Topic 1
Course Introduction, Syllabus, and Software Tools

Chapman: I didn't expect a kind of Spanish Inquisition.
Cardinal Ximinez: NOBODY expects the Spanish Inquisition! Our chief weapon is surprise...surprise and fear...fear and surprise.... Our two weapons are fear and surprise...and ruthless efficiency.... Our **three** weapons are fear, surprise, and ruthless efficiency...and an almost fanatical devotion to the Pope.... Our **four**...no... **Amongst** our weapons.... Amongst our weaponry...are such diverse elements as fear, surprise....

Mike Scott, Painter Hall 5.68,
scottm@cs.utexas.edu
www.cs.utexas.edu/~scottm/cs307

Who Am I?

- Lecturer in CS department since 2000
- Undergrad Stanford, MSCS RPI
- US Navy for 8 years, submarines
- 2 years Round Rock High School
- Wife (Kelly) is a nurse.
  - 2 daughters, Olivia and Isabelle

Rensselaer

What We Will Do Today

- Discuss
  - course content
  - procedures
  - tools

Formal Prerequisites

- One year of programming in high school, a grade of at least C in CS303E or CS 305J or consent of instructor (very rarely given).
- Credit or registration for M408C or M408K, or a score of at least 520 on the SAT II Math Level 1 or Math Level 2 test.

Are you in the right place? Required Programming Knowledge and Experience for 307 – (Informal Prerequisites)

- variables and data types
- expressions, order of operations
- decision making (if statements)
  - including boolean logic and boolean expressions
- loops (fixed and variable repetition)
- procedures or functions
- parameters (reference and value parameters, local variables, scope, problem generalization)
- structures or records or objects
- arrays (vectors, lists)
- top down design (breaking big rocks into little rocks)
  - algorithm and data design
  - create and implement program of at least 200 - 300 loc
  - could you write a program to let 2 people play connect 4?

What We Will Do in 307

- A second course in programming with a focus on canonical data structures, algorithms on those data structures, and object oriented programming
- Java Basics and Review (1 week)
- Object Oriented Basics (3 weeks)
  - classes and objects, encapsulation, inheritance, polymorphism
- Fundamental of programming (2 weeks)
  - algorithm analysis, recursion, sorting and searching
- Introduction, application, and implementation of basic abstract data types (9 weeks)
  - lists, iterators, stacks, queues, trees, sets (hash tables, maps/dictionaries, heaps)

Course Materials and Procedures

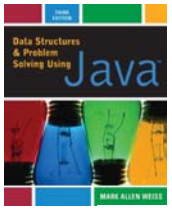
- If you are new to university level classes, you may be surprised by how much of the responsibility for knowing what to do in a class is up to you.
- You are responsible for a great number of things!

Course Materials and Procedures

- web site
  - userweb.cs.utexas.edu/~scottm/cs307/ most materials you need are on the web site
  - links, assignments, schedule, coding samples, study materials, section problems
- schedule
  - on the web site
  - schedule of topics
  - required readings, many from the web
  - links to the slides I use in class
    - Slides are a reference only.
    - We will diverge from the slides on many occasions.
  - due dates

Course Materials and Procedures

- syllabus
  - very important
  - like a contract between instructor and students
  - policies for the course
  - online with links to more information
- books
  - books are recommended not required
  - Weiss book -> data structures
  - On to Java-> Java reference
  - Thinking Recursively in Java (not in Co-op)



Course Materials and Procedures

- Lecture
  - lecture / discussion with instructor
  - not just lecture, I ask questions of you and I encourage you to ask questions of me
  - iClicker questions
- Discussion Section
  - with graduate teaching assistant
  - coding quiz at the start of each, similar in nature to test questions
    - quizzes cannot be made up
  - your chance to ask questions on the assignments
  - cover materials from section handouts which are available on the class web site

Attendance Question 1

Which of these best describes you?

A. First semester at college, recent high school grad.

B. First semester at UT, transferring from another school.

C. In second year at UT.

D. Have been at UT for 2 or more years

Attendance Question 2

Which computer programming language are you most comfortable with?

A. Java

B. C or C++

C. Python

D. PHP

E. C#

Course Materials and Procedures

- class listserv
  - sign up for the listserv, procedure in syllabus and on assignment 1
  - learn to set up a filter in your email client
  - post questions about class, assignments, material, concepts
  - answer your classmates questions
  - updates and information from me will come via the listserv
  - no large chunks (> 3 lines) of solution code on the listserv
  - additional test cases are okay

Graded Course Components

- Attendance, 41 lectures, 1 point each, 41 points total
- Discussion section quizzes, 13 quizzes, 5 points each, 65 points total
- Javabat problems, 7 problem sets, 7 points each, 49 points total
- Programming projects, 12 projects, 10 or 20 points each, 220 points total
- Midterm 1: 170 points
- Midterm 2: 200 points
- Final: 290 points

- Attendance, Quizzes, Javabat, Programming capped at 340 points.
- 35 points of "slack" among those 4 components

Grades and Performance

- Final grade determined by final point total and a 900 – 800 – 700 – 600 scale
  - Will be adjusted with plusses and minuses if within 25 points of cutoff: 875 – 899: B+, 900 – 924: A-

- Last semester **134** students enrolled in the course.
  - **100** students got a C or better. (47 As, 31 Bs, 22 Cs)
  - **24** students got a D or F.
  - **10** students dropped or withdrew.

- The majority of students getting Ds or Fs missed 1 or more exams without an excuse and / or had a failing average on non exam components. (assignments, attendance, javabat, and quizzes)

Course Materials and Procedures

- Assignments
  - where ~80% of your learning will take place
  - constant feedback -> good news / bad news
  - for learning, not evaluation -> low point value
  - posted to class web site
  - see assignment page for general guidelines
  - creating programs using Java
  - usually creating *parts* of programs based on provided code
  - sometimes a complete program
  - some assignments done as individual, some can be done with a partner

Course Materials and Procedures

- More on assignments
  - some test cases provided
  - some provided test cases may have errors
  - use class listserv to discuss and resolve errors in provided test cases
  - create your own test cases
  - graded on correctness, style, efficiency, generality, comments, testing
    - not graded on a linear scale or on effort
  - program must work, compile errors / runtime errors lose all correctness points

Course Materials and Procedures

- Still more on assignments
  - ***VERY IMPORTANT***: must get account for CS department labs -> see syllabus for procedure
  - turn in assignments to your lab account via the turnin program – DEMO
  - ***turn in the right thing! (source code now, jar files later, correct name)***
  - slip days, 6 total for the semester
  - no provisions other than slip days and “slack" in grading scheme for late / missed assignments
  - slip days and “slack” are for emergencies!

Course Materials and Procedures

- And yet more on assignments
  - graded by teaching assistant and proctor
  - scores posted to egradebook -> link on class web site
  - individual assignments are just that, individual
  - copying solution code or giving code to someone else is **CHEATING** -> F in the course
  - solutions checked with plagiarism detection software
  - sharing test cases okay and encouraged
  - read the portion of the syllabus regarding cheating and collaboration

Javabat Problems

- Small scale problems
- 7 sets
- create account, grant access to TA / Grader

Course Materials - Exams

- Out of class midterms.
- Midterm 1: Wednesday, March 2, 6 – 8 pm
- Midterm 2: Wednesday, April 20, 6 – 8 pm

- If you have a conflict, relax. We will determine a makeup time. Email me ASAP.

- Final Exam: Uniform Time to be determined.
- registrar.utexas.edu/students/exams/

More on Exams

- old tests on line – study materials
- tests consist of short answer questions and coding questions
- test emphasize problem solving, algorithm implementation, some syntax
- tests scores curved up if instructor feels necessary.

Succeeding in the Course

- Randy Pausch,
  CS Professor at CMU said:

- *"When I got tenure a year early at Virginia, other Assistant Professors would come up to me and say, 'You got tenure early!?!?! What's your secret?!?!?' and I would tell them, 'Call me in my office at 10pm on Friday night and I'll tell you.' "*

- Meaning: Some things don't have an easy solution.
- Some things simply require a lot of hard work.

Succeeding in the Course

- do the readings
- start on assignments early
- get help from the teaching staff when you get stuck on an assignment
- attend lecture and discussion sections
- participate on the listserv
- do the Javabat problems
- do the extra section problems
- study for tests using the old tests
- study for tests in groups
- ask questions and get help when needed

Course Materials and Procedures

- Software
  - can work in CS department microlab, 5th floor of Painter Hall
  - login via CS account name and password
  - can work at home if you wish
  - Java.
    - Free.
    - Web page has details under Software. (JDK 6.0)
  - Optional IDE.
    - Recommended IDE is Eclipse, also free

Topic 2
Java Basics

"On the other hand, Java has already been a big **win** in academic circles, where it has taken the place of **Pascal** as the preferred tool for teaching the basics of good programming…"

-*The New Hacker's Dictionary* version 4.3.1

www.tuxedo.org/~esr/jargon/html/The-Jargon-Lexicon-framed.html

Agenda

- Brief History of Java and overview of language
- Solve a problem to demonstrate Java syntax
- Discuss coding issues and style via example
- Slides include more details on syntax
  - may not cover everything in class, but you are expected to know these

Brief History of Java and Overview of Langauge

java.sun.com/features/1998/05/birthday.html

A brief history of Java

- "Java, whose original name was Oak, was developed as a part of the Green project at Sun. It was started in December '90 by Patrick Naughton, Mike Sheridan and James Gosling and was chartered to spend time trying to figure out what would be the "next wave" of computing and how we might catch it. They came to the conclusion that at least one of the waves was going to be the convergence of digitally controlled consumer devices and computers. "

Applets and Applications

- "The team returned to work up a Java technology-based clone of Mosaic they named "WebRunner" (after the movie *Blade Runner*), later to become officially known as the HotJava™ browser. It was 1994. WebRunner was just a demo, but an impressive one: It brought to life, for the first time, animated, moving objects and dynamic executable content inside a Web browser. That had never been done. [At the TED conference.]"

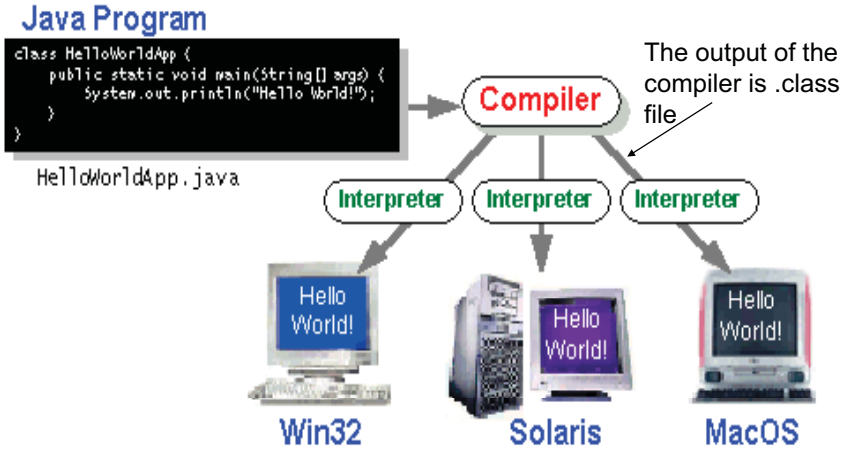
How Java Works

- Java's platform independence is achieved by the use of the *Java Virtual Machine*
- A Java program consists of one or more files with a .java extension
  - these are plain old text files
- When a Java program is compiled the .java files are fed to a compiler which produces a .class file for each .java file
- The .class file contains Java bytecode.
- Bytecode is like machine language, but it is intended for the Java Virtual Machine not a specific chip such as a Pentium or PowerPC chip

More on How Java Works

- To run a Java program the bytecode in a .class file is fed to an interpreter which converts the byte code to machine code for a specific chip (IA-32, PowerPC)
- Some people refer to the interpreter as "The Java Virtual Machine" (JVM)
- The interpreter is platform specific because it takes the platform independent bytecode and produces machine language instructions for a particular chip
- So a Java program could be run an any type of computer that has a JVM written for it.
  - PC, Mac, Unix, Linux, BeaOS, Sparc

A Picture is Worth…



The Interpreter's are sometimes referred to as the Java Virtual Machines

So What!

- The platform independence of Java may be a huge marketing tool, but is actually of little use to people learning Object Oriented Programming and Abstract Data Types
- What is of use is the simplicity of the Java syntax and programming concepts
- Java is a "pure" Object Oriented Language
  - encapsulation, inheritance, and polymorphism
  - all code must be contained in a class
  - no free functions (functions that do not belong to some class) like C++, although someone who wants to write messy Java code certainly can
  - Is OO the best programming paradigm?

HelloWorld.java

```
/**
 * A simple program
 */

public class HelloWorld
{
        public static void main(String[] args)
        {
                System.out.println("HELLO CS307!");
        }
}
```


More on Java Programs

- All code part of some *class*

```
public class Foo
{      //start of class Foo
      /*all code in here!*/
} // end of class Foo
```

- The code for class Foo will be in a file named Foo.java
  - just a text file with the .java extension
  - a class is a programmer defined data type
- A complete program will normally consist of many different classes and thus many different files

Attendance Question 1

What does 6,967 * 7,793 equal?

A. 10,000

B. 23,756,201

C. 54,293,831

D. 2,147,483,647

E. - 2,147,483,648

Attendance Question 2

How many factors does 54,161,329 have?

A. 2

B. 3

C. 4

D. 6

E. more than 6

Bonus question. What are they?

Example Problem

- Determine if a given integer is prime
  - problem definition
  - really naïve algorithm
  - implementation
  - testing
  - a small improvement
  - another improvement
  - yet another improvement
  - always another way ...
  - what about really big numbers? (Discover AKS Primality Testing)

Error Types

- Syntax error / Compile errors
  - caught at compile time.
  - compiler did not understand or compiler does not allow
- Runtime error
  - something “Bad” happens at runtime. Java breaks these into Errors and Exceptions
- Logic Error
  - program compiles and runs, but does not do what you intended or want

Java Language
Review of Basic Features

Basic Features

- Data Types
  - primitives
  - classes / objects
- Expressions and operators
- Control Structures
- Arrays
- Methods
- Programming for correctness
  - pre and post conditions
  - assertions

Java Data Types

Identifiers in Java

- letters, digits, _, and $ (don't use $. Can confuse the runtime system)
- start with letter, _, or $
- by convention:
  1. start with a letter
  2. variables and method names, lowercase with internal words capitalized e.g. honkingBigVariableName
  3. constants all caps with _ between internal words e.g. ANOTHER_HONKING_BIG_INDENTIFIER
  4. classes start with capital letter, internal words capitalized, all other lowercase e.g HonkingLongClassName
- Why? To differentiate identifiers that refer to classes from those that refer to variables

Data Types

- Primitive Data Types
  - byte short int long float double boolean char

```
//dataType identifier;
int x;
int y = 10;
int z, zz;
double a = 12.0;
boolean done = false, prime = true;
char mi = 'D';
```

  - stick with int for integers, double for real numbers
- Classes and Objects
  - pre defined or user defined data types consisting of constructors, methods, and fields (constants and fields (variables) which may be primitives or objects.)

Java Primitive Data Types

Data Type	Characteristics	Range
`byte`	8 bit signed integer	-128 to 127
`short`	16 bit signed integer	-32768 to 32767
`int`	32 bit signed integer	-2,147,483,648 to 2,147,483,647
`long`	64 bit signed integer	-9,223,372,036,854,775,808 to-9,223,372,036,854,775,807
`float`	32 bit floating point number	± 1.4E-45 to ± 3.4028235E+38
`double`	64 bit floating point number	± 4.9E-324 to ± 1.7976931348623157E+308
`boolean`	`true` or `false`	NA, note Java booleans cannot be converted to or from other types
`char`	16 bit, Unicode	Unicode character, `\u0000` to `\uFFFF` Can mix with integer types

What are Classes and Objects?

- Class is synonymous with data type
- Object is like a variable
  - The data type of the Object is some Class
  - referred to as an *instance of a Class*
- Classes contain:
  - the implementation details of the data type
  - and the interface for programmers who just want to use the data type
- Objects are complex variables
  - usually multiple pieces of internal data
  - various behaviors carried out via *methods*

Creating and Using Objects

- Declaration - DataType identifier

```
Rectangle r1;
```

- Creation - new operator and specified constructor

```
r1 = new Rectangle();
Rectangle r2 = new Rectangle();
```

- Behavior - via the dot operator

```
r2.setSize(10, 20);
String s2 = r2.toString();
```

- Refer to documentation for available behaviors (methods)

Built in Classes

- Java has a large built in library of classes with lots of useful methods
- Ones you should become familiar with quickly
- String
- Math
- Integer, Character, Double
- System
- Arrays
- Scanner
- File
- Object
- Random
- Look at the Java API page

import

- import is a reserved word
- packages and classes can be imported to another class
- does not actually import the code (unlike the C++ `include` preprocessor command)
- statement outside the class block

```
import java.util.ArrayList;
import java.awt.Rectangle;
public class Foo{
   // code for class Foo
}
```

More on import

- can include a whole package
  - `import java.util.*;`
- or list a given class
  - `import java.util.Random;`
- instructs the compiler to look in the package for types it can't find defined locally
- the `java.lang.*` package is automatically imported to all other classes.
- Not required to import classes that are part of the same project in Eclipse

The String Class

- String is a standard Java class
  - a whole host of behaviors via methods
- also special (because it used so much)
  - String literals exist (no other class has literals)

```
String name = "Mike D.";
```

  - String concatenation through the + operator

```
String firstName = "Mike";
String lastName = "Scott";
String wholeName = firstName + lastName;
```

  - Any primitive or object on other side of + operator from a String automatically converted to String

Standard Output

- To print to standard output use

```
System.out.print( expression ); // no newline
System.out.println( expression ); // newline
System.out.println( ); // just a newline
```

common idiom is to build up expression to be printed out

```
System.out.println( "x is: " + x + " y is: " + y );
```


Constants

- Literal constants - "the way you specify values that are not computed and recomputed, but remain, well, constant for the life of a program."
  - true, false, null, 'c', "C++", 12, -12, 12.12345
- Named constants
  - use the keyword **`final`** to specify a constant
  - scope may be local to a method or to a class
- By convention any numerical constant besides -1, 0, 1, or 2 requires a named constant

```
final int NUM_SECTIONS = 3;
```

Expressions and Operators

Operators

- Basic Assignment: =
- Arithmetic Operators: +, -, *, /, %(remainder)
  - integer, floating point, and mixed arithmetic and expressions
- Assignment Operators: +=, -=, *=, /=, %=
- increment and decrement operators: ++, --
  - prefix and postfix.
  - avoid use inside expressions.

```
int x = 3;
x++;
```


Expressions

- Expressions are evaluated based on the precedence of operators
- Java will automatically convert numerical primitive data types but results are sometimes surprising
  - take care when mixing integer and floating point numbers in expressions
- The meaning of an operator is determined by its operands

  /

  is it integer division or floating point division?

Casting

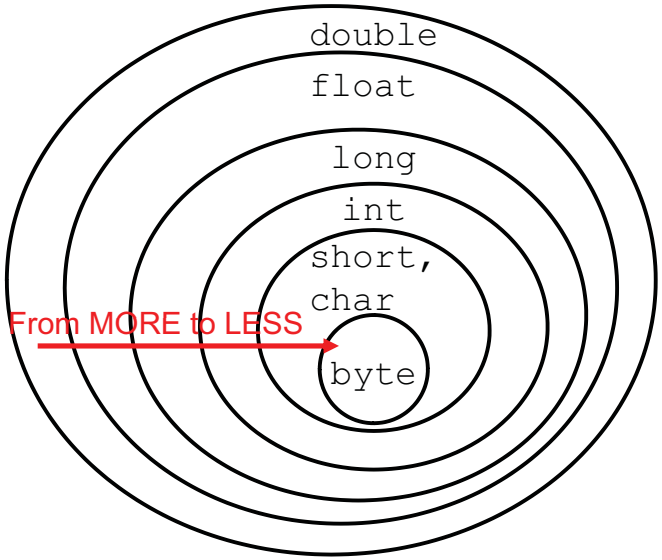
- *Casting* is the temporary conversion of a variable from its original data type to some other data type.
  - *Like being cast for a part in a play or movie*
- With primitive data types if a cast is necessary from a less inclusive data type to a more inclusive data type it is done automatically.

```
int x = 5;
double a = 3.5;
double b = a * x + a / x;
double c = x / 2;
```

- if a cast is necessary from a more inclusive to a less inclusive data type the class must be done explicitly by the programmer
  - failure to do so results in a compile error.

```
double a = 3.5, b = 2.7;
int y = (int) a / (int) b;
y = (int)( a / b );
y = (int) a / b; //syntax error
```

Primitive Casting



Outer ring is most inclusive data type. Inner ring is least inclusive.

In expressions variables and sub expressions of less inclusive data types are automatically cast to more inclusive.

If trying to place expression that is more inclusive into variable that is less inclusive, explicit cast must be performed.

Java Control Structures

Control Structures

- linear flow of control
  - statements executed in consecutive order
- Decision making with if - else statements

```
if(boolean-expression)
   statement;
if(boolean-expression)
{  statement1;
   statement2;
   statement3;
}
```

A single statement could be replaced by a statement block, braces with 0 or more statements inside

Boolean Expressions

- boolean expressions evaluate to true or false
- Relational Operators: >, >=, <, <=, ==, !=
- Logical Operators: &&, ||, !
  - && and || cause short circuit evaluation
  - if the first part of `p && q` is false then q is not evaluated
  - if the first part of `p || q` is true then q is not evaluated

```
//example
if( x <= X_LIMIT && y <= Y_LIMIT)
   //do something
```

More Flow of Control

- if-else:

```
if(boolean-expression)
   statement1;
else
   statement2;
```

- multiway selection:

```
if(boolean-expression1)
   statement1;
else if(boolean-expression2)
   statement2;
else
   statement3;
```

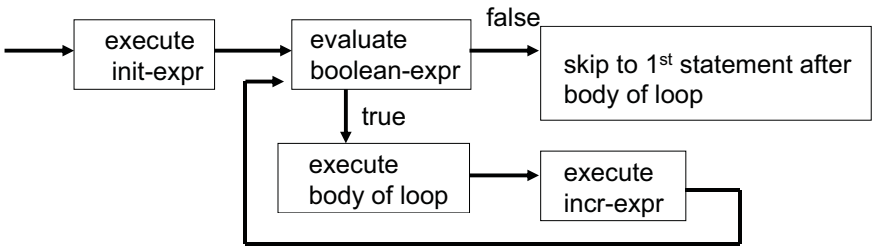
- individual statements could be replaced by a statement block, a set of braces with 0 or more statements
- Java also has the switch statement, but not part of our subset

for Loops

- for loops

```
for(init-expr;boolean-expr;incr-expr)
   statement;
```

- init-expr and incr-expr can be more zero or more expressions or statements separated by commas
- statement could be replaced by a statement block



while loops

- while loops

```
while(boolean-expression)
   statement; //or statement block
```

- do-while loop part of language

```
do
   statement;
while(boolean-expression);
```

- Again, could use a statement block
- `break`, `continue`, and labeled breaks
  - referred to in the Java tutorial as *branching statements*
  - keywords to override normal loop logic
  - use them judiciously (which means not much)

Attendance Question 3

True or false: Strings are a primitive data type in Java.

A. TRUE

B. FALSE

Attendance Question 4

What is output by the following Java code?

```
int x = 3;
double a = x / 2 + 3.5;
System.out.println(a);
```

A. a

B. 5

C. 4.5

D. 4

E. 5.0

Arrays

Arrays in Java

- "Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration. "
  - *S. Kelly-Bootle*
- Java has built in arrays. a.k.a. native arrays
- arrays hold elements of the same type
  - primitive data types or classes
  - space for array must be dynamically allocated with new operator. (Size is any *integer expression*. Due to dynamic allocation does not have to be constant.)

```
public void arrayExamples()
{    int[] intList = new int[10];
     for(int i = 0; i < intList.length; i++)
     {    assert 0 >= i && i < intList.length;
          intList[i] = i * i * i;
     }
     intList[3] = intList[4] * intList[3];
}
```


Array Details

- all arrays must be dynamically allocated
- arrays have a public, final field called *length*
  - built in size field, no separate variable needed
  - don't confuse length (capacity) with elements in use
- elements start with an index of zero, last index is length - 1
- trying to access a non existent element results in an ArrayIndexOutOfBoundsException (AIOBE)

Array Initialization

- Array variables are object variables
- They hold the memory address of an array object
- The array must be dynamically allocated
- All values in the array are initialized (0, 0.0, char 0, false, or null)
- Arrays may be initialized with an initializer list:

```
int[] intList = {2, 3, 5, 7, 11, 13};
double[] dList = {12.12, 0.12, 45.3};
String[] sList = {"Olivia", "Kelly", "Isabelle"};
```


Arrays of objects

- A native array of objects is actually a native array of *object variables*
  - all object variables in Java are really what?
  - Pointers!

```
public void objectArrayExamples()
{     Rectangle[] rectList = new Rectangle[10];
      // How many Rectangle objects exist?

      rectList[5].setSize(5,10);
      //uh oh!

      for(int i = 0; i < rectList.length; i++)
      {     rectList[i] = new Rectangle();
      }

      rectList[3].setSize(100,200);
}
```


Array Utilities

- In the `Arrays` class, static methods
- `binarySearch`, `equals`, `fill`, and `sort` methods for arrays of all primitive types (except boolean) and arrays of Objects
  - overloaded versions of these methods for various data types
- In the System class there is an `arraycopy` method to copy elements from a specified part of one array to another
  - can be used for arrays of primitives or arrays of objects

The `arraycopy` method

- `static void` **`arraycopy`**`(Object src, int srcPos, Object dest, int destPos, int length)`

Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

```
int[] list = new int[10];
// code to fill list
// list needs to be resized
int[] temp = new int[list.length * 2];
System.arraycopy(list, 0, temp, 0,
        list.length);
list = temp;
```

2D Arrays in Java

- Arrays with multiple dimensions may be declared and used

```
int[][] mat = new int[3][4];
```

- the number of pairs of square brackets indicates the dimension of the array.
- by convention, in a 2D array the first number indicates the row and the second the column
- Java multiple dimensional arrays are handles differently than in many other programming languages.

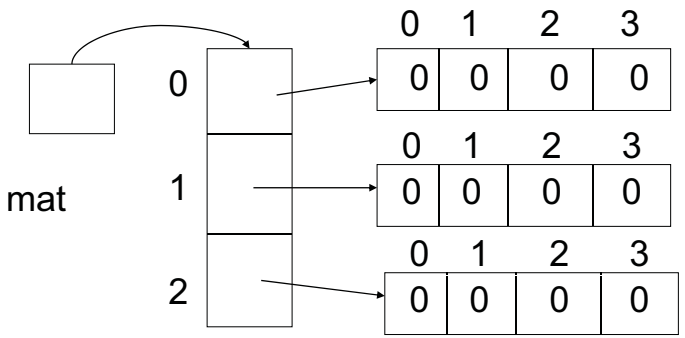
Two Dimensional Arrays

row \ column	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0

This is our abstract picture of the 2D array and treating it this way is fine.

```
mat[2][1] = 12;
```


The Real Picture



`mat` holds the memory address of an array with 3 elements. Each element holds the memory address of an array of 4 `int`s

Arrays of Multiple Dimension

- because multiple dimensional arrays are treated as arrays of arrays of arrays……multiple dimensional arrays can be *ragged*
  - each row does not have to have the same number of columns

```
int[][] raggedMat = new int[5][];
for(int i = 0; i < raggedMat.length; i++)
     raggedMat[i] = new int[i + 1];
```

  - each row array has its own length field

Ragged Arrays

- Ragged arrays are sometime useful, but normally we deal with *rectangular* matrices
  - each row has the same number of columns as every other row
  - use this a lot as precondition to methods that work on matrices
- working on matrices normally requires nested loops
  - why is this so hard?

Enhanced for loop

- New in Java 5.0
- a.k.a. the for-each loop
- useful short hand for accessing all elements in an array (or other types of structures) if no need to alter values
- alternative for iterating through a set of values

```
for(Type loop-variable : set-expression)
    statement
```

- logic error (not a syntax error) if try to modify an element in array via enhanced for loop

Enhanced for loop

```
public static int sumListOld(int[] list)
{    int total = 0;
     for(int i = 0; i < list.length; i++)
     {    total += list[i];
          System.out.println( list[i] );
     }
     return total;
}
```

```
public static int sumListEnhanced(int[] list)
{    int total = 0;
     for(int val : list)
     {    total += val;
          System.out.println( val );
     }
     return total;
}
```


Attendance Question 5

What is output by the code to the right when method d1 is called?

A. 322
B. 323
C. 363
D. 366
E. 399

```
public void d2(int x){
  x *= 2;
  System.out.print(x);
}

public void d1(){
  int x = 3;
  System.out.print(x);
  d2(x);
  System.out.print(x);
}
```

Attendance Question 6

What is output by the code to the right?

```
int[] list = {5, 1, 7, 3};
System.out.print( list[2] );
System.out.print( list[4] );
```

A. Output will vary from one run of program to next

B. 00

C. 363

D. 7 then a runtime error

E. No output due to syntax error

Methods

Methods

- methods are analogous to procedures and functions in other languages
  - local variables, parameters, *instance variables*
  - must be comfortable with variable scope: where is a variable defined?
- methods are the means by which objects are manipulated (objects *state* is changed) - much more on this later
- method header consists of
  - access modifier(**public**, package, protected, **private**)
  - static keyword (optional, class method)
  - return type (void or any data type, primitive or class)
  - method name
  - parameter signature

More on Methods

- local variables can be declared within methods.
  - Their scope is from the point of declaration until the end of the methods, unless declared inside a smaller block like a loop
- methods contain statements
- methods can call other methods
  - in the same class: `foo();`
  - methods to perform an operation on an object that is in scope within the method: `obj.foo();`
  - static methods in other classes:

```
double x = Math.sqrt(1000);
```

static methods

- the main method is where a stand alone Java program normally begins execution
- common compile error, trying to call a non static method from a static one

```
public class StaticExample
{   public static void main(String[] args)
    {     //starting point of execution
          System.out.println("In main method");
          method1();
          method2(); //compile error;
    }

    public static void method1()
    {     System.out.println( "method 1");    }

    public void method2()
    {     System.out.println( "method 2");    }
}
```


Method Overloading and Return

- a class may have multiple methods with the same name as long as the parameter signature is unique
  - may not overload on return type
- methods in different classes may have same name and signature
  - this is a type of polymorphism, not method overloading
- if a method has a return value other than void it must have a return statement with a variable or expression of the proper type
- multiple return statements allowed, the first one encountered is executed and method ends
  - style considerations

Method Parameters

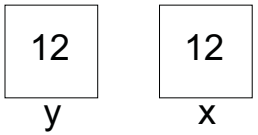
- a method may have any number of parameters
- each parameter listed separately
- no VAR (Pascal), &, or const & (C++)
- final can be applied, but special meaning
- all parameters are pass by value
- Implications of pass by value???

Value Parameters vs. Reference Parameters

- A value parameter makes a copy of the argument it is sent.
  - Changes to parameter do not affect the argument.
- A reference parameter is just another name for the argument it is sent.
  - changes to the parameter are really changes to the argument and thus are permanent

Value vs. Reference

```
// value
void add10(int x)
{    x += 10;      }

void calls()
{    int y = 12;
     add10(y);
     // y = ?
}
```




```
// C++, reference
void add10(int& x)
{    x += 10;      }

void calls()
{    int y = 12;
     add10(y);
     // y = ?
}
```

y x

Programming for Correctness

Creating Correct Programs

- methods should include *pre conditions* and *post conditions*
- Preconditions are things that must be true before a method is called
- Postconditions are things that will be true after a method is complete if the preconditions were met
- it is the responsibility of the caller of a method to ensure the preconditions are met
  - the class must provide a way of ensuring the precondition is true
  - the preconditions must be stated in terms of the interface, not the implementation
- it is the responsibility of the class (supplier, server) to ensure the postconditions are met

Programming by Contract

- preconditions and postconditions create a contract between the client (class or object user) and a supplier (the class or object itself)
  - example of a contract between you and me for a test

	Obligations	Benefits
Client (Student)	*(Must ensure preconditions)* Be at test on time, bring pencil and eraser, write legibly, answer questions in space provided	*(May benefit from postcondition)* Receive fair and accurate evaluation of test to help formulate progress in course
Supplier (Mike)	*(Must ensure postcondition)* Fairly and accurately grade test based on universal guidelines applied to all tests	*(May assume preconditions)* No need to grade test or questions that are illegible, on wrong part of exam, or give makeup exams for unexcused absences

Thinking about pre and postconditions

- pre and postconditions are part of design
- coming up with pre and postconditions at the time of implementation is too late
- the pre and post conditions drive the implementation and so must exist before the implementation can start
  - The sooner you start to code, the longer your program will take.

    -Roy Carlson, U Wisconsin
- *You must spend time on design*

Precondition Example

```
/**
* Find all indices in <tt>source</tt> that are the start of a complete
* match of <tt>target</tt>.
* @param source != null, source.length() > 0
* @param target != null, target.length() > 0
* @return an ArrayList that contains the indices in source that are the
* start of a complete match of target. The indices are stored in
* ascending order in the ArrayList
*/
public static ArrayList<Integer> matches(String source, String target) {
       // check preconditions
       assert (source != null) && (source.length() > 0)
              && (target != null) && (target.length() > 0)
              : "matches: violation of precondition";
```


Creating Correct Programs

- Java features has a mechanism to check the correctness of your program called *assertions*
- Assertions are statements that are executed as normal statements if assertion checking is on
  - you should always have assertion checking on when writing and running your programs
- Assertions are boolean expressions that are evaluated when reached. If they evaluate to true the program continues, if they evaluate to false then the program halts
- logical statements about the condition or state of your program

Assertions

- Assertions have the form

```
assert boolean expression : what to output
  if assertion is false
```

- Example

```
if ( (x < 0) || (y < 0) )
{ // we know either x or y is < 0
  assert x < 0 || y < 0 : x + " " + y;
  x += y;
}
else
{ // we know both x and y are not less than zero
  assert x >= 0 && y >= 0 : x + " " + y;
  y += x;
}
```

- Use assertion liberally in your code
  - part of style guide

Assertions Uncover Errors in Your Logic

```
if ( a < b )
{ // we a is less than b
   assert a < b : a + " " + b;
   System.out.println(a + " is smaller than " + b);
}
else
{ // we know b is less than a
   assert b < a : a + " " + b;
   System.out.println(b + " is smaller than " + a);
}
```

- Use assertions in code that other programmers are going to use.
- In the real world this is the majority of your code!

javadoc

- javadoc is a program that takes the comments in Java source code and creates the html documentation pages
- Open up Java source code. (Found in the src.zip file when you download the Java sdk.)
- Basic Format

```
/** Summary sentence for method foo. More details. More details.
   pre: list preconditions
   post: list postconditions
   @param x describe what x is
   @param y describe what y is
   @return describe what the method returns
*/
public int foo(int x, double y)
```

- Comments interpreted as html

Topic 3
References and Object Variables

"Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end."

- Henry Spencer

Object Variables

- object variables are declared by stating the class name / data type and then the variable name
  - same as primitives
  - in Java there are hundreds of built in classes.
    - show the API page
  - don't learn the classes, learn how to read and use a class interface (the users manual)
- objects are *complex* variables.
  - They have an internal *state* and various *behaviors* that can either change the state or simply tell something about the object

Object Variables

```
public void objectVaraiables()
{     Rectangle rect1;
      Rectangle rect2;
      // 2 Rectangle objects exist??
      // more code to follow
}
```

- So now there are 2 Rectangle objects right?
- Not so much.
- Object variables in Java are actually *references* to objects, not the objects themselves!
  - object variables store the memory address of an object of the proper type *not* an object of the proper type.
  - contrast this with primitive variables

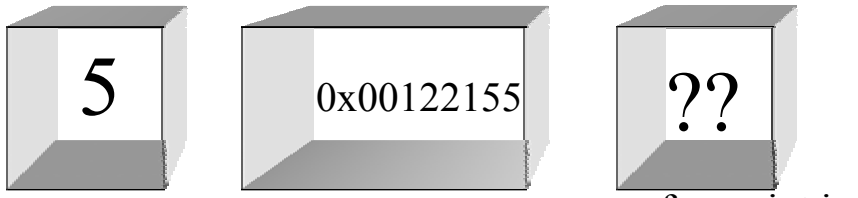
The Pointer Sidetrack

- IMPORTANT!! This material may seem a bit abstract, but it is often the cause of many a programmers logic error
- A pointer is a variable that stores the memory address of where another variable is stored
- In some languages you can have *bound* variables and dynamic variables of any type
  - a bound variable is one that is associated with a particular portion of memory that cannot be changed
- Example C++, can have an integer variable or a integer pointer (which is still a variable)

```
int intVar; // a int var
int * intPtr; //pointer to an int var
```

Pointer Variables in C++

```
int intVar = 5; // a int var
int * intPtr; //pointer to an int var
intPtr = new int;
/* dynamically allocate an space to store an int.
intPtr holds the memory address of this space*/
```



Pointer Complications

- C++ allows actual variables and pointers to variables of any type. Things get complicated and confusing very quickly

```
int intVar = 5; // a int var
int * intPtr; //pointer to an int var
intPtr = new int; // allocate memory
*intPtr = 12; /* assign the integer being
     pointed to the value of 12.  Must
     dereference the pointer.  i.e. get to
     the thing being pointed at*/
cout << intPtr << "\t" << *intPtr << "\t"
     << &intPtr << endl;
// 3 different ways of manipulating intPtr
```

- In C++ you can work directly with the memory address stored in intPtr
  - increment it, assign it other memory addresses, pointer "arithmetic"

Attendance Question 1

Given the following C++ declarations how would the variable `intPtr` be made to refer to the variable `intVar`?

```
intVar = 5;
intPtr = new int;
```

A. `intPtr = intVar;`

B. `intPtr = *intVar;`

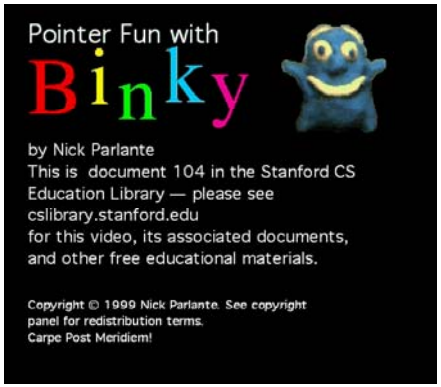
C. `*intPtr = intVar;`

D. `intPtr = &intVar;`

E. `intPtr = intVar;`

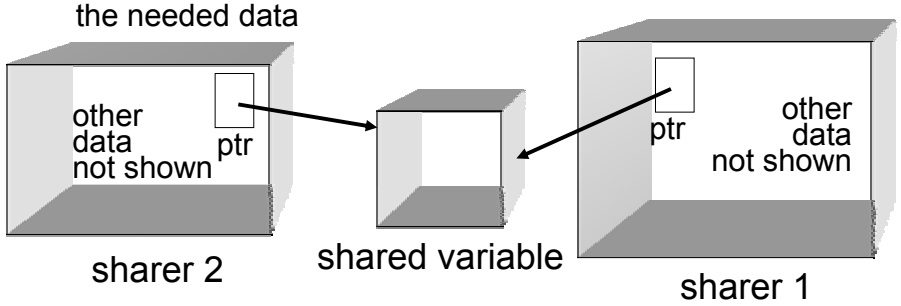
And Now for Something Completely Different…

- Thanks Nick…
- Link to Bink



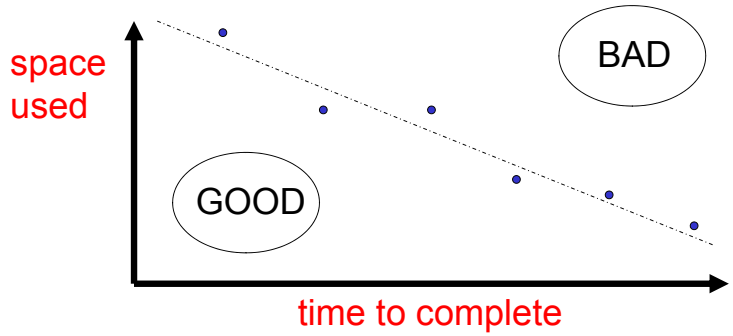
Benefit of Pointers

- Why have pointers?
- To allow the sharing of a variable
  - If several variables(objects, records, structs) need access to another single variable two alternatives
    1. keep multiple copies of variable.
    2. share the data with each variable keeping a reference to the needed data



Time Space Trade Off

- Often the case that algorithms / solutions an be made faster by using more space (memory) or can use less space at the expense of being slower.



More Benefits

- Allow dynamic allocation of memory
  - get it only when needed (*stack* memory and *heap* memory)
- Allow linked data structures such as *linked lists* and *binary trees*
  - incredibly useful for certain types of problems
- Pointers are in fact necessary in a language like Java where *polymorphism* is so prevalent (more on this later)
- Now the good news
  - In Java most of the complications and difficulties inherent with dealing with pointers are removed by some simplifications in the language

Dynamic Memory Allocation

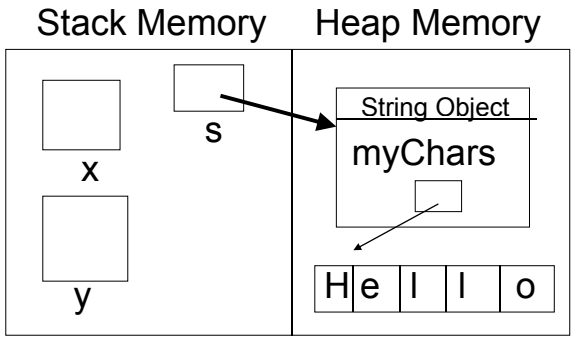
Your program has two chunks of memory to work with: *Stack memory* (or the *runtime Stack*) and *Heap memory*

When a Java program starts it receives two chunks of memory one for the Stack and one for the Heap.

Things that use Stack memory: local variables, parameters, and information about methods that are in progress.

Things that use Heap memory: everything that is allocated using the `new` operator.

The Picture




```
void toyCodeForMemory(int x)
{   int y = 10;
    x += y;
    String s = new String("Hello");
    System.out.println(x + " " + y + s);
}
```


How Much Memory?

How big is the Heap?

```
System.out.println("Heap size is " +
Runtime.getRuntime().totalMemory());
```

How much of the Heap is available?

```
System.out.println("Available memory: " +
Runtime.getRuntime().freeMemory());
```


References in Java

- In Java all primitive variables are value variables. (real, actual, direct?)
  - it is impossible to have an integer pointer or a pointer to any variable of one of the primitive data types
- All object variables are actually *reference variables* (essentially store a memory address) to objects.
  - it is impossible to have anything but references to objects. You can never have a plain object variable

Back to the Rectangle Objects

- rect1 and rect2 are variables that store the memory addresses of Rectangle objects
- right now they are uninitialized and since they are local, variables may not be used until they are given some value

```
public void objectVaraiables()
{     Rectangle rect1;
      Rectangle rect2;
      // rect1 = 0;       // syntax error, C++ style
      // rect1 = rect2; // syntax error, unitialized
      rect1 = null;  // pointing at nothing
      rect2 = null;  // pointing at nothing
}
```

- `null` is used to indicate an object variable is not pointing / naming / referring to any Rectangle object.

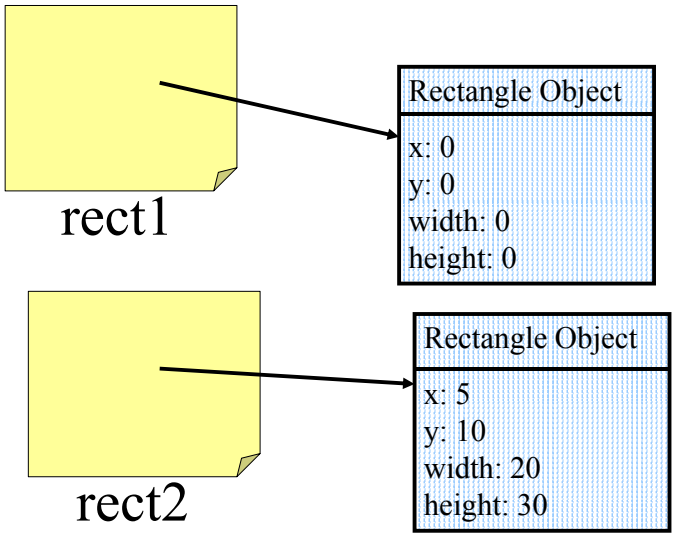
Creating Objects

- Declaring object variables does *not* create objects.
  - It merely sets aside space to hold the memory address of an object.
  - The object must be created by using the `new` operator and calling a *constructor* for that object

```
public void objectVaraiables()
{     Rectangle rect1;
      rect1 = new Rectangle();
      Rectangle rect2 = new Rectangle(5,10,20,30);
      // (x, y, width, height)
      // rect1 and rect2 now refer to Rectangle objects
}
```

- For all objects, the memory needed to store the objects, is allocated dynamically using the `new` operator and a constructor call. (Strings are a special case.)
  - constructors are similar to methods, but they are used to initialize objects

The Yellow Sticky Analogy



Pointers in Java

- Is this easier?
  - primitives one thing, objects another?
- can't get at the memory address the pointer stores as in C++

  although try this:

```
Object obj = new Object();
System.out.println( obj.toString() );
```

- dereferencing occurs automatically
- because of the consistency the distinction between an object and an object reference can be blurred
  - "pass an object to the method" versus "pass an object reference to the method
- Need to be clear when dealing with memory address of object and when dealing with the object itself

Working with Objects

- Once an object is created and an object variable points to it then Object may be manipulated via its methods

```
Rectangle r1 = new Rectangle();
r1.resize(100, 200);
r1.setLocation(10, 20);
int area = r1.getWidth() * r1.getHeight();
Rectangle r2 = null;
r2.resize( r1.getWidth(), r1.getHeight() * 2 );
// uh-oh!
```

- Use the dot operator to deference an object variable and *invoke* one of the objects behaviors
- Available behaviors are spelled out in the class of the object, (the data type of the object)

What's the Output?

```
public void objectVariables()
{       Rectangle rect1 = new Rectangle(5, 10, 15, 20);
        Rectangle rect2 = new Rectangle(5, 10, 15, 20);;
        System.out.println("rect 1: " + rect1.toString() );
        System.out.println("rect 2: " + rect2.toString() );
        // Line 1
        System.out.println("rect1 == rect2:  " + (rect1 == rect2));
        rect1 = rect2;
        rect2.setSize(50, 100); // (newWidth, newHeight)
        // Line 2
        System.out.println("rect 1: " + rect1.toString() );
        System.out.println("rect 2: " + rect2.toString() );
        System.out.println("rect1 == rect2:  " + (rect1 == rect2));
        int x = 12;
        int y = 12;
        // Line 3
        System.out.println("x == y: " + (x == y) );
        x = 5;
        y = x;
        x = 10;
        System.out.println("x == y: " + (x == y) );
        // Line 4
        System.out.println("x value: " + x + ", y value: " + y);
}
```


Attendance Question 2

What is output by the line of code marked Line 1?

A. `rect1 == rect2: true`

B. `rect1 == rect2: rect1 == rect2`

C. `rect1 == rect2: false`

D. `intPtr = &intVar;`

E. `rect1 == rect2: 0`

Attendance Question 3

What will be the width and height of the `Rectangle` object `rect1` refers to at the line of code marked Line 2?

A. width = 15, height = 20

B. width = 20, height = 15

C. width = -1, height = -1

D. width = 0, height = 0

E. width = 50, height = 100

Attendance Question 4

What is output by the line of code marked Line 3?

A. `x == y: 0`

B. `x == y: 1`

C. `x == y: true`

D. `x == y: x == y`

E. `x == y: false`

Attendance Question 5

What is output by the line of code marked Line 4?

A. `x value: 5, y value: 5`

B. `x value: 10, y value: 5`

C. `x value: 0, y value: 0`

D. `x value: 5, y value: 10;`

E. `x value: 10, y value: 10`

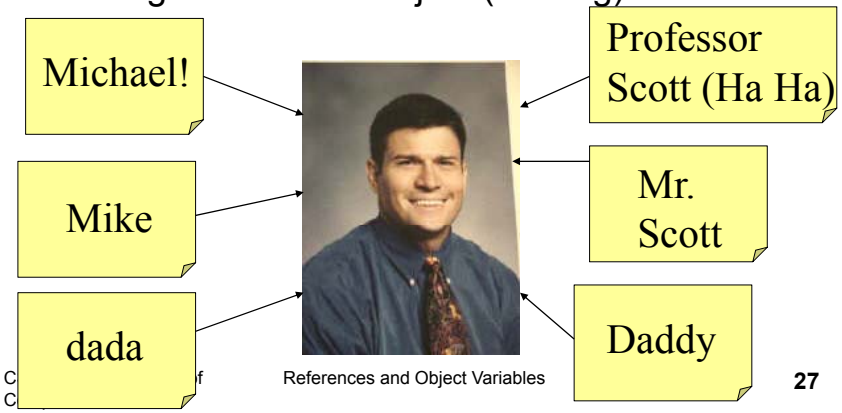
Equality versus Identity

A man walks into a pizza parlor, sits down, and tells the waiter, "I'll have what that lady over there is eating." The waiter walks over to the indicated lady, picks up the pizza that is resting in front of her, and sets it back down in from of the man's table.

- confusion over equality and identity
- identity: two things are in fact the same thing
- equality: two things are for all practical purposes alike, but not the exact same thing
- == versus the .equals method
  - use the .equals method when you want to check the contents of the pointee, use == when you want to check memory addresses

Just Like the Real World

- Objects variables are merely names for objects
- Objects may have multiple names
  - meaning there are multiple object variables referring to the same object (sharing)



The Garbage Collector

```
Rectangle rect1 = new Rectangle(2,4,10,10);
Rectangle rect2 = new Rectangle(5,10,20,30);
// (x, y, width, height)
rect1 = rect2;
/*     what happened to the Rectangle Object
       rect1 was pointing at?
*/
```

- If objects are allocated dynamically with new how are they deallocated?
  - `delete` in C++
- If an object becomes isolated (no longer is in scope), that is has no references to it, it is garbage and the Java Virtual Machine *garbage collector* will reclaim this memory AUTOMATICALLY!

Objects as Parameters

- All parameters in Java are *value* parameters
- The method receives a copy of the parameter, not the actual variable passed
- Makes it impossible to change a primitive parameter
- implications for objects? (which are references)
  - behavior that is similar to a reference parameter, with a few minor, but crucial differences
  - "Reference parameter like behavior for the pointee."

Immutable Objects

- Some classes create *immutable* objects
- Once created these objects cannot be changed
  - note the difference between objects and object variables
- Most immediate example is the String class
- String objects are immutable
- Why might this be useful?

```
String name = "Mike";
String sameName = name;
name += " " + "David" + " " + "Scott";
System.out.println( name );
System.out.println( sameName );
```

Topic 4
Exceptions and File I/O

"A slipping gear could let your M203 grenade launcher fire when you least expect it. That would make you quite unpopular in what's left of your unit."

- *THE U.S. Army's PS magazine, August 1993,* quoted in *The Java Programming Language, 3rd edition*

When Good Programs Go Bad

- A variety of errors can occur when a program is running. For example:
  - (real) user input error. bad url
  - device errors. remote server unavailable
  - physical limitations. full disk
  - code errors. interact with code that does not fulfill its contact (pre and post conditions)
- when an error occurs
  - return to safe state, save work, exit gracefully
- error handling code may be far removed from code that caused the error

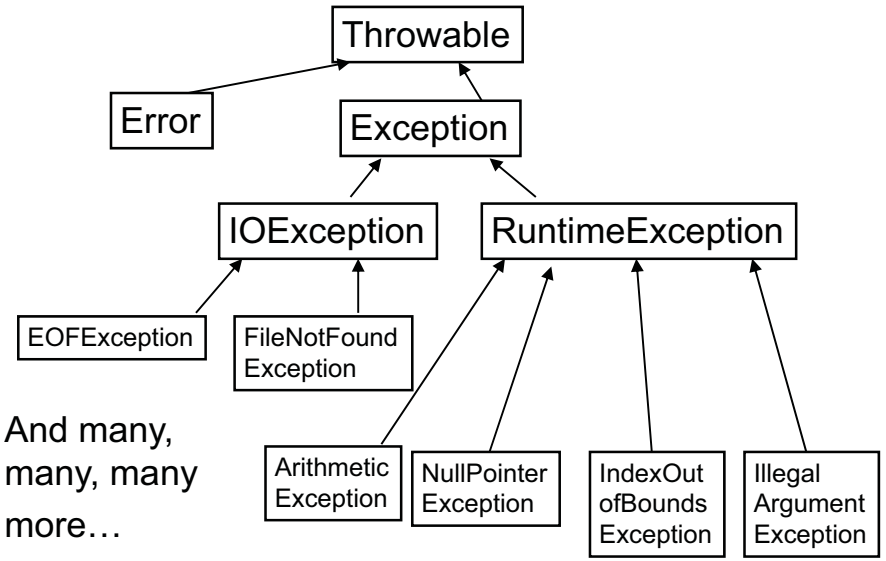
How to Handle Errors?

- It is possible to detect and handle errors of various types.
- Problem: this complicates the code and makes it harder to understand.
  - the error detection and error handling code have little or nothing to do with the *real* code is trying to do.
- A tradeoff between ensuring correct behavior under all possible circumstances and clarity of the code

Exceptions

- Many languages, including Java use a mechanism know as *Exceptions* to handle errors at runtime
  - In Java Exception is a class with many descendants.
  - **ArrayIndexOutOfBoundsException**
  - **NullPointerException**
  - **FileNotFoundException**
  - **ArithmeticException**
  - **IllegalArgumentException**

Partial Exceptions Hierarchy



And many, many, many more…

Creating Exceptions

- As a program runs, if a situation occurs that is handled by exceptions then an Exception is *thrown.*
  - An Exception object of the proper type is created
  - flow of control is transferred from the current block of code to code that can handle or deal with the exception
  - the normal flow of the program stops and error handling code takes over (if it exists.)

Attendance Question 1

Is it possible for the following method to result in an exception?

```
// pre: word != null
public static void printLength(String word){
   String output = "Word length is " + word.length();
   System.out.println( output );
}
```

A. Yes

B. No

Unchecked Exceptions

- Exceptions in Java fall into two different categories
  - *checked (other than Runtime) and unchecked (Runtime)*
- unchecked exceptions are ***completely preventable*** and should never occur.
  - They are caused by logic errors, created by us, the programmers.
- Descendents of the RuntimeException class
- Examples: ArrayIndexOutOfBoundsException, NullPointerException, ArithmeticException
- There does not *need* to be special error handling code
  - just regular error prevention code
- If error handling code was required programs would be unwieldy because so many Java statements have the possibility of generating or causing an unchecked Exception

Checked Exceptions

- "Checked exceptions represent conditions that, although exceptional, can reasonably be expected to occur, and if they do occur must be dealt with in some way.[other than the program terminating.]"
  - *Java Programming Language third edition*
- Unchecked exceptions are due to a programming logic error, our fault and preventable if coded correctly.
- Checked exceptions represent errors that are unpreventable by us!

Required Error Handling Code

- If you call a method that can generate a checked exception you must choose how to deal with that possible error
- For example one class for reading from files is the FileReader class

```
public FileReader(String fileName)
  throws FileNotFoundException
```

- This constructor has the possibility of throwing a `FileNotFoundException`
- `FileNotFoundException` is a checked exception

Checked Exceptions in Code

- If we have code that tries to build a FileReader we must deal with the possibility of the exception

```
import java.io.FileReader;

public class Tester
{
	public int countChars(String fileName)
	{	FileReader r = new FileReader(fileName);
		int total = 0;
		while( r.ready() )
		{	r.read();
			total++;
		}
		r.close();
		return total;
	}
}
```

- The code contains a syntax error. "unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown."

Handling Checked Exceptions

- In the code on the previous slide there are in fact 4 statements that can generate checked exceptions.
  - The FileReader constructor
  - the ready method
  - the read method
  - the close method
- To deal with the exceptions we can either state this method throws an Exception of the proper type or handle the exception within the method itself

Methods that throw Exceptions

- It may be that we don't know how to deal with an error within the method that can generate it
- In this case we will pass the buck to the method that called us
- The keyword `throws` is used to indicate a method has the possibility of generating an exception of the stated type
- Now any method calling ours must also throw an exception or handle it

Using the `throws` Keyword

```
public int countChars(String fileName)
      throws FileNotFoundException, IOException
{     int total = 0;
      FileReader r = new FileReader(fileName);
      while( r.ready() )
      {     r.read();
            total++;
      }
      r.close();
      return total;
}
```

- Now any method calling ours must also throw an exception or handle it

Using `try-catch` Blocks

- If you want to handle the a checked exception locally then use use the keywords `try` and `catch`
- the code that could cause an exception is placed in a block of code preceded by the keyword `try`
- the code that will handle the exception if it occurs is placed in a block of code preceded by the keyword `catch`

Sample `try` and `catch` Blocks

```
public int countChars(String fileName)
{     int total = 0;
      try
      {     FileReader r = new FileReader(fileName);
            while( r.ready() )
            {     r.read();
                  total++;
            }
            r.close();
      }
      catch(FileNotFoundException e)
      {     System.out.println("File named "
                  + fileName + "not found. " + e);
            total = -1;
      }
      catch(IOException e)
      {     System.out.println("IOException occured " +
                  "while counting chars. " + e);
            total = -1;
      }
      return total;
}
```

Mechanics of `try` and `catch`

- Code that could cause the checked exception is placed in a try block
  - note how the statements are included in one try block.
  - Each statement could be in a separate try block with an associated catch block, but that is very unwieldy (see next slide)
- Each try block must have 1 or more associated catch blocks
  - code here to handle the error. In this case we just print out the error and set result to -1

Gacky `try catch` Block

```
public int countChars3(String fileName)
{       int total = 0;
        FileReader r = null;
        try
        {       r = new FileReader(fileName);      }
        catch(FileNotFoundException e)
        {       System.out.println("File named "
                        + fileName + "not found. " + e);
                total = -1;       }
        try
        {       while( r.ready() )
                {       try
                        {       r.read();         }
                        catch(IOException e)
                        {       System.out.println("IOException "
                                        + "occurred while counting "
                                        + "chars. " + e);
                                total = -1;
                        }
                        total++;
                }
        }
        catch(IOException e)
        {     System.out.println("IOException occurred while counting chars. " + e);
              total = -1;}
        try
        {     r.close();
        }
        catch(IOException e)
        {     System.out.println("IOException occurred while counting chars. " + e);
              total = -1;}
        return total;
}
```


More `try catch` Mechanics

- If you decide to handle the possible exception locally in a method with the `try` block you must have a corresponding `catch` block
- the `catch` blocks have a parameter list of 1
- the parameter must be `Exception` or a descendant of `Exception`
- Use multiple `catch` blocks with one `try` block in case of multiple types of `Exceptions`

What Happens When Exceptions Occur

- If an exception is thrown then the normal flow of control of a program halts
- Instead of executing the regular statements the Java Runtime System starts to search for a matching catch block
- The first matching catch block based on data type is executed
- When the catch block code is completed the program does not "go back" to where the exception occurred.
  - It finds the next regular statement after the catch block

Counting Chars Again

```
public int countChars(String fileName)
{  int total = 0;
   try
   {    FileReader r = new FileReader(fileName);
        while( r.ready() )
        {       r.read();// what happens in an exception occurs?
                total++;
        }
        r.close();
   }
   catch(FileNotFoundException e)
   {    System.out.println("File named "
                     + fileName + "not found. " + e);
        total = -1;
   }
   catch(IOException e)
   {    System.out.println("IOException occured " +
                "while counting chars. " + e);
        total = -1;
   }
   return total;
}
```


Throwing Exceptions Yourself

- if you wish to throw an exception in your code you use the `throw` keyword
- Most common would be for an unmet precondition

```
public class Circle
{      private int iMyRadius;

       /** pre: radius > 0
        */
       public Circle(int radius)
       {      if (radius <= 0)
                    throw new IllegalArgumentException
                          ("radius must be > 0. "
                          + "Value of radius: " + radius);
              iMyRadius = radius;
       }
}
```


Attendance Question 2

What is output by the method `badUse` if it is called with the following code?

```
int[] nums = {3, 2, 6, 1};
badUse( nums );

public static void badUse(int[] vals){
     int total = 0;
     try{
         for(int i = 0; i < vals.length; i++){
             int index = vals[i];
             total += vals[index];
         }
     }
     catch(Exception e){
         total = -1;
     }
     System.out.println(total);
}
```

A. `1` B. `0` C. `3` D. `-1` E. `5`

Attendance Question 3

Is the use of a try-catch block on the previous question a proper use of try-catch blocks?

A. Yes

B. No

Error Handling, Error Handling Everywhere!

- Seems like a lot of choices for error prevention and error handling
  - normal program logic, e.g. if's for loop counters
  - assertions
  - try – catch block
- When is it appropriate to use each kind?

Error Prevention

- Us program logic, (ifs , fors) to prevent logic errors and unchecked exceptions
  - dereferencing a null pointer, going outside the bounds of an array, violating the preconditions of a method you are calling. e.g. the `charAt` method of the `String` class
  - use assertions as checks on your logic
    - you checked to ensure the variable `index` was within the array bounds with an if 10 lines up in the program and you are SURE you didn't alter it.
    - Use an `assert` right before you actually access the array

```
if( inbounds(index) )
{   // lots of related code
    // use an assertion before accessing
    arrayVar[index] = foo;
}
```


Error Prevention

- in 307 asserts can be used to check preconditions
  - Standard Java style is to use Exceptions
- Use try/catch blocks on checked exceptions
  - In general, don't use them to handle unchecked exceptions like NPE or AIOBE
- One place it is reasonable to use try / catch is in testing suites.
  - put each test in a try / catch. If an exception occurs that test fails, but other tests can still be run

File Input and Output

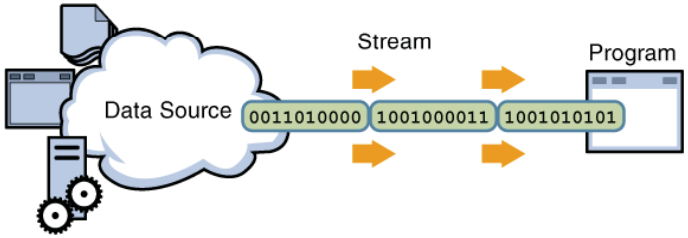
- Programs must often read from and write to files
  - large amounts of data
  - data not known at runtime
  - data that changes over time
- Each programming language has its own way of handling input and output
  - involves dealing with the operating system
  - if possible try to hide that fact as much as possible
- Java attempts to standardize input and output with the notion of a *stream,* an ordered sequence of data that has a source or destination.

Streams?

Dr. Egon Spengler:	Don't cross the streams.
Dr: Peter Venkman:	Why not?
Dr. Egon Spengler:	It would be bad.
Dr. Peter Venkman:	I'm fuzzy on the whole good/bad thing. what do you mean by "bad"?
Dr. Egon Spengler:	Try to imagine all life as you know it stopping instantaneously and every molecule in your body exploding at the speed of light.
Dr. Peter Venkman:	That's bad. Okay. Alright, important safety tip. Thanks Egon.

Streams

- A stream serves as a connection between your program and an external source or destination for bytes and bits
  - could be standard input or output, files, network connections, or other programs



Lots of Streams

- Java has over sixty (60!) different stream types in its java.io package
- part of the reason input and output are so difficult to understand in Java is the size and diversity of the IO library
- The type of stream you use depends on what you are trying to do
  - even then there are multiple options

Working with Files in Java

- Data is stored in digital form, 1s and 0s
- Work with these in packages of 8, the byte
- The IO library creates higher level abstractions so we think we are working with characters, Strings, or whole objects
- Some abstract classes
  - InputStream, OutputStream, Reader, and Writer
  - InputStream and OutputStream represent the flow of data (a stream)
  - Reader and Writer are used to read the data from a stream or put the data in a stream
  - *convenience classes* exist to make things a little easier

The Scanner class

- A class to make reading from source of input easier. New in Java 5.0
- Constructors

  Scanner(InputStream)

  Scanner(File)
- Methods to read lines from input

  boolean hasNextLine()
  String nextLine()
- Methods to read ints

  int readInt(), boolean hasNext()

Scanner and Keyboard Input

- No exceptions thrown!
  – no try – catch block necessary!
- Set delimiters with regular expressions, default is whitespace

```
Scanner s = new Scanner(System.in);

System.out.print("Enter your name: ");
String name = s.nextLine();

System.out.print("Press Enter to continue: ");
s.nextLine();
```


Hooking a Scanner up to a File

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

public class ReadAndPrintScores
{
    public static void main(String[] args)
    {   try
        {   Scanner s = new Scanner( new File("scores.dat") );
            while( s.hasNextInt() )
            {   System.out.println( s.nextInt() );
            }
             s.close();
        }
        catch(IOException e)
        {       System.out.println( e );
        }
    }
}
```

```
12 35 12
12 45
12

12

13 57
```

scores.dat

Writing to a File

```
//sample code to write 100 random ints to a file, 1 per line

import java.io.PrintStream;
import java.io.IOException;
import java.io.File;

import java.util.Random;

public class WriteToFile
{       public static void main(String[] args)
        {       try
                {       PrintStream writer = new PrintStream( new
                               File("randInts.txt"));
                        Random r = new Random();
                        final int LIMIT = 100;

                        for(int i = 0; i < LIMIT; i++)
                        {       writer.println( r.nextInt() );
                        }
                        writer.close();
                }
                catch(IOException e)
                {       System.out.println("An error occurred " +
                        + "while trying to write to the file");
                }
        }
}
```

Reading From a Web Page

```
public static void main(String[] args) {
   try {
      String siteUrl = "http://www.cs.utexas.edu/~scottm/cs307");
    URL mySite = new URL(siteURL);
      URLConnection yc = mySite.openConnection();
      Scanner in =
         new Scanner(new InputStreamReader(yc.getInputStream()));
      int count = 0;
      while (in.hasNext()) {
         System.out.println(in.next());
         count++;
      }
      System.out.println("Number of tokens: " + count);
      in.close();
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Topic 5
Implementing Classes

"And so, from Europe, we get things such ... object-oriented analysis and design (a clever way of breaking up software programming instructions and data into small, reusable objects, based on certain abstraction principles and design hierarchies.)"

-Michael A. Cusumano, The Business Of Software

Definitions

Object Oriented Programming

- What is object oriented programming?
- "Object-oriented programming is a method of programming based on a hierarchy of classes, and well-defined and cooperating objects. "
- What is a class?
- "A class is a structure that defines the data and the methods to work on that data. When you write programs in the Java language, all program data is wrapped in a class, whether it is a class you write or a class you use from the Java platform API libraries."
  - Sun code camp

Classes Are ...

- Another, simple definition:
- A *class* is a programmer defined data type.
- A *data type* is a set of possible values and the operations that can be performed on those values
- Example:
  - single digit positive base 10 ints
  - 1, 2, 3, 4, 5, 6, 7, 8, 9
  - operations: add, subtract
  - problems?

Data Types

- Computer Languages come with built in data types
- In Java, the primitive data types, native arrays
- Most computer languages provide a way for the programmer to define their own data types
  - Java comes with a large library of classes
- So object oriented programming is a way of programming that is dominated by creating new data types to solve a problem.
- We will look at how to create a new data type

A Very Short and Incomplete History of Object Oriented Programming. (OOP)

OOP is not new.

- Simula 1 (1962 - 1965) and Simula 67 (1967) Norwegian Computing Center, Oslo, Norway by Ole-Johan Dahl and Kristen Nygaard.

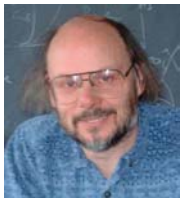
Dahl and Nygaard at the time of Simula's development

Turing Award Winners - 2001

OOP Languages

- Smalltalk (1970s), Alan Kay's group at Xerox PARC

- C++ (early 1980s), Bjarne Stroustrup, Bell Labs



OOP Languages

- Modula – 3, Oberon, Eiffel, Java, C#, Python
  - many languages have some Object Oriented version or capability
- One of the dominant styles for implementing complex programs with large numbers of interacting components
  - … but not the only programming paradigm and there are variations on object oriented programming

Program Design in OOP

- OOP breaks up problems based on the data types found in the problem
  - as opposed to breaking up the problem based on the algorithms involved
- Given a problem statement, what *things* appear in the problem?
- The nouns of the problem are candidate classes.
- The actions and verbs of the problems are candidate methods of the classes

Short Object Oriented Programming Design Example

Attendance Question 1

The process of taking a large problem and breaking it up into smaller parts is known as:

A. Functional programming
B. Object oriented programming
C. Top down design
D. Bottom up design
E. Waterfall method

Monopoly

If we had to start from scratch what classes would we need to create?

Individual Class Design

The Steps of Class Design

- Requirements
  - what is the problem to be solved
  - detailed requirements lead to specifications
- Nouns may be classes
- Verbs signal behavior and thus methods (also defines a classes responsibilities)
- walkthrough scenarios to find nouns and verbs
- implementing and testing of classes
- design rather than implementation is normally the hardest part
  - planning for reuse

Class Design

- Classes should be *cohesive*.
  - They should be designed to do one thing well.

- Classes should be *loosely coupled.*
  - Changing the **internal** implementation details of a class should not affect other classes.
  - loose coupling can also be achieved within a class itself

Encapsulation

- Also know as separation of concerns and information hiding
- When creating new data types (classes) the details of the actual data and the way operations work is hidden from the other programmers who will use those new data types
  - So they don't have to worry about them
  - So they can be changed without any ill effects (loose coupling)
- Encapsulation makes it easier to be able to use something
  - microwave, radio, ipod, the Java String class

Design to Implementation

- Design must me implemented using the syntax of the programming language
- In class example with a list of integers
- Slides include another example of creating a class to represent a playing die

A List of ints

The Problem with Arrays

- Suppose I need to store a bunch of film titles from a file

The Godfather
The Princess Bride
The Incredible

```
String[] titles = new String[100];
// I never know how much
// space I need!
```

- I want the array to grow and shrink

Lists

- I need a *list*.
- A list is a collection of items with a definite order.
- Our example will be a list of integers.
- Design and then implement to demonstrate the Java syntax for creating a class.

Attendance Question 2

When adding a new element to a list what should be the default location to add?

A. The beginning

B. The end

C. The middle

D. A random location

IntList Design

- Create a new, empty IntList

```
new IntList -> []
```

- The above is not code. It is a notation that shows what the results of operations. `[]` is an empty list.
- add to a list.

```
[].add(1) -> [1]
[1].add(5) -> [1, 5]
[1, 5].add(4) -> [1, 5, 4]
```

- elements in a list have a definite order and a position.
  - zero based position or 1 based positioning?

Instance Variables

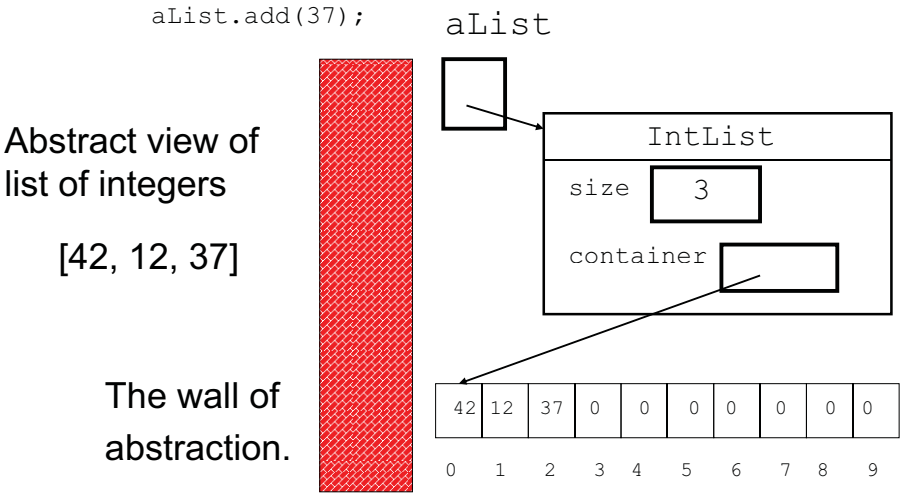
- Internal data
  - also called instance variables because every instance (object) of this class has its own copy of these
  - something to store the elements of the list
  - size of internal storage container?
  - if not what else is needed
- Must be clear on the difference between the internal data of an IntList object and the IntList that is being represented
- Why make internal data private?

Attendance Question 3

Our `IntList` class will have an instance variable of `int`s (`int[] container`). What should the capacity of this internal array be?

A. less than or equal to the size of the list
B. greater than or equal to the size of the list
C. equal to the size of the list
D. some fixed amount that never changes
E. 0


```
IntList aList = new IntList();
aList.add(42);
aList.add(12);
aList.add(37);
```



Abstract view of list of integers

[42, 12, 37]

The wall of abstraction.

Constructors

- For initialization of objects
- IntList constructors
  - default
  - initial capacity?
- redirecting to another constructor

```
this(10);
```

- class constants
  - what `static` means

Default add method

- where to add?
- what if not enough space?

```
[].add(3) -> [3]
[3].add(5) -> [3, 5]
[3, 5].add(3) -> [3, 5, 3]
```

- Testing, testing, testing!
  - a `toString` method would be useful

toString method

- return a Java String of list
- empty list -> `[]`
- one element -> `[12]`
- multiple elements -> `[12, 0, 5, 4]`
- Beware the performance of `String` concatenation.
- `StringBuffer` alternative

Attendance Question 4

What is output by the following code?

```
IntList list = new IntList();
System.out.println( list.size() );
```

A. `10`

B. `0`

C. `-1`

D. unknown

E. No output due to runtime error.

get and size methods

- get
  - access element from list
  - preconditions?

`[3, 5, 2].get(0)` returns `3`

`[3, 5, 2].get(1)` returns `5`

- size
  - number of elements in the list
  - Do not confuse with the capacity of the internal storage container
  - The array is not the list!

`[4, 5, 2].size()` returns `3`

insert method

- add at someplace besides the end

`[3, 5].insert(1, 4) -> [3, 4, 5]`

where what

`[3, 4, 5].insert(0, 4) -> [4, 3, 4, 5]`

- preconditions?
- overload add?
- chance for internal loose coupling

Attendance Question 5

What is output by the following code?

```
IntList list = new IntList();
list.add(3);
list.insert(0, 4);
list.insert(1, 1);
list.add(5);
list.insert(2, 9);
System.out.println( list.toString() );
```

A. `[4, 1, 3, 9, 5]`
B. `[3, 4, 1, 5, 9]`
C. `[4, 1, 9, 3, 5]`
D. `[3, 1, 4, 9, 5]`
E. No output due to runtime error.

remove method

- remove an element from the list based on location

`[3, 4, 5].remove(0) -> [4, 5]`

`[3, 5, 6, 1, 2].remove(2) ->`
`[3, 5, 1, 2]`

- preconditions?
- return value?
  - accessor methods, mutator methods, and mutator methods that return a value

Attendance Question 6

What is output by the following code?

```
IntList list = new IntList();
list.add(12);
list.add(15);
list.add(12);
list.add(17);
list.remove(1);
System.out.println( list );
```

A. `[15, 17]`
B. `[12, 17]`
C. `[12, 0, 12, 17]`
D. `[12, 12, 17]`
E. `[15, 12, 17]`

insertAll method

- add all elements of one list to another starting at a specified location

`[5, 3, 7].insertAll(2, [2, 3]) ->`
`[5, 3, 2, 3, 7]`

The parameter `[2, 3]` would be unchanged.

- Working with other objects of the same type
  - `this`?
  - where is private private?
  - loose coupling vs. performance

Class Design and Implementation – Another Example

This example will not be covered in class.

The Die Class

- Consider a class used to model a die
- What is the interface? What actions should a die be able to perform?
- The methods or behaviors can be broken up into constructors, mutators, accessors

The Die Class Interface

- Constructors (used in creation of objects)
  - default, single int parameter to specify the number of sides, int and boolean to determine if should roll
- Mutators (change state of objects)
  - roll
- Accessors (do not change state of objects)
  - getResult, getNumSides, toString
- Public constants
  - DEFAULT_SIDES

Visibility Modifiers

- All parts of a *class* have visibility modifiers
  - Java keywords
  - **public**, protected, **private,** (no modifier means package access)
  - do not use these modifiers on local variables (syntax error)
- **public** means that constructor, method, or field may be accessed outside of the class.
  - part of the interface
  - constructors and methods are generally public
- **private** means that part of the class is hidden and inaccessible by code outside of the class
  - part of the implementation
  - data fields are generally private

The Die Class Implementation

- Implementation is made up of constructor code, method code, and private data members of the class.
- scope of data members / instance variables
  - *private data members may be used in any of the constructors or methods of a class*
- Implementation is hidden from users of a class and can be changed without changing the interface or affecting clients (other classes that use this class)
  - Example: Previous version of Die class, DieVersion1.java
- Once Die class completed can be used in anything requiring a Die or situation requiring random numbers between 1 and N
  - DieTester class. What does it do?

DieTester method

```
public static void main(String[] args) {
       final int NUM_ROLLS = 50;
       final int TEN_SIDED = 10;
       Die d1 = new Die();
       Die d2 = new Die();
       Die d3 = new Die(TEN_SIDED);
       final int MAX_ROLL = d1.getNumSides() +
              d2.getNumSides() + d3.getNumSides();


       for(int i = 0; i < NUM_ROLLS; i++)
       {      d1.roll();
              d2.roll();
              System.out.println("d1: " + d1.getResult()
                     + " d2: " + d2.getResult() + " Total: "
                     + (d1.getResult() + d2.getResult() ) );
       }
```


DieTester continued

```
int total = 0;
int numRolls = 0;
do
{      d1.roll();
       d2.roll();
       d3.roll();
       total = d1.getResult() + d2.getResult()
              + d3.getResult();
       numRolls++;
}
while(total != MAX_ROLL);

System.out.println("\n\nNumber of rolls to get "
       + MAX_ROLL + " was " + numRolls);
```


Correctness Sidetrack

- When creating the public interface of a class give careful thought and consideration to the *contract* you are creating between yourself and users (other programmers) of your class
- Use *preconditions* to state what you assume to be true before a method is called
  - caller of the method is responsible for making sure these are true
- Use post*conditions* to state what you guarantee to be true after the method is done if the preconditions are met
  - implementer of the method is responsible for making sure these are true

Precondition and Postcondition Example

```
/* pre: numSides > 1
   post: getResult() = 1, getNumSides() = sides
*/
 public Die(int numSides)
 {   assert (numSides > 1) : "Violation of precondition: Die(int)";
      iMyNumSides = numSides;
      iMyResult = 1;
      assert getResult() == 1 && getNumSides() == numSides;
  }
```

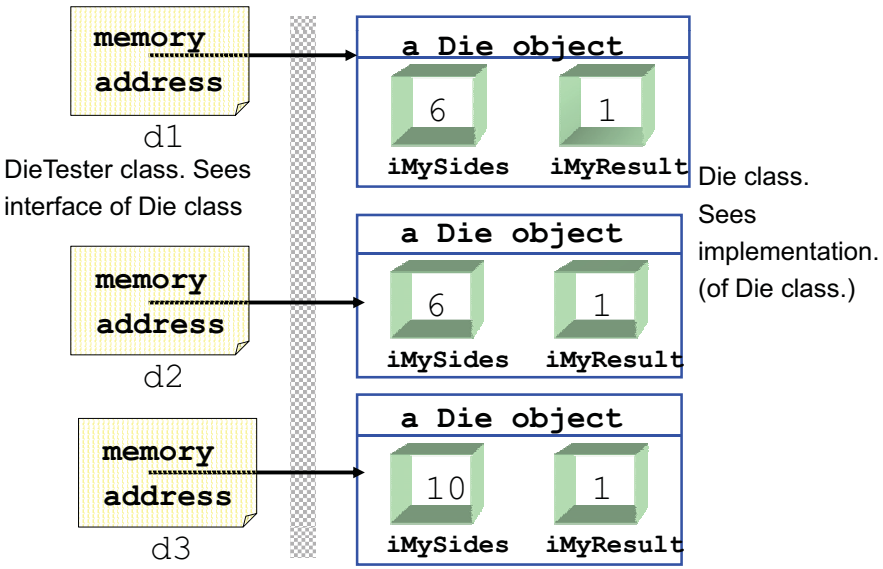

Object Behavior - Instantiation

- Consider the DieTester class

```
Die d1 = new Die();
Die d2 = new Die();
Die d3 = new Die(10);
```

- When the new operator is invoked control is transferred to the Die class and the specified constructor is executed, based on parameter matching
- Space(memory) is set aside for the new object's fields
- The memory address of the new object is passed back and stored in the object variable (pointer)
- After creating the object, methods may be called on it.

Creating Dice Objects



Objects

- Every Die object created has its own instance of the variables declared in the class blueprint

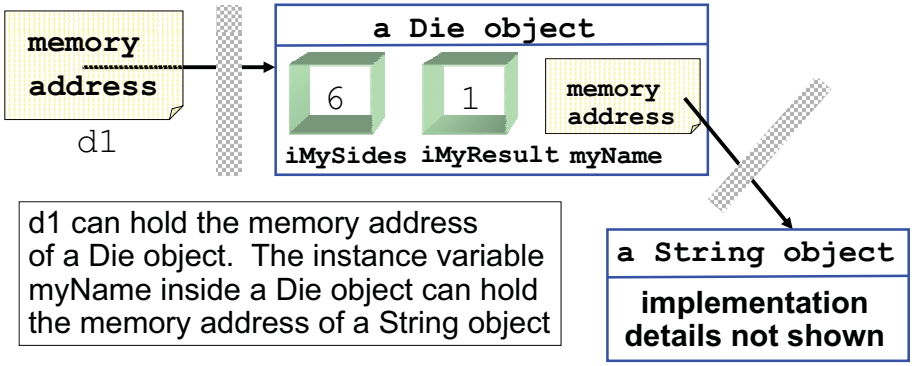
```
private int iMySides;
private int iMyResult;
```

- thus the term *instance variable*
- the instance vars are part of the hidden implementation and may be of *any* data type
  - unless they are public, which is almost always a bad idea if you follow the tenets of information hiding and encapsulation

Complex Objects

- What if one of the instance variables is itself an object?
- add to the Die class

```
private String myName;
```



d1 can hold the memory address of a Die object. The instance variable myName inside a Die object can hold the memory address of a String object

The Implicit Parameter

- Consider this code from the Die class

```
public void roll()
{   iMyResult =
        ourRandomNumGen.nextInt(iMySides) + 1;
}
```

- Taken in isolation this code is rather confusing.
- what is this iMyResult thing?
  - It's not a parameter or local variable
  - why does it exist?
  - *it belongs to the Die object that called this method*
  - if there are numerous Die objects in existence
  - Which one is used depends on which object called the method.

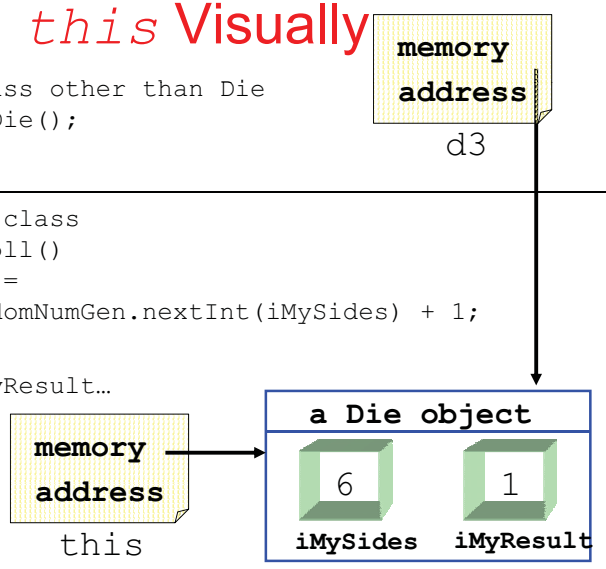
The *this* Keyword

- When a method is called it may be necessary for the calling object to be able to refer to itself
  - most likely so it can pass itself somewhere as a parameter
- when an object calls a method an implicit reference is assigned to the calling object
- the name of this implicit reference is `this`
- `this` is a reference to the current calling object and may be used as an object variable (may not declare it)

this Visually

```
// in some class other than Die
Die d3 = new Die();
d3.roll();
```

```
// in the Die class
public void roll()
{   iMyResult =
        ourRandomNumGen.nextInt(iMySides) + 1;
    /* OR
      this.iMyResult…
    */
}
```



An equals method

- working with objects of the same type in a class can be confusing
- write an equals method for the Die class. assume every Die has a myName instance variable as well as iMyNumber and iMySides

A Possible Equals Method

```
public boolean equals(Object otherObject)
{     Die other = (Die)otherObject;
      return iMySides == other.iMySides
            && iMyResult== other.iMyResult
            && myName.equals( other.myName );
}
```

- Declared Type of Parameter is Object not Die
- override (replace) the equals method instead of overload (present an alternate version)
  - easier to create generic code
- we will see the equals method is *inherited* from the Object class
- access to another object's private instance variables?

Another equals Methods

```
public boolean equals(Object otherObject)
{     Die other = (Die)otherObject;
      return this.iMySides == other.iMySides
            && this.iMyNumber == other.iMyNumber
            && this.myName.equals( other.myName );
}
```

Using the this keyword / reference to access the implicit parameters instance variables is unnecessary.
If a method within the same class is called within a method, the original calling object is still the calling object

A "Perfect" Equals Method

- From Cay Horstmann's *Core Java*

```
public boolean equals(Object otherObject)
{     // check if objects identical
      if( this == otherObject)
            return true;
      // must return false if explicit parameter null
      if(otherObject == null)
            return false;
      // if objects not of same type they cannot be equal
      if(getClass() != otherObject.getClass() )
            return false;
      // we know otherObject is a non null Die
      Die other = (Die)otherObject;
      return iMySides == other.iMySides
            && iMyNumber == other.iMyNumber
            && myName.equals( other.myName );
}
```

the instanceof Operator

- `instanceof` is a Java keyword.
- part of a boolean statement

```
public boolean equals(Object otherObj)
{  if otherObj instanceof Die
   {    //now go and cast
        // rest of equals method
   }
}
```

- Should not use instanceof in equals methods.
- `instanceof` has its uses but not in equals because of the contract of the equals method

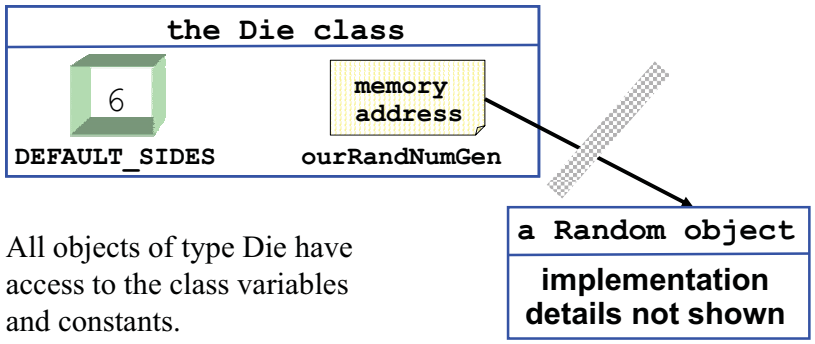
Class Variables and Class Methods

- Sometimes every object of a class does not need its own copy of a variable or constant
- The keyword `static` is used to specify class variables, constants, and methods

```
private static Random ourRandNumGen
   = new Random();
public static final int DEFAULT_SIDES = 6;
```

- The most prevalent use of static is for class constants.
  - if the value can't be changed why should every object have a copy of this non changing value

Class Variables and Constants



All objects of type Die have access to the class variables and constants.

A public class variable or constant may be referred to via the class name.

Syntax for Accessing Class Variables

```
public class UseDieStatic
{  public static void main(String[] args)
   {  System.out.println( "Die.DEFAULT_SIDES "
         + Die.DEFAULT_SIDES );
      // Any attempt to access Die.ourRandNumGen
      // would generate a syntax error

      Die d1 = new Die(10);

      System.out.println( "Die.DEFAULT_SIDES "
         + Die.DEFAULT_SIDES );
      System.out.println( "d1.DEFAULT_SIDES "
         + d1.DEFAULT_SIDES );

      // regardless of the number of Die objects in
      // existence, there is only one copy of DEFAULT_SIDES
      // in the Die class

   } // end of main method
} // end of UseDieStatic class
```

Static Methods

- `static` has a somewhat different meaning when used in a method declaration
- static methods may not manipulate any instance variables
- in non static methods, some object invokes the method

```
d3.roll();
```

- the object that makes the method call is an implicit parameter to the method

Static Methods Continued

- Since there is no implicit object parameter sent to the static method it does not have access to a copy of any objects instance variables
  - unless of course that object is sent as an explicit parameter
- Static methods are normally utility methods or used to manipulate static variables (class variables)
- The Math and System classes are nothing but static methods

static and this

- Why does this work (added to Die class)

```
public class Die
{
     public void outputSelf()
     { System.out.println( this );
     }
}
```

- but this doesn't?

```
public class StaticThis
{
     public static void main(String[] args)
     { System.out.println( this );
     }
}
```

Topic 6

Inheritance and Polymorphism

"Question: What is the object oriented way of getting rich?
Answer: Inheritance."

"Inheritance is new code that reuses old code. Polymorphism is old code that reuses new code."

Outline

- Explanation of inheritance.
- Using inheritance to create a SortedIntList.
- Explanation of polymorphism.
- Using polymorphism to make a more generic List class.

Explanation of Inheritance

Main Tenets of OO Programming

- Encapsulation
  - abstraction, information hiding
- Inheritance
  - code reuse, specialization "New code using old code."
- Polymorphism
  - do X for a collection of various types of objects, where X is *different* depending on the type of object
  - "Old code using new code."

Things and Relationships

- Object oriented programming leads to programs that are models
  - sometimes models of things in the real world
  - sometimes models of contrived or imaginary things
- There are many types of relationships between the things in the models
  - chess piece has a position
  - chess piece has a color
  - chess piece moves (changes position)
  - chess piece is taken
  - a rook is a type of chess piece

The "has-A" Relationship

- Objects are often made up of many parts or have sub data.
  - chess piece: position, color
  - die: result, number of sides
- This "has-a" relationship is modeled by composition
  - the instance variables or fields internal to objects
- Encapsulation captures this concept

The "is-a" relationship

- Another type of relationship found in the real world
  - a rook is a chess piece
  - a queen is a chess piece
  - a student is a person
  - a faculty member is a person
  - an undergraduate student is a student
- "is-a" usually denotes some form of specialization
- it is not the same as "has-a"

Inheritance

- The "is-a" relationship, and the specialization that accompanies it, is modeled in object oriented languages via *inheritance*
- Classes can inherit from other classes
  - base inheritance in a program on the real world things being modeled
  - does "an A is a B" make sense? Is it logical?

Nomenclature of Inheritance

- In Java the `extends` keyword is used in the class header to specify which preexisting class a new class is inheriting from

```
public class Student extends Person
```

- Person is said to be
  - the parent class of Student
  - the super class of Student
  - the base class of Student
  - an ancestor of Student
- Student is said to be
  - a child class of Person
  - a sub class of Person
  - a derived class of Person
  - a descendant of Person

Results of Inheritance

public class A

public class B extends A

- the sub class inherits (gains) all instance variables and instance methods of the super class, **automatically**
- additional methods can be added to class B (specialization)
- the sub class can replace (redefine, override) methods from the super class

Attendance Question 1

What is the primary reason for using inheritance when programming?

A. To make a program more complicated

B. To duplicate code between classes

C. To reuse pre-existing code

D. To hide implementation details of a class

E. To ensure pre conditions of methods are met.

Inheritance in Java

- Java is a pure object oriented language
- all code is part of some class
- all classes, except one, must inherit from exactly one other class
- The `Object` class is the *cosmic super class*
  - The `Object` class does not inherit from any other class
  - The `Object` class has several important methods: `toString`, `equals`, `hashCode`, `clone`, `getClass`
- implications:
  - all classes are descendants of `Object`
  - all classes and thus all objects have a `toString`, `equals`, `hashCode`, `clone`, and `getClass` method
    - `toString`, `equals`, `hashCode`, `clone` normally overridden

Inheritance in Java

- If a class header does not include the extends clause the class extends the `Object` class by default

  `public class Die`

  - `Object` is an ancestor to all classes
  - it is the only class that does not extend some other class
- A class extends exactly one other class
  - extending two or more classes is *multiple inheritance.* Java does not support this directly, rather it uses *Interfaces*.

Overriding methods

- any method that is not `final` may be overridden by a descendant class
- same signature as method in ancestor
- may not reduce visibility
- may use the original method if simply want to add more behavior to existing

Attendance Question 2

What is output when the `main` method is run?

```
public class Foo{
    public static void main(String[] args){
        Foo f1 = new Foo();
        System.out.println( f1.toString() );
    }
}
```

A. `0`

B. `null`

C. Unknown until code is actually run.

D. No output due to a syntax error.

E. No output due to a runtime error.

Shape Classes

- Declare a class called `ClosedShape`
  - assume all shapes have x and y coordinates
  - override `Object`'s version of `toString`
- Possible sub classes of `ClosedShape`
  - `Rectangle`
  - `Circle`
  - `Ellipse`
  - `Square`
- Possible hierarchy

  `ClosedShape` <- `Rectangle` <- `Square`

A ClosedShape class

```
public class ClosedShape
{  private double myX;
   private double myY;

   public ClosedShape()
   {  this(0,0);  }

   public ClosedShape (double x, double y)
   {  myX = x;
      myY = y;
   }

   public String toString()
   { return "x: " + getX() + " y: " + getY();   }

   public double getX(){ return myX; }
   public double getY(){ return myY; }
}
// Other methods not shown
```


Constructors

- Constructors handle initialization of objects
- When creating an object with one or more ancestors (every type except Object) a chain of constructor calls takes place
- The reserved word `super` may be used in a constructor to call a one of the parent's constructors
  - must be first line of constructor
- if no parent constructor is explicitly called the default, 0 parameter constructor of the parent is called
  - if no default constructor exists a syntax error results
- If a parent constructor is called another constructor in the same class may no be called
  - no `super();this();` allowed. One or the other, not both
  - good place for an initialization method

A Rectangle Constructor

```
public class Rectangle extends ClosedShape
{  private double myWidth;
   private double myHeight;

   public Rectangle( double x, double y,
          double width, double height )
   {   super(x,y);
       // calls the 2 double constructor in
       // ClosedShape
       myWidth = width;
       myHeight = height;

   }

    // other methods not shown
}
```


A Rectangle Class

```
public class Rectangle extends ClosedShape
{  private double myWidth;
   private double myHeight;

   public Rectangle()
   {  this(0, 0);
   }

   public Rectangle(double width, double height)
   {  myWidth = width;
      myHeight = height;
   }

   public Rectangle(double x, double y,
               double width, double height)
   {   super(x, y);
       myWidth = width;
       myHeight = height;
   }

   public String toString()
   {  return super.toString() + " width " + myWidth
         + " height " + myHeight;
   }
}
```

The Keyword `super`

- `super` is used to access something (any protected or public field or method) from the super class that has been overridden
- Rectangle's `toString` makes use of the `toString` in ClosedShape my calling `super.toString()`
- without the super calling `toString` would result in infinite recursive calls
- Java does not allow nested supers

  `super.super.toString()`

  results in a syntax error even though technically this refers to a valid method, `Object`'s toString
- Rectangle *partially* overrides ClosedShapes `toString`

Initialization method

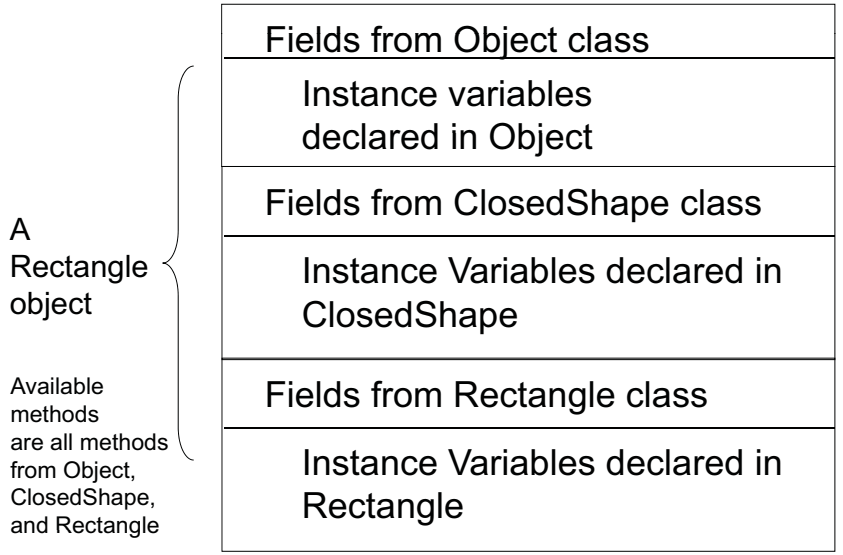
```
public class Rectangle extends ClosedShape
{  private double myWidth;
   private double myHeight;

   public Rectangle()
   {  init(0, 0);
   }

   public Rectangle(double width, double height)
   {  init(width, height);
   }

   public Rectangle(double x, double y,
                    double width, double height)
   {  super(x, y);
      init(width, height);
   }

   private void init(double width, double height)
   {  myWidth = width;
      myHeight = height;
   }
```


Result of Inheritance

Do any of these cause a syntax error?
What is the output?

```
Rectangle r = new Rectangle(1, 2, 3,
4);
ClosedShape s = new CloseShape(2, 3);
System.out.println( s.getX() );
System.out.println( s.getY() );
System.out.println( s.toString() );
System.out.println( r.getX() );
System.out.println( r.getY() );
System.out.println( r.toString() );
System.out.println( r.getWidth() );
```


The Real Picture



Access Modifiers and Inheritance

- public
  - accessible to all classes
- private
  - accessible only within that class. Hidden from all sub classes.
- protected
  - accessible by classes within the same *package* and all descendant classes
- Instance variables *should* be private
- protected methods are used to allow descendant classes to modify instance variables in ways other classes can't

Why private Vars and not protected?

- In general it is good practice to make instance variables private
  - hide them from your descendants
  - if you think descendants will need to access them or modify them provide protected methods to do this
- Why?
- Consider the following example

Required update

```
public class GamePiece
{       private Board myBoard;
        private Position myPos;

        // whenever my position changes I must
        // update the board so it knows about the change

        protected void alterPos( Position newPos )
        {       Position oldPos = myPos;
                myPos = newPos;
                myBoard.update( oldPos, myPos );
        }
```


Creating a SortedIntList

A New Class

- Assume we want to have a list of ints, but that the ints must always be maintained in ascending order

  `[-7, 12, 37, 212, 212, 313, 313, 500]`

  `sortedList.get(0)` returns the min

  `sortedList.get( list.size() - 1 )` returns the max

Implementing `SortedIntList`

- Do we have to write a whole new class?
- Assume we have an `IntList` class.
- Which of the following methods would have to be changed?

```
add(int value)
int get(int location)
String toString()
int size()
int remove(int location)
```


Overriding the `add` Method

- First attempt
- Problem?
- solving with `protected`
  - What `protected` really means
- solving with insert method
  - double edged sort

Problems

- What about this method?

```
void insert(int location, int val)
```

- What about this method?

```
void insertAll(int location,
    IntList otherList)
```

- `SortedIntList` is not the cleanest application of inheritance.

Explanation of Polymorphism

Polymorphism

- Another feature of OOP
- literally "having many forms"
- object variables in Java are polymorphic
- object variables can refer to objects or their declared type AND any objects that are descendants of the declared type

```
ClosedShape s = new
ClosedShape();
s = new Rectangle(); // legal!
s = new Circle(); //legal!
Object obj1; // = what?
```


Data Type

- object variables have:
  - a *declared type*. Also called the static type.
  - a *dynamic type*. What is the actual type of the pointee at run time or when a particular statement is executed.
- Method calls are syntactically legal if the method is in the declared type *or any ancestor* of the declared type
- **The actual method that is executed at runtime is based on the dynamic type**
  - dynamic dispatch

Attendance Question 3

Consider the following class declarations:

```
public class BoardSpace
public class Property extends BoardSpace
public class Street extends Property
public class Railroad extends Property
```

Which of the following statements would cause a syntax error? Assume all classes have a default constructor.

A. `Object obj = new Railroad();`
B. `Street s = new BoardSpace();`
C. `BoardSpace b = new Street();`
D. `Railroad r = new Street();`
E. More than one of these

What's the Output?

```
ClosedShape s = new ClosedShape(1,2);
System.out.println( s.toString() );
s = new Rectangle(2, 3, 4, 5);
System.out.println( s.toString() );
s = new Circle(4, 5, 10);
System.out.println( s.toString() );
s = new ClosedShape();
System.out.println( s.toString() );
```


Method LookUp

- To determine if a method is legal the compiler looks in the class based on the declared type
  - if it finds it great, if not go to the super class and look there
  - continue until the method is found, or the Object class is reached and the method was never found. (Compile error)
- To determine which method is actually executed the run time system
  - starts with the actual run time class of the object that is calling the method
  - search the class for that method
  - if found, execute it, otherwise go to the super class and keep looking
  - repeat until a version is found
- Is it possible the runtime system won't find a method?

Attendance Question 4

What is output by the code to the right when run?

A. `!!live`
B. `!eggegg`
C. `!egglive`
D. `!!!`
E. `eggegglive`

```
public class Animal{
  public String bt(){ return "!"; }
}

public class Mammal extends Animal{
  public String bt(){ return "live"; }
}

public class Platypus extends Mammal{
  public String bt(){ return "egg";}
}

Animal a1 = new Animal();
Animal a2 = new Platypus();
Mammal m1 = new Platypus();
System.out.print( a1.bt() );
System.out.print( a2.bt() );
System.out.print( m1.bt() );
```


Why Bother?

- Inheritance allows programs to model relationships in the real world
  - if the program follows the model it may be easier to write
- Inheritance allows code reuse
  - complete programs faster (especially large programs)
- Polymorphism allows code reuse in another way (We will explore this next time)
- Inheritance and polymorphism allow programmers to create *generic algorithms*

Genericity

- One of the goals of OOP is the support of code reuse to allow more efficient program development
- If a algorithm is essentially the same, but the code would vary based on the data type genericity allows only a single version of that code to exist
  - some languages support genericity via *templates*
  - in Java, there are 2 ways of doing this
    - polymorphism and the inheritance requirement
    - generics

the createASet example

```
public Object[] createASet(Object[] items)
{    /*
     pre: items != null, no elements
     of items = null
     post: return an array of Objects
     that represents a set of the elements
     in items. (all duplicates removed)
     */
```

```
{5, 1, 2, 3, 2, 3, 1, 5} -> {5, 1, 2, 3}
```


createASet examples

```
String[] sList = {"Texas", "texas", "Texas",
                  "Texas", "UT", "texas"};
Object[] sSet = createASet(sList);
for(int i = 0; i < sSet.length; i++)
    System.out.println( sSet[i] );

Object[] list = {"Hi", 1, 4, 3.3, true,
                 new ArrayList(), "Hi", 3.3, 4};
Object[] set = createASet(list);
for(int i = 0; i < set.length; i++)
    System.out.println( set[i] );
```


A Generic List Class

Back to `IntList`

- We may find `IntList` useful, but what if we want a List of `String`s? `Rectangle`s? `List`s?
  - What if I am not sure?
- Are the List algorithms going to be very different if I am storing `String`s instead of `int`s?
- How can we make a generic List class?

Generic List Class

- required changes
- How does `toString` have to change?
  - why?!?!
  - A good example of why keyword `this` is necessary from `toString`
- What can a `List` hold now?
- How many `List` classes do I need?

Writing an `equals` Method

- How to check if two objects are equal?

```
if(objA == objA)
    // does this work?
```

- Why not this

```
public boolean equals(List other)
```

- Because

```
public void foo(List a, Object b)
    if( a.equals(b) )
      System.out.println( same )
```

  - what if b is really a `List`?

`equals` method

- read the javadoc carefully!
- don't rely on `toString` and `String`'s equal
- lost of cases

Topic 7

Interfaces and Abstract Classes

"I prefer Agassiz in the abstract, rather than in the concrete."

Interfaces

Multiple Inheritance

- The are classes where the "is-a" test is true for more than one other class
  - a graduate teaching assistant is a graduate students
  - a graduate teaching assistant is a faculty member
- Java requires all classes to inherit from exactly one other class
  - does not allow multiple inheritance
  - some object oriented languages do

Problems with Multiple Inheritance

- Suppose multiple inheritance was allowed

```
public class GradTA extends Faculty, GradStudent
```

- Suppose Faculty overrides toString and that GradStudent overrides toString as well

```
GradTA ta1 = new GradTA();
System.out.println( ta1.toString() );
```

- What is the problem
- Certainly possible to overcome the problem
  - provide access to both (scope resolution in C++)
  - require GradTA to pick a version of toString or override it itself (Eiffel)

Interfaces – Not quite Multiple Inheritance

- Java does not allow multiple inheritance
  - syntax headaches not worth the benefits
- Java has a mechanism to allow specification of a data type with NO implementation
  - *interfaces*
- Pure Design
  - allow a form of multiple inheritance without the possibility of conflicting implementations

A List Interface

- What if we wanted to specify the operations for a List, but no implementation?
- Allow for multiple, different implementations.
- Provides a way of creating *abstractions.*
  - a central idea of computer science and programming.
  - specify "what" without specifying "how"
  - "Abstraction is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time. "

Interface Syntax

```
public interface List{
  public void add(Object val);
  public int size();
  public Object get(int location);
  public void insert(int location,
         Object val);
  public void addAll(List other);
  public Object remove(int location);
}
```


Interfaces

- All methods in interfaces are public and abstract
  - can leave off those modifiers in method headers
- No constructors
- No instance variables
- can have class constants

```
public static final int DEFAULT_SIDES = 6
```

Implementing Interfaces

- A class inherits (extends) exactly one other class, but …
- A class can *implement* as many interfaces as it likes

```
public class ArrayList implements List
```

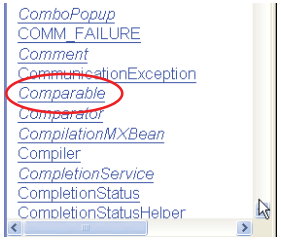
- A class that implements an interface must provide implementations of all method declared in the interface or the class must be abstract
- interfaces can extend other interfaces

Why interfaces?

- Interfaces allow the creation of *abstract data types*
  - "A set of data values and associated operations that are precisely specified independent of any particular implementation. "
  - multiple implementations allowed
- Interfaces allow a class to be specified without worrying about the implementation
  - do design first
  - What will this data type do?
  - Don't worry about implementation until design is done.
  - separation of concerns
- allow a form of multiple inheritance

The Comparable Interface

- The Java Standard Library contains a number of interfaces
  - names are italicized in the class listing
- One of the most important interfaces is the Comparable interface



Comparable Interface version 1.4

```
package java.lang

public interface Comparable
{
     public int compareTo( Object other );
}
```

- compareTo should return an int <0 if the calling object is less than the parameter, 0 if they are equal, and an int >0 if the calling object is greater than the parameter

Implementing Comparable

- Any class that has a *natural ordering* of its objects (that is objects of that type can be sorted based on some internal attribute) should implement the Comparable interface
- Back to the `ClosedShape` example
- Suppose we want to be able to sort `ClosedShape`s and it is to be based on area

Example `compareTo`

- Suppose we have a class to model playing cards
  - Ace of Spades, King of Hearts, Two of Clubs
- each card has a suit and a value, represented by ints
- this version of `compareTo` will compare values first and then break ties with suits

`compareTo` in a `Card` class

```
public class Card implements Comparable
{
      public int compareTo(Object otherObject)
      {     Card other = (Card)otherObject;
            int result = this.myRank - other.myRank;
            if(result == 0)
                  result = this.mySuit - other.mySuit;
            return result
      }
      // other methods not shown
}
```

Assume ints for ranks (2, 3, 4, 5, 6,...) and suits (0 is clubs, 1 is diamonds, 2 is hearts, 3 is spades).

Interfaces and Polymorphism

- Interfaces may be used as the data type for object variables
- Can't simply create objects of that type
- Can refer to any objects that implement the interface or descendants
- Assume `Card` implements `Comparable`

```
Card c = new Card();
Comparable comp1 = new Card();
Comparable comp2 = c;
```

Polymorphism Again! What can this Sort?

```
public static void SelSort(Comparable[] list)
{  Comparable temp;
   int smallest;
   for(int i = 0; i < list.length - 1; i++)
   {  small = i;
      for(int j = i + 1; j < list.length; j++)
      {  if( list[j].compareTo(list[small]) < 0)
            small = j;
      } // end of j loop
      temp = list[i];
      list[i] = list[small];
      list[small] = temp;
   } // end of i loop
}
```


Abstract Classes

Part Class, part Interface

Back to the ClosedShape Example

- One behavior we might want in ClosedShapes is a way to get the area
- problem: How do I get the area of something that is “just a ClosedShape”?

The ClosedShape class

```
public class ClosedShape
{  private double myX;
   private double myY;

   public double getArea()
   {  //Hmmmm?!?!
   }

   //

}
// Other methods not shown
```

Doesn’t seem like we have enough information to get the area if all we know is it is a ClosedShape.

Options

1. Just leave it for the sub classes.
   - Have each sub class define getArea() if they want to.
2. Define getArea() in ClosedShape and simply return 0.
   - Sub classes can override the method with more meaningful behavior.

Leave it to the Sub - Classes

```
// no getArea() in ClosedShape

public void printAreas(ClosedShape[] shapes)
{
     for( ClosedShape s : shapes )
     {    System.out.println( s.getArea() );
     }
}

ClosedShape[] shapes = new ClosedShape[2];
shapes[0] = new Rectangle(1, 2, 3, 4);
shapes[1] = new Circle(1, 2, 3);
printAreas( shapes );
```

Will the above code compile?

How does the compiler determine if a method call is allowed?

Fix by Casting

```
// no getArea() in ClosedShape

public void printAreas(ClosedShape[] shapes)
{
   for( ClosedShape s : shapes )
   {  if( s instanceof Rectangle )
         System.out.println( ((Rectangle)s).getArea() );
      else if( s instanceof Circle )
            System.out.println( ((Circle)s).getArea() );
   }
}

ClosedShape[] shapes = new ClosedShape[2];
shapes[0] = new Rectangle(1, 2, 3, 4);
shapes[1] = new Circle(1, 2, 3);
printAreas( shapes );
```

What happens as we add more sub classes of `ClosedShape`?

What happens if one of the objects is just a `ClosedShape`?

Fix with Dummy Method

```
// getArea() in ClosedShape returns 0

public void printAreas(ClosedShape[] shapes)
{
     for( ClosedShape s : shapes )
     {    System.out.println( s.getArea() );
     }
}

ClosedShape[] shapes = new ClosedShape[2];
shapes[0] = new Rectangle(1, 2, 3, 4);
shapes[1] = new Circle(1, 2, 3);
printAreas( shapes );
```

What happens if sub classes don't override getArea()?

Does that make sense?

A Better Fix

- We know we want to be able to find the area of objects that are instances of `ClosedShape`
- The problem is we don't know how to do that if all we know is it a `ClosedShape`
- Make `getArea` an `abstract` method
- Java keyword

Making getArea Abstract

```
public class ClosedShape
{  private double myX;
   private double myY;

   public abstract double getArea();
   // I know I want it.
   // Just don't know how, yet…

}
// Other methods not shown
```

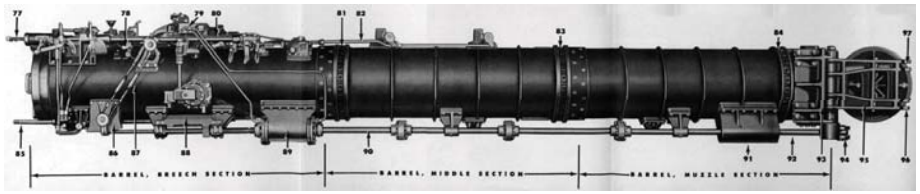
Methods that are declared abstract have no body an undefined behavior.

All methods in an interface are abstract.

Problems with Abstract Methods

Given `getArea()` is now an abstract method what is wrong with the following code?

```
ClosedShape s = new ClosedShape();
System.out.println( s.getArea() );
```



Undefined Behavior = Bad

- Not good to have undefined behaviors
- If a class has 1 or more abstract methods, the class must also be declared abstract.
  - version of `ClosedShape` shown would cause a compile error
- Even if a class has zero abstract methods a programmer can still choose to make it abstract
  - if it models some abstract thing
  - is there anything that is just a "Mammal"?

Abstract Classes

```
public abstract class ClosedShape
{  private double myX;
   private double myY;

   public abstract double getArea();
   // I know I want it.
   // Just don't know how, yet…

}
// Other methods not shown
```

if a class is abstract the compiler will not allow constructors of that class to be called

```
ClosedShape s = new ClosedShape(1,2);
//syntax error
```

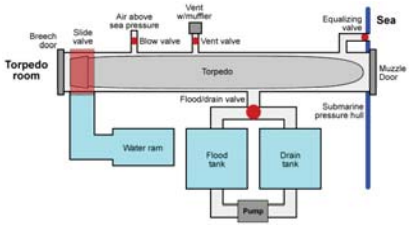

Abstract Classes

- In other words you can't create instances of objects where the lowest or most specific class type is an abstract class
- Prevents having an object with an undefined behavior
- Why would you still want to have constructors in an abstract class?
- Object variables of classes that are abstract types may still be declared

```
ClosedShape s; //okay
```


Sub Classes of Abstract Classes

- Classes that extend an abstract class must provided a working version of any abstract methods from the parent class
  - or they must be declared to be abstract as well
  - could still decide to keep a class abstract regardless of status of abstract methods



Implementing getArea()

```
public class Rectangle extends ClosedShape
{  private double myWidth;
   private double myHeight;

   public double getArea()
   {   return myWidth * myHeight;   }

   // other methods not shown
}

public class Square extends Rectangle
{  public Square()
   { }

   public Square(double side)
   { super(side, side); }

   public Square(double x, double y, double side)
   { super(side, side, x, y); }
}
```

A Circle Class

```
public class Circle extends ClosedShape
{   double dMyRadius;

    public Circle()
    { super(0,0); }

    public Circle(double radius)
    {  super(0,0);
       dMyRadius = radius;
    }

    public Circle(double x, double y, double radius)
    {   super(x,y);
        dMyRadius = radius;
    }

    public double getArea()
    {   return Math.PI * dMyRadius * dMyRadius; }

    public String toString()
    { return super.toString() + " radius: " + dMyRadius; }
}
```


Polymorphism in Action

```
public class UsesShapes
{   public static void go()
    {   ClosedShape[] sList = new ClosedShape[10];
        double a, b, c, d;
        int x;
        for(int i = 0; i < 10; i++)
        {   a = Math.random() * 100;
            b = Math.random() * 100;
            c = Math.random() * 100;
            d = Math.random() * 100;
            x = (int)(Math.random() * 3 );
            if( x == 0 )
                sList[i] = new Rectangle(a,b,c,d);
            else if(x == 1)
                sList[i] = new Square(a, c, d);
            else
                sList[i] = new Circle(a, c, d);
        }
        double total = 0.0;
        for(int i = 0; i < 10; i++)
        {   total += sList[i].getArea();
            System.out.println( sList[i] );
        }
    }
}
```


The Kicker

- We want to expand our pallet of shapes
- Triangle could also be a sub class of ClosedShape.
  - it would *inherit* from ClosedShape

```
public double getArea()
{ return 0.5 * dMyWidth * dMyHeight;}
```

- What changes do we have to make to the code on the previous slide for totaling area so it will now handle Triangles as well?
- Inheritance is can be described as new code using old code.
- **Polymorphism can be described as old code using new code.**

Comparable in ClosedShape

```
public abstract class ClosedShape implements Comparable
{  private double myX;
   private double myY;

   public abstract double getArea();

   public int compareTo(Object other)
   {   int result;
       ClosedShape otherShape = (ClosedShape)other;
       double diff = getArea() - otherShape.getArea();
       if( diff == 0 )
              result = 0;
       else if( diff < 0 )
              result = -1;
       else
              result = 1;
       return result
   }
}
```

About ClosedShapes compareTo

- don't have to return -1, 1.
  - Any int less than 0 or int greater than 0 based on 2 objects
- the `compareTo` method makes use of the `getArea()` method which is abstract in `ClosedShape`
  - how is that possible?

Topic Number 8
Algorithm Analysis

"bit twiddling: 1. (pejorative) An exercise in tuning (see *tune*) in which incredible amounts of time and effort go to produce little noticeable improvement, often with the result that the code becomes incomprehensible."

- The Hackers Dictionary, version 4.4.7

Is This Algorithm Fast?

- Problem: given a problem, how fast does this code solve that problem?
- Could try to measure the time it takes, but that is subject to lots of errors
  - multitasking operating system
  - speed of computer
  - language solution is written in

Attendance Question 1

- "My program finds all the primes between 2 and 1,000,000,000 in 1.37 seconds."
  - how good is this solution?

A. Good

B. Bad

C. It depends

Grading Algorithms

- What we need is some way to grade algorithms and their representation via computer programs for efficiency
  - both time and space efficiency are concerns
  - are examples simply deal with time, not space
- The grades used to characterize the algorithm and code should be independent of platform, language, and compiler
  - We will look at Java examples as opposed to pseudocode algorithms

Big O

- The most common method and notation for discussing the execution time of algorithms is "Big O"
- Big O is the *asymptotic execution time* of the algorithm
- Big O is an upper bounds
- It is a mathematical tool
- Hide a lot of unimportant details by assigning a simple grade (function) to algorithms

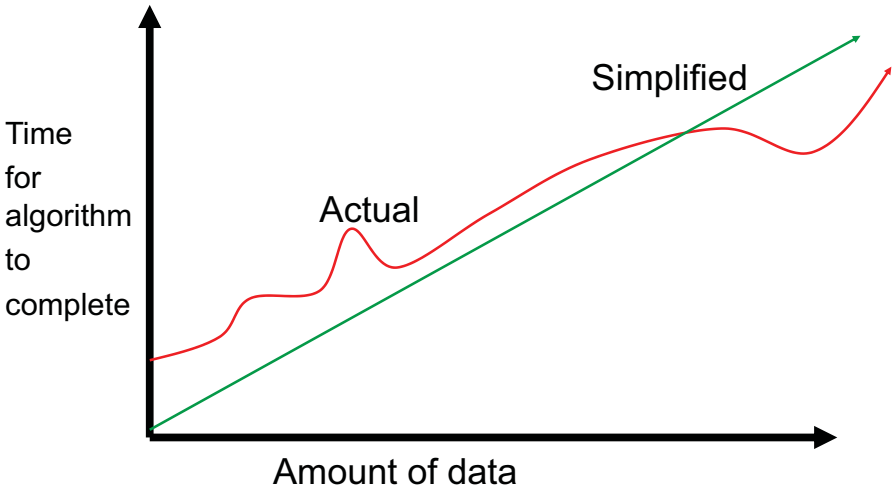
Typical Big O Functions – "Grades"

Function	Common Name
N!	factorial
2^N	Exponential
N^d, d > 3	Polynomial
N^3	Cubic
N^2	Quadratic
$N\sqrt{N}$	N Square root N
N log N	N log N
N	Linear
$\sqrt{N}$	Root - n
log N	Logarithmic
1	Constant

Big O Functions

- N is the size of the data set.
- The functions do not include less dominant terms and do not include any coefficients.
- $4N^2 + 10N – 100$ is not a valid F(N).
  - It would simply be $O(N^2)$
- It is possible to have two independent variables in the Big O function.
  - example O(M + log N)
  - M and N are sizes of two different, but interacting data sets

Actual vs. Big O



Formal Definition of Big O

- T(N) is O(F(N)) if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N \geq N_0$
  - N is the size of the data set the algorithm works on
  - T(N) is a function that characterizes the *actual* running time of the algorithm
  - F(N) is a function that characterizes an upper bounds on T(N). It is a limit on the running time of the algorithm. (The typical Big functions table)
  - c and N_0 are constants

What it Means

- T(N) is the actual growth rate of the algorithm
  - can be equated to the number of executable statements in a program or chunk of code
- F(N) is the function that bounds the growth rate
  - may be upper or lower bound
- T(N) may not necessarily equal F(N)
  - constants and lesser terms ignored because it is a *bounding function*

Yuck

- How do you apply the definition?
- Hard to measure time without running programs and that is full of inaccuracies
- Amount of time to complete should be directly proportional to the number of statements executed for a given amount of data
- Count up statements in a program or method or algorithm as a function of the amount of data
  - This is one technique
- Traditionally the amount of data is signified by the variable N

Counting Statements in Code

- So what constitutes a statement?
- Can't I rewrite code and get a different answer, that is a different number of statements?
- Yes, but the beauty of Big O is, in the end you get the same answer
  - remember, it is a simplification

Assumptions in For Counting Statements

- Once found accessing the value of a primitive is constant time. This is one statement:

```
x = y; //one statement
```

- mathematical operations and comparisons in boolean expressions are all constant time.

```
x = y * 5 + z % 3; // one statement
```

- if statement constant time if test and maximum time for each alternative are constants

```
if( iMySuit == DIAMONDS || iMySuit == HEARTS )
  return RED;
else
  return BLACK;
// 2 statements (boolean expression + 1 return)
```


Counting Statements in Loops Attendenance Question 2

- Counting statements in loops often requires a bit of informal mathematical *induction*
- What is output by the following code?

```
int total = 0;
for(int i = 0; i < 2; i++)
    total += 5;
System.out.println( total );
```

A. 2 B. 5 C. 10 D. 15 E. 20

Attendances Question 3

- What is output by the following code?

```
int total = 0;
// assume limit is an int >= 0
for(int i = 0; i < limit; i++)
    total += 5;
System.out.println( total );
```

A. 0

B. limit

C. limit * 5

D. limit * limit

E. $limit^5$

Counting Statements in Nested Loops Attendance Question 4

- What is output by the following code?

```
int total = 0;
for(int i = 0; i < 2; i++)
    for(int j = 0; j < 2; j++)
        total += 5;
System.out.println( total );
```

A. 0
B. 10
C. 20
D. 30
E. 40

Attendance Question 5

- What is output by the following code?

```
int total = 0;
// assume limit is an int >= 0
for(int i = 0; i < limit; i++)
    for(int j = 0; j < limit; j++)
        total += 5;
System.out.println( total );
```

A. 5

B. limit * limit

C. limit * limit * 5

D. 0

E. $limit^5$

Loops That Work on a Data Set

- The number of executions of the loop depends on the length of the array, values.

```
public int total(int[] values)
{  int result = 0;
   for(int i = 0; i < values.length; i++)
      result += values[i];
   return result;
}
```

- How many many statements are executed by the above method
- N = values.length. What is T(N)? F(N)?

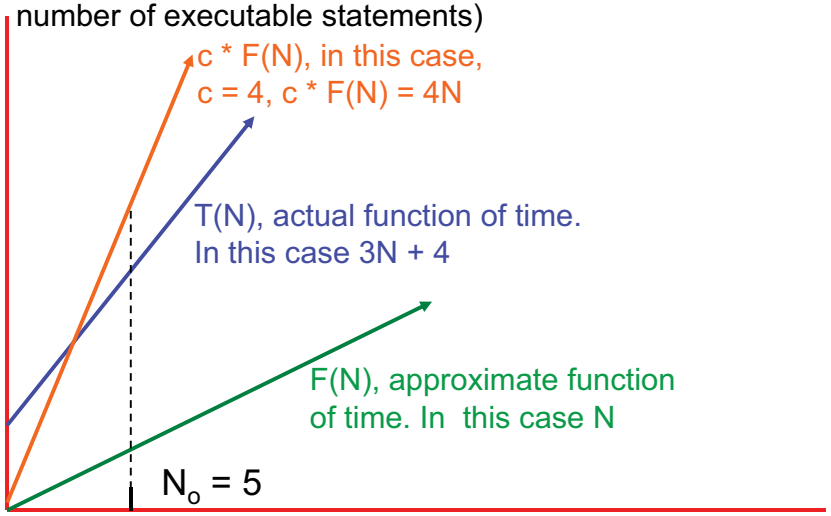
Counting Up Statements

- `int result = 0;` 1 time
- `int i = 0;` 1 time
- `i < values.length`; N + 1 times
- `i++` N times
- `result += values[i];` N times
- `return total;` 1 time
- T(N) = 3N + 4
- F(N) = N
- Big O = O(N)

Showing O(N) is Correct

- Recall the formal definition of Big O
  - T(N) is O(F(N)) if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N > N_0$
- In our case given T(N) = 3N + 4, prove the method is O(N).
  - F(N) is N
- We need to choose constants c and N_0
- how about c = 4, N_0 = 5 ?

vertical axis: time for algorithm to complete. (approximate with number of executable statements)



horizontal axis: N, number of elements in data set

Attendance Question 6

- Which of the following is true?

A. Method `total` is O(N)
B. Method `total` is $O(N^2)$
C. Method `total` is O(N!)
D. Method `total` is $O(N^N)$
E. All of the above are true

Just Count Loops, Right?

```
// assume mat is a 2d array of booleans
// assume mat is square with N rows,
// and N columns

int numThings = 0;
for(int r = row - 1; r <= row + 1; r++)
     for(int c = col - 1; c <= col + 1; c++)
          if( mat[r][c] )
               numThings++;
```

What is the order of the above code?

A. O(1) B. O(N) C. $O(N^2)$ D. $O(N^3)$ E. $O(N^{1/2})$

It is Not Just Counting Loops

```
// Second example from previous slide could be
// rewritten as follows:
int numThings = 0;
if( mat[r-1][c-1] ) numThings++;
if( mat[r-1][c] ) numThings++;
if( mat[r-1][c+1] ) numThings++;
if( mat[r][c-1] ) numThings++;
if( mat[r][c] ) numThings++;
if( mat[r][c+1] ) numThings++;
if( mat[r+1][c-1] ) numThings++;
if( mat[r+1][c] ) numThings++;
if( mat[r+1][c+1] ) numThings++;
```

Sidetrack, the logarithm

- Thanks to Dr. Math
- $3^2 = 9$
- likewise $\log_3 9 = 2$
  - "The log to the base 3 of 9 is 2."
- The way to think about log is:
  - "the log to the base x of y is the number you can raise x to to get y."
  - Say to yourself "The log is the exponent." (and say it over and over until you believe it.)
  - In CS we work with base 2 logs, a lot
- $\log_2 32 = ?$ $\log_2 8 = ?$ $\log_2 1024 = ?$ $\log_{10} 1000 = ?$

When Do Logarithms Occur

- Algorithms have a logarithmic term when they use a divide and conquer technique
- the data set keeps getting divided by 2

```
public int foo(int n)
{    // pre n > 0
     int total = 0;
     while( n > 0 )
     {    n = n / 2;
          total++;
     }
     return total;
}
```

- What is the order of the above code?

A. O(1) B. O(logN) C. O(N)
D. O(Nlog N) E. $O(N^2)$

Dealing with other methods

- What do I do about method calls?

```
double sum = 0.0;
for(int i = 0; i < n; i++)
    sum += Math.sqrt(i);
```

- Long way
  - go to that method or constructor and count statements
- Short way
  - substitute the simplified Big O function for that method.
  - if Math.sqrt is constant time, O(1), simply count `sum += Math.sqrt(i);` as one statement.

Dealing With Other Methods

```
public int foo(int[] list){
    int total = 0;
    for(int i = 0; i < list.length; i++){
        total += countDups(list[i], list);
    }
    return total;
}
// method countDups is O(N) where N is the
// length of the array it is passed
```

What is the Big O of `foo`?

A. O(1) B. O(N) C. O(NlogN)
D. $O(N^2)$ E. O(N!)

Quantifiers on Big O

- It is often useful to discuss different cases for an algorithm
- Best Case: what is the best we can hope for?
  - least interesting
- Average Case (a.k.a. expected running time): what usually happens with the algorithm?
- Worst Case: what is the worst we can expect of the algorithm?
  - very interesting to compare this to the average case

Best, Average, Worst Case

- To Determine the best, average, and worst case Big O we must make assumptions about the data set
- Best case -> what are the properties of the data set that will lead to the fewest number of executable statements (steps in the algorithm)
- Worst case -> what are the properties of the data set that will lead to the largest number of executable statements
- Average case -> Usually this means assuming the data is randomly distributed
  - or if I ran the algorithm a large number of times with different sets of data what would the average amount of work be for those runs?

Another Example

```
public double minimum(double[] values)
{   int n = values.length;
    double minValue = values[0];
    for(int i = 1; i < n; i++)
       if(values[i] < minValue)
          minValue = values[i];
    return minValue;
}
```

- T(N)? F(N)? Big O? Best case? Worst Case? Average Case?
- If no other information, assume asking average case

Independent Loops

```
// from the Matrix class
public void scale(int factor){
     for(int r = 0; r < numRows(); r++)
          for(int c = 0; c < numCols(); c++)
               iCells[r][c] *= factor;
}
```

Assume an `numRows()` = N and `numCols()` = N.

In other words, a square Matrix.

What is the T(N)? What is the Big O?

A. O(1) B. O(N) C. O(NlogN)

D. $O(N^2)$ E. O(N!)

Significant Improvement – Algorithm with Smaller Big O function

- Problem: Given an array of ints replace any element equal to 0 with the maximum value to the right of that element.

Given:

```
[0, 9, 0, 8, 0, 0, 7, 1, -1, 0, 1, 0]
```

Becomes:

```
[9, 9, 8, 8, 7, 7, 7, 1, -1, 1, 1, 0]
```


Replace Zeros – Typical Solution

```
public void replace0s(int[] data){
  int max;
  for(int i = 0; i < data.length -1; i++){
    if( data[i] == 0 ){
      max = 0;
      for(int j = i+1; j<data.length;j++)
        max = Math.max(max, data[j]);
      data[i] = max;
    }
  }
}
```

Assume most values are zeros.

Example of a **dependent loops**.

Replace Zeros – Alternate Solution

```
public void replace0s(int[] data){
  int max =
     Math.max(0, data[data.length - 1]);
  int start = data.length - 2;
  for(int i = start; i >= 0; i--){
    if( data[i] == 0 )
          data[i] = max;
     else
          max = Math.max(max, data[i]);
  }
}
```

Big O of this approach?

A.O(1) B. O(N) C. O(NlogN)

D. $O(N^2)$ E. O(N!)

A Caveat

- What is the Big O of this statement in Java?

```
int[] list = new int[n];
```

A. O(1) B. O(N) C. O(NlogN)

D. $O(N^2)$ E. O(N!)

- Why?

Summing Executable Statements

- If an algorithms execution time is $N^2 + N$ the it is said to have $O(N^2)$ execution time not $O(N^2 + N)$
- When adding algorithmic complexities the larger value dominates
- formally a function f(N) dominates a function g(N) if there exists a constant value n_0 such that for all values $N > N_0$ it is the case that $g(N) < f(N)$

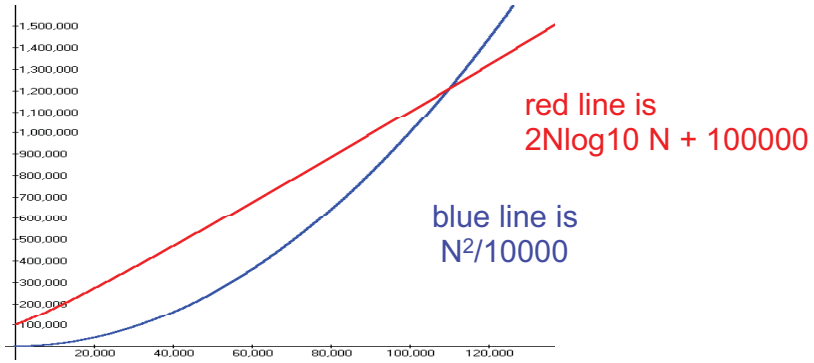
Example of Dominance

- Look at an extreme example. Assume the actual number as a function of the amount of data is:

  $N^2/10000 + 2N\log_{10} N + 100000$

- Is it plausible to say the N^2 term dominates even though it is divided by 10000 and that the algorithm is $O(N^2)$?
- What if we separate the equation into $(N^2/10000)$ and $(2N \log_{10} N + 100000)$ and graph the results.

Summing Execution Times



- For large values of N the N^2 term dominates so the algorithm is $O(N^2)$
- When does it make sense to use a computer?

Comparing Grades

- Assume we have a problem
- Algorithm A solves the problem correctly and is $O(N^2)$
- Algorithm B solves the same problem correctly and is $O(N \log_2 N)$
- Which algorithm is faster?
- One of the assumptions of Big O is that the data set is large.
- The "grades" should be accurate tools if this is true

Running Times

- Assume N = 100,000 and processor speed is 1,000,000,000 operations per second

Function	Running Time
2^N	3.2×10^{30086} years
N^4	3171 years
N^3	11.6 days
N^2	10 seconds
$N\sqrt{N}$	0.032 seconds
N log N	0.0017 seconds
N	0.0001 seconds
$\sqrt{N}$	3.2×10^{-7} seconds
log N	1.2×10^{-8} seconds

Theory to Practice OR Dykstra says: "Pictures are for the Weak."

	1000	2000	4000	8000	16000	32000	64000	128K
O(N)	$2.2x10^{-5}$	$2.7x10^{-5}$	$5.4x10^{-5}$	$4.2x10^{-5}$	$6.8x10^{-5}$	$1.2x10^{-4}$	$2.3x10^{-4}$	$5.1x10^{-4}$
O(NlogN)	$8.5x10^{-5}$	$1.9x10^{-4}$	$3.7x10^{-4}$	$4.7x10^{-4}$	$1.0x10^{-3}$	$2.1x10^{-3}$	$4.6x10^{-3}$	$1.2x10^{-2}$
$O(N^{3/2})$	$3.5x10^{-5}$	$6.9x10^{-4}$	$1.7x10^{-3}$	$5.0x10^{-3}$	$1.4x10^{-2}$	$3.8x10^{-2}$	0.11	0.30
$O(N^2)$ ind.	$3.4x10^{-3}$	$1.4x10^{-3}$	$4.4x10^{-3}$	0.22	0.86	3.45	13.79	(55)
$O(N^2)$ dep.	$1.8x10^{-3}$	$7.1x10^{-3}$	$2.7x10^{-2}$	0.11	0.43	1.73	6.90	(27.6)
$O(N^3)$	3.40	27.26	(218)	(1745) 29 min.	(13,957) 233 min	(112k) 31 hrs	(896k) 10 days	(7.2m) 80 days

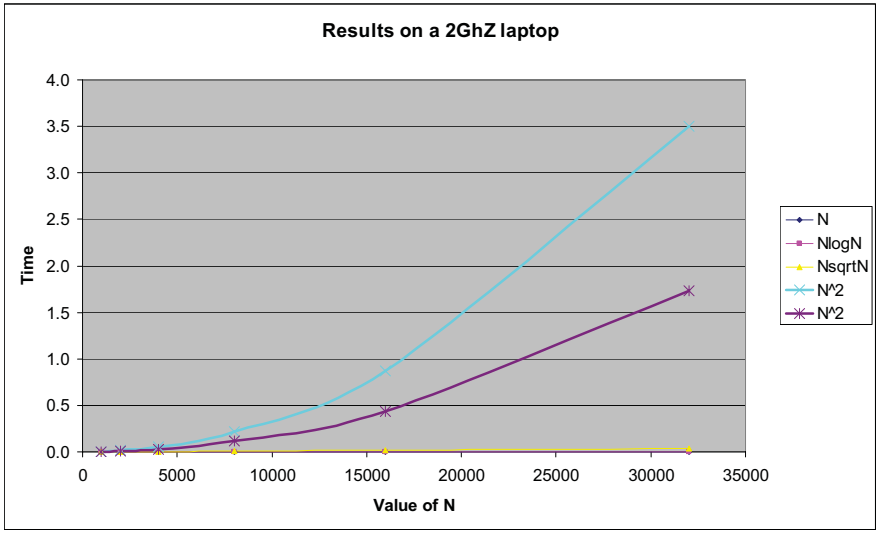
Times in Seconds. Red indicates predicated value.

Change between Data Points

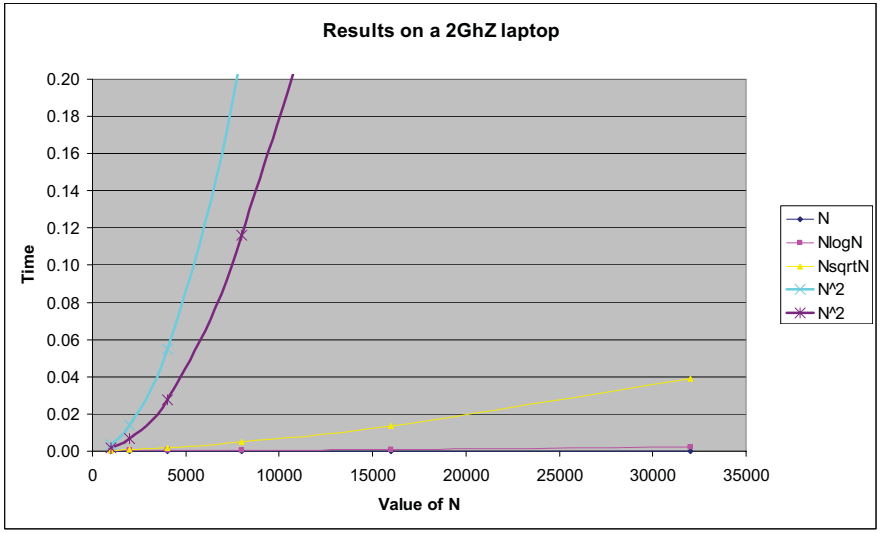
	1000	2000	4000	8000	16000	32000	64000	128K	256k	512k
O(N)	-	1.21	2.02	0.78	1.62	1.76	1.89	2.24	2.11	1.62
O(NlogN)	-	2.18	1.99	1.27	2.13	2.15	2.15	2.71	1.64	2.40
$O(N^{3/2})$	-	1.98	2.48	2.87	2.79	2.76	2.85	2.79	2.82	2.81
$O(N^2)$ ind	-	4.06	3.98	3.94	3.99	4.00	3.99	-	-	-
$O(N^2)$ dep	-	4.00	3.82	3.97	4.00	4.01	3.98	-	-	-
$O(N^3)$	-	8.03	-	-	-	-	-	-	-	-

Value obtained by $Time_x$ / $Time_{x-1}$

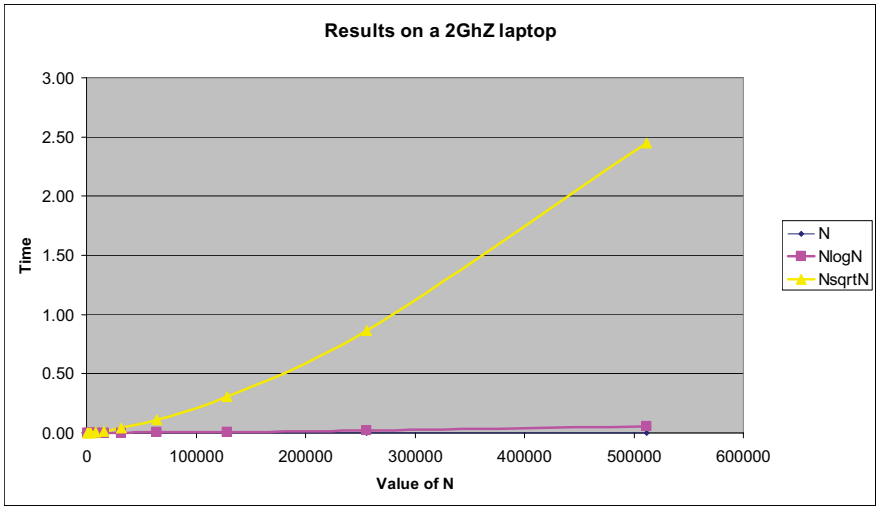
Okay, Pictures



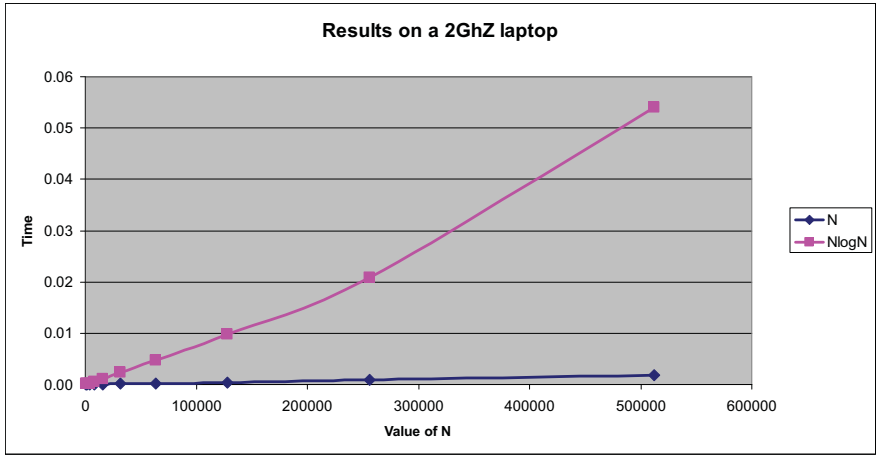
Put a Cap on Time



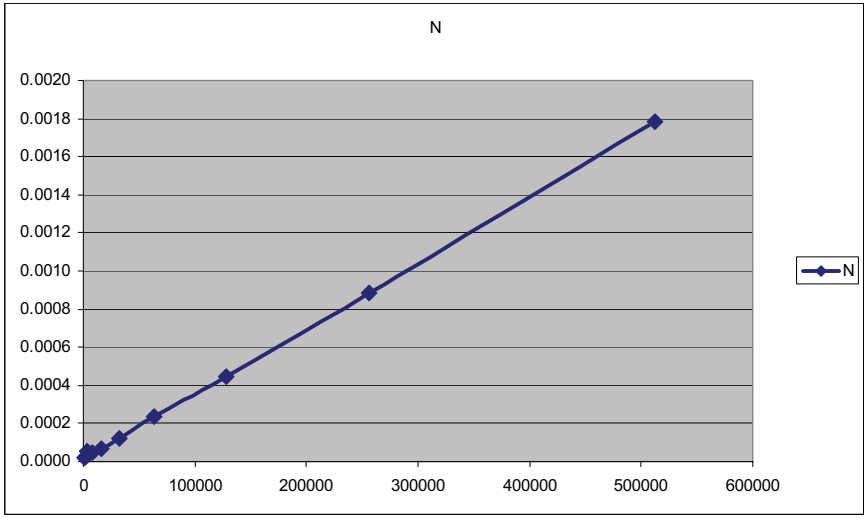
No O(N^2) Data



Just O(N) and O(NlogN)



Just O(N)



Reasoning about algorithms

- We have an O(N) algorithm,
  - For 5,000 elements takes 3.2 seconds
  - For 10,000 elements takes 6.4 seconds
  - For 15,000 elements takes ….?
  - For 20,000 elements takes ….?
- We have an $O(N^2)$ algorithm
  - For 5,000 elements takes 2.4 seconds
  - For 10,000 elements takes 9.6 seconds
  - For 15,000 elements takes …?
  - For 20,000 elements takes …?

A Useful Proportion

- Since F(N) is characterizes the running time of an algorithm the following proportion should hold true:

$F(N_0) / F(N_1) \sim= time_0 / time_1$

- An algorithm that is $O(N^2)$ takes 3 seconds to run given 10,000 pieces of data.
  - How long do you expect it to take when there are 30,000 pieces of data?
  - common mistake
  - logarithms?

10^9 instructions/sec, runtimes

N	*O(log N)*	*O(N)*	*O(N log N)*	*O(N^2)*
10	0.000000003	0.00000001	0.000000033	0.0000001
100	0.000000007	0.00000010	0.000000664	0.0001000
1,000	0.000000010	0.00000100	0.000010000	0.001
10,000	0.000000013	0.00001000	0.000132900	0.1 min
100,000	0.000000017	0.00010000	0.001661000	10 seconds
1,000,000	0.000000020	0.001	0.0199	16.7 minutes
1,000,000,000	0.000000030	1.0 second	30 seconds	31.7 years

Why Use Big O?

- As we build data structures Big O is the tool we will use to decide under what conditions one data structure is better than another
- Think about performance when there is a lot of data.
  - "It worked so well with small data sets..."
  - Joel Spolsky, Schlemiel the painter's Algorithm
- Lots of trade offs
  - some data structures good for certain types of problems, bad for other types
  - often able to trade SPACE for TIME.
  - Faster solution that uses more space
  - Slower solution that uses less space

Big O Space

- Less frequent in early analysis, but just as important are the space requirements.
- Big O could be used to specify how much space is needed for a particular algorithm

Formal Definition of Big O (repeated)

- T(N) is O(F(N)) if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N \geq N_0$
  - N is the size of the data set the algorithm works on
  - T(N) is a function that characterizes the *actual* running time of the algorithm
  - F(N) is a function that characterizes an upper bounds on T(N). It is a limit on the running time of the algorithm
  - c and N_0 are constants

More on the Formal Definition

- There is a point N_0 such that for all values of N that are past this point, T(N) is bounded by some multiple of F(N)
- Thus if T(N) of the algorithm is O(N^2) then, ignoring constants, at some point we can *bound* the running time by a quadratic function.
- given a *linear* algorithm it is *technically correct* to say the running time is O(N ^ 2). O(N) is a more precise answer as to the Big O of the linear algorithm
  - thus the caveat “pick the most restrictive function” in Big O type questions.

What it All Means

- T(N) is the actual growth rate of the algorithm
  - can be equated to the number of executable statements in a program or chunk of code
- F(N) is the function that bounds the growth rate
  - may be upper or lower bound
- T(N) may not necessarily equal F(N)
  - constants and lesser terms ignored because it is a *bounding function*

Other Algorithmic Analysis Tools

- *Big Omega* T(N) is Ω(F(N)) if there are positive constants c and N_0 such that T(N) $\geq$ cF(N)) when N $\geq$ N_0
  - Big O is similar to less than or equal, an upper bounds
  - Big Omega is similar to greater than or equal, a lower bound
- *Big Theta* T(N) is θ(F(N)) if and only if T(N) is O(F(N))and T(N) is Ω(F(N)).
  - Big Theta is similar to equals

Relative Rates of Growth

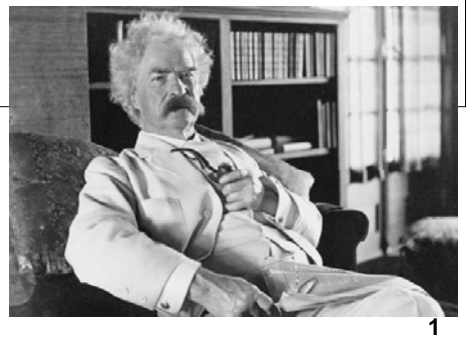
Analysis Type	Mathematical Expression	Relative Rates of Growth
Big O	T(N) = O(F(N))	T(N) $\leq$ F(N)
Big Ω	T(N) = Ω(F(N))	T(N) $\geq$ F(N)
Big θ	T(N) = θ(F(N))	T(N) = F(N)

"In spite of the additional precision offered by Big Theta, Big O is more commonly used, except by researchers in the algorithms analysis field" - Mark Weiss

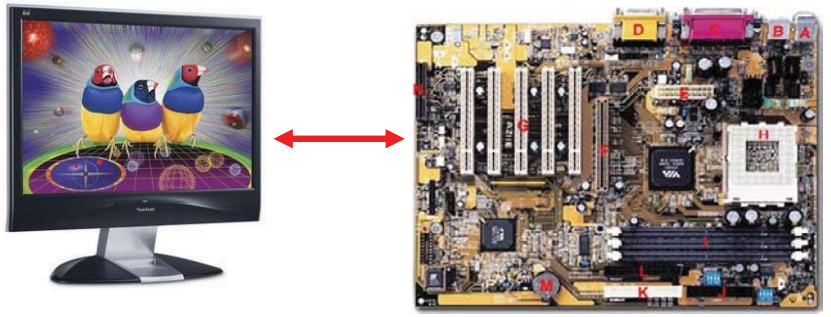
Topic 9
Introduction to Recursion

"To a man with a hammer, everything looks like a nail"

-Mark Twain



Underneath the Hood.



The Program Stack

- When you invoke a method in your code what happens when that method is completed?

```
FooObject f = new FooObject();
int x = 3;
f.someFooMethod(x);
f.someBarMethod(x);
```

- How does that happen?
- What makes it possible?

Methods for Illustration

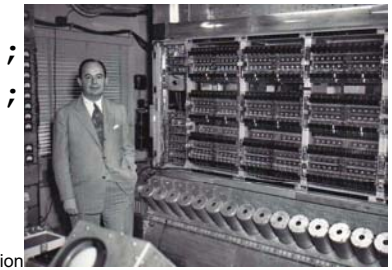
```
200 public void someFooMethod(int z)
201{     int x = 2 * z;
202      System.out.println(x);
   }


300 public void someBarMethod(int y)
301 {    int x = 3 * y;
302      someFooMethod(x);
303      System.out.println(x);
    }
```

The Program Stack

- When your program is executed on a processor the commands are converted into another set of instructions and assigned memory locations.
  - normally a great deal of expansion takes place

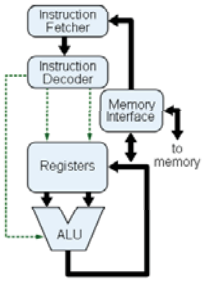
```
101 FooObject f = new FooObject();
102 int x = 3;
103 f.someFooMethod(x);
104 f.someBarMethod(x);
```

- Von Neumann Architecture



Basic CPU Operations

- A CPU works via a fetch command / execute command loop and a program counter
- Instructions stored in memory (Just like data!)




```
101 FooObject f = new FooObject();
102 int x = 3;
103 f.someFooMethod(x);
104 f.someBarMethod(x);
```

- What if `someFooMethod` is stored at memory location 200?

More on the Program Stack

```
101 FooObject f = new FooObject();
102 int x = 3;
103 f.someFooMethod(x);
104 f.someBarMethod(x);
```

- Line 103 is really saying *go to line 200 with f as the implicit parameter and x as the explicit parameter*
- When someFooMethod is done what happens?

A. Program ends B. goes to line 103

C. Goes back to whatever method called it

Activation Records and the Program Stack

- When a method is invoked all the relevant information about the current method (variables, values of variables, next line of code to be executed) is placed in an *activation record*
- The activation record is *pushed* onto the *program stack*
- A *stack* is a data structure with a single access point, the *top.*

The Program Stack

- Data may either be added (*pushed)* or removed *(popped)* from a stack but it is always from the top.
  - A stack of dishes
  - which dish do we have easy access to?

Using Recursion

A Problem

- Write a method that determines how much space is take up by the files in a directory
- A directory can contain files and directories
- How many directories does our code have to examine?
- How would you add up the space taken up by the files in a single directory
  - Hint: don't worry about any sub directories at first
- `Directory` and `File` classes
- in the `Directory` class:

```
public File[] getFiles()
public Directory[] getSubdirectories()
```

- in the `File` class

```
public int getSize()
```


Attendance Question 2

- How many levels of directories have to be visited?

A. 0

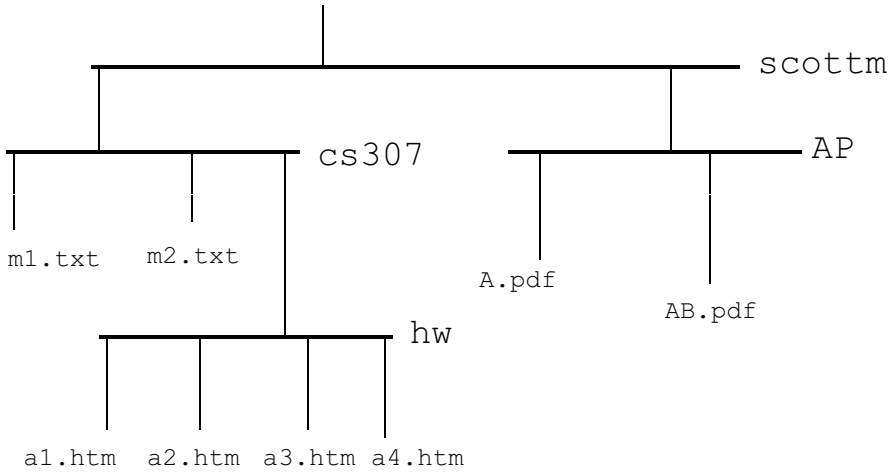
B. Unknown

C. Infinite

D. 1

E. 8

Sample Directory Structure



Code for `getDirectorySpace()`

```
public int getDirectorySpace(Directory d)
{ int total = 0;
  File[] fileList = d.getFiles();
  for(int i = 0; i < fileList.length; i++)
     total += fileList[i].getSize();
  Directory[] dirList = d.getSubdirectories();
  for(int i = 0; i < dirList.length; i++)
     total += getDirectorySpace(dirList[i]);
  return total;
}
```


Attendance Question 3

- Is it possible to write a non recursive method to do this?

A. Yes

B. No

Iterative `getDirectorySpace()`

```
public int getDirectorySpace(Directory d)
{  ArrayList dirs = new ArrayList();
   File[] fileList;
   Directory[] dirList;
   dirs.add(d);
   Directory temp;
   int total = 0;
   while( ! dirs.isEmpty() )
   {    temp = (Directory)dirs.remove(0);
        fileList = temp.getFiles();
        for(int i = 0; i < fileList.length; i++)
              total += fileList[i].getSize();
        dirList = temp.getSubdirectories();
        for(int i =0; i < dirList.length; i++)
              dirs.add( dirList[i] );
   }
   return total;
}
```

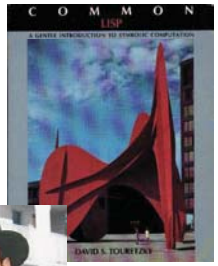
Simple Recursion Examples

Wisdom for Writing Recursive Methods

The 3 plus 1 rules of Recursion

1. Know when to stop
2. Decide how to take one step
3. Break the journey down into that step and a smaller journey
4. Have faith

From *Common Lisp: A Gentle Introduction to Symbolic Computation* by David Touretzky



Writing Recursive Methods

- Rules of Recursion
  1. Base Case: Always have at least one case that can be solved without using recursion
  2. Make Progress: Any recursive call must progress toward a base case.
  3. "You gotta believe." Always assume that the recursive call works. (Of course you will have to design it and test it to see if it works or prove that it always works.)

  A recursive solution solves a small part of the problem and leaves the rest of the problem in the same form as the original

N!

- the classic first recursion problem / example
- N!

  5! = 5 * 4 * 3 * 2 * 1 = 120

```
int res = 1;
for(int i = 2; i <= n; i++)
      res *= i;
```


Factorial Recursively

- Mathematical Definition of Factorial

```
0! = 1
N! = N * (N - 1)!
```

The definition is recursive.

```
// pre n >= 0
public int fact(int n)
{ if(n == 0)
      return 1;
  else
      return n * fact(n-1);
}
```


Big O and Recursion

- Determining the Big O of recursive methods can be tricky.
- A *recurrence relation* exits if the function is defined recursively.
- The T(N), actual running time, for N! is recursive
- $T(N)_{fact} = T(N-1)_{fact} + O(1)$
- This turns out to be O(N)
  - There are N steps involved

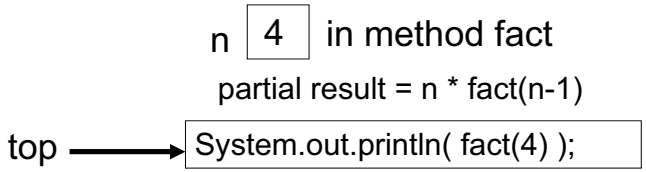
Common Recurrence Relations

- T(N) = T(N/2) + O(1) -> O(logN)
  - binary search
- T(N) = T(N-1) + O(1) -> O(N)
  - sequential search, factorial
- T(N) = T(N/2) + T(N/2) + O(1) -> O(N),
  - tree traversal
- T(N) = T(N-1) + O(N) -> O(N^2)
  - selection sort
- T(N) = T(N/2) + T(N/2) + O(N) -> O(NlogN)
  - merge sort
- T(N) = T(N-1) + T(N-1) + O(1) -> O(2^N)
  - Fibonacci

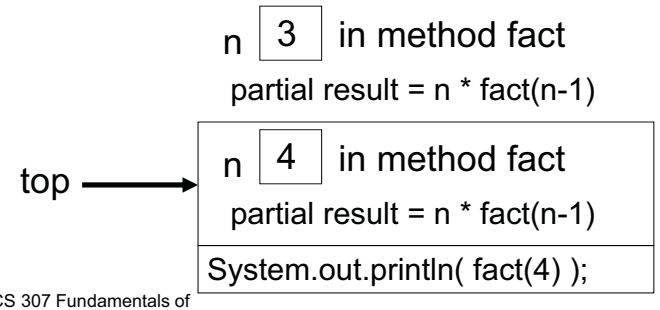
Tracing Fact With the Program Stack

System.out.println(fact(4));

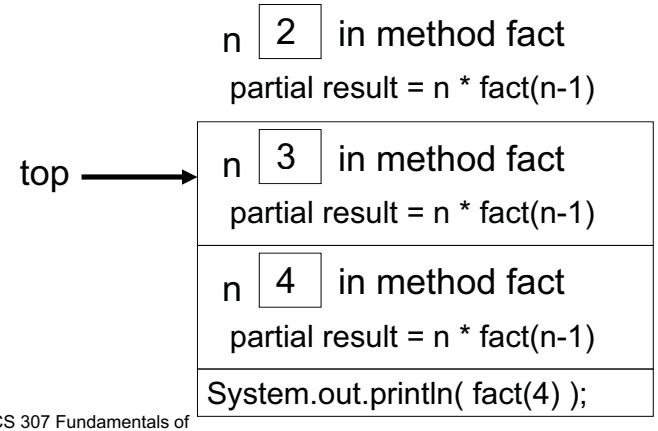
Calling fact with 4



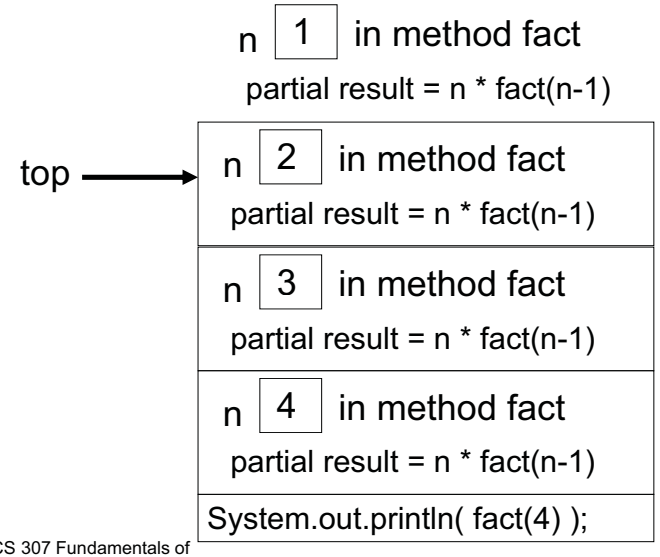
Calling fact with 3



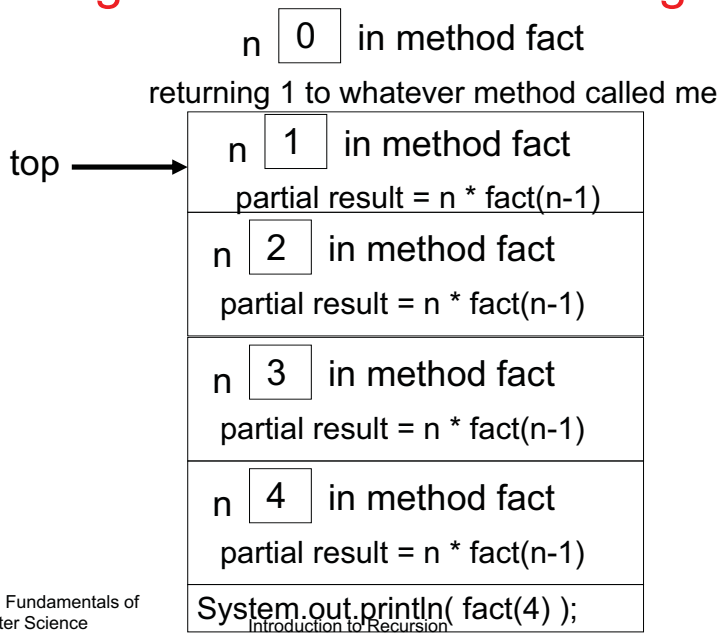
Calling fact with 2



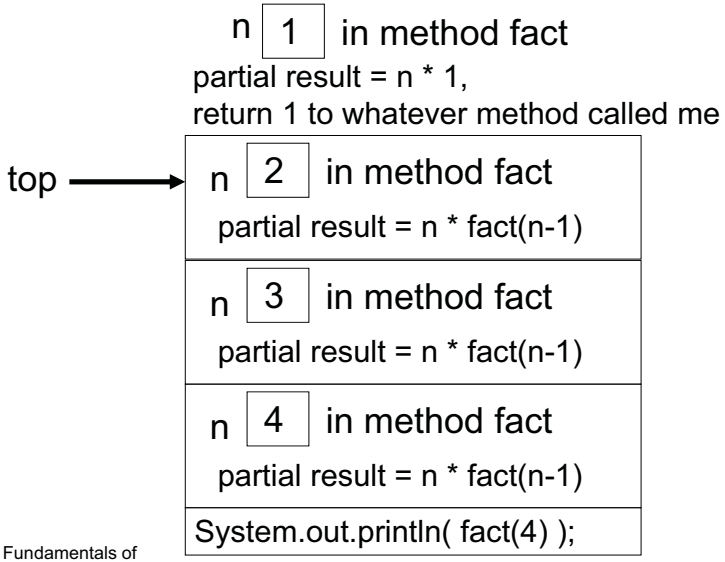
Calling fact with 1



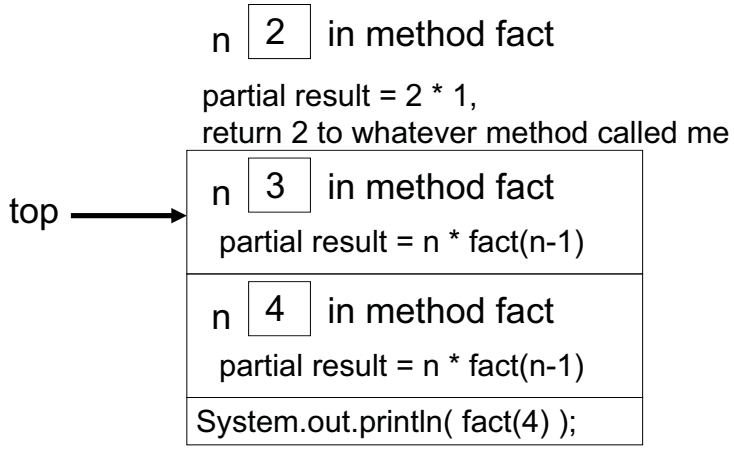
Calling fact with 0 and returning 1



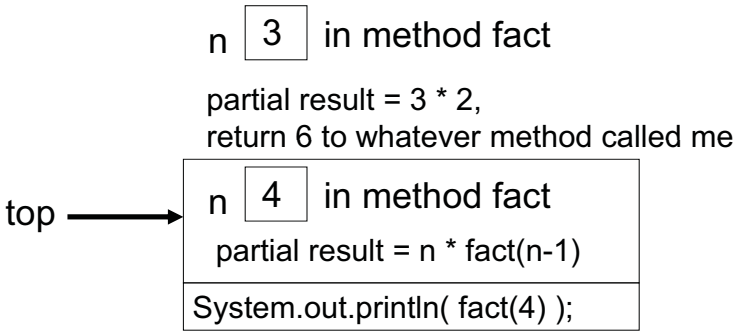
Returning 1 from fact(1)



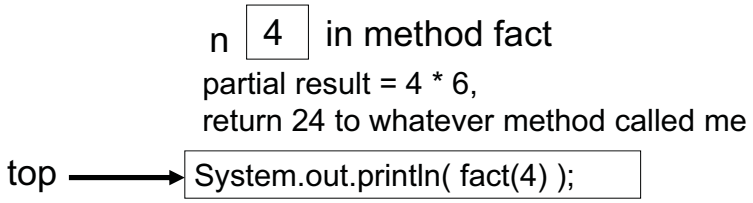
Returning 2 from fact(2)



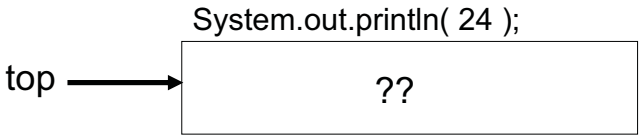
Returning 6 from fact(3)



Returning 24 from fact(4)



Calling System.out.println



Evaluating Recursive Methods

Evaluating Recursive Methods

- you must be able to evaluate recursive methods

```
public static int mystery (int n){
    if( n == 0 )
        return 1;
    else
        return 3 * mystery(n-1);
}
// what is returned by mystery(5)
```


Evaluating Recursive Methods

- Draw the program stack!

m(5) = 3 * m(4)
m(4) = 3 * m(3)
m(3) = 3 * m(2)
m(2) = 3 * m(1)
m(1) = 3 * m(0)
m(0) = 1
-> 3^5 = 243

- with practice you can see the result

Attendance Question 4

- What is returned by `mystery(-3)` ?

A. 0
B. 1
C. Infinite loop
D. Syntax error
E. Runtime error due to stack overflow

Evaluating Recursive Methods

- What about multiple recursive calls?

```
public static int bar(int n){
    if( n <= 0 )
        return 2;
    else
        return 3 + bar(n-1) + bar(n-2);
}
```

- Draw the program stack and REMEMBER your work

Evaluating Recursive Methods

- What is returned by `bar(5)`?

b(5) = 3 + b(4) + b(3)
b(4) = 3 + b(3) + b(2)
b(3) = 3 + b(2) + b(1)
b(2) = 3 + b(1) + b(0)
b(1) = 3 + b(0) + b(-1)
b(0) = 2
b(-1) = 2

Evaluating Recursive Methods

- What is returned by `bar(5)`?

b(5) = 3 + b(4) + b(3)
b(4) = 3 + b(3) + b(2)
b(3) = 3 + b(2) + b(1)
b(2) = 3 + b(1) + b(0) //substitute in results
b(1) = 3 + 2 + 2 = 7
b(0) = 2
b(-1) = 2

Evaluating Recursive Methods

- What is returned by `bar(5)`?

b(5) = 3 + b(4) + b(3)
b(4) = 3 + b(3) + b(2)
b(3) = 3 + b(2) + b(1)
b(2) = 3 + 7 + 2 =12
b(1) = 7
b(0) = 2
b(-1) = 2

Evaluating Recursive Methods

- What is returned by `bar(5)`?

b(5) = 3 + b(4) + b(3)
b(4) = 3 + b(3) + b(2)
b(3) = 3 + 12 + 7 = 22
b(2) = 12
b(1) = 7
b(0) = 2
b(-1) = 2

Evaluating Recursive Methods

- What is returned by `bar(5)`?

b(5) = 3 + b(4) + b(3)

b(4) = 3 + 22 + 12 = 37

b(3) = 22

b(2) = 12

b(1) = 7

b(0) = 2

b(-1) = 2

Evaluating Recursive Methods

- What is returned by `bar(5)`?

b(5) = 3 + 37 + 22 = 62

b(4) = 37

b(3) = 22

b(2) = 12

b(1) = 7

b(0) = 2

b(-1) = 2

Unplugged Activity

- Double the number of pieces of candy in a bowl.
- Only commands we know are:
  - take one candy out of bowl and put into infinite supply
  - take one candy from infinite supply and place in bowl
  - do nothing
  - double the number of pieces of candy in the bowl
- Thanks Stuart Reges

Recursion Practice

- Write a method raiseToPower(int base, int power)
- //pre: power >= 0
- Tail recursion refers to a method where the recursive call is the last thing in the method

Finding the Maximum in an Array

- public int max(int[] values){
- Helper method or create smaller arrays each time

Attendance Question 5

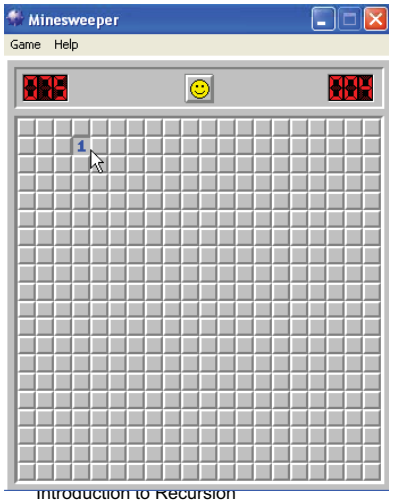
- When writing recursive methods what should be done first?

A. Determine recursive case
B. Determine recursive step
C. Make recursive call
D. Determine base case(s)
E. Determine Big O

Your Meta Cognitive State

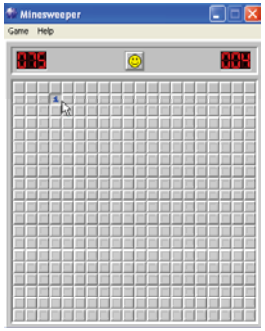
- Remember we are learning to use a tool.
- It is not a good tool for ***all*** problems.
  - In fact we will implement several algorithms and methods where an iterative (looping without recursion) solution would work just fine
- After learning the mechanics and basics of recursion the real skill is knowing what problems or class of problems to apply it to

A Harder(??) Problem



Mine Sweeper

- Game made popular due to its inclusion with Windows (from 3.1 on)
- What happens when you click on a cell that has 0 (zero) mines bordering it?



Result of clicking marked cell.

The update method

- Initially called with the x and y coordinates of a cell with a 0 inside it meaning the cell does not have any bombs bordering it.
- Must reveal all cells neighboring this one and if any of them are 0s do the same thing

-1 indicates a mine in that cell

2	-1	2	0	0	0
2	-1	3	2	2	1
1	1	3	-1	-1	1
0	0	2	-1	3	1
0	0	1	1	1	0
0	0	0	0	0	0

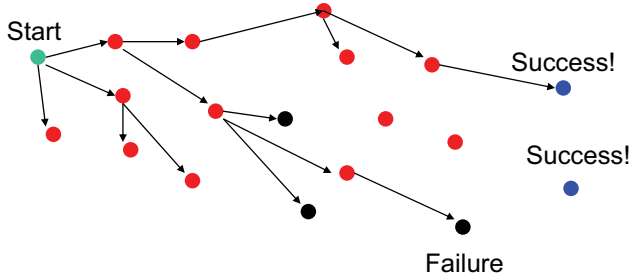
Update Code

Topic 10
Recursive Backtracking

"In ancient times, before computers were invented, alchemists studied the mystical properties of numbers. Lacking computers, they had to rely on dragons to do their work for them. The dragons were clever beasts, but also lazy and bad-tempered. The worst ones would sometimes burn their keeper to a crisp with a single fiery belch. But most dragons were merely uncooperative, as violence required too much energy. This is the story of how Martin, an alchemist's apprentice, discovered recursion by outsmarting a lazy dragon."

- David S. Touretzky, *Common Lisp: A Gentle Introduction to Symbolic Computation*

Backtracking



Problem space consists of states (nodes) and actions (paths that lead to new states). When in a node can can only see paths to connected nodes

If a node only leads to failure go back to its "parent" node. Try other alternatives. If these all lead to failure then more backtracking may be necessary.

A More Concrete Example

- Sudoku
- 9 by 9 matrix with some numbers filled in
- all numbers must be between 1 and 9
- Goal: Each row, each column, and each mini matrix must contain the numbers between 1 and 9 once each
  - no duplicates in rows, columns, or mini matrices

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Solving Sudoku – Brute Force

- A *brute force* algorithm is a simple but general approach
- Try all combinations until you find one that works
- This approach isn't clever, but computers are fast
- Then try and improve on the brute force resuts

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Solving Sudoku

- Brute force Sudoku Soluton
  - if not open cells, solved
  - scan cells from left to right, top to bottom for first open cell
  - When an open cell is found start cycling through digits 1 to 9.
  - When a digit is placed check that the set up is legal
  - now solve the board

5	3	1		7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

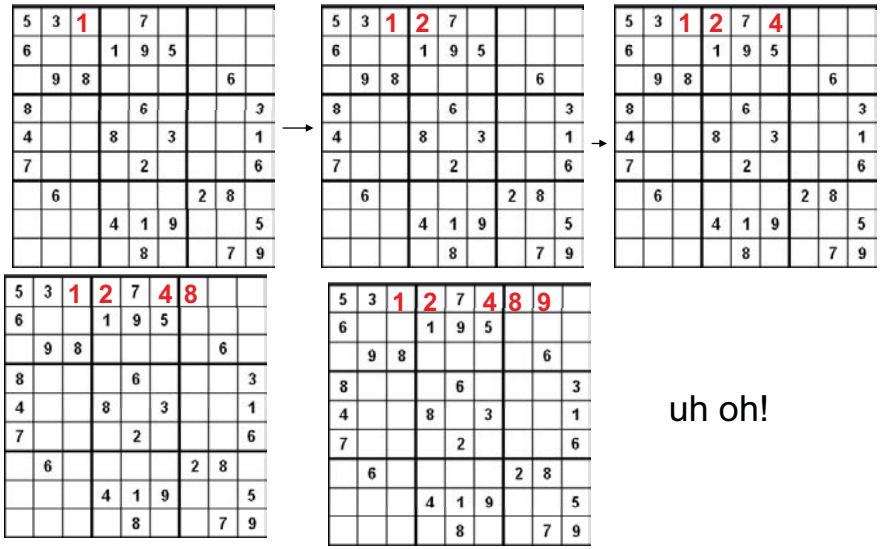
Attendance Question 1

- After placing a number in a cell is the remaining problem very similar to the original problem?

A. Yes

B. No

Solving Sudoku – Later Steps



uh oh!

Sudoku – A Dead End

- We have reached a dead end in our search

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- With the current set up none of the nine digits work in the top right corner

Backing Up

- When the search reaches a dead end in ***backs up*** to the previous cell it was trying to fill and goes onto to the next digit
- We would back up to the cell with a 9 and that turns out to be a dead end as well so we back up again
  - so the algorithm needs to remember what digit to try next
- Now in the cell with the 8. We try and 9 and move forward again.

5	3	1	2	7	4	8	9	
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	1	2	7	4	9		
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Characteristics of Brute Force and Backtracking

- Brute force algorithms are slow
- The don't employ a lot of logic
  - For example we know a 6 can't go in the last 3 columns of the first row, but the brute force algorithm will plow ahead any way
- But, brute force algorithms are fairly easy to implement as a first pass solution
  - backtracking is a form of a brute force algorithm

Key Insights

- After trying placing a digit in a cell we want to solve the new sudoku board
  - Isn't that a smaller (or simpler version) of the same problem we started with?!?!?!?
- After placing a number in a cell the we need to remember the next number to try in case things don't work out.
- We need to know if things worked out (found a solution) or they didn't, and if they didn't try the next number
- If we try all numbers and none of them work in our cell we need to *report back* that things didn't work

Recursive Backtracking

- Problems such as Suduko can be solved using recursive backtracking
- recursive because later versions of the problem are just slightly simpler versions of the original
- backtracking because we may have to try different alternatives

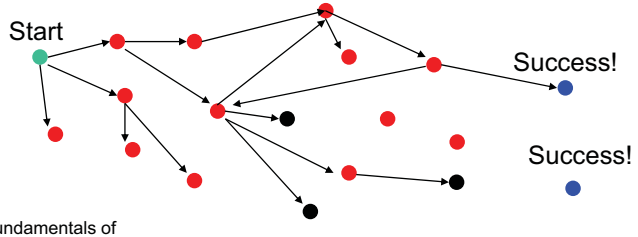
Recursive Backtracking

Pseudo code for recursive backtracking algorithms

If at a solution, report success
for(every possible choice from current state / node)
  Make that choice and take one step along path
  Use recursion to solve the problem for the new node / state
  If the recursive call succeeds, report the success to the next high level
  Back out of the current choice to restore the state at the beginning of the loop.
Report failure

Goals of Backtracking

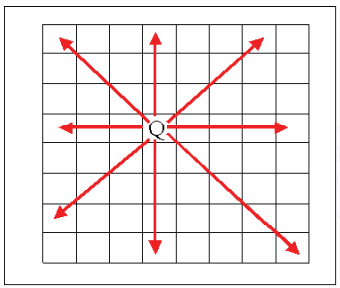
- Possible goals
  - Find a path to success
  - Find all paths to success
  - Find the best path to success
- Not all problems are exactly alike, and finding one success node may not be the end of the search

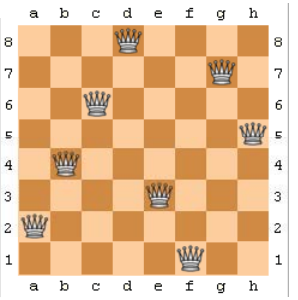


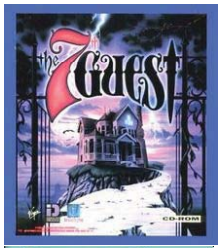
The 8 Queens Problem

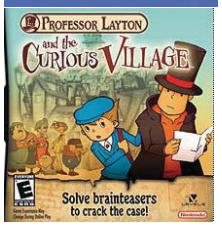
The 8 Queens Problem

- A classic chess puzzle
  - Place 8 queen pieces on a chess board so that none of them can attack one another









The N Queens Problem

- Place N Queens on an N by N chessboard so that none of them can attack each other
- Number of possible placements?
- In 8 x 8

  64 * 63 * 62 * 61 * 60 * 59 * 58 * 57
  = 178,462, 987, 637, 760 / 8!
  = 4,426,165.368

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdots (n-k+1)}{k \cdot (k-1) \cdots 1} = \frac{n!}{k!(n-k)!} \quad \text{if } 0 \le k \le n \qquad (1)$$

  n choose k
  – How many ways can you choose k things from a set of n items?
  – In this case there are 64 squares and we want to choose 8 of them to put queens on

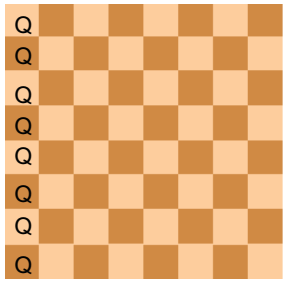
Attendance Question 2

- For valid solutions how many queens can be placed in a give column?

A. 0
B. 1
C. 2
D. 3
E. 4
F. Any number

Reducing the Search Space

- The previous calculation includes set ups like this one



- Includes lots of set ups with multiple queens in the same column
- How many queens can there be in one column?
- Number of set ups
  8 * 8 * 8 * 8 * 8 * 8 * 8 * 8 = 16,777,216
- We have reduced search space by two orders of magnitude by applying some logic

A Solution to 8 Queens

- If number of queens is fixed and I realize there can't be more than one queen per column I can iterate through the rows for each column

```
for(int c0 = 0; c0 < 8; c0++){
      board[c0][0] = 'q';
       for(int c1 = 0; c1 < 8; c1++){
            board[c1][1] = 'q';
            for(int c2 = 0; c2 < 8; c2++){
                  board[c2][2] = 'q';
                  // a little later
                  for(int c7 = 0; c7 < 8; c7++){
                        board[c7][7] = 'q';
                        if( queensAreSafe(board) )
                              printSolution(board);
                        board[c7][7] = ' '; //pick up queen
                  }
                  board[c6][6] = ' '; // pick up queen
```

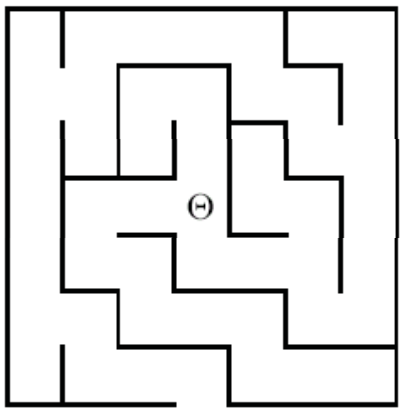
N Queens

- The *problem* with N queens is you don't know how many for loops to write.
- Do the problem recursively
- Write recursive code with class and demo
  - show backtracking with breakpoint and debugging option

Recursive Backtracking

- You must practice!!!
- Learn to recognize problems that fit the pattern
- Is a *kickoff* method needed?
- All solutions or a solution?
- Reporting results and acting on results

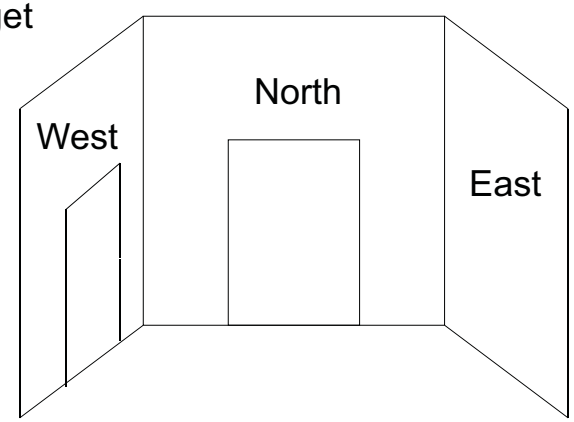
Another Backtracking Problem A Simple Maze



Search maze until way out is found. If no way out possible report that.

The Local View

Which way do I go to get out?

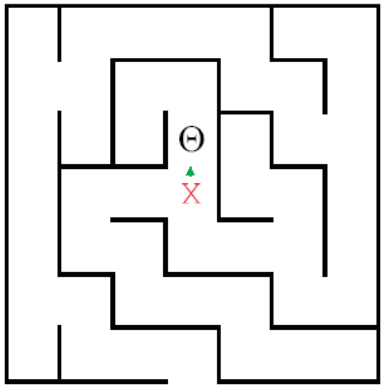


Behind me, to the South is a door leading South

Modified Backtracking Algorithm for Maze

- If the current square is outside, return TRUE to indicate that a solution has been found.
  If the current square is marked, return FALSE to indicate that this path has been tried.
  Mark the current square.
  for (each of the four compass directions)
  { if (this direction is not blocked by a wall)
     { Move one step in the indicated direction from the current square.
        Try to solve the maze from there by making a recursive call.
        If this call shows the maze to be solvable, return TRUE to indicate that fact.
     }
  }
  Unmark the current square.
  Return FALSE to indicate that none of the four directions led to a solution.

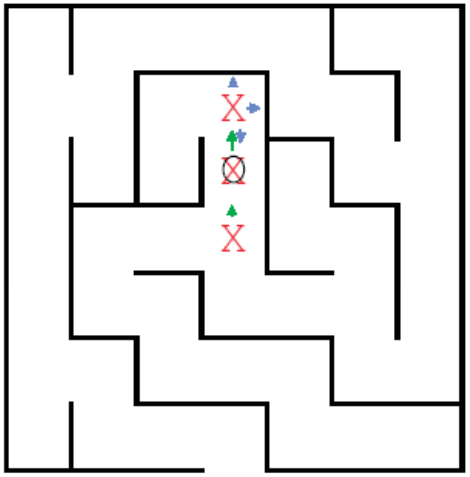
Backtracking in Action



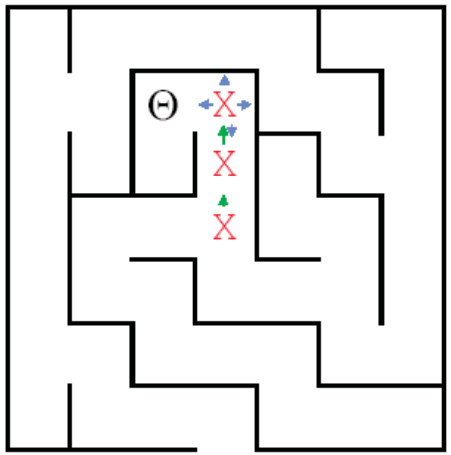
The crucial part of the algorithm is the for loop that takes us through the alternatives from the curren square. Here we have move to the North.

```
for (dir = North; dir <= West; dir++)
{     if (!WallExists(pt, dir))
      {if (SolveMaze(AdjacentPoint(pt, dir)))
            return(TRUE);
}
```


Backtracking in Action

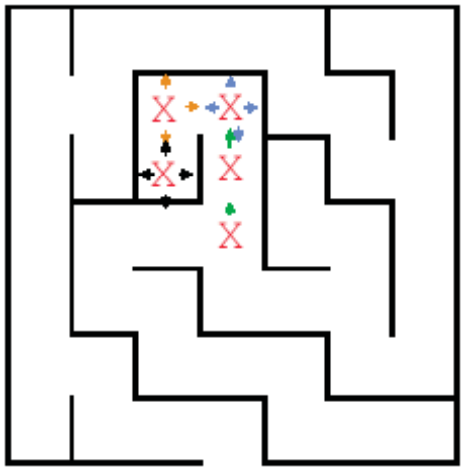


Here we have moved North again, but there is a wall to the North . East is also blocked, so we try South. That call discovers that the square is marked, so it just returns.



So the next move we can make is West.

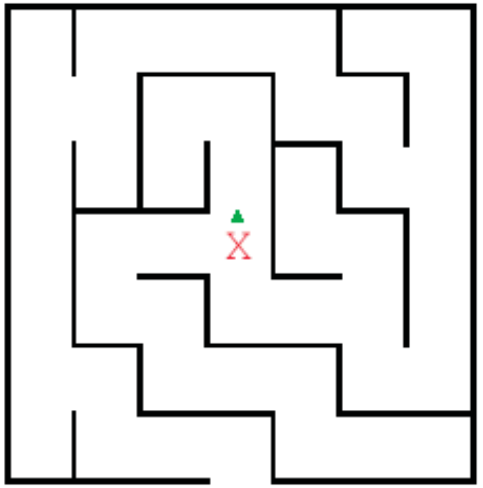
Where is this leading?



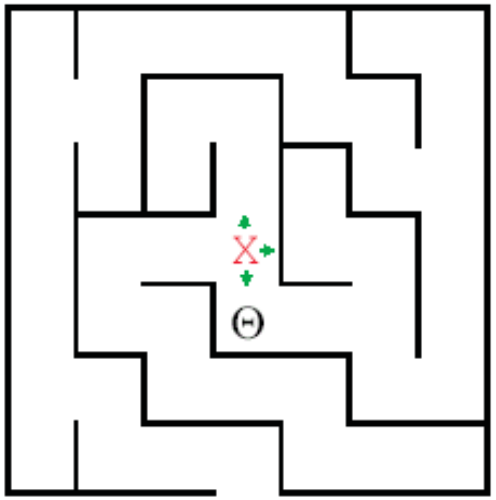
This path reaches a dead end.

Time to backtrack!

Remember the program stack!

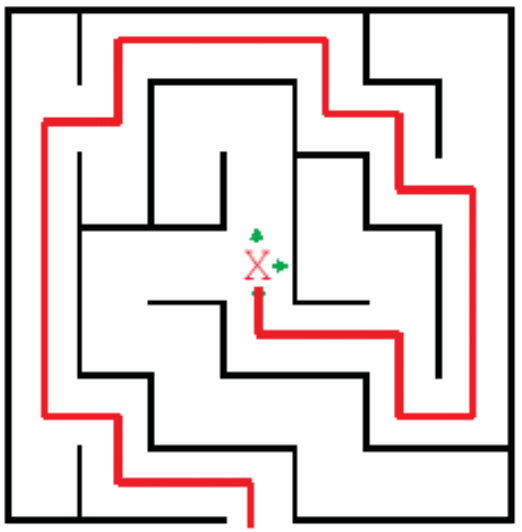


The recursive calls end and return until we find ourselves back here.



And now we try South

Path Eventually Found



More Backtracking Problems

Other Backtracking Problems

- Knight's Tour
- Regular Expressions
- Knapsack problem / Exhaustive Search
  - Filling a knapsack. Given a choice of items with various weights and a limited carrying capacity find the optimal load out. 50 lb. knapsack. items are 1 40 lb, 1 32 lb. 2 22 lbs, 1 15 lb, 1 5 lb. A greedy algorithm would choose the 40 lb item first. Then the 5 lb. Load out = 45lb. Exhaustive search 22 + 22 + 5 = 49.

The CD problem

- We want to put songs on a Compact Disc. 650MB CD and a bunch of songs of various sizes.

```
If there are no more songs to consider return result
else{
    Consider the next song in the list.
        Try not adding it to the CD so far and use recursion to evaluate best
                without it.
        Try adding it to the CD, and use recursion to evaluate best with it
        Whichever is better is returned as absolute best from here
    }
```


Another Backtracking Problem

- Airlines give out frequent flier miles as a way to get people to always fly on their airline.
- Airlines also have partner airlines. Assume if you have miles on one airline you can redeem those miles on any of its partners.
- Further assume if you can redeem miles on a partner airline you can redeem miles on any of its partners and so forth...
  - Airlines don't usually allow this sort of thing.
- Given a list of airlines and each airlines partners determine if it is possible to redeem miles on a given airline A on another airline B.

Airline List – Part 1

- Delta
  - partners: Air Canada, Aero Mexico, OceanAir
- United
  - partners: Aria, Lufthansa, OceanAir, Quantas, British Airways
- Northwest
  - partners: Air Alaska, BMI, Avolar, EVA Air
- Canjet
  - partners: Girjet
- Air Canda
  - partners: Areo Mexico, Delta, Air Alaska
- Aero Mexico
  - partners: Delta, Air Canda, British Airways

Airline List - Part 2

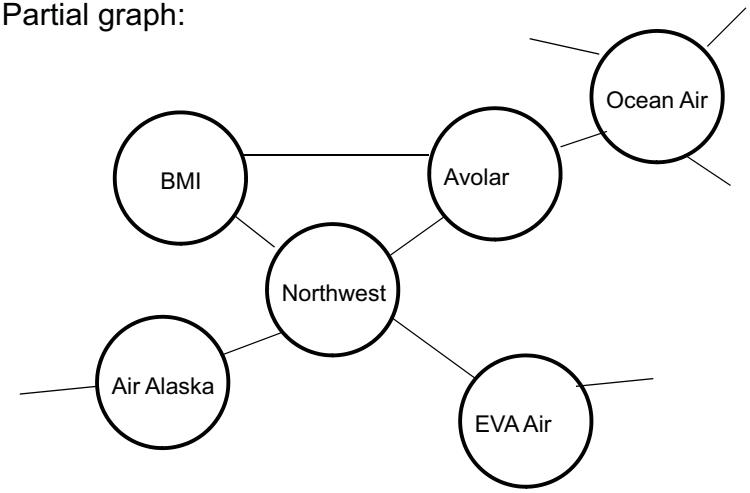
- Ocean Air
  - partners: Delta, United, Quantas, Avolar
- AlohaAir
  - partners: Quantas
- Aria
  - partners: United, Lufthansa
- Lufthansa
  - partners: United, Aria, EVA Air
- Quantas
  - partners: United, OceanAir, AlohaAir
- BMI
  - partners: Northwest, Avolar
- Maxair
  - partners: Southwest, Girjet

Airline List - Part 3

- Girjet
  - partners: Southwest, Canjet, Maxair
- British Airways
  - partners: United, Aero Mexico
- Air Alaska
  - partners: Northwest, Air Canada
- Avolar
  - partners: Northwest, Ocean Air, BMI
- EVA Air
  - partners: Northwest, Luftansa
- Southwest
  - partners: Girjet, Maxair

Problem Example

- If I have miles on Northwest can I redeem them on Aria?
- Partial graph:



Topic 11
Sorting and Searching

"There's nothing in your head the sorting hat can't see. So try me on and I will tell you where you ought to be."

-The Sorting Hat, *Harry Potter and the Sorcerer's Stone*

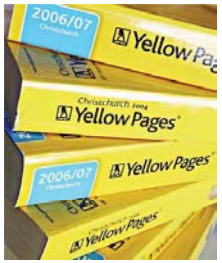
Sorting and Searching

- Fundamental problems in computer science and programming
- Sorting done to make searching easier
- Multiple different algorithms to solve the same problem
  - How do we know which algorithm is "better"?
- Look at searching first
- Examples will use arrays of ints to illustrate algorithms

Searching

Searching

- Given a list of data find the location of a particular value or report that value is not present
- linear search
  - intuitive approach
  - start at first item
  - is it the one I am looking for?
  - if not go to next item
  - repeat until found or all items checked
- If items not sorted or unsortable this approach is necessary



Linear Search

```
/*    pre: list != null
      post: return the index of the first occurrence
      of target in list or -1 if target not present in
      list
*/
public int linearSearch(int[] list, int target) {
    for(int i = 0; i < list.length; i++)
        if( list[i] == target )
            return i;
    return -1;
}
```


Linear Search, Generic

```
/*    pre: list != null, target != null
      post: return the index of the first occurrence
      of target in list or -1 if target not present in
      list
*/
public int linearSearch(Object[] list, Object target) {
    for(int i = 0; i < list.length; i++)
        if( list[i] != null && list[i].equals(target) )
            return i;
    return -1;
}
```

T(N)? Big O? Best case, worst case, average case?

Attendance Question 1

- What is the average case Big O of linear search in an array with N items, if an item is present?

A. O(N)

B. $O(N^2)$

C. O(1)

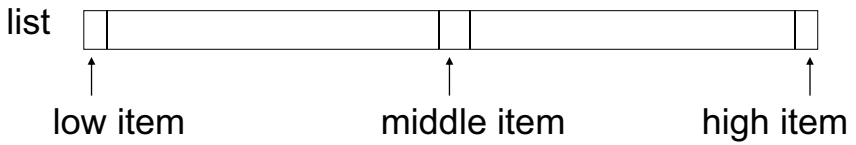
D. O(logN)

E. O(NlogN)

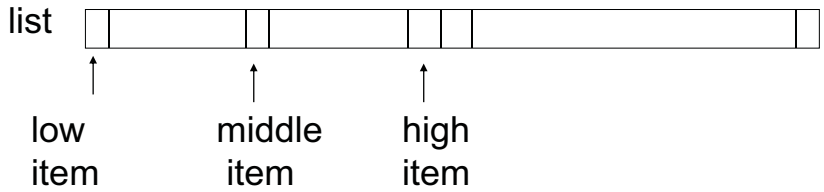
Searching in a Sorted List

- If items are sorted then we can *divide and conquer*
- dividing your work in half with each step
  - generally a good thing
- The Binary Search on List in Ascending order
  - Start at middle of list
  - is that the item?
  - If not is it less than or greater than the item?
  - less than, move to second half of list
  - greater than, move to first half of list
  - repeat until found or sub list size = 0

Binary Search



Is middle item what we are looking for? If not is it more or less than the target item? (Assume lower)



and so forth…

Binary Search in Action

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53

```
public static int bsearch(int[] list, int target)
{       int result = -1;
        int low = 0;
        int high = list.length - 1;
        int mid;
        while( result == -1 && low <= high )
        {       mid = low + ((high - low) / 2);
                if( list[mid] == target )
                        result = mid;
                else if( list[mid] < target)
                        low = mid + 1;
                else
                        high = mid - 1;
        }
        return result;
}
// mid = ( low + high ) / 2; // may overflow!!!
// or mid = (low + high) >>> 1; using bitwise op
```


Trace When Key == 3
Trace When Key == 30

Variables of Interest?

Attendance Question 2

What is the worst case Big O of binary search in an array with N items, if an item is present?

A. O(N)
B. $O(N^2)$
C. O(1)
D. O(logN)
E. O(NlogN)

Generic Binary Search

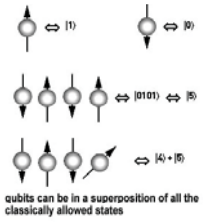
```
public static int bsearch(Comparable[] list, Comparable target)
{       int result = -1;
        int low = 0;
        int high = list.length - 1;
        int mid;
        while( result == -1 && low <= high )
        {       mid = low + ((high - low) / 2);
                if( target.equals(list[mid]) )
                        result = mid;
                else if(target.compareTo(list[mid]) > 0)
                        low = mid + 1;
                else
                        high = mid - 1;
        }
        return result;
}
```


Recursive Binary Search

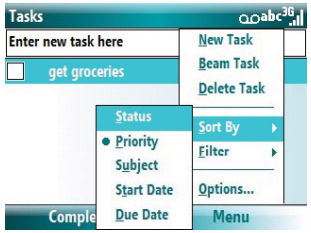
```
public static int bsearch(int[] list, int target){
    return bsearch(list, target, 0, list.length - 1);
}

public static int bsearch(int[] list, int target,
                                  int first, int last){
    if( first <= last ){
        int mid = low + ((high - low) / 2);
        if( list[mid] == target )
            return mid;
        else if( list[mid] > target )
            return bsearch(list, target, first, mid - 1);
        else
            return bsearch(list, target, mid + 1, last);
    }
    return -1;
}
```


Other Searching Algorithms

- Interpolation Search
  - more like what people really do
- Indexed Searching
- Binary Search Trees
- Hash Table Searching
- Grover's Algorithm (Waiting for quantum computers to be built)
- best-first
- A*



Sorting



U.S. All-time List - Marathon

As of 4/24/08

Women

1	1	2:19:36	Deena Kastor nee Drossin
2		2:21:16	Drossin (2)
3	2	2:21:21	Joan Benoit Samuelson
4		2:21:25	Kastor (3)
5		2:22:43a	Benoit (2)
6		2:24:52a	Benoit (3)
7		2:26:11	Benoit (4)
8	3	2:26:26a	Julie Brown
9	4	2:26:40a	Kim Jones

Sorting Fun
Why Not Bubble Sort?

Sorting

- A fundamental application for computers
- Done to make finding data (searching) faster
- Many different algorithms for sorting
- One of the difficulties with sorting is working with a fixed size storage container (array)
  - if resize, that is expensive (slow)
- The "simple" sorts run in quadratic time $O(N^2)$
  - bubble sort
  - selection sort
  - insertion sort

Stable Sorting

- A property of sorts
- If a sort guarantees the relative order of equal items stays the same then it is a *stable sort*
- [7_1, 6, 7_2, 5, 1, 2, 7_3, -5]
  - subscripts added for clarity
- [-5, 1, 2, 5, 6, 7_1, 7_2, 7_3]
  - result of stable sort
- Real world example:
  - sort a table in Wikipedia by one criteria, then another
  - sort by country, then by major wins

Selection sort

- Algorithm
  - Search through the list and find the smallest element
  - swap the smallest element with the first element
  - repeat starting at second element and find the second smallest element

```
public static void selectionSort(int[] list)
{     int min;
      int temp;
      for(int i = 0; i < list.length - 1; i++) {
            min = i;
            for(int j = i + 1; j < list.length; j++)
                  if( list[j] < list[min] )
                        min = j;
            temp = list[i];
            list[i] = list[min];
            list[min] = temp;
       }
}
```

Selection Sort in Practice

44 68 191 119 119 37 83 82 191 45 158 130 76 153 39 25

What is the T(N), *actual* number of statements executed, of the selection sort code, given a list of N elements? What is the Big O?

Generic Selection Sort

```
public void selectionSort(Comparable[] list)
{  int min; Comparable temp;
   for(int i = 0; i < list.length - 1; i++) {
   {  min = i;
      for(int j = i + 1; j < list.length; j++)
         if( list[min].compareTo(list[j]) > 0 )
            min = j;
      temp = list[i];
      list[i] = list[min];
      list[min] = temp;
   }
}
```

- Best case, worst case, average case Big O?

Attendance Question 3

Is selection sort always stable?

A. Yes

B. No

Insertion Sort

- Another of the O(N^2) sorts
- The first item is sorted
- Compare the second item to the first
  - if smaller swap
- Third item, compare to item next to it
  - need to swap
  - after swap compare again
- And so forth…

Insertion Sort Code

```
public void insertionSort(int[] list)
{    int temp, j;
     for(int i = 1; i < list.length; i++)
     {    temp = list[i];
          j = i;
          while( j > 0 && temp < list[j - 1])
          {    // swap elements
               list[j] = list[j - 1];
               list[j - 1] = temp;
               j--;
          }
     }
}
```

- Best case, worst case, average case Big O?

Attendance Question 4

- Is the version of insertion sort shown always stable?

A. Yes

B. No

Comparing Algorithms

- Which algorithm do you think will be faster given random data, selection sort or insertion sort?
- Why?

Sub Quadratic Sorting Algorithms

Sub Quadratic means having a Big O better than $O(N^2)$

ShellSort

- Created by Donald Shell in 1959
- Wanted to stop moving data small distances (in the case of insertion sort and bubble sort) and stop making swaps that are not helpful (in the case of selection sort)
- Start with sub arrays created by looking at data that is far apart and then reduce the gap size

ShellSort in practice

46 2 83 41 102 5 17 31 64 49 18

Gap of five. Sort sub array with 46, 5, and 18
5 2 83 41 102 18 17 31 64 49 46

Gap still five. Sort sub array with 2 and 17
5 2 83 41 102 18 17 31 64 49 46

Gap still five. Sort sub array with 83 and 31
5 2 31 41 102 18 17 83 64 49 46

Gap still five Sort sub array with 41 and 64
5 2 31 41 102 18 17 83 64 49 46

Gap still five. Sort sub array with 102 and 49
5 2 31 41 49 18 17 83 64 102 46

Continued on next slide:

Completed Shellsort

5 2 31 41 49 18 17 83 64 102 46

Gap now 2: Sort sub array with 5 31 49 17 64 46
5 2 17 41 31 18 46 83 49 102 64

Gap still 2: Sort sub array with 2 41 18 83 102
5 2 17 18 31 41 46 83 49 102 64

Gap of 1 (Insertion sort)

2 5 17 18 31 41 46 49 64 83 102

Array sorted

Shellsort on Another Data Set

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
44	68	191	119	119	37	83	82	191	45	158	130	76	153	39	25

Initial gap = length / 2 = 16 / 2 = 8

initial sub arrays indices:

{0, 8}, {1, 9}, {2, 10}, {3, 11}, {4, 12}, {5, 13}, {6, 14}, {7, 15}
next gap = 8 / 2 = 4

{0, 4, 8, 12}, {1, 5, 9, 13}, {2, 6, 10, 14}, {3, 7, 11, 15}
next gap = 4 / 2 = 2

{0, 2, 4, 6, 8, 10, 12, 14}, {1, 3, 5, 7, 9, 11, 13, 15}

final gap = 2 / 2 = 1

ShellSort Code

```
public static void shellsort(Comparable[] list)
{  Comparable temp; boolean swap;
   for(int gap = list.length / 2; gap > 0; gap /= 2)
      for(int i = gap; i < list.length; i++)
      {   Comparable tmp = list[i];
          int j = i;
          for( ; j >= gap &&
                 tmp.compareTo( list[j - gap] ) < 0;
                 j -= gap )
              list[ j ] = list[ j - gap ];
          list[ j ] = tmp;
       }
}
```


Comparison of Various Sorts

Num Items	Selection	Insertion	Shellsort	Quicksort
1000	16	5	0	0
2000	59	49	0	6
4000	271	175	6	5
8000	1056	686	11	0
16000	4203	2754	32	11
32000	16852	11039	37	45
64000	expected?	expected?	100	68
128000	expected?	expected?	257	158
256000	expected?	expected?	543	335
512000	expected?	expected?	1210	722
1024000	expected?	expected?	2522	1550

times in milliseconds

Quicksort

- Invented by C.A.R. (Tony) Hoare
- A divide and conquer approach that uses recursion

1. If the list has 0 or 1 elements it is sorted
2. otherwise, pick any element p in the list. This is called the pivot value
3. Partition the list minus the pivot into two sub lists according to values less than or greater than the pivot. (equal values go to either)
4. return the quicksort of the first list followed by the quicksort of the second list

Quicksort in Action

39 23 17 90 33 72 46 79 11 52 64 5 71

Pick middle element as pivot: 46

Partition list

23 17 5 33 39 11 46 79 72 52 64 90 71

quick sort the less than list

Pick middle element as pivot: 33

23 17 5 11 33 39

quicksort the less than list, pivot now 5

{} 5 23 17 11

quicksort the less than list, base case

quicksort the greater than list

Pick middle element as pivot: 17

and so on….

Quicksort on Another Data Set

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
44	68	191	119	119	37	83	82	191	45	158	130	76	153	39	25

Big O of Quicksort?


```
public static void swapReferences( Object[] a, int index1, int index2 )
{   Object tmp = a[index1];
    a[index1] = a[index2];
    a[index2] = tmp;
}

 public void quicksort( Comparable[] list, int start, int stop )
 {    if(start >= stop)
        return; //base case list of 0 or 1 elements

      int pivotIndex = (start + stop) / 2;

      // Place pivot at start position
      swapReferences(list, pivotIndex, start);
      Comparable pivot = list[start];

      // Begin partitioning
      int i, j = start;

      // from first to j are elements less than or equal to pivot
      // from j to i are elements greater than pivot
      // elements beyond i have not been checked yet
      for(i = start + 1; i <= stop; i++ )
      {   //is current element less than or equal to pivot
          if(list[i].compareTo(pivot) <= 0)
          {   // if so move it to the less than or equal portion
              j++;
              swapReferences(list, i, j);
          }
      }

      //restore pivot to correct spot
      swapReferences(list, start, j);
      quicksort( list, start, j - 1 );    // Sort small elements
      quicksort( list, j + 1, stop );   // Sort large elements
  }
```


Attendance Question 5

- What is the best case and worst case Big O of quicksort?

	Best	Worst
A.	O(NlogN)	$O(N^2)$
B.	$O(N^2)$	$O(N^2)$
C.	$O(N^2)$	O(N!)
D.	O(NlogN)	O(NlogN)
E.	O(N)	O(NlogN)

Quicksort Caveats

- Average case Big O?
- Worst case Big O?
- Coding the partition step is usually the hardest part

Attendance Question 6

- You have 1,000,000 items that you will be searching. How many searches need to be performed before the data is changed to make sorting worthwhile?

A. 10

B. 40

C. 1,000

D. 10,000

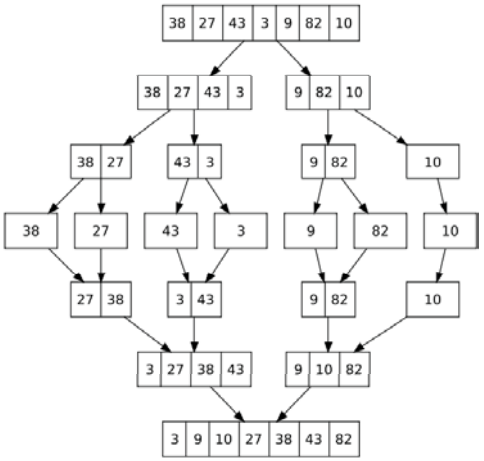
E. 500,000

Merge Sort Algorithm

Don Knuth cites John von Neumann as the creator of this algorithm

1. If a list has 1 element or 0 elements it is sorted
2. If a list has more than 2 split into into 2 separate lists
3. Perform this algorithm on each of those smaller lists
4. Take the 2 sorted lists and merge them together

Merge Sort



When implementing one temporary array is used instead of multiple temporary arrays.

Why?

Merge Sort code

```
/**
 * perform a merge sort on the data in c
 * @param c c != null, all elements of c
 * are the same data type
 */
public static void mergeSort(Comparable[] c)
{       Comparable[] temp = new Comparable[ c.length ];
        sort(c, temp, 0, c.length - 1);
}

private static void sort(Comparable[] list, Comparable[] temp,
                                int low, int high)
{       if( low < high){
                int center = (low + high) / 2;
                sort(list, temp, low, center);
                sort(list, temp, center + 1, high);
                merge(list, temp, low, center + 1, high);
        }
}
```

Merge Sort Code

```
private static void merge( Comparable[] list, Comparable[] temp,
                           int leftPos, int rightPos, int rightEnd){
          int leftEnd = rightPos - 1;
          int tempPos = leftPos;
          int numElements = rightEnd - leftPos + 1;
          //main loop
          while( leftPos <= leftEnd && rightPos <= rightEnd){
                    if( list[ leftPos ].compareTo(list[rightPos]) <= 0){
                              temp[ tempPos ] = list[ leftPos ];
                              leftPos++;
                    }
                    else{
                              temp[ tempPos ] = list[ rightPos ];
                              rightPos++;
                    }
                    tempPos++;
          }
          //copy rest of left half
          while( leftPos <= leftEnd){
                    temp[ tempPos ] = list[ leftPos ];
                    tempPos++;
                    leftPos++;
          }
          //copy rest of right half
          while( rightPos <= rightEnd){
                    temp[ tempPos ] = list[ rightPos ];
                    tempPos++;
                    rightPos++;
          }
          //Copy temp back into list
          for(int i = 0; i < numElements; i++, rightEnd--)
                    list[ rightEnd ] = temp[ rightEnd ];
}
```


Final Comments

- Language libraries often have sorting algorithms in them
  - Java Arrays and Collections classes
  - C++ Standard Template Library
  - Python sort and sorted functions
- Hybrid sorts
  - when size of unsorted list or portion of array is small use insertion sort, otherwise use O(N log N) sort like Quicksort of Mergesort
- Many other sorting algorithms exist.

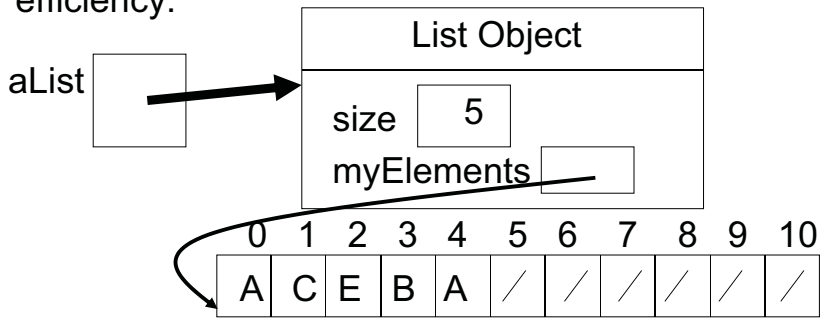
Topic 12

ADTS, Data Structures, Java Collections and Generic Data Structures

"Get your data structures correct first, and the rest of the program will write itself."
- *David Jones*

Data Structures

- A *Data Structure* is:
  - an implementation of an abstract data type *and*
  - "An organization of information, usually in computer memory", for better algorithm efficiency."



Data Structure Concepts

- Data Structures are containers:
  - they hold other data
  - arrays are a data structure
  - ... so are lists
- Other types of data structures:
  - stack, queue, tree, binary search tree, hash table, dictionary or map, set, and on and on
  - www.nist.gov/dads/
  - en.wikipedia.org/wiki/List_of_data_structures
- Different types of data structures are optimized for certain types of operations

Core Operations

- Data Structures will have 3 core operations
  - a way to add things
  - a way to remove things
  - a way to access things
- Details of these operations depend on the data structure
  - Example: List, add at the end, access by location, remove by location
- More operations added depending on what data structure is designed to do

ADTs and Data Structures in Programming Languages

- Modern programming languages usually have a library of data structures
  - Java collections framework
  - C++ standard template library
  - .Net framework (small portion of VERY large library)
  - Python lists and tuples
  - Lisp lists

Data Structures in Java

- Part of the Java Standard Library is the *Collections Framework*
  - In class we will create our own data structures and discuss the data structures that exist in Java
- A library of data structures
- Built on two interfaces
  - Collection
  - Iterator
- http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html

The Java Collection interface

- A generic collection
- Can hold any object data type
- Which type a particular collection will hold is specified when declaring an instance of a class that implements the Collection interface
- Helps guarantee *type safety* at compile time

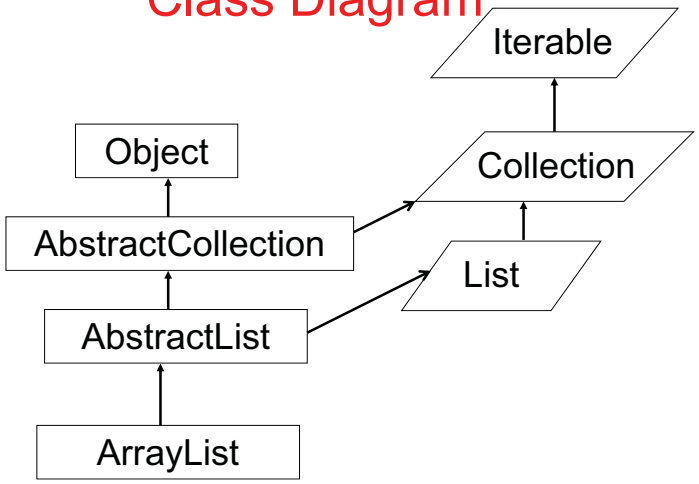
Methods in the Collection interface

```
public interface Collection<E>
{     public boolean add(E o)
      public boolean addAll(Collection<? extends E> c)
      public void clear()
      public boolean contains(Object o)
      public boolean containsAll(Collection<?> c)
      public boolean equals(Object o)
      public int hashCode()
      public boolean isEmpty()
      public Iterator<E> iterator()
      public boolean remove(Object o)
      public boolean removeAll(Collection<?> c)
      public boolean retainAll(Collection<?> c)
      public int size()
      public Object[] toArray()
      public <T> T[] toArray(T[] a)
}
```

The Java ArrayList Class

- Implements the List interface and uses an array as its *internal storage container*
- It is a list, not an array
- The array that actual stores the elements of the list is hidden, not visible outside of the ArrayList class
- all actions on ArrayList objects are via the methods
- ArrayLists are generic.
  - They can hold objects of any type!

ArrayList's (Partial) Class Diagram



Back to our Array Based List

- Started with a list of ints
- Don't want to have to write a new list class for every data type we want to store in lists
- Moved to an array of `Object`s to store the elements of the list

```
// from array based list
private Object[] myCon;
```


Using Object

- In Java, all classes inherit from exactly one other class except Object which is at the top of the class hierarchy
- Object variables can point at objects of their declared type and any descendants
  - polymorphism
- Thus, if the internal storage container is of type Object it can hold anything
  - primitives handled by *wrapping* them in objects. int – Integer, char - Character

Difficulties with Object

- *Creating* generic containers using the Object data type and polymorphism is relatively straight forward
- Using these generic containers leads to some difficulties
  - Casting
  - Type checking
- Code examples on the following slides

Attendance Question 1

- What is output by the following code?

```
ArrayList list = new ArrayList();
String name = "Olivia";
list.add(name);
System.out.print( list.get(0).charAt(2) );
```

A. `i`

B. `O`

C. `l`

D. No output due to syntax error.

E. No output due to runtime error.

Code Example - Casting

- Assume a list class

```
ArrayList li = new ArrayList();
li.add("Hi");
System.out.println( li.get(0).charAt(0) );
// previous line has syntax error
// return type of get is Object
// Object does not have a charAt method
// compiler relies on declared type
System.out.println(
      ((String)li.get(0)).charAt(0) );
// must cast to a String
```


Code Example – type checking

```
//pre: all elements of li are Strings
public void printFirstChar(ArrayList li){
     String temp;
     for(int i = 0; i < li.size(); i++)
     {    temp = (String)li.get(i);
          if( temp.length() > 0 )
               System.out.println(
                    temp.charAt(0) );
     }
}
// what happens if pre condition not met?
```

Too Generic?

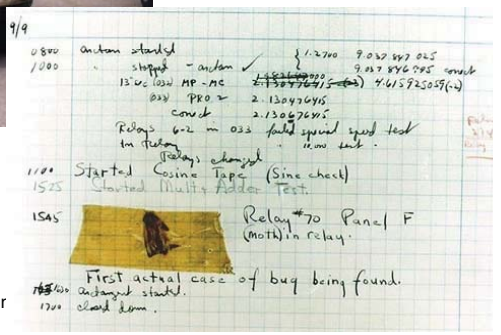
- Does the compiler allow this?

```
ArrayList list = new ArrayList();
list.add( "Olivia" );
list.add( new Integer(12) );
list.add( new Rectangle() );
list.add( new ArrayList() );
```

A. Yes

B. No

Is this a bug or a feature?



"Fixing" the Method

```
//pre: all elements of li are Strings
public void printFirstChar(ArrayList li){
     String temp;
     for(int i = 0; i < li.size(); i++){
          if( li.get(i) instanceof String ){
               temp = (String)li.get(i);
               if( temp.length() > 0 )
                    System.out.println(
                         temp.charAt(0) );
          }
     }
}
```


Generic Types

- Java has syntax for *parameterized data types*
- Referred to as *Generic Types* in most of the literature
- A traditional parameter *has* a data type and can store various values just like a variable

```
public void foo(int x)
```

- Generic Types are like parameters, but the data type for the parameter is *data type*
  - like a variable that stores a data type

Making our Array List Generic

- Data type variables declared in class header

```
public class GenericList<E> {
```

- The `<E>` is the declaration of a data type parameter for the class
  - any legal identifier: `Foo`, `AnyType`, `Element`, `DataTypeThisListStores`
  - Sun style guide recommends terse identifiers
- The value E stores will be filled in whenever a programmer creates a new `GenericList`

```
GenericList<String> li =
            new GenericList<String>();
```


Modifications to GenericList

- instance variable

```
private E[] myCon;
```

- Parameters on
  - add, insert, remove, insertAll
- Return type on
  - get
- Changes to creation of internal storage container

```
myCon = (E[])new Object[DEFAULT_SIZE];
```

- Constructor header does not change

Using Generic Types

- Back to Java's ArrayList

```
ArrayList list1 = new ArrayList();
```

  - still allowed, a "raw" ArrayList
  - works just like our first pass at GenericList
  - casting, lack of type safety

Using Generic Types

```
ArrayList<String> list2 =
            new ArrayList<String>();
```

  - for list2 `E` stores `String`

```
list2.add( "Isabelle" );
System.out.println(
    list2.get(0).charAt(2) ); //ok
list2.add( new Rectangle() );
// syntax error
```

Parameters and Generic Types

- Old version

```
//pre: all elements of li are Strings
public void printFirstChar(ArrayList li){
```

- New version

```
//pre: none
public void printFirstChar(ArrayList<String> li){
```

- Elsewhere

```
ArrayList<String> list3 = new ArrayList<String>();
printFirstChar( list3 ); // ok
ArrayList<Integer> list4 = new ArrayList<Integer>();
printFirstChar( list4 ); // syntax error
```


Generic Types and Subclasses

```
ArrayList<ClosedShape> list5 =
        new ArrayList<ClosedShape>();
list5.add( new Rectangle() );
list5.add( new Square() );
list5.add( new Circle() );
// all okay
```

- `list5` can store `ClosedShape`s and any descendants of `ClosedShape`

Topic 14
Iterators

"First things first, but not necessarily in that order "

-Dr. Who

A Question

```
public class WordList {
  private ArrayList<String> myList;

  // pre: none
  // post: all words that are exactly len
  // characters long have been removed from
  // this WordList with the order of the
  // remaining words unchanged
  public void removeWordsOfLength(int len){
     for(int i = 0; i < myList.size(); i++){
          if( myList.get(i).length() == len )
               myList.remove(i);
     }
```


Attendance Question 1

- When does method `removeWordsOfLength` work as intended?

A. Always

B. Sometimes

C. Never

```
// original list = [“dog”, “cat”, “hat”, “sat”]
// resulting list after removeWordsOfLength(3) ?
```


The Remove Question

- Answer?

```
public void removeWordsOfLength(int len) {
  Iterator<String> it = myList.iterator();
  while( it.hasNext() )
     if( it.next().length() == len )
          it.remove();


  }
}
// original list = [“dog”, “cat”, “hat”, “sat”]
// resulting list after removeWordsOfLength(3) ?
```

Iterators

- ArrayList is part of the Java Collections framework
- Collection is an interface that specifies the basic operations every collection (data structure) should have
- Some Collections don't have a definite order
  - Sets, Maps, Graphs
- How to access all the items in a Collection with no specified order?

Access All Elements - ArrayList

```
public void printAll(ArrayList list){
    for(int i = 0; i < list.size(); i++)
        System.out.println(list.get(i));
}
```

- How do I access all the elements of a Set? The elements don't have an index.
- *Iterator* objects provide a way to go through all the elements of a Collection, one at a time

Iterator Interface

- An iterator object is a "one shot" object
  - it is designed to go through all the elements of a Collection once
  - if you want to go through the elements of a Collection again you have to get another iterator object
- Iterators are obtained by calling a method from the Collection

Iterator Methods

- The Iterator interface specifies 3 methods:

```
boolean hasNext()
//returns true if this iteration has more elements

Object next()
//returns the next element in this iteration
//pre: hastNext()

void remove()
/*Removes from the underlying collection the last element
   returned by the iterator.
  pre: This method can be called only once per call to next.
   After calling, must call next again before calling remove
   again.
*/
```

Attendance Question 2

- Which of the following produces a syntax error?

```
ArrayList list = new ArrayList();
Iterator it1 = new Iterator(); // I
Iterator it2 = new Iterator(list); // II
Iterator it3 = list.iterator(); // III
```

A. I
B. II
C. III
D. I and II
E. II and III

Typical Iterator Pattern

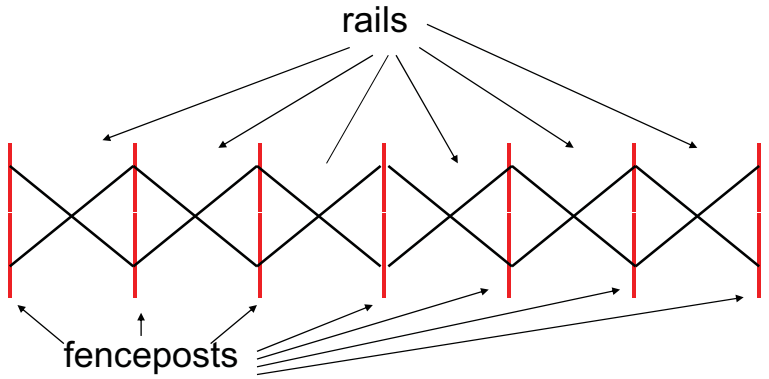
```
public void printAll(ArrayList list){
    Iterator it = list.iterator();
    Object temp;
    while( it.hasNext() ) {
        temp = it.next();
        System.out.println( temp );
    }
}
```


Typical Iterator Pattern 2

```
public void printAll(ArrayList list){
    Iterator it = list.iterator();
    while( it.hasNext() )
        System.out.println( it.next() );
}

// go through twice?
public void printAllTwice(ArrayList list){
    Iterator it = list.iterator();
    while( it.hasNext() )
        System.out.println( it.next() );
    it = list.iterator();
    while( it.hasNext() )
        System.out.println( it.next() );
}
```

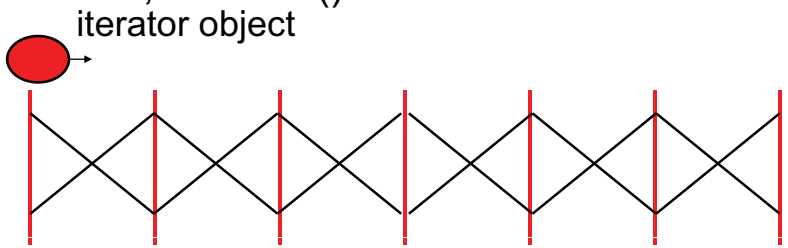

A Picture of an Iterator

- Imagine a fence made up of fence posts and rail sections



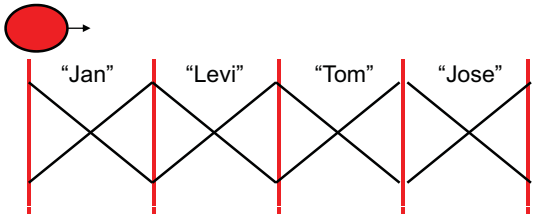
Fence Analogy

- The iterator lives on the fence posts
- The data in the collection are the rails
- Iterator created at the far left post
- As long as a rail exists to the right of the Iterator, hasNext() is true



Fence Analogy

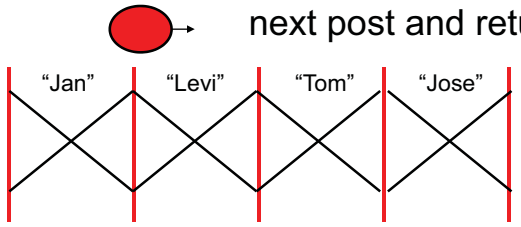
```
ArrayList<String> names =
                    new ArrayList<String>();
names.add("Jan");
names.add("Levi");
names.add("Tom");
names.add("Jose");
Iterator<String> it = names.iterator();
int i = 0;
```



Fence Analogy

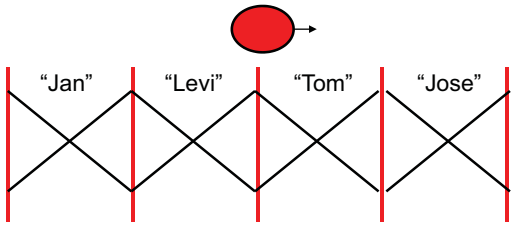
```
while( it.hasNext() ) {
    i++;
    System.out.println( it.next() );
}
// when i == 1, prints out Jan
```

first call to next moves iterator to next post and returns "Jan"



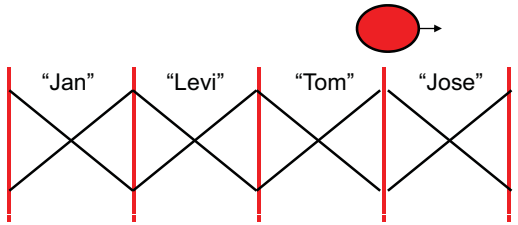
Fence Analogy

```
while( it.hasNext() ) {
     i++;
    System.out.println( it.next() );
}
// when i == 2, prints out Levi
```



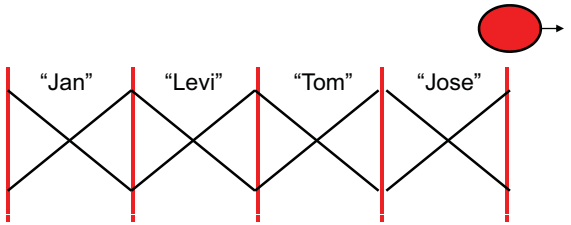
Fence Analogy

```
while( it.hasNext() ) {
     i++;
    System.out.println( it.next() );
}
// when i == 3, prints out Tom
```



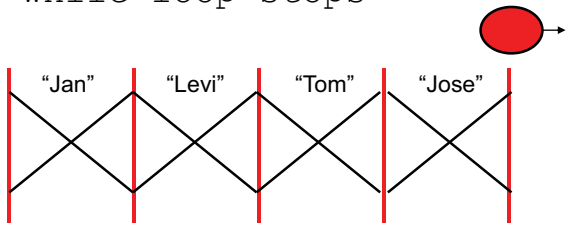
Fence Analogy

```
while( it.hasNext() ) {
     i++;
    System.out.println( it.next() );
}
// when i == 4, prints out Jose
```



Fence Analogy

```
while( it.hasNext() ) {
     i++;
    System.out.println( it.next() );
}
// call to hasNext returns false
// while loop stops
```



Attendance Question 3

- What is output by the following code?

```
ArrayList<Integer> list;
List = new ArrayList<Integer>();
list.add(3);
list.add(3);
list.add(5);
Iterator<Integer> it = list.iterator();
System.out.println(it.next());
System.out.println(it.next());
```

A. 3 B. 5 C. 3 3 5

D. 3 3 E. 3 5

Comodification

- If a `Collection` (`ArrayList`) is changed while an iteration via an iterator is in progress an Exception will be thrown the next time the `next()` or `remove()` methods are called via the iterator

```
ArrayList<String> names =
                    new ArrayList<String>();
names.add("Jan");
Iterator<String> it = names.iterator();
names.add("Andy");
it.next(); // exception will occur here
```


remove method

- Can use the `Iterator` to remove things from the `Collection`
- Can only be called once per call to `next()`

```
public void removeWordsOfLength(int len) {
  String temp;
  Iterator it = myList.iterator
  while( it.hasNext() ) {
     temp = (String)it.next();
      if( temp.length() == len )
          it.remove();
   }
}
// original list = ["dog", "cat", "hat", "sat"]
// resulting list after removeWordsOfLength(3) ?
```


Common Iterator Error

```
public void printAllOfLength(ArrayList<String> names,
                                        int len)
{   //pre: names != null, names only contains Strings
    //post: print out all elements of names equal in
    // length to len
    Iterator<String> it = names.iterator();
    while( it.hasNext() ){
        if( it.next().length() == len )
            System.out.println( it.next() );
    }
}
// given names = ["Jan", "Ivan", "Tom", "George"]
// and len = 3 what is output?
```


The Iterable Interface

- A related interface is `Iterable`
- One method in the interface:
  `public Iterator<T> iterator()`
- Why?
- Anything that implements the `Iterable` interface can be used in the for each loop.

```
ArrayList<Integer> list;
//code to create and fill list
int total = 0;
for( int x : list )
    total += x;
```

Iterable

- If you simply want to go through all the elements of a Collection (or Iterable thing) use the for each loop
  - hides creation of the Iterator

```
public void printAllOfLength(ArrayList<String> names,
                                              int len){
    //pre: names != null, names only contains Strings
    //post: print out all elements of names equal in
    // length to len
   for(String s : names){
      if( s.length() == len )
            System.out.println( s );
   }
}
```


Implementing an Iterator

- Implement an Iterator for our GenericList class
  - Nested Classes
  - Inner Classes
  - Example of encapsulation
  - checking precondition on remove
  - does our GenricList *need* an Iterator?

Topic 14
Linked Lists

"All the kids who did great in high school writing pong games in BASIC for their Apple II would get to college, take CompSci 101, a data structures course, and when they hit the pointers business their brains would just totally explode, and the next thing you knew, they were majoring in Political Science because law school seemed like a better idea."

-Joel Spolsky

Thanks to Don Slater of CMU for use of his slides.

Attendance Question 1

- What is output by the following code?

```
ArrayList<Integer> a1 = new ArrayList<Integer>();
ArrayList<Integer> a2 = new ArrayList<Integer>();
a1.add(12);
a2.add(12);
System.out.println( a1 == a2 );
```

A. No output due to syntax error

B. No output due to runtime error

C. `false`

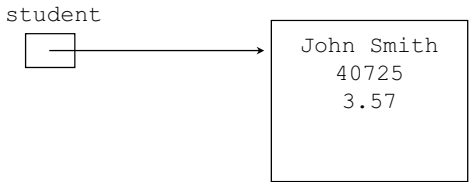
D. `true`

Dynamic Data Structures

- *Dynamic* data structures
  - They grow and shrink one element at a time, normally without some of the inefficiencies of arrays
  - as opposed to a static container like an array
- Big O of Array Manipulations
  - Access the kth element
  - Add or delete an element in the middle of the array while maintaining relative order
  - adding element at the end of array? space avail? no space avail?
  - add element at beginning of an array

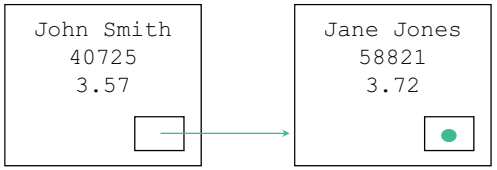
Object References

- Recall that an *object reference* is a variable that stores the address of an object
- A reference can also be called a *pointer*
- They are often depicted graphically:



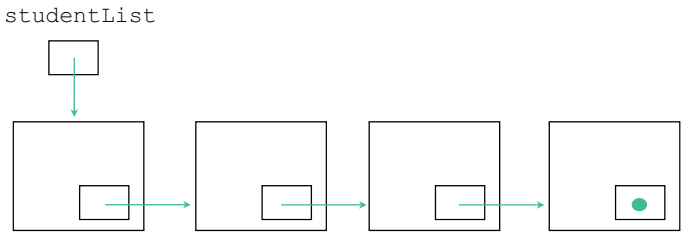
References as Links

- Object references can be used to create *links* between objects
- Suppose a `Student` class contained a reference to another `Student` object



References as Links

- References can be used to create a variety of linked structures, such as a *linked list*:



Linked Lists

- A **linear** collection of self-referential objects, called nodes, connected by other links
  - linear: **for every node in the list, there is one and only one node that precedes it (except for possibly the first node, which may have no predecessor,) and there is one and only one node that succeeds it, (except for possibly the last node, which may have no successor)**
  - self-referential: **a node that has the ability to refer to another node of the same type, or even to refer to itself**
  - node: **contains data of any type, including a reference to another node of the same data type, or to nodes of different data types**
  - **Usually a list will have a beginning and an end; the first element in the list is accessed by a reference to that class, and the last node in the list will have a reference that is set to `null`**

Advantages of linked lists

- Linked lists are dynamic, they can grow or shrink as necessary
- Linked lists can be maintained in sorted order simply by inserting each new element at the proper point in the list. Existing list elements do not need to be moved
- Linked lists are *non-contiguous;* the logical sequence of items in the structure is decoupled from any physical ordering in memory

Nodes and Lists

- A different way of implementing a list
- Each element of a Linked List is a separate Node object.
- Each Node tracks a single piece of data plus a reference (pointer) to the next
- Create a new Node very time we add something to the List
- Remove nodes when item removed from list and allow garbage collector to reclaim that memory

A Node Class

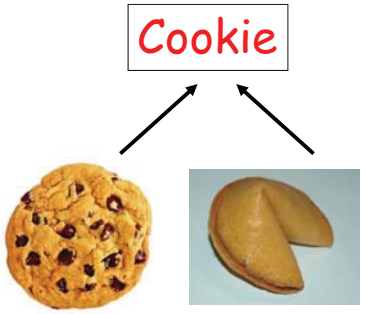
```
public class Node<E> {
        private E myData;
        private Node myNext;

        public Node()
        {       myData = null; myNext = null;     }

        public Node(E data, Node<E> next)
        {       myData = data; myNext = next; }

        public E getData()
        {       return myData;       }

        public Node<E> getNext()
        {       return myNext;       }

        public void setData(Et data)
        {       myData = data;       }

        public void setNext(Node<E> next)
        {       myNext = next;       }
}
```


One Implementation of a Linked List

- The Nodes show on the previous slide are *singly linked*
  - a node refers only to the next node in the structure
  - it is also possible to have *doubly linked* nodes.
  - The node has a reference to the next node in the structure and the *previous* node in the structure as well
- How is the end of the list indicated
  - myNext = null for last node
  - a separate dummy node class / object

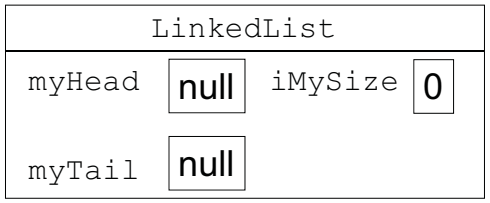
Interfaces and Standard Java

- Finally, an alternate implementation to an ADT
- Specify a List interface
  - Java has this
- Implement in multiple ways
  - ArrayList
  - LinkedList
- Which is better?



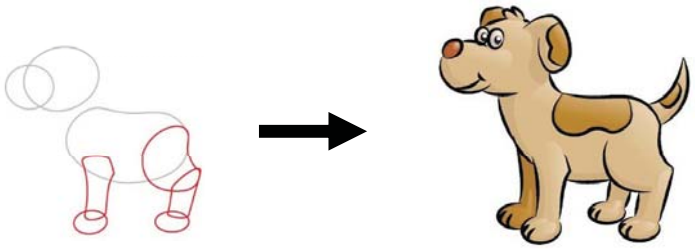
A Linked List Implementation

```
public class LinkedList<E> implements Ilist<E>
      private Node<E> head;
      private Node<E> tail;
      private int size;

      public LinkedList(){
            head = null;
            tail = null;
            size = 0;
      }
}
LinkedList<String> list = new LinkedList<String>();
```



Writing Methods

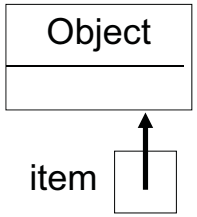
- When trying to code methods for Linked Lists ***draw pictures!***
  - If you don't draw pictures of what you are trying to do it is very easy to make mistakes!



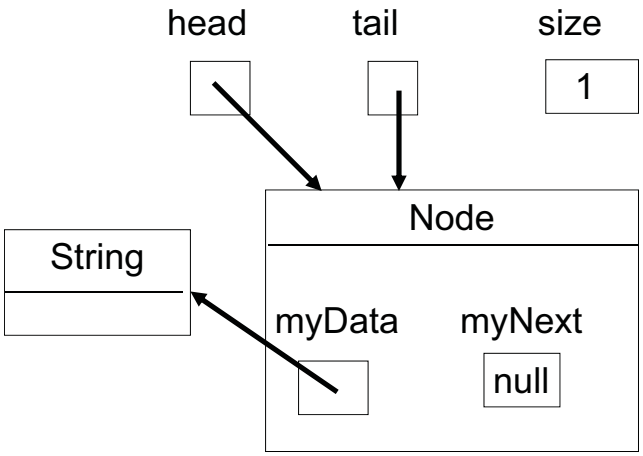
add method

- add to the end of list
- special case if empty
- steps on following slides
- public void add(Object obj)

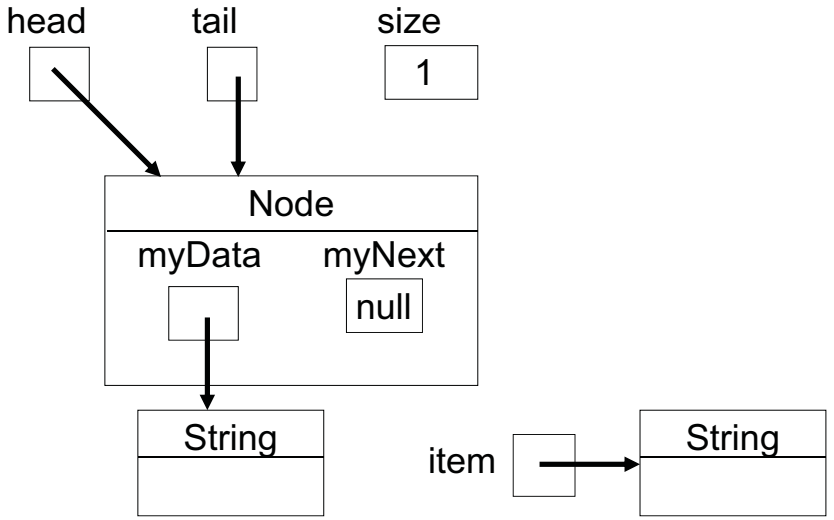
Add Element - List Empty (Before)



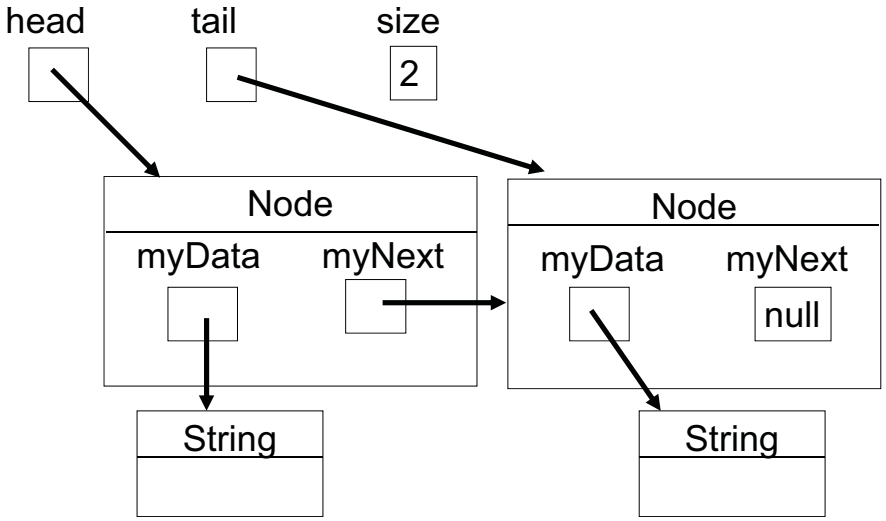
Add Element - List Empty (After)



Add Element - List Not Empty (Before)



Add Element - List Not Empty (After)



Code for default add

- public void add(Object obj)

Attendance Question 2

- What is the worst case Big O for adding to the end of an array based list and a linked list? The lists already contains N items.

	Array based	Linked
A.	O(1)	O(1)
B.	O(N)	O(N)
C.	O(logN)	O(1)
D.	O(1)	O(N)
E.	O(N)	O(1)

Code for addFront

- add to front of list
- public void addFront(Object obj)
- How does this compare to adding at the front of an array based list?

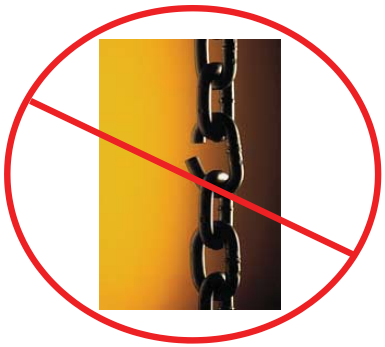
Attendance Question 3

- What is the Big O for adding to the front of an array based list and a linked list? The lists already contains N items.

	Array based	Linked
A.	O(1)	O(1)
B.	O(N)	O(1)
C.	O(logN)	O(1)
D.	O(1)	O(N)
E.	O(N)	O(N)

Code for Insert

- public void insert(int pos, Object obj)
- Must be careful not to break the chain!
- Where do we need to go?
- Special cases?



Attendance Question 4

- What is the Big O for inserting an element into the middle of an array based list and a linked list? The lists contains N items.

	Array based	Linked
A.	O(N)	O(N)
B.	O(N)	O(1)
C.	O(logN)	O(1)
D.	O(logN)	O(logN))
E.	O(1)	O(N)

Attendance Question 5

- What is the Big O for getting an element based on position from an array based list and a linked list? The lists contain N items.

	Array based	Linked
A.	O(1)	O(N)
B.	O(N)	O(1)
C.	O(logN)	O(1)
D.	O(logN)	O(N)
E.	O(N)	O(N)

Code for get

- public Object get(int pos)
- The downside of Linked Lists

Code for remove

- public Object remove(int pos)

Why Use Linked List

- What operations with a Linked List faster than the version from ArrayList?

Remove Back Method

- public Object removeBack()

- Big O?

Iterators for Linked Lists

- What is the Big O of the following code?

```
LinkedList<Integer> list;
list = new LinkedList<Integer>();
// code to fill list with N elements

//Big O of following code?
for(int i = 0; i < list.size(); i++)
   System.out.println( list.get(i) );
```


Attendance Question 6

- What is the Big O of the code on the previous slide?

A. O(N)

B. $O(2^N)$

C. O(NlogN)

D. $O(N^2)$

E. $O(N^3)$

Other Possible Features of Linked Lists

- Doubly Linked
- Circular
- Dummy Nodes for first and last node in list

```
public class DLNode<E> {
       private E myData;
       private DLNode<E> myNext;
       private DLNode<E> myPrevious;

}
```


Dummy Nodes

- Use of Dummy Nodes for a Doubly Linked List removes most special cases
- Also could make the Double Linked List circular

Doubly Linked List addFront

- public void addFront(Object obj)

Insert for Doubly Linked List

- public void insert(int pos, Object obj)

Topic 15
Implementing and Using Stacks

"stack n.
The set of things a person has to do in the future. "I haven't done it yet because every time I pop my stack something new gets pushed." If you are interrupted several times in the middle of a conversation, "My stack overflowed" means "I forget what we were talking about."

-The Hacker's Dictionary

Friedrich L. Bauer
German computer scientist who proposed "stack method of expression evaluation" in 1955.

Stack Overflow

Sharper Tools

Lists

Stacks

Stacks

- Access is allowed only at one point of the structure, normally termed the *top* of the stack
  - access to the most recently added item only
- Operations are limited:
  - push (add item to stack)
  - pop (remove top item from stack)
  - top (get top item without removing it)
  - clear
  - isEmpty
  - size?
- Described as a "Last In First Out" (LIFO) data structure

Stack Operations

Assume a simple stack for integers.

Stack s = new Stack();

s.push(12);

s.push(4);

s.push(s.top() + 2);

s.pop()

s.push(s.top());

//what are contents of stack?

Stack Operations

Write a method to print out contents of stack in reverse order.

Common Stack Error

```
Stack s = new Stack();
// put stuff in stack
for(int i = 0; i < 5; i++)
    s.push( i );
// print out contents of stack
// while emptying it. (??)
for(int i = 0; i < s.size(); i++)
    System.out.print( s.pop() + " ");

// What is output?
```


Attendance Question 1

- What is output of code on previous slide?

A `0 1 2 3 4`

B `4 3 2 1 0`

C `4 3 2`

D `2 3 4`

E No output due to runtime error.

Corrected Version

```
Stack s = new Stack();
// put stuff in stack
for(int i = 0; i < 5; i++)
    s.push( i );
// print out contents of stack
// while emptying it
int limit = s.size();
for(int i = 0; i < limit; i++)
    System.out.print( s.pop() + " ");
//or
//   while( !s.isEmpty() )
//       System.out.println( s.pop() );
```

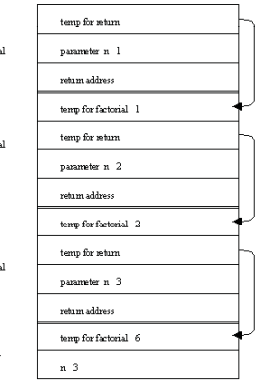

Implementing a stack

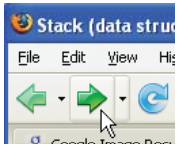
- need an underlying collection to hold the elements of the stack
- 2 basic choices
  - array (native or ArrayList)
  - linked list
- array implementation
- linked list implementation
- Some of the uses for a stack are much more interesting than the implementation of a stack

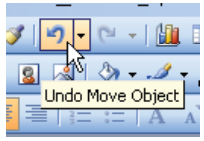
Applications of Stacks

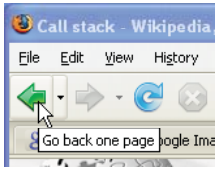
Problems that Use Stacks

- The runtime stack used by a process (running program) to keep track of methods in progress
- Search problems
- Undo, redo, back, forward









Mathematical Calculations

What is 3 + 2 * 4? 2 * 4 + 3? 3 * 2 + 4?

The precedence of operators affects the order of operations. A mathematical expression cannot simply be evaluated left to right.

A challenge when evaluating a program.

Lexical analysis is the process of interpreting a program.

Involves Tokenization

What about 1 - 2 - 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 3

Infix and Postfix Expressions

- The way we are use to writing expressions is known as infix notation
- Postfix expression does not
- require any precedence rules
- 3 2 * 1 + is postfix of 3 * 2 + 1
- evaluate the following postfix expressions and write out a corresponding infix expression:

2 3 2 4 * + * 1 2 3 4 ^ * +

1 2 - 3 2 ^ 3 * 6 / + 2 5 ^ 1 -

Attendance Question 2

- What does the following postfix expression evaluate to?

  6 3 2 + *

A. 18

B. 36

C. 24

D. 11

E. 30

Evaluation of Postfix Expressions

- Easy to do with a stack
- given a proper postfix expression:
  - get the next token
  - if it is an operand push it onto the stack
  - else if it is an operator
    - pop the stack for the right hand operand
    - pop the stack for the left hand operand
    - apply the operator to the two operands
    - push the result onto the stack
  - when the expression has been exhausted the result is the top (and only element) of the stack

Infix to Postfix

- Convert the following equations from infix to postfix:

  2 ^ 3 ^ 3 + 5 * 1

  11 + 2 - 1 * 3 / 3 + 2 ^ 2 / 3

  Problems:

  Negative numbers?

  parentheses in expression

Infix to Postfix Conversion

- Requires operator precedence parsing algorithm
  - parse v. To determine the syntactic structure of a sentence or other utterance

Operands: add to expression

Close parenthesis: pop stack symbols until an open parenthesis appears

Operators:

Have an on stack and off stack precedence

Pop all stack symbols until a symbol of lower precedence appears. Then push the operator

End of input: Pop all remaining stack symbols and add to the expression

Simple Example

Infix Expression: 3 + 2 * 4

PostFix Expression:

Operator Stack:

Precedence Table

Symbol	Off Stack Precedence	On Stack Precedence
+	1	1
-	1	1
*	2	2
/	2	2
^	10	9
(	20	0

Simple Example

Infix Expression: + 2 * 4

PostFix Expression: 3

Operator Stack:

Precedence Table

Symbol	Off Stack Precedence	On Stack Precedence
+	1	1
-	1	1
*	2	2
/	2	2
^	10	9
(	20	0

Simple Example

Infix Expression: 2 * 4

PostFix Expression: 3

Operator Stack: +

Precedence Table

Symbol	Off Stack Precedence	On Stack Precedence
+	1	1
-	1	1
*	2	2
/	2	2
^	10	9
(	20	0

Simple Example

Infix Expression: * 4

PostFix Expression: 3 2

Operator Stack: +

Precedence Table

Symbol	Off Stack Precedence	On Stack Precedence
+	1	1
-	1	1
*	2	2
/	2	2
^	10	9
(	20	0

Simple Example

Infix Expression: 4

PostFix Expression: 3 2

Operator Stack: + *

Precedence Table

Symbol	Off Stack Precedence	On Stack Precedence
+	1	1
-	1	1
*	2	2
/	2	2
^	10	9
(	20	0

Simple Example

Infix Expression:

PostFix Expression: 3 2 4

Operator Stack: + *

Precedence Table

Symbol	Off Stack Precedence	On Stack Precedence
+	1	1
-	1	1
*	2	2
/	2	2
^	10	9
(	20	0

Simple Example

Infix Expression:

PostFix Expression: 3 2 4 *

Operator Stack: +

Precedence Table

Symbol	Off Stack Precedence	On Stack Precedence
+	1	1
-	1	1
*	2	2
/	2	2
^	10	9
(	20	0

Simple Example

Infix Expression:

PostFix Expression: 3 2 4 * +

Operator Stack:

Precedence Table

Symbol	Off Stack Precedence	On Stack Precedence
+	1	1
-	1	1
*	2	2
/	2	2
^	10	9
(	20	0

Example

1 - 2 ^ 3 ^ 3 - (4 + 5 * 6) * 7

Show algorithm in action on above equation

Balanced Symbol Checking

- In processing programs and working with computer languages there are many instances when symbols must be balanced

  { } , [ ] , ( )

  A stack is useful for checking symbol balance. When a closing symbol is found it must match the most recent opening symbol of the same type.

Algorithm?

Algorithm for Balanced Symbol Checking

- Make an empty stack
- read symbols until end of file
  - if the symbol is an opening symbol push it onto the stack
  - if it is a closing symbol do the following
    - if the stack is empty report an error
    - otherwise pop the stack. If the symbol popped does not match the closing symbol report an error
- At the end of the file if the stack is not empty report an error

Algorithm in practice

- list[i] = 3 * (44 - method(foo(list[2 * (i + 1) + foo(list[i - 1])) / 2 *) - list[method(list[0])];
- Complications
  - when is it not an error to have non matching symbols?
- Processing a file
  - *Tokenization*: the process of scanning an input stream. Each independent chunk is a token.
- Tokens may be made up of 1 or more characters

Topic 16
Queues

> **"FISH queue**: n.
>
> [acronym, by analogy with FIFO (First In, First Out)] 'First In, Still Here'. A joking way of pointing out that processing of a particular sequence of events or requests has stopped dead. Also FISH mode and FISHnet; the latter may be applied to any network that is running really slowly or exhibiting extreme flakiness."
>
> -The Jargon File 4.4.7

Queues

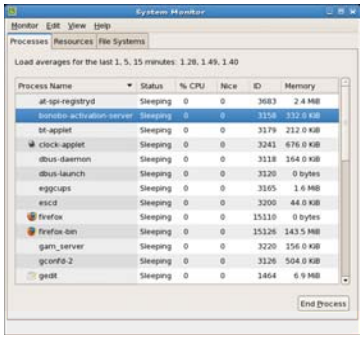
- Similar to Stacks
- Like a line
  - In Britain people don't "get in line" they "queue up".

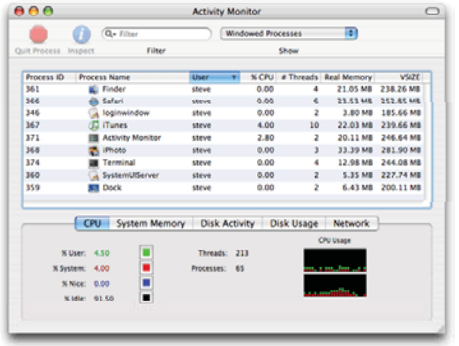
Queue Properties

- Queues are a first in first out data structure
  - FIFO (or LILO, but that sounds a bit silly)
- Add items to the end of the queue
- Access and remove from the front
  - Access to the element that has been in the structure the ***longest*** amount of time
- Used extensively in operating systems
  - Queues of processes, I/O requests, and much more

Queues in Operating Systems

- On a computer with 1 CPU, but many processes how many processes can actually use the CPU at a time?
- One job of OS, schedule the processes for the CPU
- issues: fairness, responsiveness, progress





Queue operations

- `add(Object item)`
  - a.k.a. `enqueue(Object item)`
- `Object get()`
  - a.k.a. `Object front(), Object peek()`
- `Object remove()`
  - a.k.a. `Object dequeue()`
- `boolean isEmpty()`
- Specify in an interface, allow varied implementations

Queue interface, version 1

```
public interface Queue
{    //place item at back of this Queue
     enqueue(Object item);

     //access item at front of this queue
     //pre: !isEmpty()
     Object front();

     //remove item at front of this queue
     //pre: !isEmpty()
     Object dequeue();

     boolean isEmpty();
}
```


Implementing a Queue

- Given the internal storage container and choice for front and back of queue what are the Big O of the queue operations?

	ArrayList	LinkedList (Singly Linked)	LinkeList (Doubly Linked)
enqueue			
front			
dequeue			
isEmpty			

Attendance Question 1

- If implementing a queue with a singly linked list with references to the first and last nodes (head and tail) which end of the list should be the front of the queue in order to have all queue operations O(1)?

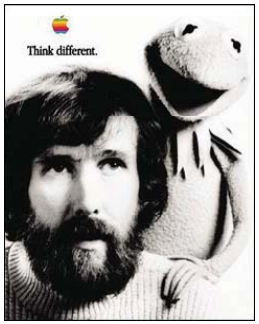
A. The front of the list should be the front of the queue

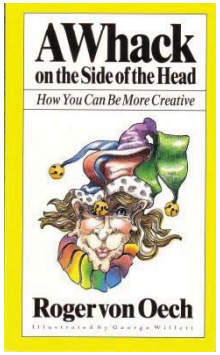
B. The back of the list should be the front of the queue.

C. D. E. I don't know, but I am sure looking forward to taking 307 again some time.

Alternate Implementation

- How about implementing a Queue with a native array?
  - Seems like a step backwards





Application of Queues

- Radix Sort
  - radix is a synonym for *base.* base 10, base 2
- Multi pass sorting algorithm that ***only*** looks at individual digits during each pass
- Use queues as *buckets* to store elements
- Create an array of 10 queues
- Starting with the least significant digit place value in queue that matches digit
- empty queues back into array
- repeat, moving to next least significant digit

Radix Sort in Action: 1s

- original values in array

  113, 70, 86, 12, 93, 37, 40, 252, 7, 79, 12
- Look at ones place

  113, 70, 86, 12, 93, 37, 40, 252, 7, 79, 12
- Queues:

0 70, 40	5
1	6 86
2 12, 252, 12	7 37, 7
3 113, 93	8
4	9 9, 79

Radix Sort in Action: 10s

- Empty queues in order from 0 to 9 back into array

  70, 40, 12, 252, 12, 113, 93, 86, 37, 7, 9, 79
- Now look at 10's place

  70, 40, 12, 252, 12, 113, 93, 86, 37, 7, 9, 79
- Queues:

0 7, 9	5 252
1 12, 12, 113	6
2	7 70, 79
3 37	8 86
4 40	9 93

Radix Sort in Action: 100s

- Empty queues in order from 0 to 9 back into array

  7, 9, 12, 12, 113, 37, 40, 252, 70, 79, 86, 93
- Now look at 100's place

  __7, __9, _12, _12, 113, _37, _40, 252, _70, _79, _86, _93
- Queues:

0	_7, _9, _12, _12, _40, _70, _79, _86, _93	5
1	113	6
2	252	7
3		8
4		9

Radix Sort in Action: Final Step

- Empty queues in order from 0 to 9 back into array

  7, 9, 12, 12, 40, 70, 79, 86, 93, 113, 252

Radix Sort Code

```
public static void sort(int[] list){
    ArrayList<Queue<Integer>> queues = new ArrayList<Queue<Integer>>();
    for(int i = 0; i < 10; i++)
        queues.add( new LinkedList<Integer>() );
    int passes = numDigits( list[0] );
    int temp;
    for(int i = 1; i < list.length; i++){
        temp = numDigits(list[i]);
        if( temp > passes )
            passes = temp;
    }
    for(int i = 0; i < passes; i++){
        for(int j = 0; j < list.length; j++){
            queues.get(valueOfDigit(list[j], i)).add(list[j]);
        }
        int pos = 0;
        for(Queue<Integer> q : queues){
            while( !q.isEmpty())
                list[pos++] = q.remove();
        }
    }
}
```

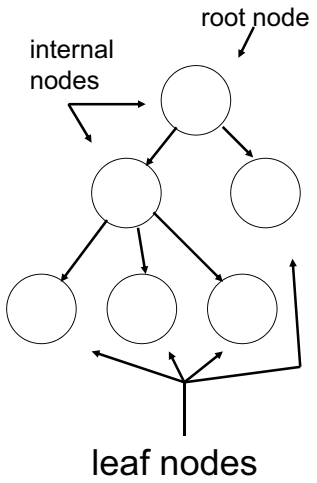
Topic 17
Introduction to Trees

"A tree may grow a thousand feet tall, but its leaves will return to its roots."

-Chinese Proverb

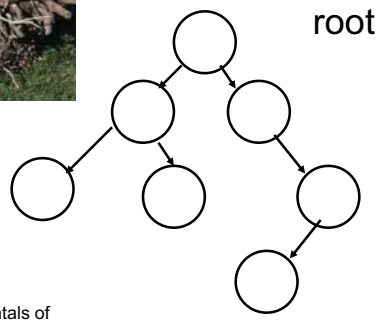
Definitions

- A *tree* is an abstract data type
  - one entry point, the ***root***
  - Each node is either a ***leaf*** or an *internal node*
  - An internal node has 1 or more ***children***, nodes that can be reached directly from that internal node.
  - The internal node is said to be the ***parent*** of its child nodes



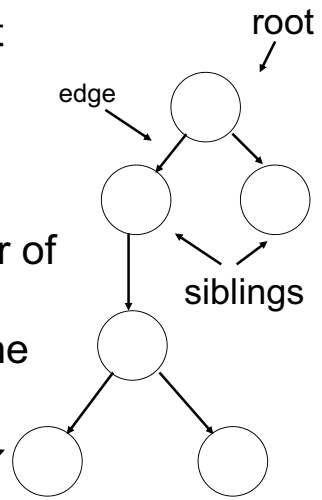
Properties of Trees

- Only access point is the root
- All nodes, except the root, have one parent
  - like the inheritance hierarchy in Java
- Traditionally trees drawn upside down



Properties of Trees and Nodes

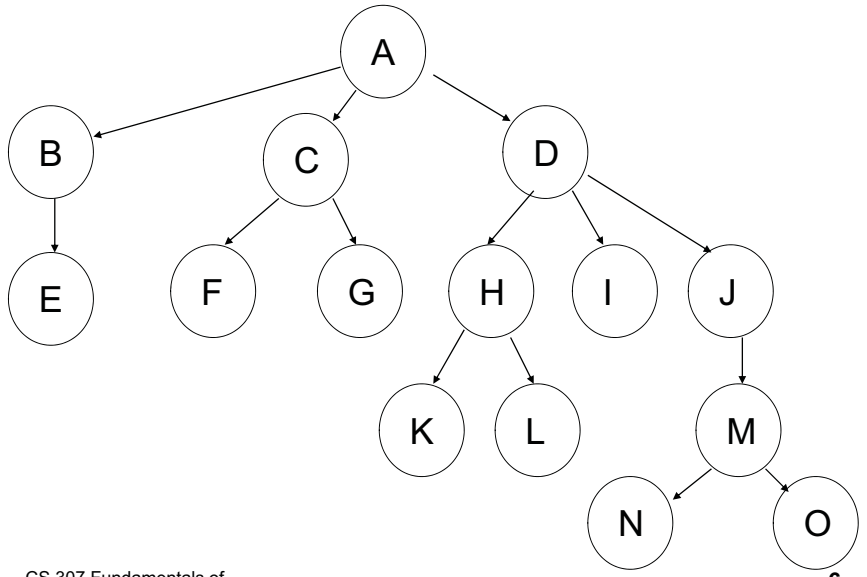
- *siblings:* two nodes that have the same parent
- *edge:* the link from one node to another
- *path length:* the number of edges that must be traversed to get from one node to another



More Properties of Trees

- *depth:* the path length from the root of the tree to this node
- *height of a node:* The maximum distance (path length) of any leaf from this node
  - a leaf has a height of 0
  - the height of a tree is the height of the root of that tree
- *descendants*: any nodes that can be reached via 1 or more edges from this node
- *ancestors:* any nodes for which this node is a descendant

Tree Visualization



Attendance Question 1

- What is the depth of the node that contains M on the previous slide?

A. -1

B. 0

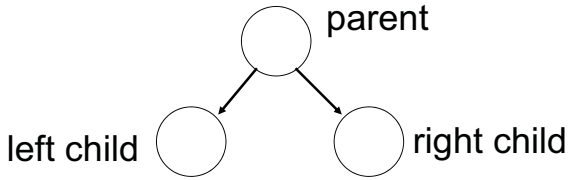
C. 1

D. 2

E. 3

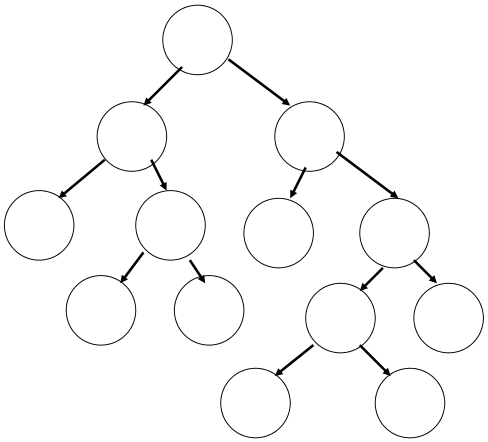
Binary Trees

- There are many variations on trees but we will work with *binary trees*
- *binary tree:* a tree with at most two children for each node
  - the possible children are normally referred to as the left and right child



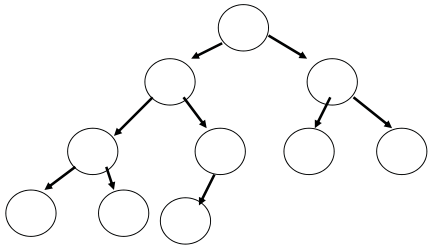
Full Binary Tree

- *full binary tree:* a binary tree is which each node was exactly 2 or 0 children



Complete Binary Tree

- *complete binary tree:* a binary tree in which every level, except possibly the deepest is completely filled. At depth n, the height of the tree, all nodes are as far left as possible



Where would the next node go to maintain a complete tree?

Perfect Binary Tree

- *perfect binary tree:* a binary tree with all leaf nodes at the same depth. All internal nodes have exactly two children.
- a perfect binary tree has the maximum number of nodes for a given height
- a perfect binary tree has $2^{(n+1)} - 1$ nodes where n is the height of a tree
  - height = 0 -> 1 node
  - height = 1 -> 3 nodes
  - height = 2 -> 7 nodes
  - height = 3 -> 15 nodes

A Binary Node class

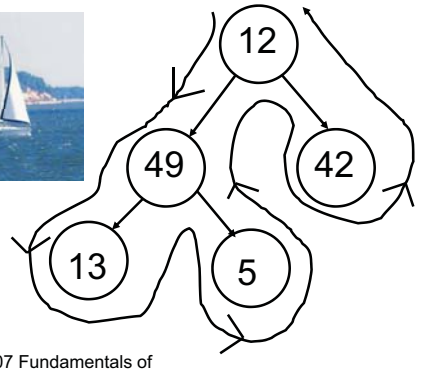
```
public class BNode
{    private Object myData;
     private BNode myLeft;
     private BNode myRight;

     public BNode();
     public BNode(Object data, BNode left,
          BNode right)
     public Object getData()
     public BNode getLeft()
     public BNode getRight()

     public void setData(Object data)
     public void setLeft(BNode left)
     public void setRight(BNode right)
}
```

Binary Tree Traversals

- Many algorithms require all nodes of a binary tree be visited and the contents of each node processed.
- There are 4 traditional types of traversals
  - preorder traversal: process the root, then process all sub trees (left to right)
  - in order traversal: process the left sub tree, process the root, process the right sub tree
  - post order traversal: process the left sub tree, process the right sub tree, then process the root
  - level order traversal: starting from the root of a tree, process all nodes at the same depth from left to right, then proceed to the nodes at the next depth.

Results of Traversals

- To determine the results of a traversal on a given tree draw a path around the tree.
  - start on the left side of the root and trace around the tree. The path should stay close to the tree.

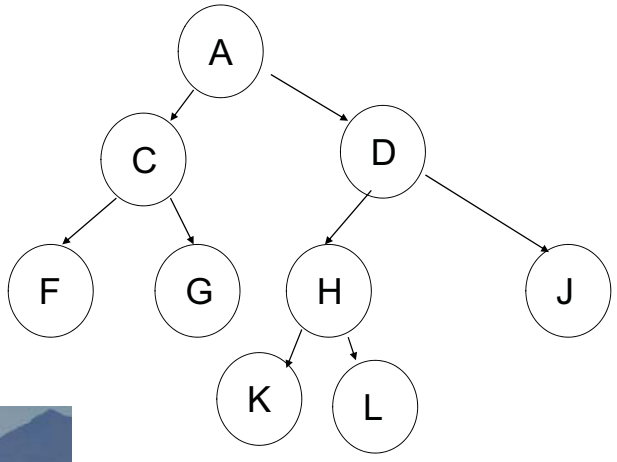


pre order: process when pass down left side of node
12 49 13 5 42

in order: process when pass underneath node
13 49 5 12 42

post order: process when pass up right side of node
13 5 49 42 12

Tree Traversals



Attendance Question 2

- What is a the result of a post order traversal of the tree on the previous slide?

A. F C G A K H L D J
B. F G C K L H J D A
C. A C F G D H K L J
D. A C D F G H J K L
E. L K J H G F D C A

Implement Traversals

- Implement preorder, inorder, and post order traversal
  - Big O time and space?
- Implement a level order traversal using a queue
  - Big O time and space?
- Implement a level order traversal without a queue
  - target depth
- Different kinds of Iterators for traversals?

Topic 18
Binary Search Trees

"Yes. Shrubberies are my trade. I am a shrubber. My name is 'Roger the Shrubber'. I arrange, design, and sell shrubberies."

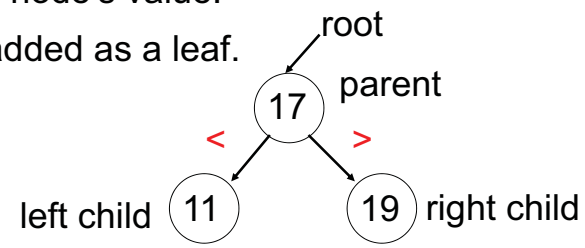
-Monty Python and The Holy Grail

The Problem with Linked Lists

- Accessing a item from a linked list takes O(N) time for an arbitrary element
- Binary trees can improve upon this and reduce access to O(log N) time for the average case
- Expands on the binary search technique and allows insertions and deletions
- Worst case degenerates to O(N) but this can be avoided by using balanced trees (AVL, Red-Black)

Binary Search Trees

- A binary tree is a tree where each node has at most two children, referred to as the left and right child
- A binary search tree is a binary tree in which every node's left subtree holds values less than the node's value, and every right subtree holds values greater than the node's value.
- A new node is added as a leaf.



Attendance Question 1

- After adding N distinct elements in random order to a Binary Search Tree what is the expected height of the tree?

A. $O(N^{1/2})$
B. O(logN)
C. O(N)
D. O(NlogN)
E. $O(N^2)$

Implementation of Binary Node

```
public class BSTNode
{     private Comparable myData;
      private BSTNode myLeft;
      private BSTNode myRightC;

      public BinaryNode(Comparable item)
      {     myData = item;    }

      public Object getValue()
      {     return myData;    }

      public BinaryNode getLeft()
      {     return myLeft;    }

      public BinaryNode getRight()
      {     return myRight;   }

      public void setLeft(BSTNode b)
      {     myLeft = b; }
      // setRight not shown
}
```


Sample Insertion

- 100, 164, 130, 189, 244, 42, 141, 231, 20, 153 (from HotBits: www.fourmilab.ch/hotbits/)

If you insert 1000 random numbers into a BST using the naïve algorithm what is the expected height of the tree? (Number of links from root to deepest leaf.)

Worst Case Performance

- In the worst case a BST can degenerate into a singly linked list.
- Performance goes to O(N)
- 2 3 5 7 11 13 17

More on Implementation

- Many ways to implement BSTs
- Using nodes is just one and even then many options and choices

```
public class BinarySearchTree
{     private TreeNode root;
      private int size;

      public BinarySearchTree()
      {     root = null;
            size = 0;
      }
```

Add an Element, Recursive

Add an Element, Iterative

Attendance Question 2

- What is the best case and worst case Big O to add N elements to a binary search tree?

	Best	Worst
A.	O(N)	O(N)
B.	O(NlogN)	O(NlogN)
C.	O(N)	O(NlogN)
D.	O(NlogN)	$O(N^2)$
E.	$O(N^2)$	$O(N^2)$

Performance of Binary Trees

- For the three core operations (add, access, remove) a binary search tree (BST) has an average case performance of O(log N)
- Even when using the *naïve insertion / removal algorithms*
- no checks to maintain balance
- balance achieved based on the randomness of the data inserted

Remove an Element

- Three cases
  - node is a leaf, 0 children (easy)
  - node has 1 child (easy)
  - node has 2 children (interesting)

Properties of a BST

- The minimum value is in the left most node
- The maximum value is in the right most node
  - useful when removing an element from the BST
- An *inorder traversal* of a BST provides the elements of the BST in ascending order

Using Polymorphism

- Examples of dynamic data structures have relied on *null terminated ends.*
  - Use null to show end of list, no children
- Alternative form
  - use structural recursion and polymorphism

BST Interface

```
public interface BST {
    public int size();
    public boolean contains(Comparable obj);
    public boolean add(Comparable obj);
}
```

EmptyBST

```
public class EmptyBST implements BST {

   private static EmptyBST theOne = new EmptyBST();

   private EmptyBST(){}

   public static EmptyBST getEmptyBST(){ return theOne; }

   public NEBST add(Comparable obj) { return new NEBST(obj); }

   public boolean contains(Comparable obj) { return false; }

   public int size() { return 0; }
}
```


Non Empty BST – Part 1

```
public class NEBST implements BST {

   private Comparable data;
   private BST left;
   private BST right;

   public NEBST(Comparable d){
      data = d;
      right = EmptyBST.getEmptyBST();
      left = EmptyBST.getEmptyBST();
   }

   public BST add(Comparable obj) {
      int val = obj.compareTo( data );
      if( val < 0 )
        left = left.add( obj );
      else if( val > 0 )
        right = right.add( obj );
      return this;
   }
```


Non Empty BST – Part 2

```
   public boolean contains(Comparable obj){
         int val = obj.compareTo(data);
         if( val == 0 )
            return true;
         else if (val < 0)
            return left.contains(obj);
         else
            return right.contains(obj);
      }

      public int size() {
         return 1 + left.size() + right.size();
      }

   }
```

Topic 19
Red Black Trees

"People in every direction
No words exchanged
No time to exchange
And all the little ants are marching
Red and black antennas waving"
-*Ants Marching,* Dave Matthew's Band

"Welcome to L.A.'s Automated Traffic Surveillance and Control Operations Center. See, they use video feeds from intersections and specifically designed algorithms to predict traffic conditions, and thereby control traffic lights. So all I did was come up with my own... kick ass algorithm to sneak in, and now we own the place."
-Lyle, the Napster, (Seth Green), *The Italian Job*

Attendance Question 1

- 2000 elements are inserted one at a time into an initially empty binary search tree using the traditional algorithm. What is the maximum possible height of the resulting tree?

A. 1
B. 11
C. 1000
D. 1999
E. 4000

Binary Search Trees

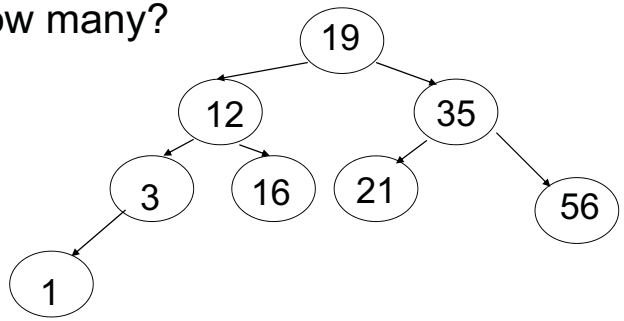
- Average case and worst case Big O for
  - insertion
  - deletion
  - access
- Balance is important. Unbalanced trees give worse than log N times for the basic tree operations
- Can balance be guaranteed?

Red Black Trees

- A BST with more complex algorithms to ensure balance
- Each node is labeled as Red or Black.
- Path: A unique series of links (edges) traverses from the root to each node.
  - The number of edges (links) that must be followed is the path length
- In Red Black trees paths from the root to elements with 0 or 1 child are of particular interest

Paths to Single or Zero Child Nodes

- How many?

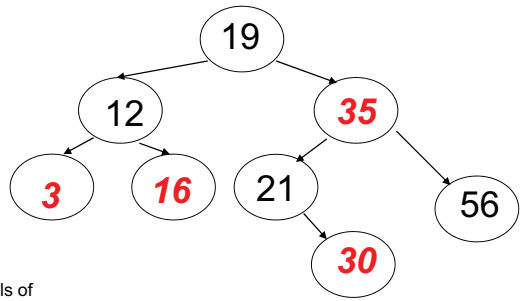


Red Black Tree Rules

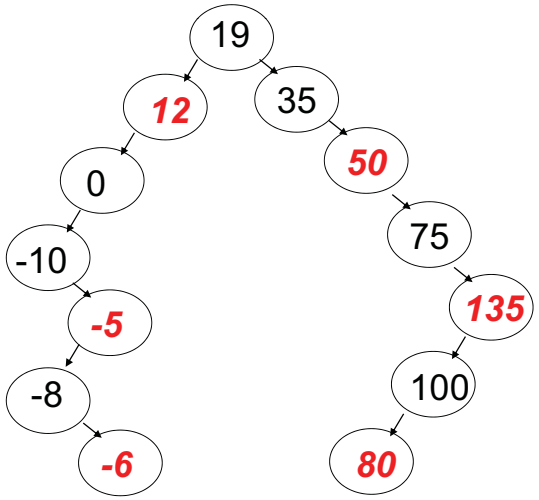
1. Every node is colored either Red or black
2. The root is black
3. If a node is red its children must be black. (a.k.a. the red rule)
4. Every path from a node to a null link must contain the same number of black nodes (a.k.a. the path rule)

Example of a Red Black Tree

- The root of a Red Black tree is black
- Every other node in the tree follows these rules:
  - Rule 3: If a node is Red, all of its children are Black
  - Rule 4: The number of Black nodes must be the same in all paths from the root node to null nodes



Red Black Tree?

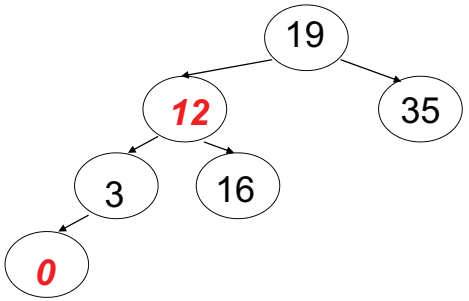


Attendance Question 2

- Is the tree on the previous slide a binary search tree? Is it a red black tree?

	BST?	Red-Black?
A.	No	No
B.	No	Yes
C.	Yes	No
D.	Yes	Yes

Red Black Tree?



Perfect?
Full?
Complete?

Attendance Question 3

- Is the tree on the previous slide a binary search tree? Is it a red black tree?

	BST?	Red-Black?
A.	No	No
B.	No	Yes
C.	Yes	No
D.	Yes	Yes

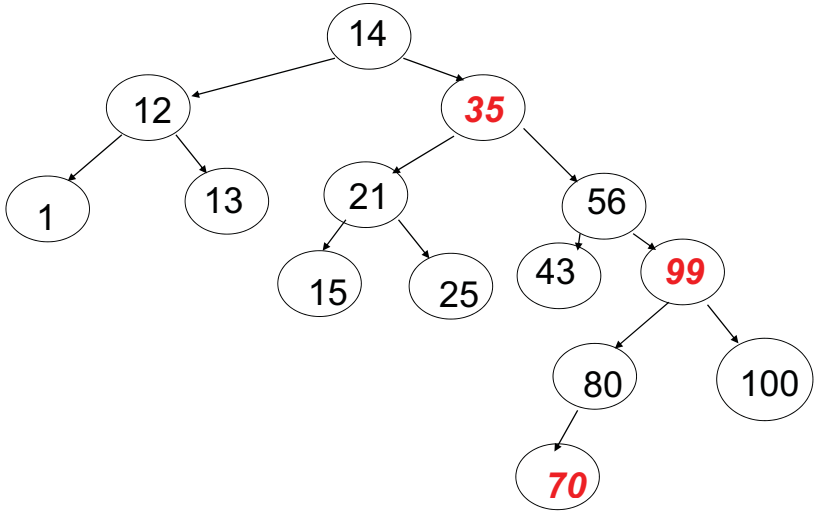
Implications of the Rules

- If a Red node has any children, it must have two children and they must be Black. (Why?)
- If a Black node has only one child that child must be a Red leaf. (Why?)
- Due to the rules there are limits on how unbalanced a Red Black tree may become.
  - on the previous example may we hang a new node off of the leaf node that contains 0?

Properties of Red Black Trees

- If a Red Black Tree is complete, with all Black nodes except for Red leaves at the lowest level the height will be minimal, ~log N
- To get the max height for N elements there should be as many Red nodes as possible down one path and all other nodes are Black
  - This means the max height would be < 2 * log N
  - see example on next slide

Max Height Red Black Tree

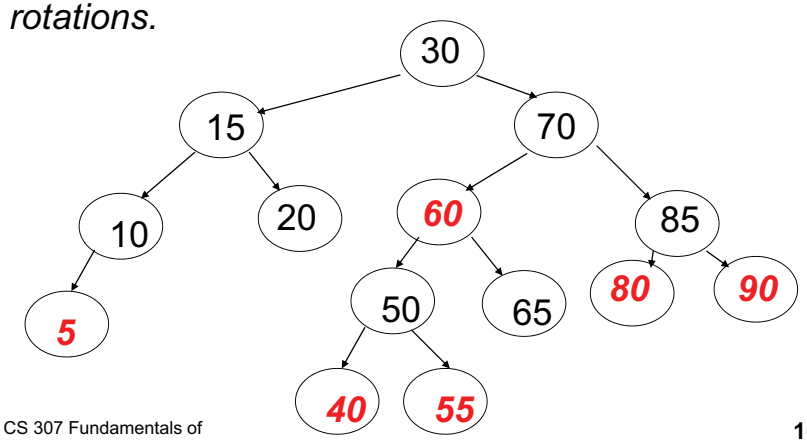


Maintaining the Red Black Properties in a Tree

- Insertions
- Must maintain rules of Red Black Tree.
- New Node always a leaf
  - can't be black or we will violate rule 4
  - therefore the new leaf must be red
  - If parent is black, done (trivial case)
  - if parent red, things get interesting because a red leaf with a red parent violates rule 3

Insertions with Red Parent - Child

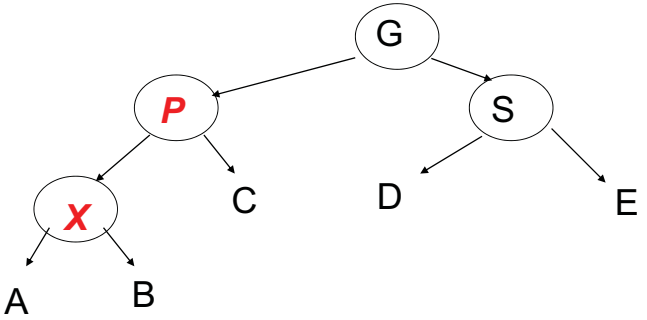
Must modify tree when insertion would result in Red Parent - Child pair using color changes and *rotations.*



Case 1

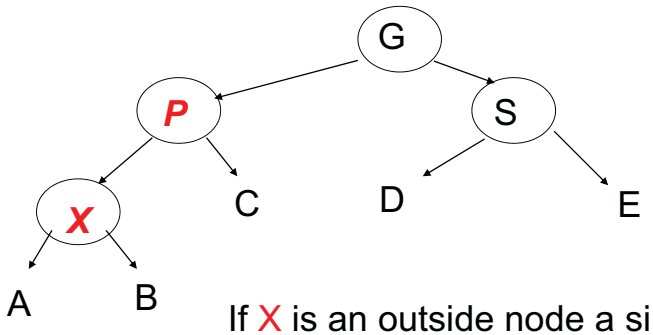
- Suppose sibling of parent is Black.
  - by convention null nodes are black
- In the previous tree, true if we are inserting a 3 or an 8.
  - What about inserting a 99? Same case?
- Let X be the new leaf Node, P be its Red Parent, S the Black sibling and G, P's and S's parent and X's grandparent
  - What color is G?

Case 1 - The Picture



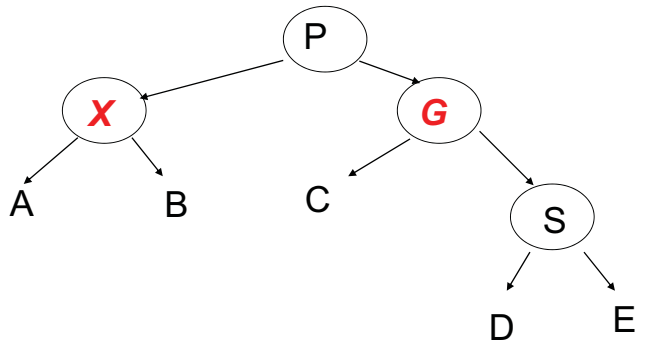
Relative to G, X could be an *inside* or *outside* node.
Outside -> left left or right right moves
Inside -> left right or right left moves

Fixing the Problem



If X is an outside node a single *rotation* between P and G fixes the problem.
A rotation is an exchange of roles between a parent and child node. So P becomes G's parent. Also must recolor P and G.

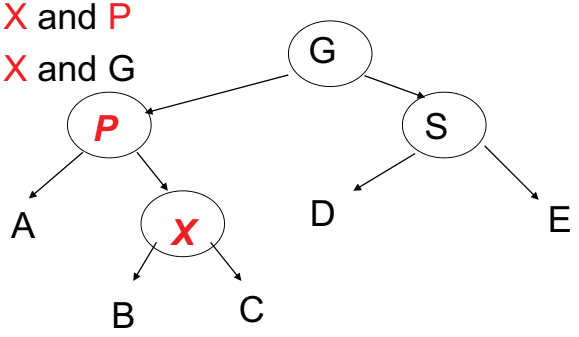
Single Rotation



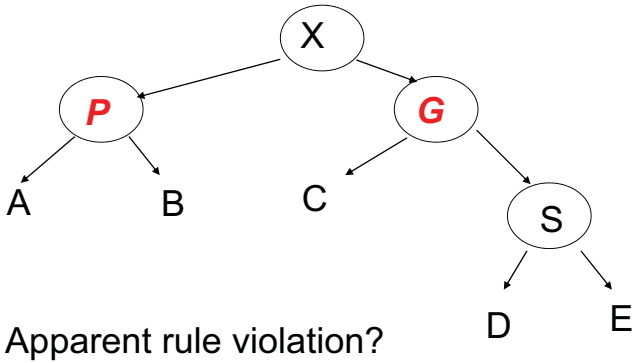
Apparent rule violation?

Case 2

- What if X is an inside node relative to G?
  - a single rotation will not work
- Must perform a double rotation
  - rotate X and P
  - rotate X and G

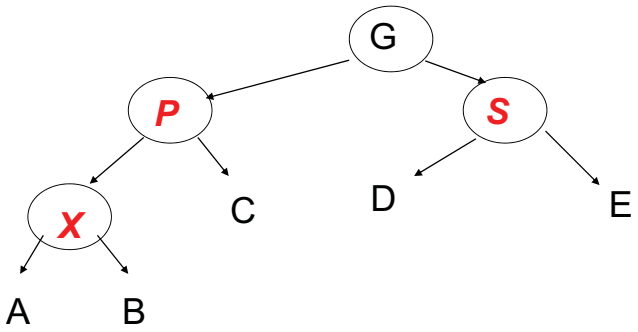


After Double Rotation



Apparent rule violation?

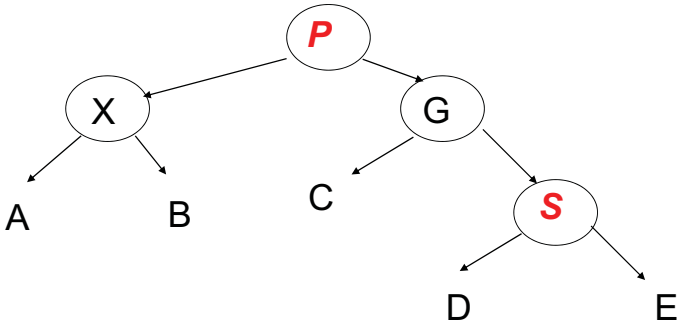
Case 3
Sibling is Red, not Black



Any problems?

Fixing Tree when S is Red

- Must perform single rotation between parent, P and grandparent, G, and then make appropriate color changes

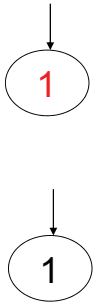


More on Insert

- Problem: What if on the previous example G's parent had been red?
- Easier to never let Case 3 ever occur!
- On the way down the tree, if we see a node X that has 2 Red children, we make X Red and its two children black.
  - if recolor the root, recolor it to black
  - the number of black nodes on paths below X remains unchanged
  - If X's parent was Red then we have introduced 2 consecutive Red nodes.(violation of rule)
  - to fix, apply rotations to the tree, same as inserting node

Example of Inserting Sorted Numbers

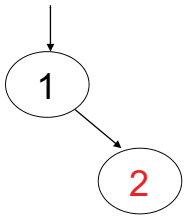
- 1 2 3 4 5 6 7 8 9 10



Insert 1. A leaf so red. Realize it is root so recolor to black.

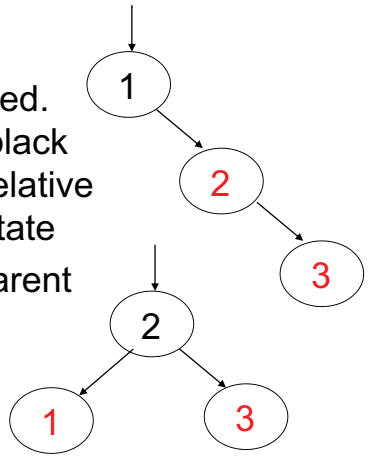
Insert 2

make 2 red. Parent is black so done.



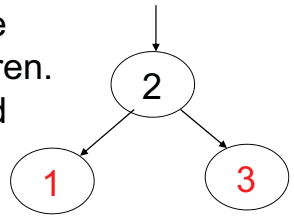
Insert 3

Insert 3. Parent is red. Parent's sibling is black (null) 3 is outside relative to grandparent. Rotate parent and grandparent

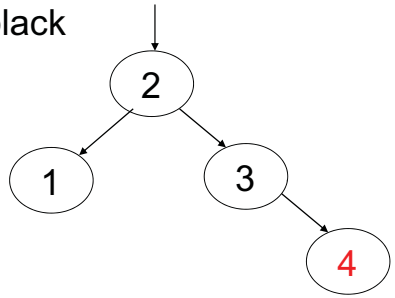


Insert 4

On way down see 2 with 2 red children. Recolor 2 red and children black. Realize 2 is root so color back to black

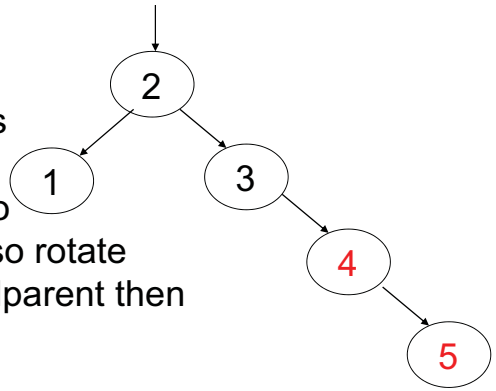


When adding 4 parent is black so done.

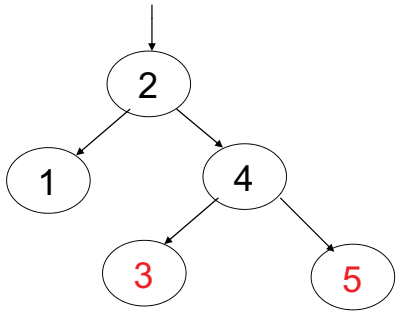


Insert 5

5's parent is red. Parent's sibling is black (null). 5 is outside relative to grandparent (3) so rotate parent and grandparent then recolor

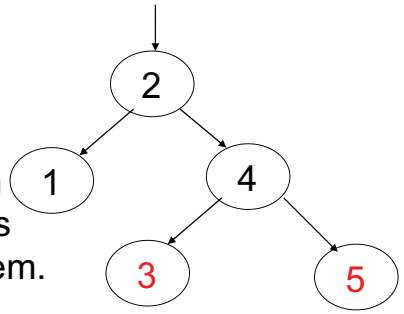


Finish insert of 5



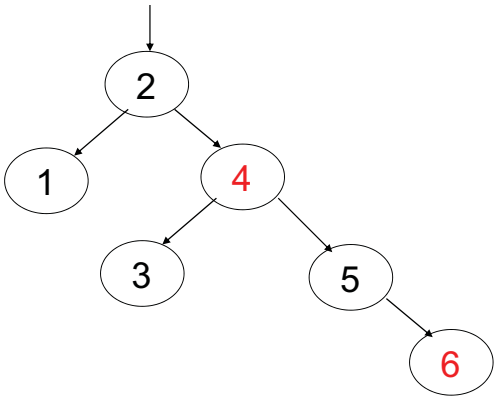
Insert 6

On way down see 4 with 2 red children. Make 4 red and children black. 4's parent is black so no problem.



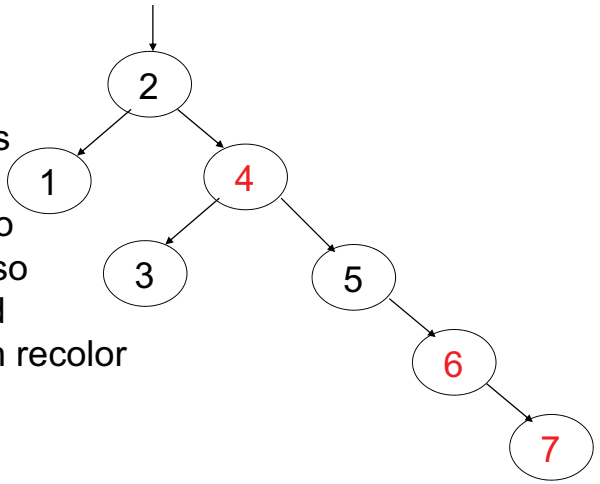
Finishing insert of 6

6's parent is black so done.

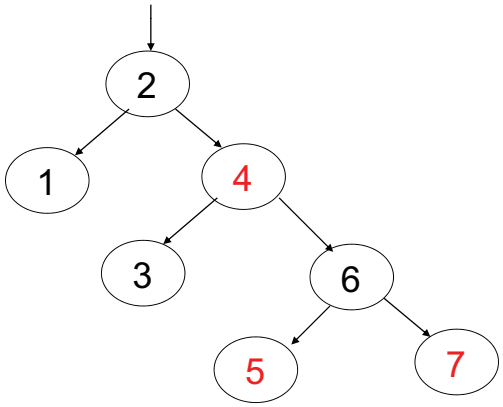


Insert 7

7's parent is red. Parent's sibling is black (null). 7 is outside relative to grandparent (5) so rotate parent and grandparent then recolor

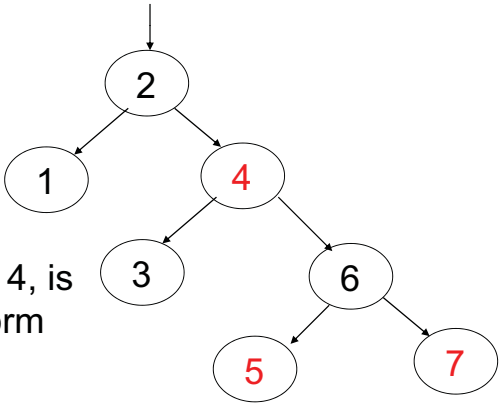


Finish insert of 7



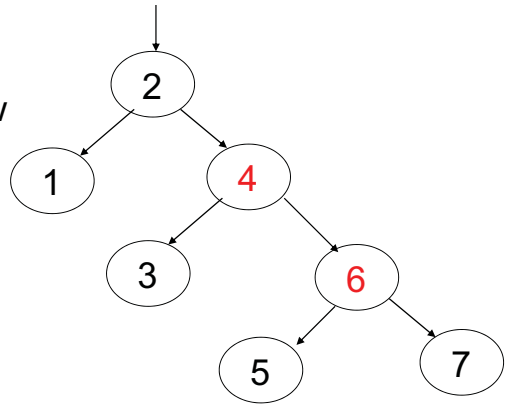
Insert 8

On way down see 6 with 2 red children. Make 6 red and children black. This creates a problem because 6's parent, 4, is also red. Must perform rotation.



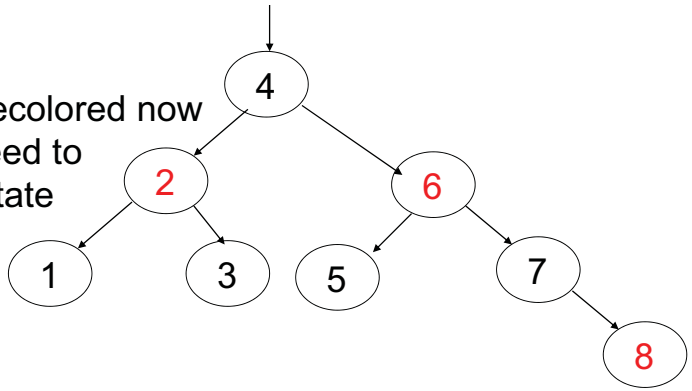
Still Inserting 8

Recolored now need to rotate

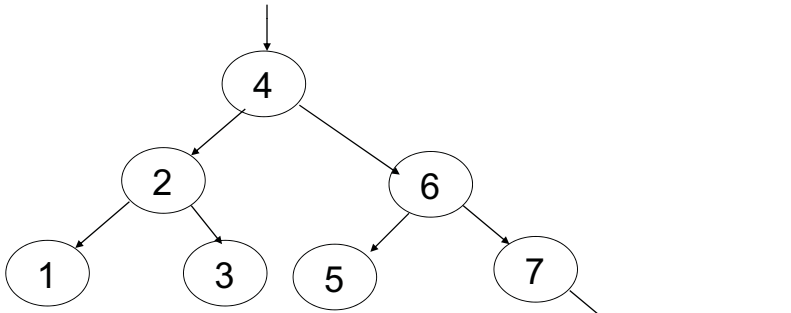


Finish inserting 8

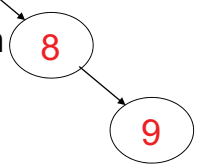
Recolored now need to rotate



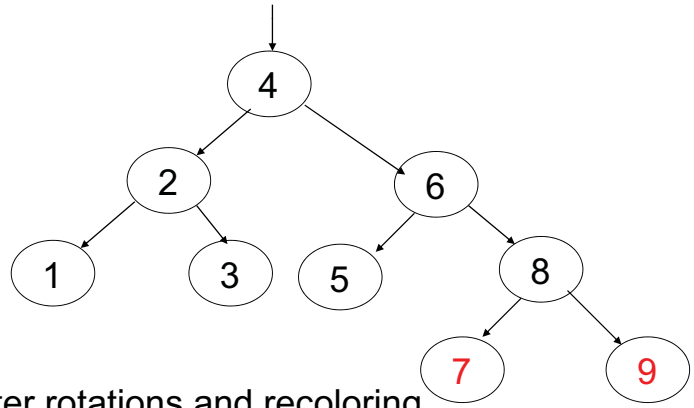
Insert 9



On way down see 4 has two red children so recolor 4 red and children black.
Realize 4 is the root so recolor black

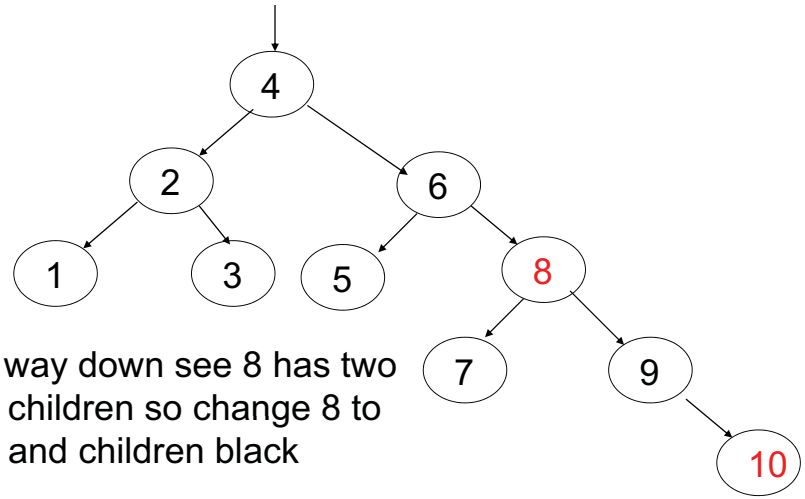


Finish Inserting 9



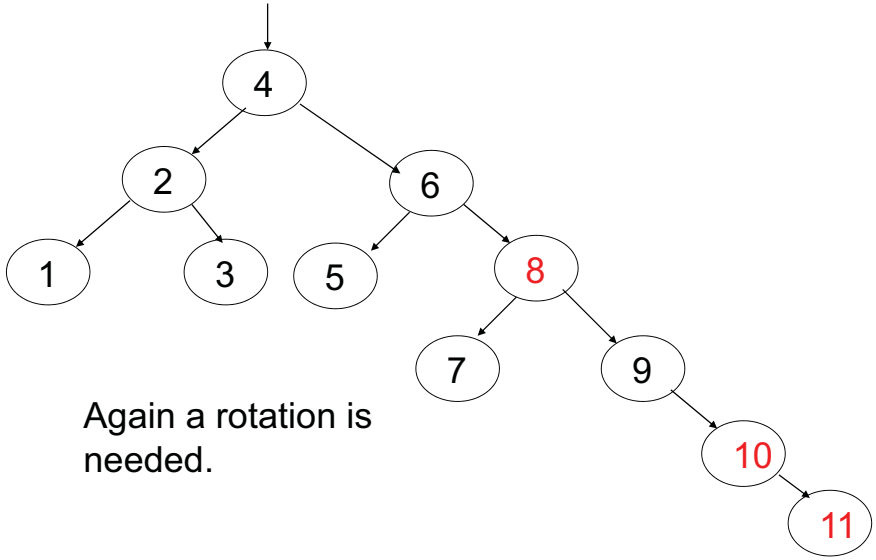
After rotations and recoloring

Insert 10



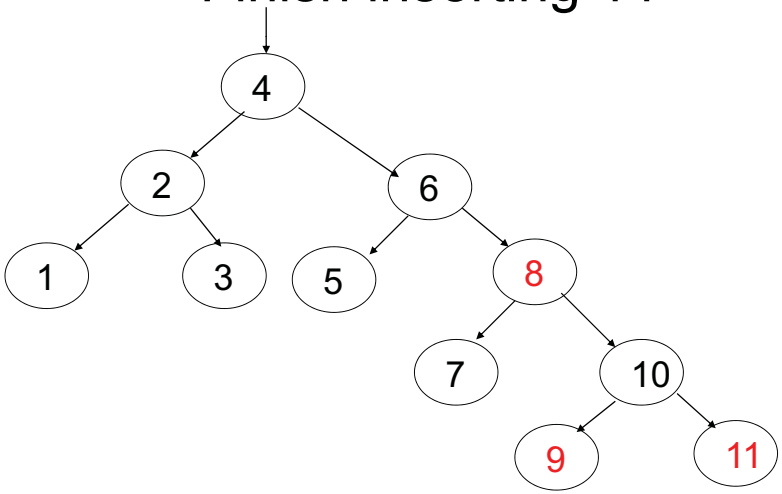
On way down see 8 has two red children so change 8 to red and children black

Insert 11



Again a rotation is needed.

Finish inserting 11



Topic 20
Data Structure Potpourri: Hash Tables and Maps

"**hash collision** n.
[from the techspeak] (var. `hash clash') When used of people, signifies a confusion in associative memory or imagination, especially a persistent one (see **thinko**). True story: One of us was once on the phone with a friend about to move out to Berkeley. When asked what he expected Berkeley to be like, the friend replied: "Well, I have this mental picture of naked women throwing Molotov cocktails, but I think that's just a collision in my hash tables."

-The Hacker's Dictionary

Programming Pearls by Jon Bentley

- Jon was *senior programmer* on a large programming project.
- Senior programmer spend a lot of time helping junior programmers.
- Junior programmer to Jon: "I need help writing a *sorting algorithm*."

A Problem

- From *Programming Pearls (Jon in Italics)*

Why do you want to write your own sort at all? Why not use a sort provided by your system?
I need the sort in the middle of a large system, and for obscure technical reasons, I can't use the system file-sorting program.
What exactly are you sorting? How many records are in the file? What is the format of each record?
The file contains at most ten million records; each record is a seven-digit integer.
Wait a minute. If the file is that small, why bother going to disk at all? Why not just sort it in main memory?
Although the machine has many megabytes of main memory, this function is part of a big system. I expect that I'll have only about a megabyte free at that point.
Is there anything else you can tell me about the records?
Each one is a seven-digit positive integer with no other associated data, and no integer can appear more than once.

Questions

- When did this conversation take place?
- What were they sorting?
- How do you sort data when it won't all fit into main memory?
- Speed of file i/o?

A Solution

```
/* phase 1: initialize set to empty */
for i = [0, n)
    bit[i] = 0

/* phase 2: insert present elements into the set */
for each i in the input file
    bit[i] = 1

/* phase 3: write sorted output */
for i = [0, n)
    if bit[i] == 1 write i on the output file
```


Some Structures so Far

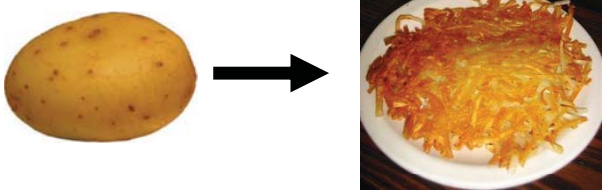
- ArrayLists
  - O(1) access
  - O(N) insertion (average case), better at end
  - O(N) deletion (average case)
- LinkedLists
  - O(N) access
  - O(N) insertion (average case), better at front and back
  - O(N) deletion (average case), better at front and back
- Binary Search Trees
  - O(log N) access if balanced
  - O(log N) insertion if balanced
  - O(log N) deletion if balanced

Why are Binary Trees Better?

- Divide and Conquer
  - reducing work by a factor of 2 each time
- Can we reduce the work by a bigger factor? 10? 1000?
- An ArrayList does this in a way when *accessing* elements
  - *but must use an integer value*
  - *each position holds a single element*

Hash Tables

- Hash Tables overcome the problems of ArrayList while maintaining the fast access, insertion, and deletion in terms of N (number of elements already in the structure.)



Hash Functions

- Hash: "From the French hatcher, which means 'to chop'. "
- *to hash* to mix randomly or shuffle (To cut up, to slash or hack about; to mangle)
- Hash Function: Take a large piece of data and reduce it to a smaller piece of data, usually a single integer.
  - A function or algorithm
  - The input need not be integers!

Hash Function

Simple Example

- Assume we are using names as our *key*
  - *take 3rd letter of name, take int value of letter (a = 0, b = 1, ...), divide by 6 and take remainder*
- What does "Bellers" hash to?
- L -> 11 -> 11 % 6 = 5

Result of Hash Function

- Mike = (10 % 6) = 4
- Kelly = (11 % 6) = 5
- Olivia = (8 % 6) = 2
- Isabelle = (0 % 6) = 0
- David = (21 % 6) = 3
- Margaret = (17 % 6) = 5 (uh oh)
- Wendy = (13 % 6) = 1
- This is an imperfect hash function. A perfect hash function yields a one to one mapping from the keys to the hash values.
- What is the maximum number of values this function can hash perfectly?

More on Hash Functions

- Normally a two step process
  - transform the key (which may not be an integer) into an integer value
  - Map the resulting integer into a valid index for the hash table (where all the elements are stored)
- The transformation can use one of four techniques
  - mapping, folding, shifting, casting

Hashing Techniques

- Mapping
  - As seen in the example
  - integer values or things that can be easily converted to integer values in key
- Folding
  - partition key into several parts and the integer values for the various parts are combined
  - the parts may be hashed first
  - combine using addition, multiplication, shifting, logical exclusive OR

More Techniques

- Shifting
  - an alternative to folding
  - A fold function

```
int hashVal = 0;
int i = str.length() - 1;
while(i > 0){
   hashVal += (int) str.charAt(i);
   i--;
}
```

results for "dog" and "god" ?

Shifting and Casting

- More complicated with shifting

```
int hashVal = 0;
int i = str.length() - 1;
while(i > 0)
{ hashVal = (hashVal << 1) + (int) str.charAt(i);
  i--;
}
```

different answers for "dog" and "god"

Shifting may give a better range of hash values when compared to just folding

Casts

- Very simple
  - essentially casting as part of fold and shift when working with chars.

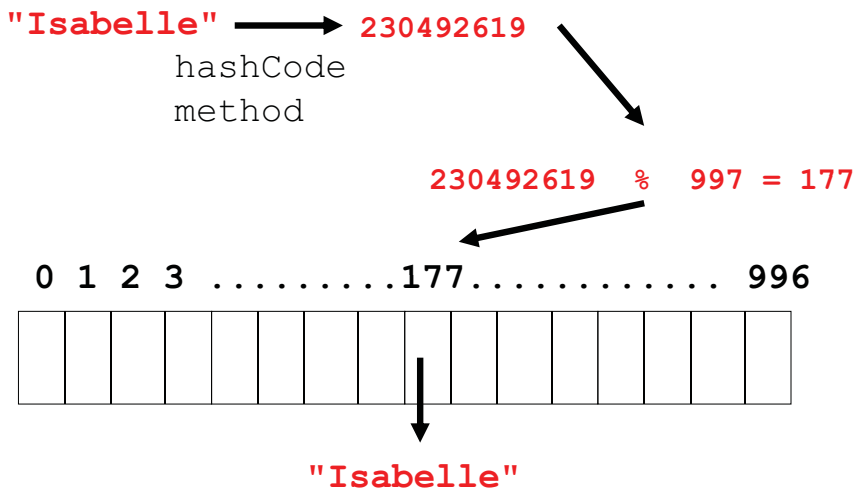
The Java String class hashCode method

```
public int hashCode()
{    int h = hash;
     if (h == 0)
     {    int off = offset;
          char val[] = value;
          int len = count;
           for (int i = 0; i < len; i++)
           {    h = 31*h + val[off++];
           }
           hash = h;
     }
     return h;
}
```

Mapping Results

- Transform hashed key value into a legal index in the hash table
- Hash table is normally uses an array as its underlying storage container
- Normally get location on table by taking result of hash function, dividing by size of table, and taking remainder
  - index = key mod n
  - n is size of hash table
  - empirical evidence shows a prime number is best
  - 1000 element hash table, make 997 or 1009 elements

Mapping Results



Handling Collisions

- What to do when inserting an element and already something present?

Open Address Hashing

- Could search forward or backwards for an open space
- Linear probing:
  - move forward 1 spot. Open?, 2 spots, 3 spots
  - reach the end?
  - When removing, insert a blank
  - null if never occupied, blank if once occupied
- Quadratic probing
  - 1 spot, 2 spots, 4 spots, 8 spots, 16 spots
- Resize when *load factor* reaches some limit

Chaining

- Each element of hash table be another data structure
  - linked list, balanced binary tree
  - More space, but somewhat easier
  - everything goes in its spot
- Resize at given load factor or when any chain reaches some limit: (relatively small number of items)
- What happens when resizing?
  - Why don't things just collide again?

Hash Tables in Java

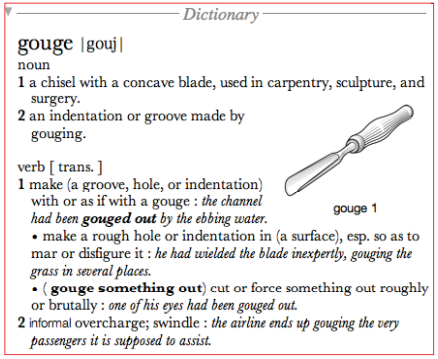
- `hashCode` method in `Object`
- `hashCode` and `equals`
  - "If two objects are equal according to the equals (`Object`) method, then calling the `hashCode` method on each of the two objects must produce the same integer result. "
  - if you override `equals` you need to override `hashCode`

Hash Tables in Java

- HashTable class
- HashSet class
  - implements Set interface with internal storage container that is a HashTable
  - compare to TreeSet class, internal storage container is a Red Black Tree
- HashMap class
  - implements the Map interface, internal storage container for keys is a hash table

Maps (a.k.a. Dictionaries)

```
A -> 65
```



Dictionary

gouge |gouj|
noun
1 a chisel with a concave blade, used in carpentry, sculpture, and surgery.
2 an indentation or groove made by gouging.

verb [trans.]
1 make (a groove, hole, or indentation) with or as if with a gouge : *the channel had been* ***gouged out*** *by the ebbing water.*
• make a rough hole or indentation in (a surface), esp. so as to mar or disfigure it : *he had wielded the blade inexpertly, gouging the grass in several places.*
• (**gouge something out**) cut or force something out roughly or brutally : *one of his eyes had been gouged out.*
2 informal overcharge; swindle : *the airline ends up gouging the very passengers it is supposed to assist.*

gouge 1

Maps

- Also known as:
  - table, search table, dictionary, associative array, or associative container
- A data structure optimized for a very specific kind of search / access
  - with a *bag* we access by asking "is X present"
  - with a *list* we access by asking "give me item number X"
  - with a *queue* we access by asking "give me the item that has been in the collection the longest."
- In a *map* we access by asking "give me the *value* associated with this *key."*

Keys and Values

- Dictionary Analogy:
  - The *key* in a dictionary is a word: *foo*
  - The *value* in a dictionary is the definition: *First on the standard list of metasyntactic variables used in syntax examples*
- A key and its associated value form a pair that is stored in a map
- To retrieve a value the key for that value must be supplied
  - A List can be viewed as a Map with integer keys

More on Keys and Values

- Keys must be unique, meaning a given key can only represent one value
  - but one value may be represented by multiple keys
  - like synonyms in the dictionary. Example: *factor: n.See coefficient of X*
  - *factor* is a key associated with the same value (definition) as the key *coefficient of X*

The Map<K, V> Interface in Java

- `void clear()`
  - Removes all mappings from this map (optional operation).
- `boolean containsKey(Object key)`
  - Returns true if this map contains a mapping for the specified key.
- `boolean containsValue(Object value)`
  - Returns true if this map maps one or more keys to the specified value.
- `Set<K> keySet()`
  - Returns a Set view of the keys contained in this map.

The Map Interface Continued

- `V get(Object key)`
  - Returns the value to which this map maps the specified key.
- `boolean isEmpty()`
  - Returns true if this map contains no key-value mappings.
- `V put(K key, V value)`
  - Associates the specified value with the specified key in this map

The Map Interface Continued

- `Vremove(Object key)`
  - Removes the mapping for this key from this map if it is present
- `int size()`
  - Returns the number of key-value mappings in this map.
- `Collection<V> values()`
  - Returns a collection view of the values contained in this map.

Implementing a Map

- Two common implementations of maps are to use a binary search tree or a hash table as the internal storage container
  - HashMap and TreeMap are two of the implementations of the Map interface
- `HashMap` uses a hash table as its internal storage container.
  - keys stored based on hash codes and size of hash tables internal array

TreeMap implementation

- Uses a Red - Black tree to implement a Map
- relies on the `compareTo` method of the keys
- somewhat slower than the HashMap
- keys stored in sorted order

Sample Map Problem

Determine the frequency of words in a file.

```
File f = new File(fileName);
Scanner s = new Scanner(f);
Map<String, Integer> counts =
      new Map<String, Integer>();
while( s.hasNext() ){
  String word = s.next();
  if( !counts.containsKey( word ) )
    counts.put( word, 1 );
  else
    counts.put( word,
        counts.get(word) + 1 );
}
```