level numbering”

  - 0th entry is skipped

  - node i’s children: 2i, 2i+1
  - node i’s parent: i/2 (round down)

  - by tracking size, knowledge of last item’s location maintained (n -> nth)

  - two means of storing the 2 variables (k,v)
    1 use two arrays (1 for k, 1 for v)
    2 store an Entry data structure containing fields for both k, v

operation implementations

```
  1 Entry min();

    return entry at root

  2 Entry insert(Object k, Object v);

    let x be new entry (k,v)

    place x in bottom level of tree at first free spot from left
      i.e. first free location in array
      (if bottom level is full, start new level & place at far left)

    x’s location may violate heap order property

      BUBBLE UP!
      x must bubble up tree until heap order property is satisfied
      repeat:
        let p = x’s parent
        if x.k < p.k
          exchange(x, p)
        else
          terminate
```

3 Entry removeMin();

```
  remove entry at root
  save said entry for return value

  replace root with last entry in tree, x

  x’s location almost certainly violates h.o.p.

    BUBBLE DOWN!

    x must bubble down tree until h.o.p. is satisfied
    repeat:
      let minC = min(x.c1, x.c2)
      if x > minC
        exchange(x, minC)
      else
        terminate
```

Bubble down explanation

- all subtrees are binary heaps
  therefore, when root is replaced by bottom-most entry,
  the subtrees rooted at the new root’s children are binary heaps
  - so to determine the “proper” root,
    just make sure the root becomes the min of r’ and its children
    - if that required a swap,
      bubble down recursively
      - the new binary heap under consideration is the subtree
        whose root bubbled up

Running times of priority queue implementations

	binary_heap	sorted_list	unsorted_list
min()	THETA(1)	THETA(1)	THETA(n)
insert()			
worst-case	THETA(log n)	THETA(n)	THETA(1)
best-case	THETA(1)	THETA(1)	THETA(1)
removeMin()			
worst-case	THETA(log n)	THETA(1)	THETA(n)
best-case	THETA(1)	THETA(1)	THETA(n)

Bottom-up heap construction

- input: unsorted collection of entries
- output: binary heap containing entries

- methods

  - naive
    1 insert each 1-by-1

    - each insertion can take THETA(log n) time
      -> THETA(n log n) time

  - void bottomUpHeap();
    1 build complete tree out of entries in any order
    2 walk backward from last internal node to root
      i.e. reverse order in array
    3 when visiting a node, bubble it down recursively

    - runtime = THETA(n)