CS 1322 Object-Oriented Programming

- Introduction
- See web page for lots of info
  - grading: “exams are it”
  - HWs: “you can drop one, like a ‘sick day’ at work”
  - book(s): “the L&L is a good book!”
  - exams
  - helpdesk: “TAs available almost all days”
  - newsgroups: “not an option, a requirement”
  - review sessions
  - collaboration policy: “feel free to shoot yourselves in the foot”
  - lecturing and slides on the web: “if you read only the slides, you will fail!”
  - make-up policy: “God forgives, professors don’t. Ask for permission beforehand! Like an airline: no-shows get no refund”
  - grade dispute policy

Java

- A popular programming language
  - used very much in web applications
-
```
// This is just a comment
public class Test {
  public static void main(String [] args) {
    System.out.println(“This is a test program”);
  }
}
```
  - different names (reserved vs. user defined)
  - whitespace irrelevant, case important
  - output with “print” (single string), “println” (whole line), both in object “System.out”

Interpreters vs. Compilers

- A program can either be interpreted (simpler) or compiled (faster)
  - *Interpret*: decode instructions, find variable values, perform instructions, for every instruction
  - *Compile*: decode instructions once, then translate them into efficient code that will perform instructions
- Java has a two-stage process:
  - compile Java program to bytecode
  - compile bytecode to machine language (or interpret)

What is Object-Oriented Programming?

- OOP = programming with objects
- *Objects* are abstractions: don't know how it works, but know what it supports
  - they usually represent entities (a car, a computer, a person)
  - they let us split our program into manageable pieces
- Objects are *instances* of *classes*. A class represents a concept
  - e.g., the concept of a car, of a computer, of an employee, etc.

Values and Types

- In a Java program, every value has a type
  - it may have many compatible types, but one of them is the most specific type
- Java has two kinds of types: classes and primitive types
  - classes can represent any concept
  - primitive types represent well-known concepts: int, long, char, byte, short, boolean, float, double, void (= nothing)
  - E.g., 3 (type = int), 3.0 (type = double), false (type = boolean), ‘a’ (type = char), 3.0F (type = float)
- For the time being, we’ll only deal with primitive types and the special class String

Defining Variables

- Types are most useful for defining *variables*. A variable has a type and a name. When the program runs, it holds a value:
  - int i = 3;
  - boolean flag = true;
  - double pi = 3.14;
  - String s = “oh, the pain!”;
  - char c = ‘a’;
- Variable values can change using *assignments*:
  - i = 5;
  - s = “hey, it’s not so bad any more”;

Type Checking

- If a value is not compatible with the type of a variable, a compile-time error occurs
  - int i = “3”;
  - float f = 0.2; // !!!
  - String s = ‘a’;
  - int num = 0.2;

Constants

- Sometimes it’s useful to give a name to constants. The *final* keyword is used.
  - final double pi = 3.14;
    final int xSize = 200, ySize = 300;

String

- The String class is special. It is a regular class, but the language has special operators for it
  - String classname = "CS" + "1322";
    - which is confusing because "+" means also "add numbers":
    - String phone = "404" + "9876543";
    - String phone = "404" + "987" + "6543";
    - String phone = "404" + "987" + 6543;
    - String phone = "404" + 987 + 6543;
      - see Facts.java
  - Special string (and char) escapes: \", \n, \t, \\
      - see Roses.java

Expressions

- Usual arithmetic operators and precedence rules used to form expressions
    - total = 2 + 3*5;
    - total = (2 + 3) * 5;
    - x = 8 – 3 * 4 + 6 / 2;
    - y = 4 / 2 – 3 + 5 * 3;
    - fahrTemp = celsTemp * 9.0/5.0 + 32;
  - also ++, --, +=, *=, /=, %=, etc. operators as shorthand
- Precedence: unary ops (-), multiply/divide (*, /, %), add (+, -), assign
  - when in doubt, use parentheses

Types

- Every expression has a (most specific) type, but we may need to convert to others
- Conversion done
  - by assigmnent
    - int sum;
      double avg;
      avg = sum; // ok
      sum = avg; // compile error
  - by arithmetic promotion (implicit conversion)
    - result = sum/avg; // promoted to double
  - by casting (explicit conversion)
    - sum = (int) avg; // type name in parentheses
  - what is the precedence of cast?
    - double t = (double) 5 / 2;
- Roughly speaking: widening is ok, narrowing needs cast

Using Classes

"Let's see what others have done"

Calling Methods

- Java code is written inside “methods” (which are written inside classes)
- A method is “called” on an object using the “.” operator. Parameters can be passed:
  - System.out.println(“I am your father, Luke!”);
  - String s = “no way, dude!”;
    System.out.println(s.length());
- “main” in our programs is a method

Useful classes: String

- Review: class = concept (e.g., car);
  object = entity (e.g., my Honda car)
  - for now, we won’t care about how classes are defined. We’ll use pre-written classes (String, Scanner, Random)
- String supports lots of useful methods:
  - char charAt(int index)
  - int length()
  - String replace(char c1, char c2)
  - String substring(int startind, int endind)
  - String toUpperCase()
    - see StringMutation.java

Useful classes: Scanner

- Programs need to communicate with the outside world. Scanner is a class that reads input from the keyboard
    - Scanner scan = new Scanner(System.in);
      int n = scan.nextInt();
      String word = scan.next();
      String line = scan.nextLine();
      boolean b = scan.hasNext();
        - see GasMileage.java
  - What's that "new" in the first line?

Creating Objects

- "new" is used to create objects from a given class
    - Dog fido = new Dog(); // Imaginary class
    - Scanner sc = new Scanner(System.in);
    - String s = new String("deja vu, all over again");
- Object references can also get "null"
    - special value, valid for any object type, but means that the reference points to nothing (i.e., cannot be used)
    - Dog fido = null;
    - Scanner sc = null;
    - String s = null;
        - but none of them can be used to call methods on

Packages

- Classes are grouped into packages
  - package = set of related classes
  - java.lang always available. String is there
  - classes in other packages need to be imported before use:
    - import java.util.Random;
    - import java.util.*;
  - others: java.awt, java.beans, java.io, java.math, java.net, java.util

Useful Classes: Random

- Random = a random number generator
    - ```
      import java.util.Random;
      ...
      Random gen = new Random();
      int i = gen.nextInt(); // all ints
      float f = gen.nextFloat();  // 0.0f to 1.0f
      i = gen.nextInt(11); // 0 to 10
      ```
  - random = pseudo-random. Can produce deterministic sequences with a seed:
    - Random gen = new Random(7);
      ```

Useful Classes: Math

- Math = a collection of useful mathematical functions
  - No need to use “new” (methods static, as we’ll see later)
  - Math.abs(-5.0) -> 5.0
  - Math.pow(3.0, 4.0) -> 81.0
  - Math.sqrt(9.0) -> 3.0
  - Math.PI
  - Math.cos(Math.PI) -> -1.0
  - Math.sin(Math.PI/2) -> 1.0
  - ...
    - all arguments double, except abs, which also supports int
    - see Quadratic.java, try with negative discriminant get NAN

Useful Classes: Wrapper Classes

- Each primitive type has a “wrapper” class
  - byte: Byte, int: Integer, double: Double, char: Character, boolean: Boolean, ...
  - for now, all we need from wrapper classes is specialized parsing functions:
    - int i = Integer.parseInt(“809”); // 809
    - Scanner sc = new Scanner(System.in);
int i = Integer.parseInt(sc.next());
  - otherwise, wrapper objects are just like primitive values:
    - Integer num = new Integer(40);
System.out.println(num.intValue()); // 40

Defining Classes

"Let's try to make some ourselves"

What Is a Class?

- It's a blueprint for creating objects
  - represents a concept (e.g., person, account, computer, car, item, etc.)
  - groups together many "members":
    - methods
    - member fields (or "member variables")
      - see Die.java
  - the main idea is called "encapsulation": a class should expose *what* it supports and hide *how* it does it

Instance Data (Member Variables)

- Member variables represent all fields of the objects built from this class
  - scope: entire class, extent: one per object
    - class Person {
      String name;
      int age;
      void display() { ... name ... age ... }
      }
    - "name", "age" can be used in methods inside Person
  - automatically initialized to 0, null

Member Methods

- Methods have two parts:
  - signature (what the method is called and how it is called)
  - body (what the method does)
    - RetType methName**(**ArgType1 argName1, ...**)** { Body }
    - e.g.,
      - void display() {
        System.out.println(name + " is " + age + " years old");
        }
      - int twice(int num) { return num + num; }
    - methods accept parameters when they are called
    - methods may return a value, use the return statement for this
      - void : "return" or nothing, non-void: "return <type-correct-value>"
  - more on how methods are called and return later
    - but you should know the basic idea from previous course

Encapsulation

- Visibility modifiers:
  - private (only accessible in class)
  - public (accessible by anyone)
  - protected (accessible by “related” classes, subclasses—more later)
  - <none> (accessible by “related” classes)
    - “related” means in same package, but ignore this for now
  - E.g.,
    - class Person {

```
  private String name;
  private int age;
  public void display() { ... name ... age ... }
}
```

  - Good practice: always explicitly use public or private
    - not always enough room to do it in these slides

Accessors and Mutators

- Methods made to provide disciplined access to private data
    - class Person {

```
  private String name;
  private int age;
  ...
  public void setAge(int a) { age = Math.abs(a); }  // mutator
  public int getAge() { return age; } // accessor
}
```

  - see also Die.java
- a.k.a. “getters”/“setters”

Constructors

- Constructor = special method used to create an object
  - this is what is called when you say “new <ClassName>()”
  - name = name of class
  - do not return anything, *not even void*
  - until the constructor has run, the object is not yet fully created. Better to not call other methods on it!
  - class Person {

```
class Person {
  private String name;
  private int age;
  public Person(int a, String n) {
     name = n.toUpperCase();
     age = Math.abs(a);
  } ...
}
```

  - all classes have a “default constructor”

Yannis Smaragdakis, CS 1322

Method Calling

- Argument values copied to formal parameters
- Can also create temporary variables

```
– class MyMath {
    int myFunction(int num) {
      int twice = num+num;
      return twice * twice + 45* twice + num – twice/5;
    }
  }

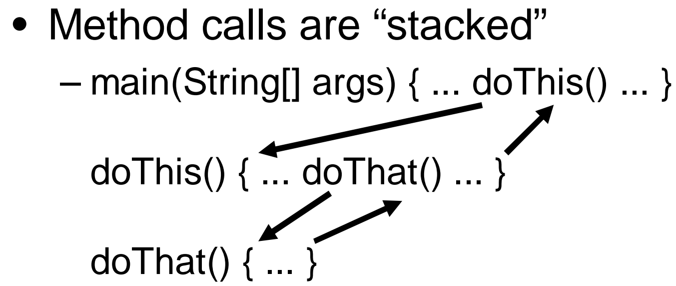
  MyMath mm = new MyMath();
  mm.myFunction(5);
    // copy of “5” created, called “num” inside “myFunction”
```

  – Parameters and temporaries disappear after method returns!

Yannis Smaragdakis, CS 1322

Method Calling

- Method calls are “stacked”



If and Loops

“let’s do some real programming”

Boolean Expressions

- In programming, logical tests are called “boolean expressions” and return a value
  - the Java type of the value is “boolean”
    - ==, < , <=, >, >=, != ...
    - logical connectives: !, &&, ||
  - e.g.:
    - boolean done = false;
      int count = 3, MAX = 10;
      boolean flag = (!done && (count < MAX));
  - shortcircuiting:&&, || evaluate only as needed
    - done && method() // method never called if !done

if (... else)

- If: conditional execution statement
  - if (<boolean cond>) <do something>
    else <do something else>
  - entire “if ... else” is one statement, “else” is optional
    - if (myCoin.isHeads())
        System.out.println(“You win”);
      else System.out.println(“I win”);
      System.out.println(“Game over”);
  - “dangling else”: which “if” matches “else”?
    - if (x < y) if (x < z) return x; else return z;
      else if (y<z) return y; else return z;
  - also “if for expressions”: <cond> ? <expr1> : <expr2>
    - return x < y ? (x < z ? x : z) : (y < z? y : z) ;

Yannis Smaragdakis, CS 1322

Comparison Advice

- Ways to compare:
  – for floats, doubles, representation is inaccurate. Compare with tolerance for equality:
    - if (Math.abs(d1 – d2) < TOLERANCE) ...
  – for chars, <, > compare them under Unicode ordering. No need to care in most cases: string comparisons
  – for strings, compare with “compareTo”
    - name.compareTo(“A”) >= 0 && name.compareTo(“Z”) <=0
  – for objects, compare with “==” to find out if they are the *same* object, compare with “.equals” to find out whether they have the same “value” (class-specific)

Switch

- An equality-based comparison statement *with fall-through*
  – for ints, chars (*not* Strings!)

```
switch (grade) {
  case ‘A’ : System.out.println(“Ah, lucky!”);
             break; // if left out, next println executes
  case ‘B’ : System.out.println(“Good job”);
             break;
  case ‘C’ ... case ‘D’ ... case ‘F’ ...
  default : System.out.println(“That’s not a grade!”);
}
```

While Loop

- while (<cond>) <statement>
  - keeps executing the statement as long as the condition is true
    - ```
      int i = 0, e = 7;
      while (i < 10)
        e = e * 2 + e;
      ```
    - statement can be multiple statements in { ... }
      - see Average.java
  - break, continue: escape early (never necessary, but convenient)

More on Loops

- What if the condition never becomes false? (*infinite loop*)
  - ```
    int count = 1;
    while (count <= 25) {
      System.out.println(count);
      count = count - 1;
    }
    ```
  - Need to be very careful when programming to ensure loops will eventually finish
- A “while” can contain another “while” (*nested loops*)
  - e.g., code a method stars(int n) that will print out n, n-1, n-2, etc. stars (each block on a new line)

Reading Text Files

- You can use a scanner on a disk file

```
Scanner sc = new Scanner(new File("file.txt"));
int i = 1;
while (sc.hasNext()) {
  String line = sc.nextLine();
  System.out.println(i + ": " + line);
  i++;
}
```


Do Loops

- Like "while", only execute first, check later
  - do <stmt> while (<cond>)
  - useful when loop body will execute at least once

```
int input = 1;
do {
  System.out.println("Enter a positive number");
  input = sc.nextInt();
} while (input <= 0);
```

    - see ReverseNumber.java (on your own)
  - could do same with "while", but body will be replicated

For Loops

- for (<init>; <cond>; <updstmt>) <stmt>
    - <init> can be a statement or a variable declaration
    - <updstmt> executed *after* <stmt>!
  - just a convenient shorthand
    - ```
      int i = 1;
      while (i <= 5) {
        System.out.println(i);
        i++;
      }
      ```
    - ```
      for (int i = 1; i <= 5; i++)
        System.out.println(i);
      ```
  - often makes it easier to avoid infinite loops: it's clear how expressions are updated so that the loop eventually finishes

Classes, references, interfaces, testing

"*Let's start going deep*"

How to Design Classes

- Recap: classes are concepts, objects are entities
  - a single class is used to produce many objects (“instances”)
- Don’t confuse classes and objects! A class is the blueprint, objects are what we produce from it
  - e.g., the concept of a “Honda Civic” vs. “my Civic”, “Mike’s Civic”, “Jennifer’s Civic”, etc.
- When designing programs, first identify concepts (“nouns”) and represent them as classes

Static Members

- A class can have a special kind of members called “static”
  - we’ve used such members many times already
    - our well-known “main” is a static method
    - Math.abs is a static method
    - System.out is a static field of class System
  - static fields are shared by all objects of the class
  - static methods are best called using the class name
  - static methods *cannot access non-static fields*, unless they are passed a specific object!
    - otherwise ambiguous: which object’s field?

Example

- What happens in each of the 4 cases (w. w/o static) ?
  - class A {

```
class A {
  ?static? int i = 0;
  ?static? void meth() {
    i += 3;
    System.out.println(i);
  }
}
...
A a1 = new A();
A a2 = new A();
a1.meth();
a2.meth();
```

Yannis Smaragdakis, CS 1322

Design Concepts

- When we design software, we first identify concepts (classes) and then their relationships
  - typically three kinds: has-a, is-a, uses
- An object can refer to other objects, using reference fields
  - references implement *has-a* (aggregation) and *uses* (dependency, delegation) relationships

Yannis Smaragdakis, CS 1322

Example

- class Car {
  Wheel frontleft, frontright, rearleft, rearright;
  Seat ...
  }

  class Wheel {
  Tire tire;
  }

  class Seat { ... }
  class Tire { ... }

```
class Car {
  Wheel frontleft, frontright, rearleft, rearright;
  Seat ...
}

class Wheel {
  Tire tire;
}

class Seat { ... }
class Tire { ... }
```


References and "this"

- A reference is either null or points to an object
- Each object can refer to itself with a reference named "this"
  - "this.foo()" or "this.field" is usually the same as just "foo()" or "field"
- "this" used when passing object as parameter and secondarily for disambiguation of names

```
class Person {
  int i;
  Person father;
  void meth(int i) {
    this.i = i;      // "this" used for disambiguation
    father.sendMoney(this); // "this" passed as parameter
  }
}
```

References: what's the deal?

- Objects in Java are accessed *only* through references.
  <u>Big difference from primitive types!!!</u>
    - Person p; // declares a reference to person
      int i; // declares an integer i, not a reference to one
    - p = new Person(); // now p points to a real person
      i = 5;
  - The real difference is when copying values and then changing them:
    - Person p = q; // now both p and q point to same person!
      int j = i; // j just gets a copy of the value of i
      p.age = 30; // q.age also becomes 30!
      j = 30; // i is not affected

Parameter Passing

- Calling methods is another way to copy values
  - recall that the parameters are copied into the formals
- Again, copying references means that the new reference points to the same object!
  - see ParameterTester.java for a long example
  - void meth(int i, Person boy, Person girl) {
    i = 3;
    boy.age = 6;
    girl = new Person();
    }
    ...
    Person boy = new Person(), girl = new Person();
    int num = 0;
    girl.age = 7;
    meth(num, boy, girl); // what is the value of num, boy.age, girl.age?

Yannis Smaragdakis, CS 1322

Aliasing

- Having multiple references point to the same object is called *aliasing*
  - aliasing is very common in Java
    - try it with every class you know
      - ```
        import java.util.Random;
        ...
        Random gen = new Random(4); // make it deterministic
        int i = gen.nextInt();
        Random gen2 = gen;
        i = gen2.nextInt();  // consumes a number. Remove to see
        i = gen.nextInt();
        ```


References and Equality

- Now it should also be clearer why we have "==" vs. "equals" for objects
  - one checks whether the two references point to same object
  - the other checks whether the two references point to equivalent objects
      - ```
        String s = "john";
        String s2 = new String("john");
        // s == s2 ? No!
        // s.equals(s2) ? Yes!
        ```

Overloading

- *Overloading*: methods in the same class can have the same name as long as there is no confusion when the method is called
  - ```
    println(String s) {...}
    println(int i) {...}
    println(double d) {...}
    ```
- Constructors can also be overloaded
  - ```
    class Team {
      int winDiff;  // Games won minus games lost
      Team(int diff) { winDiff = diff; }
      Team(int wins, int losses) {this(wins-losses); }
    }
    ```


Interfaces

- Classes are not the only types users can define in Java
- Interfaces: like classes, only without variables and without code (can have constants and method signatures)
  - then, why have them? Only to give a name to well-known concepts, so that we can write programs that deal with any class that supports the concept
    - ```
      interface Drawable {
        void drawAt(int x, int y);
        int ySize();
      }
      class Line implements Drawable { ...
        void drawAt(int x, int y) { ... }  // do the drawing
        int ySize() { ... }
      }
      ...
      void drawUnder(Drawable a, Drawable b, int x, int y) {
        a.drawAt(x,y);
        b.drawAt(x,y + a.ySize());
      }
      ```
      ```

Interface Specifics

- A class can implement many interfaces but needs to support all the methods declared in them
- A class can have additional methods, not defined in any interface it implements
- Many classes can implement the same interface
- An interface can also contain constant fields (final implied)

Useful Interfaces

- Classes we have seen often implement very common interfaces
    - class String implements Comparable ...
        – supports method “compareTo”
    - class Scanner implements Iterator ...
        – supports methods “hasNext”, “next”, “remove”
  – Good programmers write general code: e.g., when you write a method that takes in Strings, think whether it would work for all Comparables

Program Development Advice

- DON'T
  - write all first, test next
  - complain of vague errors. Be specific, show minimal example
- DO
  - develop incrementally
    - what you have should always run and then you just add more functionality
    - develop unit tests
    - isolate errors / divide and conquer
      - dynamically: what's the difference between your program and working code?
      - logically: test small pieces with unit tests
- Programming = divide and conquer

Arrays and ArrayList

"let's start remembering many things at once"

Arrays and Data Structures

- Imagine your program needs to remember several objects, but you don't know how many until run-time
  - E.g., create N planet objects, then go back and make them look bigger
- Remembering many things is the purpose of *data structures*
- Arrays are the simplest data structure
- Array = an ordered table of values
  - Typed (all values same type), zero-indexed

Yannis Smaragdakis, CS 1322

Example

- An array of integers (type: int[])
  - ```
    int [] age = new int[10];
    for (int i = 0; i < 10; i++)
      age[i] = 18+i;
    ```
  - what is age[1]? age[0]?
    - see LetterCount.java (long example, "upper[cur-'A']++" idiom)
- An array of Strings (type: String[])
  - String[] names = new String[10];
- Also, array initializer lists
  - char [] vowels = {'A', 'E', 'I', 'O', 'U'};
- Arrays are objects
  - even if they are arrays of primitives
  - e.g., only references exist until "new" or initialized

Yannis Smaragdakis, CS 1322
    ```

Bounds Checking

- Array accesses are checked for being in-bounds
  - ```
    int [] age = new int[10];
    for (int i = 0; i < 10; i++)
      age[i] = 18+i;
    ```
  - bounds: 0 to 9. What is age[10]?
- Arrays have a "length" constant defined by default
  - ```
    int [] age = new int[10];
    for (int i = 0; i < age.length; i++)
      age[i] = 18+i;
    ```
  - age[age.length] ?

Arrays Are Objects!

- Arrays can be aliased
  - ```
    int[] age = { 62, 35, 45, 12, 28 };
    ...
    int[] employeeAge = age;
    employeeAge[1] = 19;
    // age[1] ?
    ```
- Same when used as method parameters
  - ```
    int change(int [] array) { array[1] = 19; }
    change(age);
    // age[1] ?
    ```

Other Syntax and Conventions

- forall idiom:
  - ```
    String [] names = {“Jenn”, “David”};
    for (String ele : names)
       System.out.println(ele.length());
    ```
- Command line arguments: array of strings
  - ```
    public static void main(String[] args) {
      if (args.length < 3)
        System.out.println(“insufficient args”);
      ...
    }
    ```


Arrays of Objects

- Arrays of objects are just arrays of references
  - no objects are allocated!
    - ```
      Bird [] flock = new Bird[10];
      for (int i = 0; i < flock.length; i++)
        flock[i] = new Bird();
      ```
      ```

Multi-Dimensional Arrays

- Imagine a 2-d table (e.g., a photo)
- Multi-dimensional arrays: just arrays of arrays!
  - int [][] table = new int[5][10];
    // 5 arrays of length 10 each
  - int [][] scores = { {1,3,0}, {2,2,0} };
    // dimensions? elements?
- Can be jagged
  - arr[1].length != arr[0].length
  - int [][] table = new int[5][]; // empty dims in the end
    table[0] = new int[3];
    table[1] = new int[5];

ArrayList

- ArrayList is another data structure you can use for storing many objects
  - think of it as expandable array
  - removals compact array, additions move later elements
    - add(Object o);
    - add(int index, Object o);
    - remove(int index);
    - int indexOf(Object o);
    - Object get(int index);
    - int size();
    - boolean contains(Object o);

Inheritance, Object, Polymorphism

"What is this *Object-oriented* stuff?"

Subclasses

- Classes are blueprints for objects, represent concepts
- A new class can be a specialization of an existing class, without needing to redefine what the existing class has already defined
  - this is the whole purpose: reuse of code
- This is called "subclassing" or "inheritance"
  - the original class is called "parent", "superclass", or "base class"
  - the new class is called "child", "subclass", or "derived class"

When to Subclass?

- Only when the new concept specializes the old one
  - the IS-A test
  - Animal -> Dog, Vehicle -> Car, Vehicle -> Motorcycle, Person -> Man, Employee -> Professor

How to Subclass?

```
class Vehicle {
  Wheel[] wheels;
  ...
  void move()
  { for (Wheel wheel : wheels) wheel.move(); }
}
class Car extends Vehicle { ... }

Car c = new Car();
System.out.println(c.wheels.length);
c.move();
```

"Protected"

- Remember the "protected" access modifier?
  - class C { protected int i; ... }
  - It means the member is accessible to subclasses (and related classes) but not to any other classes
    - subclasses inherit private members, but cannot access them via code in the subclass

Method Overriding

- A subclass can define methods with identical signature as superclass methods and "override" them

```
class Car extends Vehicle {
  void move() { wheels[0].roll(); wheels[1].roll(); }
}
Car c = new Car();
c.move(); // calls "move" in Car, not in Vehicle
```

- Cannot override member variables
  - they are just hidden
    - avoid re-declaring variables with same name in subclasses

Super

- “super”: like “this” only for parent sub-object. Can access parent methods explicitly
  - E.g., super.move(), when the subclass has its own “move”
- Also super() for constructor calls
  - every subclass constructor implicitly first calls super()

Object

- When a class does not subclass another, it implicitly subclasses a special class called Object (java.lang.Object)
  - class T {} = class T extends Object {}
- Object provides some useful methods that all Java classes support
  - equals, toString, clone
  - can override those. E.g.:
    - to define what it means for two Person objects to be equal (i.e., exact copies)
    - toString used to print every object (in System.out.print). By default prints a unique identifier. We may override it to print more information (e.g., a person’s name and SSN)
    - class Person {

```
class Person {
  String name; String ssn;
  public String toString() { return name + ssn; }
}
```

Inheritance Hierarchies

- Classes can form hierarchies through inheritance
  - a child class can itself be a parent
    - a Man is a subclass of Person, but Father may be a subclass of Man
  - all hierarchies end at Object: the superclass of all classes
  - Terminology: we talk of "*direct* superclass/subclass" and just "superclass/subclass". Ask when it's vague whether "superclass" means "direct superclass"
- Common features should be as high up in the hierarchy as possible for best reuse

Abstract Classes

- Sometimes classes are not meant to create objects, just to be used in subclassing
- e.g., Animal. Doesn't make sense to say "new Animal()". What kind???
    - abstract class Animal {

```
abstract class Animal {
  Leg[] legs;
  public abstract void move();
  int numLegs() { return legs.length; }
}
class Dog extends Animal { ... }
```

- Subclasses need to override abstract methods
    - abstract methods only in abstract classes
    - abstract classes can have non-abstract methods

Interface Hierarchies

- Interfaces can also extend each other, form hierarchies
  - inherit all members (including method signatures and constants)
    - interface Paintable extends Drawable {
      int paint();
      }
  - class and interface hierarchies are entirely separate

Yannis Smaragdakis, CS 1322

Subtyping (***Important***)

- Recall that we said subclasses in Java should satisfy the IS-A test!
- This is because Java also supports subtyping: a subclass object can be used ***anywhere*** a superclass object is expected
    - class Farmer { void feed(Animal a) { ... } }
      Farmer f = new Farmer();
      f.feed(new Cow());
      f.feed(new Dog());
  - Or, for a simpler example:
    - Animal a = new Cow();
      Animal a = new Dog();

Yannis Smaragdakis, CS 1322

More Subtyping

- The same can be done with interfaces. “implements” causes subtyping
    - interface Drawable { void draw(); }
      class Square implements Drawable { void draw() {...} }
      class Triangle implements Drawable { void draw() {...} }
      class A { static void display(Drawable d) { ... d.draw(); ... } }
      A.display(new Square());
      A.display(new Triangle());
  - Or, for a simpler example:
    - Drawable d = new Square();
      Drawable d = new Triangle();

Wrong Subtyping

- We can cast a superclass object to subclass, but need to be sure type is right
  - e.g., casting to-and-from Object is common for writing general code
- A Dog IS-A Animal, but not the other way around
    - class Dog extends Animal {...}
      class Cat extends Animal {...}
      Dog d = new Animal(); // ?
      Dog d = new Cat(); // ?
      Animal a = new Dog(); // ?
      Dog d = (Dog) a; // ?
      Cat c = (Cat) a; // ?

Polymorphism (***Important***)

- When calling overridden methods, the version of the method called is that in the *actual* class of the object
  - ```
    class Animal { String name() {return "dunno"; } }
    class Dog extends Animal {
      String name() { return "Dog"; }
    }
    class Cat extends Animal {
      String name() { return "Cat"; }
    }
    class Farm {
      static report(Animal a) {
        System.out.println("I am a " + a.name());
      }
    }
    Farm.report(new Dog());
    Farm.report(new Cat());
    ```
  - what gets printed?

More Polymorphism

- More generally:
  - ```
    Animal a = new Dog();
    a.meth();  // method called is that of Dog, if it exists
    ```
  - ```
    Dog d = new Dog();
    ((Animal)d).meth(); // ???
    ```
  - ```
    Animal a = new Dog();
    ...
    Dog d = (Dog) a;
    a.meth(); // ???
    d.meth(); // ???
    ```
- Also called "dynamic dispatch"
  - we know what method to call only at run-time ("dynamically") as it depends on the actual type of the object

“Final”: restricting inheritance

- Classes and methods can be declared “final”
  - final methods cannot be overridden
  - final classes cannot be subclassed
    - compiler stops you if you try
- Useful for creating methods and calling them so that you are sure you know which version you are calling

Searching and Sorting

“Simple Algorithms Made General Through Inheritance”

Selection Sort

```
• void selSort(Comparable [] arr) {
    for (int fstUnsorted = 0; fstUnsorted < arr.length; fstUnsorted++) {
      int minInd = fstUnsorted;
      for (int i = fstUnsorted + 1; i < arr.length; i++) {
        if (arr[i].compareTo(arr[minInd]) < 0)
          minInd = i;
      }
      Comparable temp = arr[minInd];
      arr[minInd] = arr[fstUnsorted];
      arr[fstUnsorted] = temp;
    }
  }
```

Find min

Swap

Insertion Sort

```
• void insSort(Comparable [] arr) {
    for (int fstUnsorted = 1; fstUnsorted < arr.length; fstUnsorted++) {
      Comparable temp = arr[fstUnsorted];
      int i;
      for (i = fstUnsorted; i>0; i--)
        if (temp.compareTo(arr[i-1]) < 0)
          arr[i] = arr[i-1];
        else break;
      arr[i] = temp;
    }
  }
```

Search

- Linear search: slow, straightforward
- Binary search: assumes sorted array, but very fast
- Just like sorting, can use Subtyping (through interfaces) to express the algorithm very generally

Binary Search

-
```
Comparable binarySearch(Comparable [] sarr, Comparable ele) {
  int min = -1, max = sarr.length;
  int ind = -1;
  // if ele in array, then between min and max (but not at either)
  // invariant in loop: (max+min)/2 is always a legal array index
  while (max – min > 1) {
    int mid = (min+max)/2;
    if (sarr[mid].equals(ele))
      break;
    if (sarr[mid].compareTo(ele) < 0) min = mid;
    else max = mid;
  }
  if (max – min <= 1) // empty range
    return null;
  return sarr[ (min + max) / 2 ];
}
```

Exceptions

"How to Tell Things Went Wrong Without Checking All the Time"

Exceptions

- There is often a main processing logic in a program, but also some exceptional conditions
  - tried to open a file but it's not there
  - tried to call a method, but the argument is invalid
  - tried to look inside an array, but beyond the end
  - tried to do a division but the denominator is zero
- One way to handle those is to return error values and keep checking: did the operation finish correctly?
- This is too hard: most of our code would be checking code
- Instead: exceptions. Objects that indicate unusual conditions. Code is written as if no exception can occur, and we add "exception handling" code on the side

Uncaught Exceptions

- If an exception is “thrown” but not “caught”, the program terminates, prints stack trace
- You’ve seen this in your programs
    - NullPointerException
    - ArrayIndexOutOfBoundsException

Catching Exceptions

- Use the “try...catch” statement to catch exceptions
    - see ProductCodes.java
    - ```
      try {
        int i = Integer.parseInt(str.substring(3,5));
        ...
      }
      catch (StringIndexOutOfBoundsException e) { ... }
      catch (NumberFormatException e) { ... }
      ```
  - the “catch” says what types of exceptions it can handle
    - most generally (bad practice): catch (Exception e) {...}
    - when we catch one, we can retry the operation (e.g., to get different input)
    - we can print the stack (e.printStackTrace()), or get an explanation string (e.getMessage())
      ```

“Finally”

- A “try...catch” has an optional “finally” clause
  - after “catch” (if a “catch” exists)
  - executed always, whether or not exception is thrown
  - used for cleanup tasks

Exception Propagation

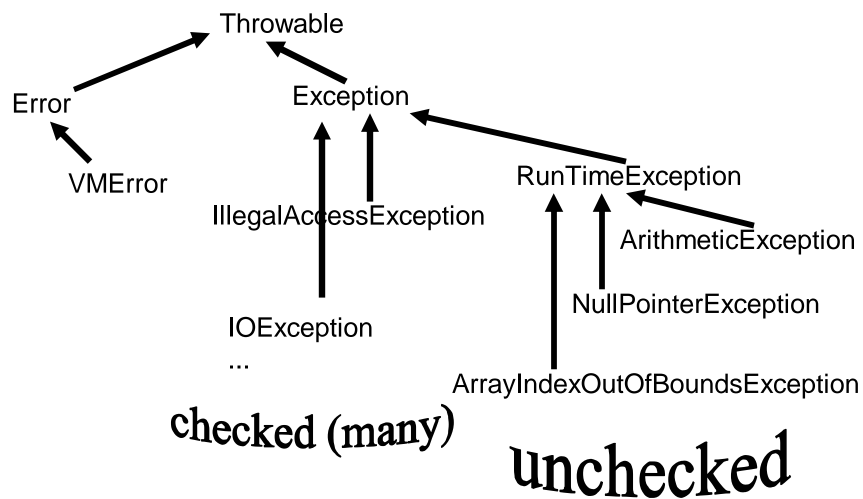
- Exceptions propagate:
  - from the operation that throws them
  - to the method that performed the operation
  - to the method that called that method
  - to the method that called that method
  - ...
  - to the main method
  - (until they are caught or the program exits)

Creating Exceptions

- There is no magic: you can create and throw exceptions yourselves
- A user-defined exception is just a subclass of Exception
  - see CreatingExceptions.java
- An exception is thrown using the “throw” clause
  - throw new MyOwnException();

Yannis Smaragdakis, CS 1322

Exception Class Hierarchy



Yannis Smaragdakis, CS 1322

Checked Exceptions

- Your own exceptions are always "checked":
  - the compiler will check whether they are caught
  - if not caught in the method thrown, the method needs to have a "throws" clause in its signature
    - void meth() throws WeirdException {
      ... throw new WeirdException();
      }
    - void meth2() {
      ...
      try { ... throw new WeirdException(); ... }
      catch (WeirdException e) {...}
      }

Yannis Smaragdakis, CS 1322

Java I/O and Exceptions

- Java I/O uses "streams" abstraction
  - e.g., System.in (InputStream), System.out, System.err (PrintStream)
  - can create many types of streams
    - on a file/socket, binary/text, buffered
      - new PrintWriter(
        new BufferedWriter(new FileWriter("test.dat")))
      - PrintWriter adds "print", "println"
      - see TestData.java
  - operations throw different IOException objects
    - e.g., "FileNotFoundException"

Yannis Smaragdakis, CS 1322

Recursion

"Defining things in terms of themselves"

Recursion

- Recursive method = method that calls itself
  - int fib(int n)
    { if (n>2) return fib(n-1) + fib(n-2) else return 1; }
- Recursive type = a type that is defined in terms of itself
  - class Person {
      String name;
      Person father, mother;
      ...
    }
- Recursive definitions in math: everything with a "..."

Infinite Recursion

- Bad recursion:
  - "Def. Recursion: see Recursion."
- Good recursion:
  - "Def. Ancestors: a person's parents or their parents' ancestors."
- The "bad" kind is "infinite" recursion. Does not terminate, cannot be used to define
- In methods, need to make sure something gets smaller with every call
- Infinite recursion leads to infinite loop in the logic and a *crash* in the program (stack overflow)

Examples

- Direct recursion
  - ```
    int fact(int n) {
      if (n > 0) return n*fact(n-1);
      return 1;
    }
    ```
  - ```
    int sum(int n) {
      if (n == 1)
       return 1;
      return sum(n-1) + n;
    }
    ```
  - what is sum(0)?
- Also, "indirect" recursion: call other method that calls back original

Implementation and Uses

- Recursion implemented through a stack, just like regular method calls
  - each invocation of the method has its own copy of all variables
- Recursion and iteration are interchangeable, but each more convenient for certain tasks
  - see examples in book (Maze.java, etc.)
- Good example: towers of Hanoi
  - moveTower(int n, int start, int end, int temp) {

```
moveTower(n-1, start, temp, end);
moveDisk(start,end);
moveTower(n-1, temp, end, start):
}
```


Data Structures

"Organizing data for fast processing"

Data Structures

- We've seen arrays and ArrayLists
  - ways to organize large amounts of data
  - we have not seen how ArrayList is defined
- Many more data structures
  - each has performance advantages for different tasks
  - e.g., ArrayList is slow if inserting, removing in middle
- The first question is "what operations does the data structure support?"

Abstract Data Types (ADTs)

- ADT: the operations supported ("what"), no clue about the "how"
  - ADTs map well (kind of) to Java interfaces

```
– interface SetADT {
    void add(Object ele);
    Object remove(Object ele);
    Object removeRandom();
    SetADT union(SetADT set);
    boolean contains(Object ele);
    boolean equals(SetADT set);
    boolean isEmpty();
    int size();
    Iterator iterator();
  }
```

    - Iterator: interface with hasNext, next
- Data structure: an implementation ("how")

Java Generics

- L&C Book talks about Data Structures using Java generics
  - a new construct, introduced in Java 5
    - ```
      interface SetADT<T> {
        void add(T ele);
        T remove(T ele);
        T removeRandom();
        SetADT<T> union(SetADT<T> set);
        boolean contains(T ele);
        boolean equals(SetADT<T> set);
        boolean isEmpty();
        int size();
        Iterator<T> iterator();
      }
      ```
  - note: no mention of “Object”. Also no need for casts!

Java Generics

- Generics get translated into regular code with casts
    - but compiler automatically ensures that casts will succeed. *No danger of exceptions*!
      - ```
        class Foo<T> { T meth (T t) { ... } }
        Foo<Dog> fd = new Foo<Dog>;
        Dog rex = fd.meth(new Dog());
                becomes
        class Foo { Object meth(Object o) { ... } }
        Foo fd = new Foo();
        Dog rex = (Dog) fd.meth(new Dog());
        ```
  - I will avoid generics, but use them if you want

Data Structures: ArraySet

- Let's implement the Set ADT with an array
  - see code in book (download) throughout these slides
  - you should be ***crystal clear*** on how to implement something like this
  - class ArraySet implements SetADT {

```
class ArraySet implements SetADT {
  ...
  private Object[] contents;
  private int count;
  public ArraySet() { count = 0; contents = new Object[100]; }
  public void add (Object ele) {
    if (!contains(ele)) {
      if (count == contents.length) expandCapacity();
      contents[count++] = element;
    }
  }
  ...
}
```

  - remove? expandCapacity? union? removeRandom?

Array Considerations

- When implementing something with an array, need to handle case that array is full
  - 3 options:
    - pass "full" flag to user as return value
      - e.g., boolean add(Object ele) {...}
    - throw exception
      - e.g., void add(Object ele) throws DSFullException {...}
    - resize
      - allocate new array, copy old data, drop old reference

Linked Data Structures

- Remember recursive types? These are the basis of “linked data structures”
  - – class Record { String name; Record next; ... }
  - all data structures are:
    - array-based (i.e., use contiguous memory)
    - or linked (i.e., use interspersed objects with links)
    - or hybrid (i.e., use arrays + links)
  - linked structures are good because there are no space constraints: just add objects wherever and link them

LinkedSet

- Let’s implement the Set ADT as a linked data structure
  - class LinkedNode { LinkedNode next; Object ele; }

```
class LinkedSet implements SetADT {
  private LinkedNode head;
  private int count;
  public void add (Object ele) {
    if (!contains(ele)) {
     LinkedNode n = new LinkedNode();
     n.ele = ele;   // best through constructor
     n.next = head;
     head = n;
     count++;
    }
  } ...
}
```

  - remove? (this is harder)
  - advantages? What if *doubly-linked*?

Stack ADT

- A Stack is probably the simplest ADT
  - LIFO policy
  - used everywhere: for recursion, for tree processing, for graph algorithms
    - see PostFix.java, MazeSearch.java for long examples
    - interface StackADT {

```
interface StackADT {
  void push (Object ele);
  Object pop();
  Object peek();
  boolean isEmpty();
  int size();
}
```

  - java has a java.util.Stack

ArrayStack

- Let's implement the Stack ADT with an array

```
class ArrayStack implements StackADT {
  private int top;
  private Object[] data;
  public ArrayStack() { top = 0; data = new Object[100]; }
  public void push (Object ele) {
    if (top == data.length()) expandCapacity();
    stack[top++] = ele;
  }
  public Object pop() throws EmptyStackException {
    if (isEmpty())
      throw new EmptyStackException();
    Object result = stack[--top];
    stack[top] = null; // why ?
    return result;
  } ... // expandCapacity? peek?
}
```

LinkedStack

- Let's implement the Stack ADT with links
  - ```
    class LinkedStack implements StackADT {
      private LinkedNode top;
      public void push(Object ele) {
        LinkedNode temp = new LinkedNode();
        temp.ele = ele; // best through constructor
        temp.next = top;
        top = temp;
      }
      public Object pop() throws EmptyStackException {
        if (isEmpty()) throw new EmptyStackException();
        Object result = top.ele;
        top = top.next;
        return result;
      } ...
    }
    ```


Queue ADT

- A queue ADT supports the FIFO policy
  - serve jobs by priority, explore solutions in a breadth-first way
    - see extended examples in book
  - ```
    interface QueueADT {
      void enqueue (Object ele);
      Object dequeue();
      Object first();
      boolean isEmpty();
      int size();
    }
    ```

ArrayQueue

- Can do it in a braindead way using first index = 0
  - very slow for dequeues: need to shift all
  - instead: circular array
  - class ArrayQueue implements QueueADT {

```
class ArrayQueue implements QueueADT {
  private int rear, front;
  private Object[] queue;
  public boolean isEmpty() { return rear == front; }
  public boolean isFull()
   { return (rear + 1) % queue.length == front; }
  public ArrayQueue() { queue = new Object[100]; }
  public void enqueue(Object ele) {
    if (isFull()) expandCapacity();
    queue[rear]  = ele;
    rear = (rear + 1) % queue.length;
  }
  public Object dequeue() throws EmptyQueueException {
    if (isEmpty()) throw new EmptyQueueException();
     Object ret = queue[front];
      front = (front + 1) % queue.length;
      return ret; // ideally also nullify reference
  }
}
```


LinkedQueue

- class LinkedQueue implements QueueADT {

```
class LinkedQueue implements QueueADT {
  private LinkedNode front, rear;
  public void enqueue(Object ele) {
    LinkedNode temp = new LinkedNode();
    temp.ele = ele; // best through constructor
    if (rear != null) rear.next = temp;
    if (front == null) front = temp;
    rear = temp;
  }
  public Object dequeue() throws EmptyQueueException {
    if (isEmpty()) throw new EmptyQueueException();
    Object result = front.ele;
    front = front.next;
    if (front == null)
      rear = null;
    return result;
  } ...
}
```

Lists

- A (linear) list is a simple kind of ADT
- There are many different list ADTs
  - implicitly ordered lists: elements in order according to some intrinsic characteristic
  - explicitly ordered lists: “addAfter” operation, possibly “next” operation
  - indexed lists: each element has a numeric index
    - this is a list because an element’s index changes when new elements get inserted before it!
    - ArrayList!

Indexed List ADT

- Interesting point: can’t tell from interface that this is a list. Other data types (e.g., an int-map) have very similar interface, but permanently associate indices with objects
  - interface IndexedListADT {

```
interface IndexedListADT {
  Object remove (Object ele);
  Object first();
  Object last();
  ...
  void add(int index, Object ele);
  void set(int index, Object ele);
  void add(Object ele); // add to end
  Object get(int index);
  int indexOf(Object ele);
  Object remove(int index);
}
```

Indexed Lists with Arrays

- This is exactly like an ArrayList
  - See the actual Sun code ArrayList.java in class
- – e.g.,
  - public Object remove(Object ele) throws NotFoundException {

```
public Object remove(Object ele) throws NotFoundException {
  int i = 0;
  for ( ; i < arr.length; i++)
    if (ele.equals(arr[i])) break;
  if (i == arr.length) throw new NotFoundException();
  Object result = arr[i];
  rear--;
  for (int scan = i; scan < rear; scan++) arr[scan] = arr[scan+1];
  arr[rear] = null;
  return result;
}
```


Lists with Single Links

- All lists can be implemented with single links, but some operations are slower, while others are cumbersome
  - e.g., remove (this is hard code because of the special cases! ***Try to write and test it yourselves***!)
- – Object remove(Object ele) throws EmptyListException,NotFoundException {

```
Object
remove(Object ele) throws EmptyListException,NotFoundException {
  if (isEmpty()) throw new EmptyListException();
  LinkedNode prev = null, cur = head;
  while (cur != null) {
    if (ele.equals(cur.ele)) break;
    prev = cur; cur = cur.next;
  }
  if (cur == null) throw new NotFoundException();
  if (head == tail) head = tail = null;
  else if (cur == head) head = cur.next;
     else if (cur == tail) { tail = prev; tail.next = null; }
       else prev.next = cur.next;
  return cur.ele;
}
```

Doubly Linked Lists

- Instead: doubly-linked lists
  - class DoubleNode { DoubleNode next, prev; Object ele; }
  - Object remove(Object ele)

```
Object remove(Object ele)
throws EmptyListException,NotFoundException
{
  if (isEmpty()) throw new EmptyListException();
  DoubleNode cur = head;
  for ( ; cur != null; cur = cur.next)
    if (ele.equals(cur.ele)) break;
  if (cur == null) throw new NotFoundException();
  if (cur == head) head = cur.next;
  else cur.prev.next = cur.next;
  if (cur == tail) tail = cur.prev;
  else cur.next.prev = cur.prev;
  return cur.ele;
}
```


More Advanced Data Structures and Algorithms

"Cool ideas for high performance"

Hash Tables

- Hash tables are a variety of hybrid data structures: using an array and linked nodes
- Very common, used to implement lots of useful ADTs: sets, maps
    - set ADT: recall that the array or linked implementations we saw are very slow when searching ("contains"), removing, etc.
    - map ADT: stores <key,value> pairs, then can use key to retrieve value

Hash Table Elements

- Main idea: assign elements to *buckets* according to some *hash function*, resolve *collisions* in some way
  - hash function: even simple is often good
    - e.g.,
      - modulo of table size
      - sum of squares of letter values
      - selecting specific digits
    - hashcode method in Object
  - collision resolution: even simple is often good
    - e.g., linked lists at each bucket (*chaining*)
- Java has Hashtable, HashSet, HashMap, ... classes pre-defined

Example Code in a HashSet with Chaining

- Just use simple linked list routines (add, contains) and a front-end that selects the right list
  - class HashSet implements SetADT {

```
class HashSet implements SetADT {
  private LinkedList [] table;
  HashSet () { table = new LinkedList[97]; }  // primes are good
  void add (Object ele) {
    int hc = ele.hashcode() % table.length;
    LinkedList lili = table[hc];
    lili.addFront(ele);
  }
  boolean contains(Object ele) {
    int hc = ele.hashcode() % table.length;
    return table[hc].contains(ele);
  }
  ...
}
```


Trees

- Trees are hierarchical recursive data structures
  - richer than linked lists (a linked list is a limited tree)
  - many different kinds, used to implement various ADTs
  - used in parsing language (syntax trees), organizing information (e.g., databases), creating fast dictionaries (compete with hash tables), biology
  - interesting concepts: *balanced*, *complete*, *binary*

```
class BinaryTreeNode {
  Object ele;
  BinaryTreeNode left, right;   // called “children”
}
```

Tree Traversals

- Inorder: visit left child, then parent node, then other children
- Postorder: visit all children before parent
- Preorder: visit parent, then children
- Level-order: visit nodes one level at a time

Level-Order Implementation

- Requires a queue:
  - void levelOrder(BinaryTreeNode root) {

```
void levelOrder(BinaryTreeNode root) {
  Queue q = new Queue();
  q.enqueue(root);
  while (!q.isEmpty()) {
    BinaryTreeNode n =
      (BinaryTreeNode) q.dequeue();
    n.process();  // e.g., print it
    if (n.left != null) q.enqueue(n.left);
    if (n.right != null) q.enqueue(n.right);
  }
}
```

Other Traversal Implementations

- Trivial with recursion, otherwise need a stack
  - void inOrder(BinaryTreeNode n) {

```
  if (n == null) return;
  inOrder(n.left);
  n.process();  // e.g., print it
  inOrder(n.right);
}
```

  - void preOrder(BinaryTreeNode n) {

```
  if (n == null) return;
  n.process();  // e.g., print it
  preOrder(n.left);
  preOrder(n.right);
}
```

  - how would you do a search for a given element?

Binary Search Trees (BSTs)

- A tree data structure with the property that every parent node has value greater than its left child and less than its right
  - class BSTNode {

```
  Comparable ele;
  BSTNode left, right;   // called “children”
}
```

- Used to implement many ADTs (including Sets, Maps)
  - example operations:
    - void addElement (Comparable element);
    - Comparable removeElement (Comparable targetElement);
    - Comparable removeMin();
    - Comparable removeMax();
    - Comparable findMin();
    - Comparable findMax();

Some Operations

- void addElement(Comparable o) {
```
  BSTNode temp = new BSTNode();
  temp.ele = o;
  if (isEmpty()) root = temp;
  else {
    BSTNode cur = root;
    while (true) {
      if (o.compareTo(cur.ele)) < 0)
        if (cur.left == null) { cur.left = temp; break; } else cur = cur.left;
      else
        if (cur.right == null) { cur.right = temp; break; } else cur = cur.right;
     }
  }
}
```
- Comparable findMin() {
```
  BSTNode cur = root;
  if (cur == null) return null;
  while (cur.left != null) cur = cur.left;
  return cur.ele;
}
```


More Operations

- removeElement and removeMin are tricky
  - need to replace removed node with other
  - removeElement, 3 cases:
    - no children (no need to replace)
    - one child (used as replacement)
    - two children (inorder successor)
  - removeMin, 3 cases:
    - root has no left child (is minimum, right child becomes root)
    - leftmost node (minimum) is leaf (no replacement)
    - leftmost node has right child (replace minimum node with right child)

More on BSTs

- Use BSTs to implement ordered list ADT: fast (logn) add, remove, etc., provided the tree stays balanced!
- Special kinds of BSTs are self-balancing
  – AVL trees, red-black trees
- Such trees are great for implementing Set and Map ADTs
  – guaranteed performance bound, unlike hash table
  – TreeSet and TreeMap in Java

Graphs

- The most general data structure is a *graph*
  – no limitations, no hierarchy
  – a tree is a special kind of graph
  – graphs used to represent all kinds of data
    - internet and other networks (e.g., power grid)
    - data dependencies (e.g., biology)
- Concepts: undirected, directed, weights, nodes (vertices), edges
- Traversals: BFT, DFT
  – very much the same as tree traversals, only remembering what we have visited

Representation

- Two main kinds:
  - adjacency lists
    - saves space for sparse graphs, but is slower for some kinds of algorithms
      - ```
        class GraphNode {
          GraphNode [] neighbors;
        }
        ```
  - adjacency matrices
    - too big for sparse graphs, but many algorithms are matrix-based
      - ```
        boolean [][] adjacency;
        // all nodes are numbered
        ```


Breadth-First Traversal

- Level-Order in trees was a BFT!
  - ```
    class GraphNode {
      GraphNode [] neighbors;
      Object ele;
      boolean visited = false;
    }
    ```
  - ```
    void BFT(GraphNode node) {
      Queue q = new Queue();
      q.enqueue(node);
      while (!q.isEmpty()) {
        GraphNode n = (GraphNode) q.dequeue();
        if (n.visited) continue;
        n.process();  // e.g., print it
        n.visited = true;
        for (int i = 0; i < n.neighbors.length; i++)
          q.enqueue(n.neighbors[i]);
      }
    }
    ```

Depth-First Traversal

- Just as in-order, post-order, etc. in trees
- Used for many graph algorithms (e.g., connected components)
  - void DFT(GraphNode n) {

```
void DFT(GraphNode n) {
    if (n == null || n.visited) return;
    n.process();  // e.g., print it
    n.visited = true;
    for (int i = 0; i < n.neighbors.length; i++)
      DFT(n.neighbors[i]);
}
```


Spanning Trees

- Spanning tree = tree that spans all graph nodes
- Minimum spanning tree (on weighted graph) = tree with lowest total weight
- Two very nice algorithms for undirected graphs
- Kruskal's algorithm (O(e*log(e))):
  - keep adding globally minimum edge that keeps current structure a tree
- Prim's algorithm ($O(v^2)$)
  - select vertex that expands current MST with minimum edge