# COMP SCI 600.226: Data Structures

**Gregory D. Hager**

http://www.cs.jhu.edu/~hager/Teaching/cs226

# What this course is about

## Data structures:

conceptual and concrete ways to organize data for efficient storage and manipulation

**DESIGN = DATA STRUCTURES + ALGORITHMS**

# Some Example Data Structures

- Lists (linear structures) … most basic of structures
  - Add some rules -- a queue for using a resource
  - Can be extended – e.g. priority queue
- Tree (nonlinear structures)
  - Useful for fast search
  - Can be implemented in many ways depending on application properties
- Graphs
  - Very widely used as representation
    - WEB, roads, molecules, networks ….
  - Many complex problems reduce to traversals
- Tries, Sets, Binomial queues, ….

# Why do we need them

Computers take on more and more complex tasks

Software implementation and maintenance is difficult

Clean conceptual framework allows for more efficient & more correct code

Understanding basic principles provides for good code reuse (tools, tools, tools …)

# A Brief Conceptual Example

**Design a WEB Search Engine (JHUgle)**

**Basic ideas:**

**We want to store terms and URLs**

**We want to look them up**

**We want to rank the results**

# Specification: What Functionality?

```
public interface TermSet {
        Boolean addTerm(String term, String URL);
        TermSet lookUp(String term);
        }
```

This uses the Java notion of an interface which corresponds roughly to what we'll refer to as an Abstract Data Type (ADT)

# Implementation: How to Support Functionality

```
public  class ArrayTermSet implements TermSet {
   int nelements;
   String[] terms;
   String[] URLs;
   ArrayTermSet(int nelements) // ... allocate things ...
   Boolean addTerm(String term, String URL)  // add new term
   TermSet lookUp(String term) // find a term
}
```

# Implementation: How to Support Functionality

```
public boolean addTerm(String term, String URL) {
  if (nelements < terms.length) {
      terms[nelements] = term;
      URLs[nelements] = URL;
      nelements++;
      return true;
  }
  else
      return false;
  }
```

# Implementation: How to Support Functionality

```
public TermSet lookUp(String term)  {
    int [] indices = new int[nelements];
    int nfound = 0;
    for (int i=0;i<nelements;i++) {
        if (terms[i] == term) {
            indices[nfound]=i;
            nfound++;
        }
    }

    ArrayTermSet x = new ArrayTermSet(nfound);
    for (int i=0;i<nfound;i++) {
        if (!x.addTerm(terms[indices[i]],URLs[indices[i]]))
            System.exit(0);
        }

    return x;
  }
```

# Can We Do Better?

**How do we make ArrayTermSet better?**

# Can We Do Better?

**How do we make ArrayTermSet better?**

- **Fixed size array (what if it fills)?**
- **Doesn't capture common URLs for term**
- **Serial search (what is faster)?**
- **Simple term queries (and, or ...)**
- **More set operations (^ ~ ∨)**
- **System memory architecture?**

# What Else Should We Support?

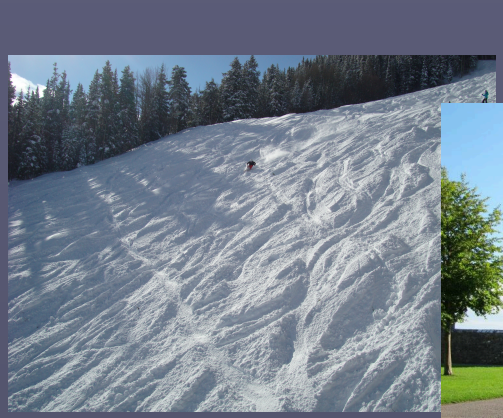
Evaluation functions: we want to rank pages based on attributes

- many terms/page (need to store more info)
- number of pages that point here (expert page)
- how often a page is accessed
- reverse lookup (find terms at URL)

Other things we haven't yet thought of …

# Another Example

Find pairs of identical images in a photo library





# The general lesson

**Good data structures mean:**

- **clear design**
- **efficiency (space and time!)**
- **generality**
- **extensibility**
- **scalability**

# What you will learn

**What are some of the common data structures**

**What are some ways we can implement them**

**How can we analyze their efficiency**

**How can we use them to solve some practical problems**

# Tool Box

Known data structures are tools for solving your future problems

Libraries and packages contain some debugged implementations of these structures

# Why do we study this?

**Argument against:**

- **Packages are already written**
- **Why not just read documentation of their interfaces and use them?**

**Argument for:**

# Some arguments for

**The more you know, the better you can choose the tools**

**You can modify tools**

**You can create entirely new tools**

**You are to become the experts!**

# Prerequisites

**At least one semester of C++ or Java programming**

**Easier if you've got two semesters**

# What you need

**Textbook**

- **Weiss. *Data Structures and Algorithm Analysis in Java, 3nd edition 2011.***

**Java reference (handy, not required)**

- **Arnold, Gosling, and Holmes. *The Java Programming Language, 4th edition.* 2005.**

**Computer with Java installation**

# What you'll be graded on

**20% : Written assignments**

**40% : Programming assignments**

**15% : Mid-term**

**25% : Final**

# What NOT to do

**Plagiarism**

**Procrastination**

# A Few Java Essentials (Chap 1 of Weiss)

(Review through 1.3 on your own)

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Greg Hager

# Java Features

- Platform independent
  - "Write once, run anywhere"
- Object-oriented
- Safe references
  - Class casting checked at run-time
  - No dangerous pointer manipulations
- Garbage collection
- Built-in exception handling
- Support for multi-threading, networking, security, web applets, etc.

# Some Shortcomings

**Rather slow compared to fully-compiled code**

- **Changing with on-the-fly compilation technology**

**Some unavoidable space inefficiencies**

- **No arrays of classes without references**

**Difficult to take advantage of platform-specific features**

# Java Tools

**javac : Java byte code compiler**

- Compiles Java source to platform-independent byte codes

**java : Java run-time environment**

- Verifies byte codes for security correctness and executes on Java Virtual Machine

**jdb : Java debugger**

**jGRASP: Graphical debugging environment**

- written in Java, for Java

# Java Basic Syntax

**I assume you know this. If you've forgotten or never knew it, I suggest you review it as we will *not* be covering it in this class**

# Some Basics

**What happens in the following lines of code?**

```
int x;
Integer x = new Integer(3);
Object[] x = new Integer[5];
((Integer)x[0]).toString();
```

# Classes

Combine *fields* (variables) and *methods* (procedures)
Fields and methods accessed by . (dot) operator

```
class MyClass {
  static int numInstances=0;
  protected int somethingImportant;
  public int tellAll() {
    return somethingImportant;
  }
}

MyClass myVar = new MyClass;
System.out.println(MyClass.numInstances +
  ", " + myVar.tellAll());
```

# Field Instantiation

## Class fields

- static - only one per class
- May be accessed without a class variable
  - <classname>.<static field>
    - e.g. Math.PI

## Instance fields

- non-static - one per class instance

# Field and Method Visibility

**public, protected, private, or "package" (default)**

| Accessible to: | public | protected | package | private |
|---|---|---|---|---|
| same class | yes | yes | yes | yes |
| class in same package | yes | yes | yes | no |
| subclass in different package | yes | yes | no | no |
| non-subclass, different package | yes | no | no | no |

# Inheritance

**Enables class extensions with reuse of some fields and methods**

- **All parent fields included in child instantiation**
- **Protected and public fields and methods directly accessible to child**
- **Parent methods may be *overridden***
- **New fields and methods may be added to child**
- **Only single inheritance (unlike C++)**

# Simple Inheritance Example

```
Class MyExtension extends MyClass {
  float newField;
  MyExtension() {
    super(); //call parent constructor
    newField = super.tellAll()*3.14;}
  int tellAll(){return (int)newField;}
}

MyClass foo = new MyExtension;
System.out.println(foo.tellAll());
```

**Method accessed is that of actual instantiation type, not variable type**

- **In C++ terminology, all functions are "virtual"**

# Casting of Class Variables

## "upward" casting

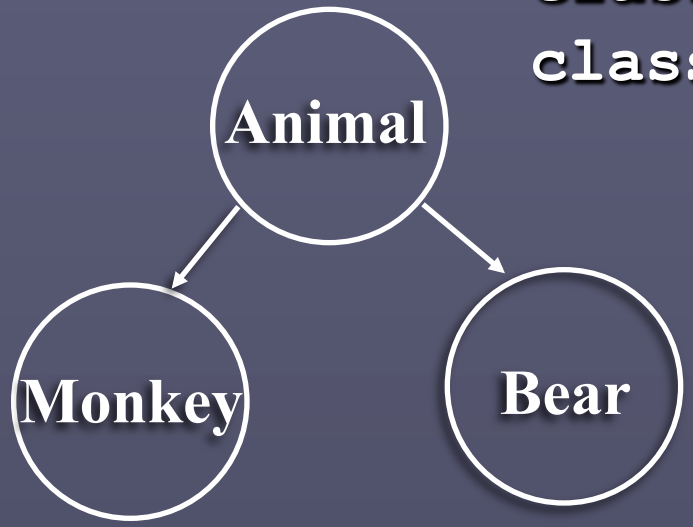
- Casting derived class variable to ancestor class is always safe (and may be done implicitly)

## "downward" casting

- Casting class variable to derived class fails if variable is not actually an instance of the derived class
  - —run-time error
- `instanceof` operator can be used to test class type before downward cast

# Class Casting Example

```
class Animal {...}
class Bear extends Animal {...}
class Monkey extends Animal {...}
```




```
Animal a;
Monkey m;
Bear b;
```

```
a = new Bear();      // legal
b = (Bear)a;         // legal
m = (Monkey)a;       // illegal
m = (Monkey)b;       // illegal
```

# Interfaces

```
interface Printable {
  print(); }
class Foo implements Printable {
  print(){...}; }
```

Similar to abstract classes

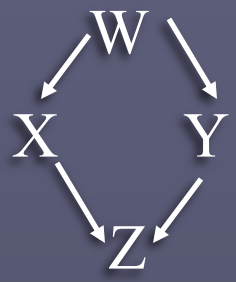
- But *no* methods implemented or fields specified

Class can be defined to *implement* one or more interfaces

- More general mechanism than just single inheritance

Variables may actually use interface as type

# "Multiple Inheritance"




**Several ways to achieve in Java**

- **combinations of interfaces and classes**
- **W and Y are interfaces, X and Z are classes**
- **W, X, and Y are interfaces, Z is class**

# Genericity

**Generic programming is a style ofcomputer programming in which algorithms are written in terms of *to-be-specified-later types that are then instantiated when needed for specific types provided as parameters.***

# Implementing Generic Components

One goal of any good design is to apply concepts as broadly as possible – i.e. independent of details of data or implementation. Java provides three mechanisms:

- **Use Object class**
- **Use Interfaces**
- **Use Generics**

# Genericity Using Objects

```
public class MemoryCell{
        public Object read( )          { return storedValue; }
        public void write( Object x )  { storedValue = x; }

        private Object storedValue;}


public class TestMemoryCell{
   public static void main( String [ ] args )
     {
       MemoryCell m = new MemoryCell( );
       m.write( "57" );
       String val = (String) m.read( );
       System.out.println( "Contents are: " + val );
}}
```

# Genericity Using Interfaces

```
public static Comparable findMax( Comparable [ ] a )
{
    int maxIndex = 0;
    for( int i = 1; i < a.length; i++ )
        if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
            maxIndex = i;
    return a[ maxIndex ];
}

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

# Genericity Using Generics

```
public class  Thing<SomeRefType>
{
  private SomeRefType thing;

  public SomeRefType get()          { return thing; }
  public void set (SomeRefType t) { thing = t; }

  public static void main(String[] args)
    {
      Thing<Integer> ithing = new  Thing<Integer>();
      ithing.set(24);                          // uses autoboxing
      int i = ithing.get();                    // auto-unboxing
      System.out.println(ithing.get());
      Thing<Card> cthing = new Thing<Card>();
     cthing.set(new Card("Jack", "Diamonds"));
     System.out.println(cthing.get());
}}
```

# Genericity Using Generics

```
public class Pair<K, V>
{  K key;  V value;

   public void set(K k, V v) {    key = k;    value = v;  }
   public K getKey() { return key; }
   public V getValue() { return value; }
   public String toString() { return "[" + getKey() + ", " + getValue() + "]";  }

   public static void main (String[] args) {
            Pair<String,Integer> pair1 = new  Pair<String,Integer>();
            pair1.set(new String("height"), new Integer(36));
            System.out.println(pair1);
            Pair<Student,Double> pair2 = new  Pair<Student,Double>();
            pair2.set(new Student("A5976","Sue",19), new Double(9.5));
            System.out.println(pair2);
}}
```

# Genericity Using Generics

```
public interface Comparable<AnyType>
{
        public int compareto ( AnyType other)
}
```

Now --- we can catch errors at compile time rather than runtime!

# New in Java 7

Instantiating generic objects requires full type specification

Thing<Integer> x = new Thing<Integer>

Isn't the type of "new" obvious?

Thing<Integer> x = new Thing<>

# Covariance

A type A is **covariant** if it preserves type ordering

- if student isa person, then student [] isa person []

```
person [] arr = new student [ 3 ];
arr[0] = new faculty [ ..  ] ;
```

- This compiles, but throws a runtime exception

- Generics are *not* covariant

```
foo someFunction( Collection<person> z) {…}
Collection<student> x = ….
y = someFunction( x ) // ← compile time error!
```

# Wildcards

- Generics are *not* covariant

  foo someFunction( Collection<person> z ) {…}

  Collection<student> x = ….

  y = someFunction( x ) // ← compile time error!

- Another try …

  foo someFunction( Collection<? extends person> z ) {…}

  Collection<student> x = ….

  y = someFunction( x ) // Now we are ok!

# Type Bounds

```
public static <   what goes here?  >
AnyType findMax( AnyType [ ] a )
{       int maxIndex = 0;
         for( int i = 1; i < a.length; i++ )
              if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
                   maxIndex = i;
          return a[ maxIndex ];
}
```

# Type Bounds

```
public static <  AnyType extends Comparable<? super AnyType>
AnyType findMax( AnyType [ ] a )
{       int maxIndex = 0;
        for( int i = 1; i < a.length; i++ )
            if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
                maxIndex = i;
        return a[ maxIndex ];
}
```

WOW!!!

# Genericity Using Function Objects

```
public static <AnyType>
AnyType findMax( AnyType [ ] a, Comparator<? super AnyType> cmp )
{
      int maxIndex = 0;
          for( int i = 1; i < a.length; i++ )
                if( cmp.compareTo(a[ i ], a[ maxIndex ] ) > 0 )
                          maxIndex = i;
           return a[ maxIndex ];
}

public interface Comparator<AnyType>
{
            int compareTo(  AnyType lhs, AnyType rhs );
}
```

# Your "Homework"

- Get a Book!
- Get a computer account and/or make sure you can access Java
- Type in and test "Hello World"
- Read the first two chapters of the book