CollegeBoard
Advanced Placement Program

AP® Computer Science

2006–2007
Professional Development
Workshop Materials

Special Focus:
Using the Java Collections Hierarchy

connect to college success™
www.collegeboard.com

CollegeBoard
Advanced Placement Program

AP® Computer Science

2006–2007
Professional Development
Workshop Materials

Special Focus:
Using the Java Collections Hierarchy

connect to college success™
www.collegeboard.com

The College Board: Connecting Students to College Success

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the association is composed of more than 5,000 schools, colleges, universities, and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,500 colleges through major programs and services in college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT®, and the Advanced Placement Program® (AP®). The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities, and concerns.

Equity Policy Statement

The College Board and the Advanced Placement Program encourage teachers, AP Coordinators, and school administrators to make equitable access a guiding principle for their AP programs. The College Board is committed to the principle that all students deserve an opportunity to participate in rigorous and academically challenging courses and programs. All students who are willing to accept the challenge of a rigorous academic curriculum should be considered for admission to AP courses. The Board encourages the elimination of barriers that restrict access to AP courses for students from ethnic, racial, and socioeconomic groups that have been traditionally underrepresented in the AP Program. Schools should make every effort to ensure that their AP classes reflect the diversity of their student population. For more information about equity and access in principle and practice, contact the National Office in New York.

Visit the College Board on the Web: www.collegeboard.com.
AP Central is the official online home for the AP Program and Pre-AP: apcentral.collegeboard.com.

Table of Contents

Special Focus: Using the Java Collections Hierarchy

Important Note: The following set of materials is organized around a particular theme, or "special focus," that reflects important topics in the AP Computer Science course. The materials are intended to provide teachers with resources and classroom ideas relating to these topics. The special focus, as well as the specific content of the materials, cannot and should not be taken as an indication that a particular topic will appear on the AP Exam.

Introduction

Fran Trees, editor
Drew University
Madison, New Jersey

Java includes a **collections** framework. A **collection** is an object that represents a group of objects.

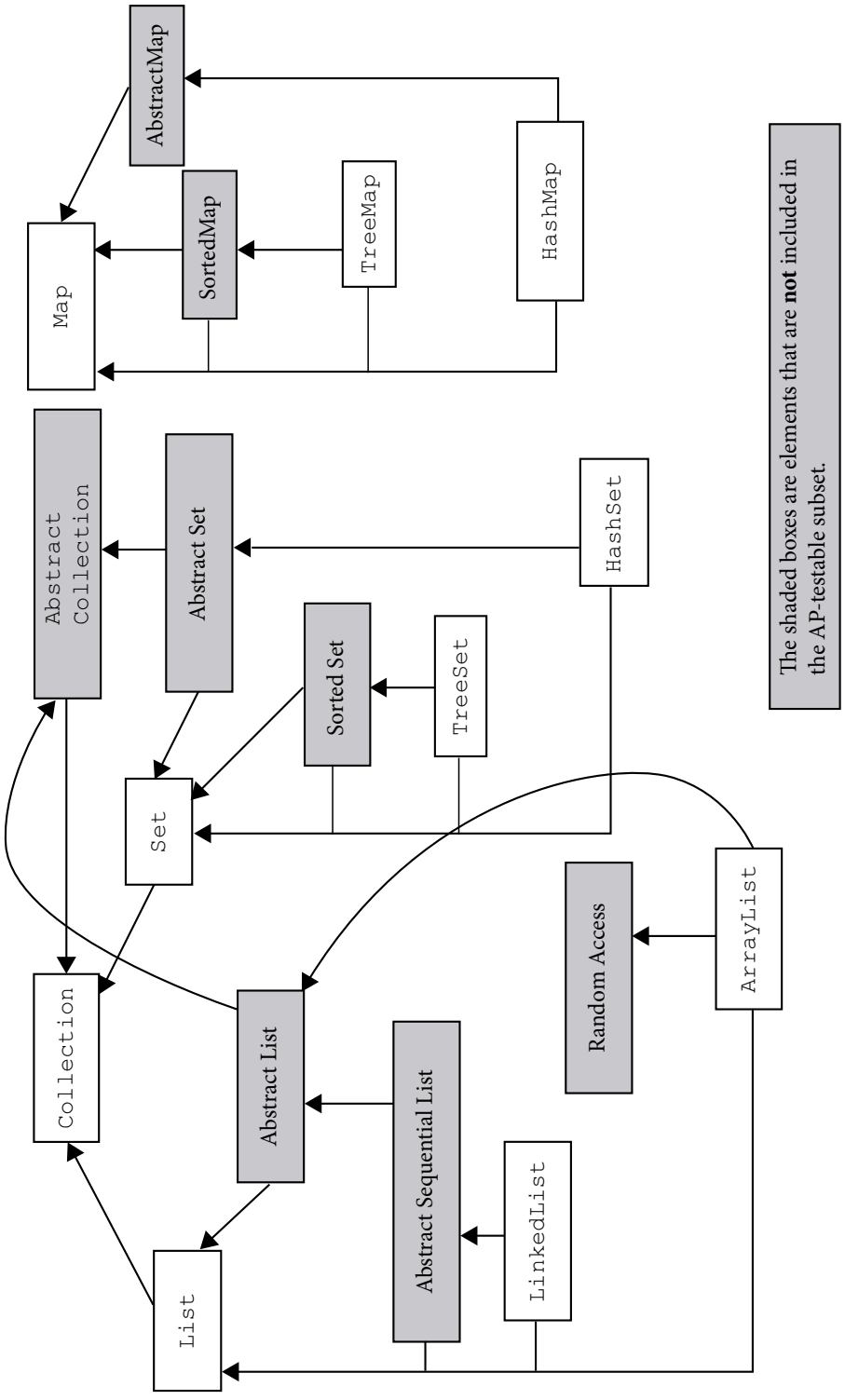
From the Java API: "The collections framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation. It reduces programming effort while increasing performance. It allows for interoperability among unrelated APIs, reduces effort in designing and learning new APIs, and fosters software reuse."

The representation of a simplified Java hierarchy included on the next page contains more than is studied in AP Computer Science and less than is contained in the complete Java hierarchy. The diagram is intended to give you an overview of a much bigger picture. The interfaces studied in AP Computer Science that are part of the basic collections framework are `Set`, `List`, and `Map`. Because `Map` represents mappings more than simple collections, it does not extend `Collection`. However, `Map` can be manipulated and viewed in similar ways to collections, so it is usually included with the hierarchy.

The primary implementations of the collection interfaces that we study in AP Computer Science are `HashSet`, `TreeSet`, `ArrayList`, `LinkedList`, `HashMap`, and `TreeMap`.

We hope that the materials contained in this section help you in the classroom and, more importantly, provide ideas and directions that will assist you in the development of your own materials. We have described teaching strategies and included sample lab assignments, problems, and worksheets. Also included is an introduction to Java 5 (Tiger) as it relates to the collections hierarchy and finally a list of Web-based resources that point to materials focusing on collections. I would like to thank the contributors for their hard work and continued commitment to our AP Computer Science family.

The Java Collections Hierarchy



Array Lists

Pat Philips
Craig High School
Janesville, Wisconsin

This review of array lists is meant to provide
- An introduction to `ArrayList` class
- Method analysis of `ArrayList` class
- Manipulation of data in array lists
- Lab practice using array lists with authentic data

The PowerPoint for this review is available on AP Central at http://apcentral.collegeboard.com/workshop/csfiles. The materials included below are meant to accompany and supplement this online presentation.

`ArrayList` Authentic Data Project

Name: _____

Complete these steps:

- View power point on the `ArrayList` class.
- Create an application that meets these specifications:
  _____Uses authentic data collected from almanacs on line
  _____Uses this data to design a class (object design)
  _____Accesses a text file to retrieve data used to instantiate objects
  _____Loads objects into an `ArrayList`
  _____Allows user to select from a menu (switch) of statistical calculations
  _____Traverses the `ArrayList` to calculate statistics
  _____Produces readable output of selected statistics

Idea

- Collect data on automobile crash tests by model.
- Create a text file that might contain information like this:
  LandRover, Model_X, 4000, ($ damage) 2, (injury rating)
  for several cars
- Design an "Auto" class with fields to store required data.
- Create methods including toString, compareTo, and accessors.
- You might want to allow the user to change the data fields of a given element This will require additional mutator methods.
- Create the main program logic to offer menu choices for display of data and statistical analysis such as mean, median, mode, greatest, least, sorts, and so on. Select those that are most meaningful for the data chosen.
- Allow the user to add additional objects to the ArrayList.
- Allow the user to remove objects from the ArrayList.
- Allow the user to save the ArrayList data back to a text file.
- Be sure the program allows for easy-to-read output with ample description of results.

Step 1: Browse the online almanacs to select a topic and gather data.

Step 2: My topic is _____, and a text file of data is attached.

Step 3: The name of my class is _____.
The fields needed for the class design include:

Step 4: Draw the UML (Unified Modeling Language) diagram for this project on the back of this page.

Step 5: Create the class code. Attach.

Step 6: The statistical analysis measures I will use:

Statistical Measure	Why This Is Useful for This Data

Step 7: Create the code for your application. Attach.

Step 8: Self-evaluate with the attached rubric.

Special Focus: Using the Java Collections Hierarchy

Selected Methods Using `ArrayList`

(These methods were chosen from a variety of projects. They are not intended to be a complete class but rather examples of various techniques created by students.)

Loading `ArrayList` with Data from a Text File

```
        int count = 0;
    String line;
    BufferedReader infile = new BufferedReader(new
    FileReader("bowlgames.txt"));
    ArrayList bowlgame = new ArrayList(10);
    line = infile.readLine();

    while (line != null)
    {
       while(count < 10)
       {
          String whichbowl = line;
          String team1 = infile.readLine();
          double score1 =
Double.valueOf(infile.readLine()).doubleValue();
          String team2 = infile.readLine();
          double score2 =
Double.valueOf(infile.readLine()).doubleValue();
          BowlGame game = new BowlGame(whichbowl, team1,
                score1,team2, score2);
          bowlgame.add(game);
          line =infile.readLine();
          count++;
       }
```

Finding an Average

```
public static void average(ArrayList incominglist)
{
   double Score1;
   double Score2;
   int index;
   double Score3= 0;
   double Score4;
   int count = 0;
   for(index = 0; index < 10; index++)
   {
   BowlGame game2 = (BowlGame)incominglist.get(index);
   Score1 = game2.getScore1();
```

```
   Score2 = game2.getScore2();
   count+= 2;
   Score3 += Score1 + Score2;
   }
   Score4 = Score3/count;
   System.out.println("average score of all Bowl Game Scores "
         + Score4);
}
```

Finding Greatest Value in an `ArrayList`

```
public static void highest(ArrayList incominglist)
{
   double max = 0;
   double Score1;
   double Score2;
   int index;
   for(index = 0; index < 10; index++)
   {
   BowlGame game2 = (BowlGame)incominglist.get(index);
   Score1 = game2.getScore1();
   if (Score1 > max)
      max = Score1;
   Score2 = game2.getScore2();
   if (Score2 > max)
      max = Score2;
   }
   System.out.println("The greatest score in list " + max);
}
```

Sorting an `ArrayList`

```
private static void sort() throws IOException
{
   System.out.println("----------------------------------------");
System.out.println("Sort Menu:" +
   "\n" + "-'d' to sort Descending" +
   "\n" + "-'a' to sort Ascending" +
   "\n" + "-'q' to quit:");
commandIn = (inRead.readLine()).charAt(0);
System.out.println(" ");
System.out.println(" ");
EntExpend localE;
int runHigh = 0;
int runLow = 0;
int run = 0;
int c;
int pastrun = 1000000;
```

```
switch (commandIn)
{
    case 'd':
          for (int i=0;i<List1.size();i++)
          {
              localE = (EntExpend)(List1.get(i));
              c=i;
              if (localE.isAge)
              {
                 run = localE.Agelow;
                    if (run >= pastrun)
                    {
                    runHigh = run;
                    }
                    if (run >= runHigh)
                    {
                       List1.remove(i);
                       List1.add(0,localE);
                    }
              }
              if (localE.isYear)
              {
                  run = localE.Exyear;
                  if (run >= pastrun)
                  {
                        runHigh = run;
                  }
                  if (run >= runHigh)
                  {
                        List1.remove(i);
                        List1.add(0,localE);
                  }
              }
              pastrun = run;
       }
       main();
       break;

   case 'a':
         for (int i=0;i<List1.size();i++)
         {
               localE = (EntExpend)(List1.get(i));
               c=i;
               if (localE.isAge)
               {
                     run = localE.Agelow;
                     if (run <= pastrun)
                     {
```

```
                        runLow = run;
                    }
                    if (run <= runLow)
                    {
                        List1.remove(i);
                        List1.add(0,localE);
                    }
                }
                if (localE.isYear)
                {
                    run = localE.Exyear;
                    if (run <= pastrun)
                    {
                        runLow = run;
                    }
                    if (run <= runLow)
                    {
                        List1.remove(i);
                        List1.add(0,localE);
                    }
                }
                pastrun = run;
            }
            main();
            break;
    }
```

ArrayList Project Rubric

Your Name:_____

Topic:_____

Rate each category according to the following scale: 9–10 = Excellent, 7–8 = Very good, 5–6 = Good, 3–4 = Satisfactory, 1–2 = Poor, and 0 = Unsatisfactory.

	Possible Points	Self-Assessment	Teacher Assessment
Topic is suitable for `ArrayList` development, and text file is correctly designed			
Project planning form is complete			
Statistics calculated are valuable to the data used and justified			
Two points for each useful statistic calculated			
Four points for each `ArrayList` method correctly and efficiently used			
Output and user interface is readable, understandable, and helpful			
Code is efficient			
Code is formatted and documented			
Test plan is thorough for both code and data			
Total Possible Points			

Collections and ObjectDraw: Using a Collection Class with Iteration in a Graphical Program

Leigh Ann Sudol
Fox Lane High School
Bedford, New York

Introduction

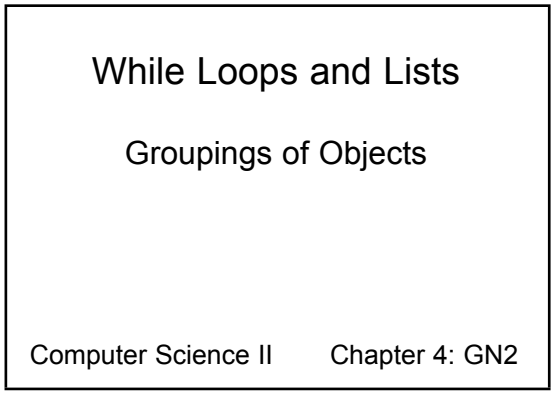
The `ObjectDraw` libraries were developed by Williams College as a way to easily integrate graphics into the introductory computer science classroom in order to teach basic principles. The `Collections` classes are part of the `java.util` package and contain implementations of lists, sets, and a variety of other data structures.

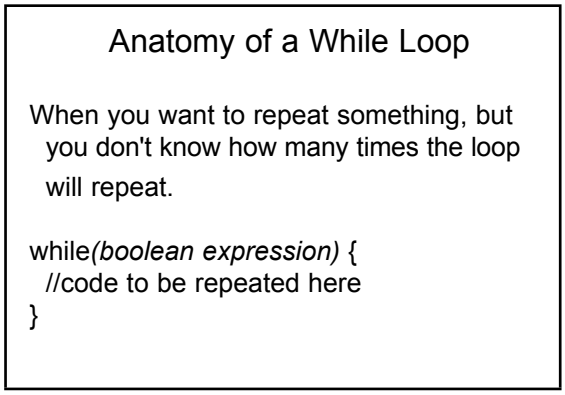
This lesson and activity is focused on combining the ease of using graphics with the `Collection` classes that are now part of the AP curriculum to provide students with a visual representation of iteration and a collection.

Description

The lesson includes several pieces and is meant to be addressed in the following manner.

PowerPoint
The PowerPoint presentation shown below outlines a sample program as well as some basics about the collection object `ArrayList` that is being used.





Special Focus: Using the Java Collections Hierarchy

Creating Groups of Objects

Previously when creating a series of dots or lines, we could only change their properties when they were first created, since there was one variable that represented them all.

We are going to use an ArrayList to hold *all* the objects (dots in this case) so that we can change their properties while the program is running.

What Is an ArrayList?

An ArrayList is a group of objects. Each object in the List has an index number.

Example with numbers:

List = {3, 4, 5, 6, 7}

Code for Creating a Set of Circles

```
import objectdraw.*;
import java.util.*;  <- needed for ArrayList

class Circles extends WindowController{

    private ArrayList circles = new ArrayList();

    public void begin(){
        for int( i = 0; i < 10; i++){
              circles.add(new FilledOval(i*10, i*10, 10, 10, canvas)

        }
//continued on next slide
```

Iterating Through the Set

We want to change one property of each of the circles. For this example, when clicked, the circles will move 10 pixels to the right.

//code on next slide

onMousePress

```
public void onMousePress(Location point){

   iterator myCircles = circles.iterator();
   //creates a way to move through the set
   while(myCircles.hasNext());
         ((FilledOval)myCircles.next()).move(10,0);
   }
}
//this code moves all the circles to the right 10
   pixels
```

Summary

- While loops are used when you don't know how many times the loop will execute.
- A List is a group of things.
- An Iterator is used to navigate a list.

Assignment

The students will be responsible for creating a program similar to the one described in the PowerPoint slides. The first part of the assignment requires students to create an `ArrayList` of circles and then iterate that `ArrayList` to change a single property of each item in the list (for example, the color of each circle) once a mouse is clicked.

The second part of the assignment requires students to think creatively to come up with a property to change on their own based upon a control that the user can implement (continuing with the first example, perhaps presenting a series of colored rectangles; when one rectangle is clicked, the color of all the circles changes to match the color of the selected rectangle).

Assignment

Phase I:

- Write a program that creates a pattern of shapes. When the user clicks on the screen, the program should change one property about the shape.

Phase II:

- Implement some type of control such that the user can influence the change (a place for them to click that performs a specific operation).

Students are given a file containing an outline for a class. They will need to instantiate a private `ArrayList` variable as well as write in the code to both create the `ArrayList` of shapes and modify the color of the shapes when the mouse is clicked.

```
public class Circles extends WindowController{

      //Create a private variable to store the ArrayList of
      //FilledOvals for the program.

      /**
       * Use this method to instantiate your ArrayList of circles.
       * Instantiate at least 30 circles and place them in the
       * ArrayList.
       **/
      public void begin(){

      }

      /**
       * Use this method to iterate through the ArrayList to change
       * the property.
       **/
      public void onMousePress(Location point){

      }
}
```

Extensions

There are several possible extensions to this program, depending upon the learning outcome desired at this point in the curriculum:

- To understand how the `Iterator` or `ArrayList` works, students could compose a role play for the program that they wrote, highlighting the use of the `ArrayList` and `Iterator`.
- To understand how looping works, the students could be asked to selectively choose which items to change (for example, every third circle).
- To understand random numbers and their application, students could assign a random color to the circles as a changed property (by either calling the `Color` constructor that receives an RGB value and using random integers for each of the three parameters, or by choosing a select list of colors—two or three—and having students pick a random number from 1 to the number of colors to choose the color with the help of if statements).
- For more practice with `ArrayLists`, students could add multiple `ArrayLists` with varying shapes (circles, rectangles, and so on).
- For an example of polymorphism, students could add both `FilledOval` and `FilledRect` objects to the `ArrayList`, and when iterated, the objects could be cast to `Drawable` before changing their color.

Lab Setup

Students will need the following for this program to be successful (in addition to a Java IDE):

1. The IDE used by students will need to be configured to use the `ObjectDraw` libraries. Information specific to your operating system and IDE can be found at http://applecore.cs.williams.edu/~cs134/eof/library.
2. Students will either need a paper copy or electronic copy of the setup file. This is not absolutely necessary but can be useful if `ObjectDraw` has not been used throughout the course.

Solution

The following solution has been implemented for phase 1 of the color example used in the "Assignment" description. (The solution code appears in boldface.)

```
public class Circles extends WindowController{

      //Create a private variable to store the ArrayList of
      //FilledOvals for the program.
      private ArrayList myCircles;

      /**
       * Use this method to instantiate your ArrayList of circles.
       * Instantiate at least 30 circles and place them in the
       * ArrayList.
       **/
      public void begin(){
            for(int i=0; i< 50; i++)
                  myCircles.add(new FilledOval(i*20, i*20, 10, 10,
canvas);    }

      /**
       * Use this method to iterate through the ArrayList to change
       * the property.
       **/
      public void onMousePress(Location point){
            Iterator it = myCircles.iterator();
            while(it.hasNext()){
                  FilledOval temp = (FilledOval)it.next();
                  temp.setColor(Color.red);
            }
      }
}
```

Reference

Bruce, Kim, Andrea Danyluk, and Thomas Murtagh. *Materials for Java: An Eventful Approach*. Prentice Hall, 2004. http://applecore.cs.williams.edu/~cs134/eof.

Special Focus: Using the Java Collections Hierarchy

Comparing Memory Representations Between ArrayList and LinkedList: Using the BlueJ Inspector to See a Representation of a Data Structure

Leigh Ann Sudol
Fox Lane High School
Bedford, New York

Introduction

BlueJ is an IDE (integrated development environment) developed by several members of a team of college educators as a visual tool for introductory programmers to learn about object orientation. BlueJ makes this activity worthwhile in that it provides an object bench for instantiating objects, but it also provides functionality for using built-in objects from any of the Java library classes.

Description

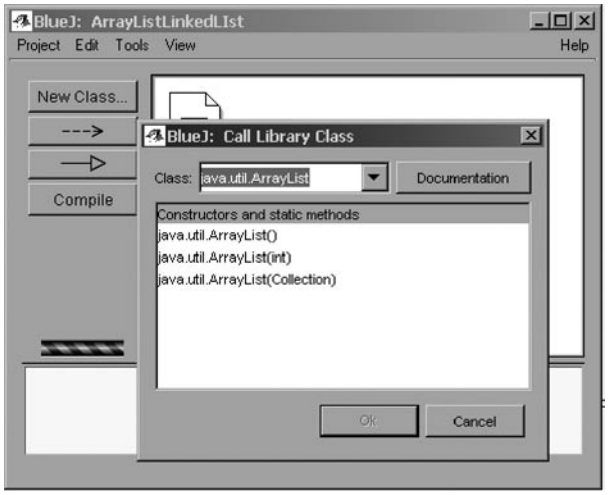
This activity uses the BlueJ IDE to view a physical representation of the difference between an `ArrayList` and a `LinkedList`. The lesson follows the steps below:

1. Introduce the definition of a `List` (numbered, linear collection of items).

2. Using a projection device for a computer screen, launch the BlueJ IDE to the starting screen (shown below).

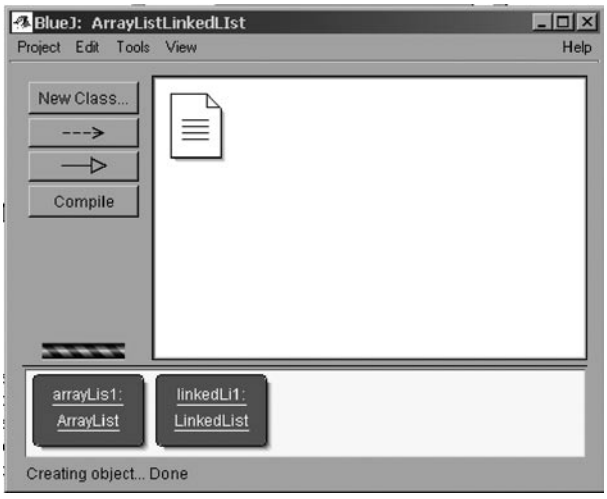
3. Create a new BlueJ project (from the "Project" menu, choose "New Project"). Name the project as appropriate and save it in an appropriate location.

4. From the "Tools" menu, choose "Use Library Class" and type `java.util.ArrayList` into the "Class" box (see graphic below).



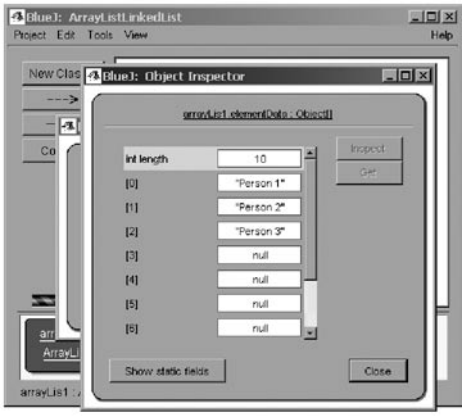
5. Once you press Enter, you will be given a list of constructors to choose from. Choose the default constructor, and click Ok. This will give you a dialog box for construction. Click Ok again, and a representation of the `ArrayList` should appear on your object bench.

6. Repeat the process, this time instantiating a `LinkedList` variable (`java.util.LinkedList`). After that object is instantiated, your BlueJ window should appear as follows:



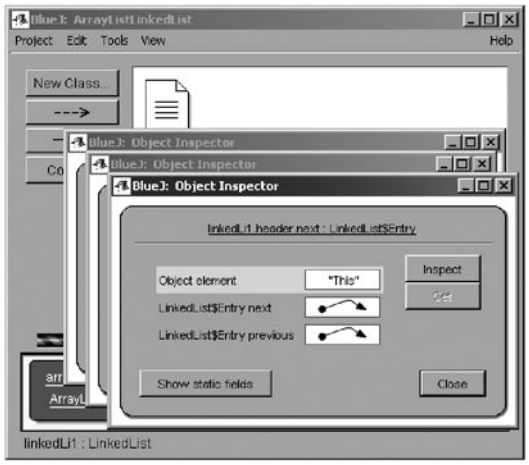
7. Each of the red boxes at the bottom of the screen is representative of the objects. By right-clicking upon the box, you can choose any of the methods of the object to execute. The next step in this process is to add some `String` variables to each of the two lists. For consistency, add the same strings to both lists.
   a. Right-click the `ArrayList` icon and choose the method `add(Object ..)`. This will give you a dialog box to enter a parameter to be added. Type a string into the box in quotes (e.g., "Someone's Name"). Repeat the process until you have added at least three names. You can verify this by calling on the size method of the `ArrayList` in order to determine how many names were successfully entered.
   b. Right-click the `LinkedList` icon and choose the method `add(Object ..)`. This will give you a dialog box to enter a parameter to be added. Type the same strings into the box that you did for the `ArrayList`.

8. You are now ready to view the differences between the two data structures. First inspect the `ArrayList` object (right-click and choose "Inspect"). The window that you will see gives the private data of the `ArrayList`. Double-click the arrow reference to private `Object[] elementData` to view the array being kept. (It will show up in a new window.) The window on the screen should resemble the following:



   a. At this point, you can engage students in a discussion that talks about what is shown by that window—the reference to the array object and the array object itself, where individual strings are stored by index, with a default size of 10.

b. Either close or move those windows to the side and now inspect the `LinkedList` object. The reference now is to a `LinkedList$Entry` named `Header`. Upon opening this object, your window should be as follows:



c. A discussion of what you are seeing should ensue. Why is the `LinkedList` set up differently than the `ArrayList`? How do you get to the "next" object in the sequence?

Assignment

Have students complete the included `List` worksheet, which asks them to compare `ArrayList` and `LinkedList` objects and make conjectures about where each would best be applied.

Extensions

There are several possible extensions to this program, depending upon the learning outcome desired at this point in the curriculum:

- Have students look at the object returned by the `Iterator` object of both the array and `ArrayList` objects to see if there are differences in the `Iterator` that parallel the way the list is stored.
- Students can make comparisons between `ArrayList`, `LinkedList`, and other collection objects in the same way as well.
- Advanced AB students can try to build their own `Entry` class from the `LinkedList` demo and construct a basic `LinkedList` whose structure would appear similar to the `java.util.LinkedList` class used in this activity.

Special Focus: Using the Java Collections Hierarchy

Lab Setup

Students will need the following for this program activity to be successful:

1. Students should have access to a machine with BlueJ and the IDE installed. They should also be given sample data sets (strings work really well) in order to construct their lists for comparison.

2. Students will also need a copy of the worksheet comparing `LinkedList` to `ArrayList`.

References

Barnes, David, and Michael Kolling. *Objects First with Java—A Practical Introduction Using BlueJ*. 2nd ed. Prentice Hall/Pearson Education, 2004.

`ArrayList` vs. `LinkedList` Worksheet

Name:_____

Part A: Memory Representations

Draw memory representations of both `ArrayList` and `LinkedList`. Label each part of the diagrams to point out the specific differences between the two data structures.

Part B: Uses and Efficiency

Describe how the structure of an `ArrayList` makes it more efficient for accessing individual elements.

Describe how the structure of `LinkedList` makes it more efficient for insertion into the middle of the list, as well as for a dynamic length list (lots of resizing).

Part C: List? ArrayList? LinkedList? What Are the Real Differences?

Fill in the following chart regarding the differences between `ArrayList` and `LinkedList`:

	ArrayList	LinkedList
Interface		
Traversing the List		
Indexing a Specific Item in the List		
Stored in Memory		

Give an example of a program where an `ArrayList` would be more beneficial to use.

Give an example of a program where a `LinkedList` would be more beneficial to use.

Special Focus: Using the Java Collections Hierarchy

Implementing the Java Marine Biology Case Study Using Maps

Christian Day
Emma Willard School
Troy, New York

The exercise below is designed to provide instructions to modify the unbounded ocean implementation of the *AP Marine Biology Simulation Case Study*. The implementation provided by the College Board uses the `ArrayList` data structure to hold the fish in the environment. This implementation modifies that implementation using a `Map` instead. The `Map` interface provides functionality for mapping keys to values in an efficient manner. In this implementation, the location of each fish will be used as a key that refers to the fish (or any class that implements the `Locatable` interface) with that key location.

The Java `Collections` framework offers two classes that implement the Map interface. These are `HashMap` and `TreeMap`. Class `HashMap` uses a hash table to store the keys. Each key includes a reference to the value it maps to; in this case, this will generally be a fish. A `TreeMap` uses a tree data structure (specifically a red-black tree) to store the keys. Just as with `HashMap`, each key includes a reference to the value it maps to.

This exercise will not try to explain the data structures underlying the Java implementation of `Maps`. The reader should know the following about the performance of `HashMaps` and `TreeMaps`:

Java Class	Add a New Element	Find an Element	Remove an Element
`HashMap`	*O(1)*	*O(1)*	*O(1)*
`TreeMap`	*O(log N)*	*O(log N)*	*O(log N)*

Where to Start

This exercise changes the underlying data structure used to store the environment in an unbounded ocean. Being a dedicated fan of encapsulation, I have completed this exercise in a manner that does not affect any of the other case study classes. While I use an unadulterated release of the case study, there is no reason why the changes implemented should not work if you have already committed time to modifying the

case study in significant ways. For example, if your case study code currently includes additional classes for creating sharks that chase the fish, or if your fish have the ability to breed and die, that should have no effect on this exercise.

Instructions

Note: The horizontal lines below delineate sets of directions.

Begin by opening the file `UnboundedEnv.java`. Save this file as `UnboundedEnvMap.java`, and rename the class as `UnboundedEnvMap`.

Add the following import to the list of packages imported. The `HashMap` and `TreeMap` classes, as well as the `Map` interface, are all located in the `java.util` package.

```
import java.util.*;
```

At the top of the class where the instance variables are declared, replace `private ArrayList objectList` with the following line:

```
   private HashMap objectMap;
```

Modify the constructor for class `UnboundedEnv`.

Begin by changing the name of the constructor to `UnboundedEnvMap` to match the name of the class.

Replace `objectList = new ArrayList();` with

```
   objectMap = new HashMap();
```

This creates a new `HashMap` with a capacity of `16` and a load factor of `0.75`. The load factor is an indication of how full the `HashMap` can get (in this case `75%` full) before the `HashMap` is resized. Resizing the `HashMap` doubles the size of the underlying hash table and then transfers the existing elements to the new, larger hash table. This is a *O(N)* operation so it is worthwhile to think about the initial size. Using the default values,

the `HashMap` will be resized after `12` keys have been placed in the map. The integer constructor for class `HashMap` describes the initial capacity of the `HashMap`:

```
objectMap = new HashMap(500)[1]
```

If you choose to make this change, you will have to return to using the default constructor in order to use the `TreeMap` class.

Methods `numRows`, `numCols`, and `isValid` remain unchanged.

Method `numObjects` requires the simple change of `objectList` to `objectMap`:

```
return objectMap.size();
```

Method `allObjects` needs significant modifications. None of the code in the current method is necessary. The `Map` method `values` is used to get a list of all *values* (not keys) in a `Map`. Method `values` returns a `Collection`, but the `allObjects` method needs to return an array of `Locatable` objects.

Fortunately, the `Collection` interface provides the method `toArray` that returns an array. Unfortunately, this method returns an array of class `Object`, and there is no way to cast the entire array[2]. The alternative method `toArray` needs to be used instead:

```
public Object[] toArray(Object[] a)
```

---

[1] The integer constructor for class `HashMap` rounds the value up to the nearest power of `2`. Using `500` will round the initial size up to `512` (2^9).

[2] Really, it's true. Your program will compile using return `(Locatable[])objectMap.values().toArray()`, but it will throw a `ClassCastException` when the line attempts to execute.

This version of `toArray` uses the parameter it receives (`Object[] a`) to determine the class that the members of the Collection should be cast to before being placed in the array. The code to be added looks like this:

```
Locatable[] locatableArray = new Locatable[0];
locatableArray =

(Locatable[])objectMap.values().toArray(locatableArray);
return locatableArray;
```

The distinction between `public Object[] toArray()` and `public Object[] toArray(Object[] a)` is significant and complex. The API documentation for this method provides the following description:

> Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array. If the collection fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this collection.

The second sentence explains why we can safely initialize `locatableArray` to an array of size `0`. When we make the call to `toArray` and pass it an array of `Locatable` objects, we do so knowing full well that any object contained within the `Collection` (which was constructed from the `Map`) is a `Locatable`. There is no way, in our program, that an object that does not implement the `Locatable` interface could end up in the `Map`. If the `toArray` method did encounter an object that did not implement the `Locatable` interface, an `ArrayStoreException` exception would be thrown.

---

[3] Indeed, we must set this to `0`. Otherwise, we risk the potential of returning an empty array of the size given. That will cause trouble in the `toString()` method, which attempts to loop through all elements in the array returned, extracting a `Locatable` from each.

Method `isEmpty` should return `true` when the `Location` given does not contain an object. In our `Map` implementation, that means that the `Location` is not used as a key in the `Map`. The method `containsKey` can be used, but it returns `true` if the `Location` *is* in the `Map`. We use the unary *not* operator to negate the result:

```
return !objectMap.containsKey(loc);
```

Method `objectAt` is designed to return the `Locatable` given a location. This is exactly what a `Map` does, so the modification in this method is simple:

```
return (Locatable)objectMap.get(loc);
```

Method `toString` will work without making any changes. The programmer of the class was clever enough to use public method `allObjects()` when getting the array. Since we have modified method `allObjects()` so that it returns the expected result, we are assured this method will work.

Method `add` needs to replace one line. With the `List` interface, we were able to add `obj` to the `List` using the `add` method. Now we need to associate a key with that value. The method already creates a `Location loc` to insure the location of `obj` is empty.

Replace `objectList.add(obj);` with

```
objectMap.put(loc, obj);
```

The code for method `remove` needs to be completely replaced. The following lines take advantage of the fact that the `Map` interface method `remove` returns `null` when the key to be removed is not in the `Map`.

```
if (objectMap.remove(obj.location()) == null) {
      throw new IllegalArgumentException("Cannot remove " +
                        obj + "; not there");
}
```

Method `recordMove` works best with the `Map` implementation if you simply use the code from the `boundedEnv` class.

To test if `obj` is at the same location as `oldLoc` (i.e., the `Fish` has not moved), we use the test from the bounded environment:

```
Location newLoc = obj.location();
if ( newLoc.equals(oldLoc) )
         return;
```

To test that the move made is valid, we again use the test from the bounded environment:

```
Locatable foundObject = objectAt(oldLoc);
if ( ! (foundObject == obj && isEmpty(newLoc)) )
  throw new IllegalArgumentException("Precondition
violation moving "
       + obj + " from " + oldLoc);
```

The only change comes in the form of removing the `Fish` at `oldLoc` and putting it at its new location. In the bounded environment, we did this by setting the old location to `null` and the new location to `obj`. We do the same thing with the `Map`, but with different code:

Replace the lines

```
theGrid[newLoc.row()][newLoc.col()] = obj;
theGrid[oldLoc.row()][oldLoc.col()] = null;
```

with

```
objectMap.remove(oldLoc);
      objectMap.put(newLoc, obj);
```

The private helper method `indexOf` is no longer necessary in the `Map` implementation.

Real-Time Performance Testing

Performing a thorough and accurate performance test is made difficult by the use of the `RandNumGenerator` class. This class is used in order to ensure the randomness of the values calculated. To accurately test our changes, we would like to have the exact same sequence of events occur. To do this, we would like to seed the random number generator with some value. There is no simple way to do this[4], so we will settle for a less precise method that will still yield meaningful results.

I tested the performance of the `Map` using the `Fish` class located in the "DynamicPopulation" folder in the core case study and the `SimpleMBSDemo2` class. Any implementation that causes the fish to breed will do, but the more rapid the explosion, the more quickly you will see results.

Timing the results:

The `getTimeInMillis()` static method in class `System` will tell you the current time in milliseconds[5]. Modify the method step in class `Simulation`:

```
long startTime = System.currentTimeMillis();
  Locatable[] theFishes = theEnv.allObjects();
```

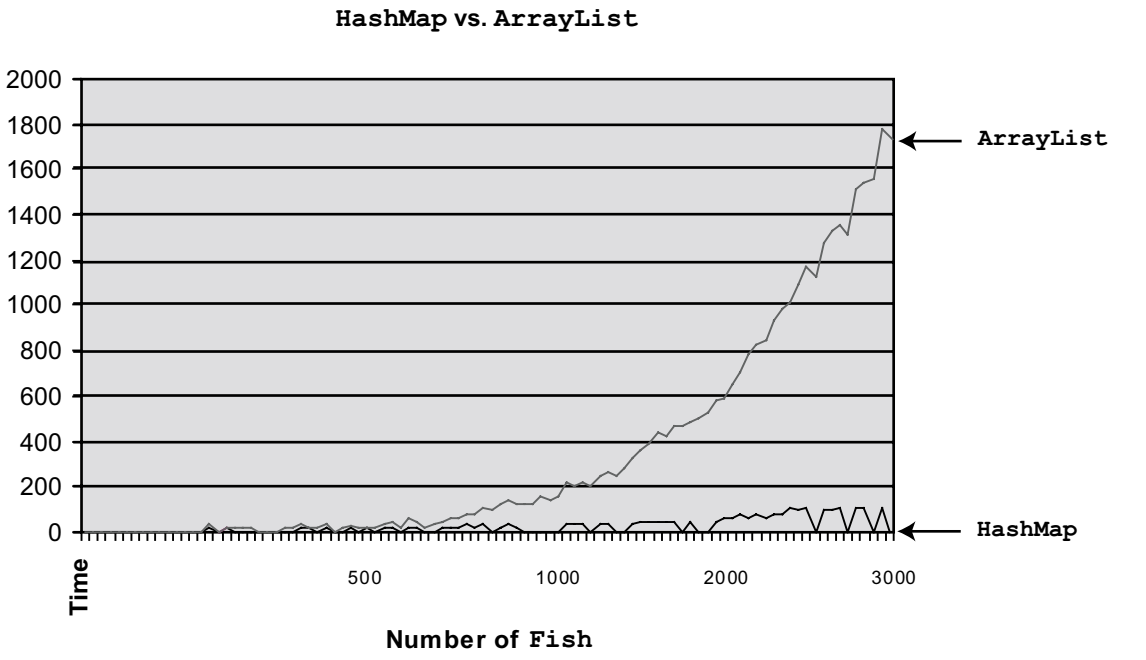
---

[4] But if you would like to try, create your own `RandNumGenerator` class with a constructor that takes an `int` that is the seed for the generator.

[5] However, this clock is only updated every 1/60th of a second, so your results may seem a little strange for small numbers.

```
for ( int index = 0; index < theFishes.length; index++ )
   {
      ((Fish)theFishes[index]).act();
   }
 long endTime = System.currentTimeMillis();
  System.out.println("The full simulation with " +
             theEnv.numObjects() + " Fish took " +
             (endTime - startTime) + " milliseconds");
```

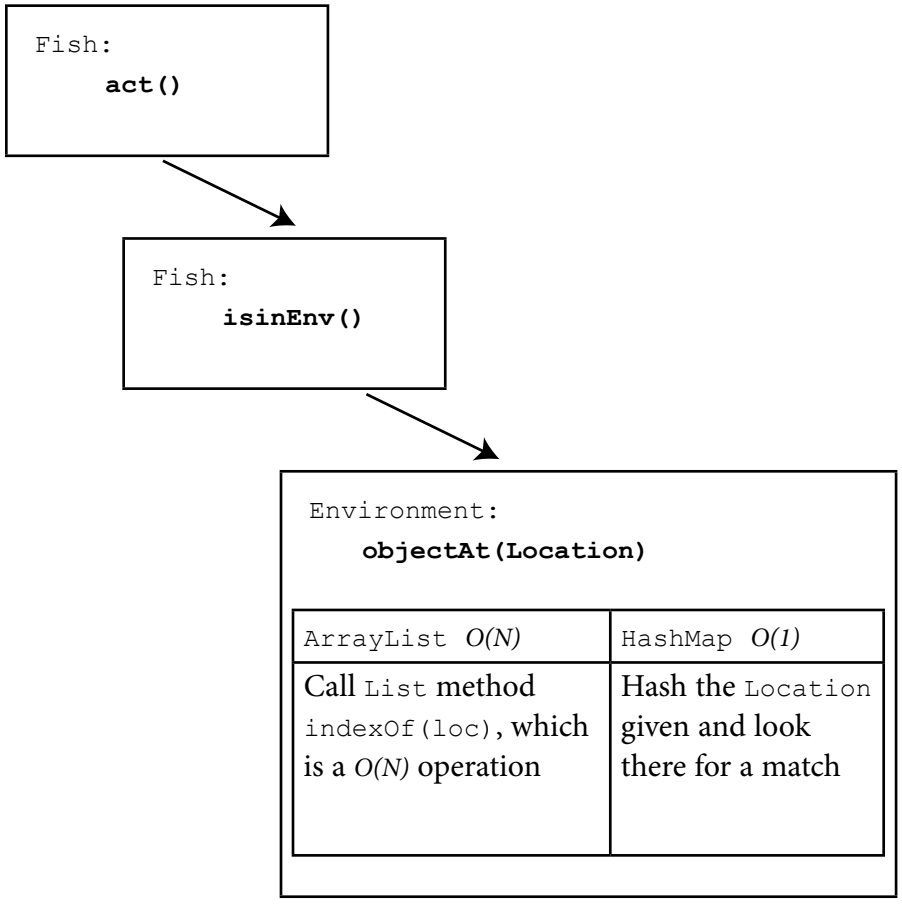
This will display the cumulative amount of time that it takes to `act` on every `Fish`. Using the `Fish` class from the "DynamicPopulation" folder, `Fish` populations grow very quickly. It doesn't take much information to be able to conclude that the `HashMap` implementation of this program provides a significant performance upgrade over the `ArrayList` implementation. The following is a chart comparing the performance of the `ArrayList` to the performance of the `HashMap` for up to 3,000 `Fish`.



As you can see, by the time there are 3,000 `Fish` in the environment, the `ArrayList` implementation is taking nearly two seconds (on a 3 GHz Intel Pentium III® with 2 GB of RAM running Microsoft Windows XP®) to act on all of the `Fish`. The `HashMap` implementation is still taking less than 100 milliseconds (1/10 of a second).

Performance Evaluation

In reality, having a fish act comprises a number of different well-defined actions. The following depicts the different actions taken in the case study while running the `Fish` class available in the "DynamicPopulation" folder. The steps taken by method `act` are depicted one at a time.



You can see that the simple call to method `isInEnv()` already causes the `List` version of this program to perform a *O(N)* operation. An attempt to map all the method calls made in method `act` is made on the following pages. The table below summarizes the complexity of different actions in different implementations of the program.

Action	2D Array	`ArrayList`	`HashMap`	`TreeMap`
Find a `Locatable` in the environment given a location	Exactly 1 lookup and comparison: *O(1)*	From 1 to N lookups and comparisons: *O(N)*	Exactly 1 lookup and comparison: *O(1)*	*O(log N)*
Remove a `Fish`	*O(1)*	*O(N)*	*O(1)*	*O(log N)*
Add a `Fish`	*O(1)*	*O(N)*	*O(1)*	*O(log N)*
Check empty neighbors	*O(1)*	*O(N)*	*O(1)*	*O(log N)*
Generate an array of `Fish` in the list	*O(N)*	*O(N)*	*O(N)*	*O(N)*

Diagram (Continued): Comparison of Algorithmic Complexity Between the `ArrayList` and the `HashMap` Implementation in the `Act` Method

Step 1: Breeding

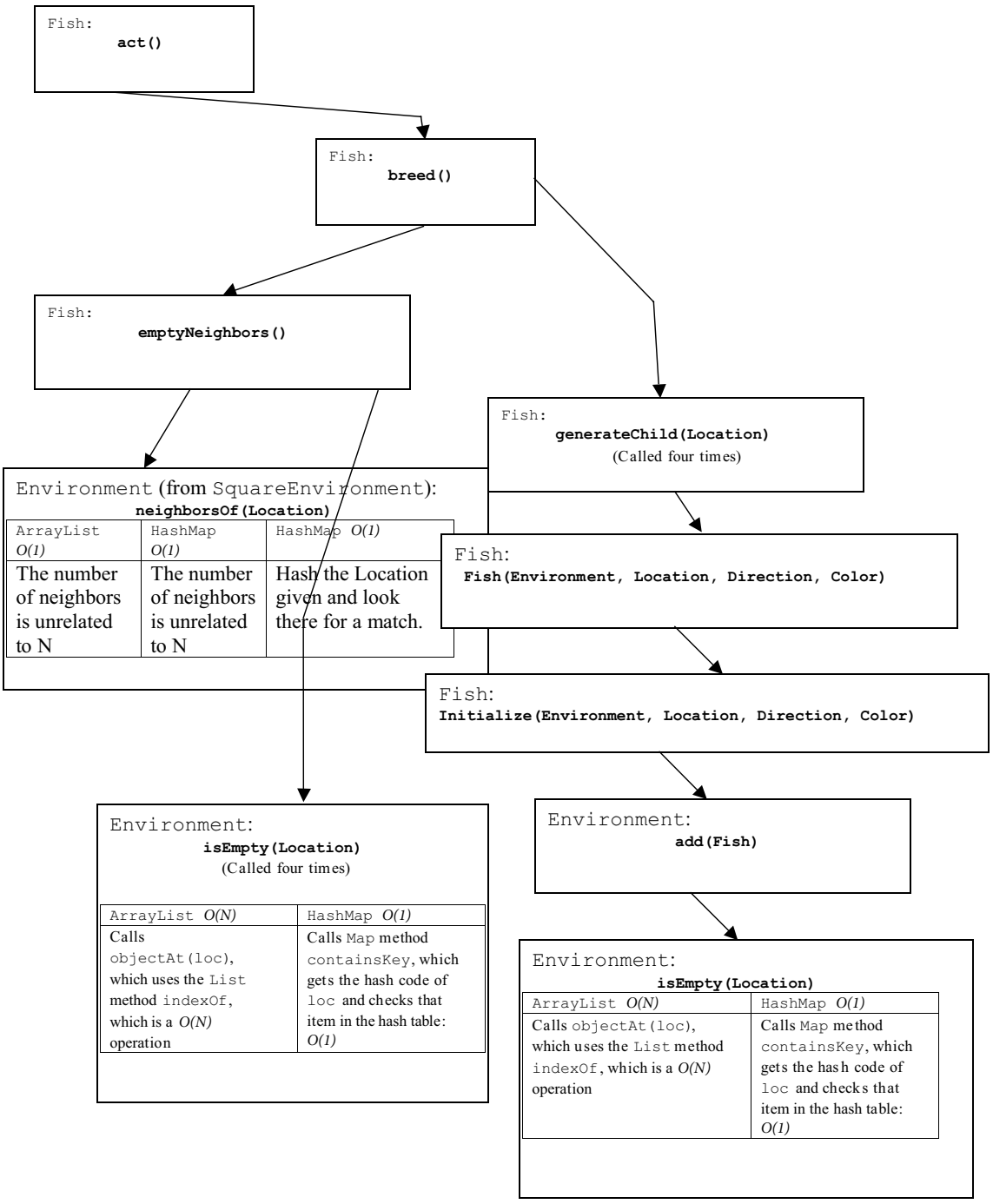
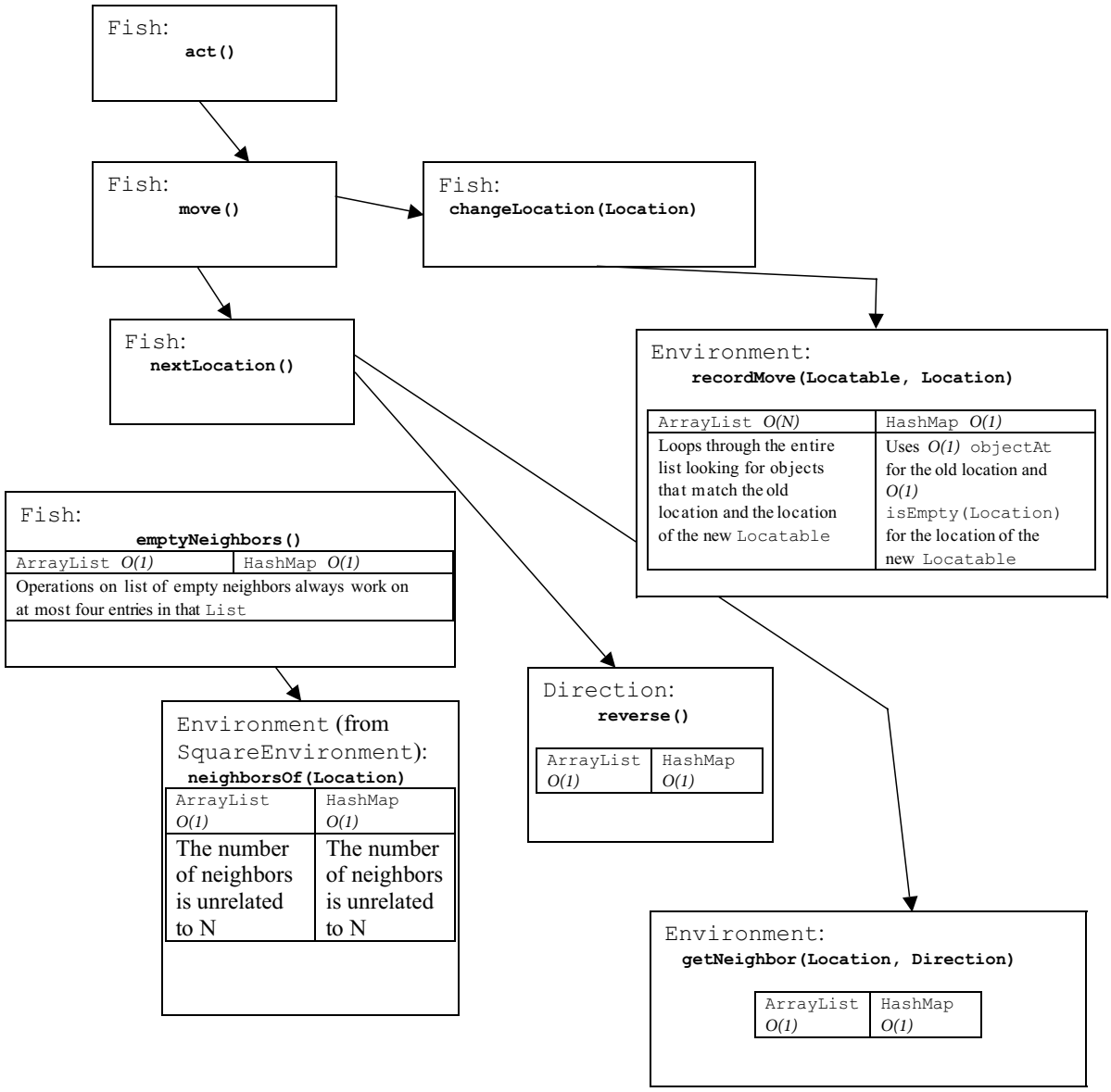


Diagram (Continued): Comparison of Algorithmic Complexity Between the `ArrayList` and the `HashMap` Implementation in the `Act` Method

Step 2: Moving



Special Focus: Using the Java Collections Hierarchy

Sets and Maps: An Excursion

Bekki George
James E. Taylor High School
Katy, Texas

Set Lesson: Teacher Notes

Prior to teaching this lesson, teachers should ensure that students are familiar with using `ListIterator` and `Iterator` on `ArrayList` and/or `LinkedList`.

Included in the lesson:

- PowerPoint presentation on `Set` that can be found on AP Central at http://apcentral.collegeboard.com/workshop/csfiles
- `Set` Worksheet (below)
- Answers to `Set` Worksheet (below)

I start the lesson by giving notes on `Set` and follow up with the worksheet. After checking the worksheet, I give a short quiz the next class day to check for understanding. I then assign these short programs that use `Set`:

1. Write a program that uses a `Set` to determine the number of **unique** words in a text file.

2. Design a class called `MathSet` that has a default constructor and the following methods:
   - `Set union(Set s)`—Example call: `s1.union(s2)`. This method returns the union of `s1` and `s2` but will not modify `s1`.
   - `Set intersect (Set s)`—Example call: `s1.intersect(s2)`. This method returns the intersection of `s1` and `s2` but will not modify `s1`.
   - `Set difference(Set s)`—Example call: `s1.difference(s2)`. This method returns the difference of `s1` and `s2` but will not modify `s1`. The difference of two sets consists of the elements of `s1` that are not contained in `s2`.
   - `boolean subset(Set s)`—This method will determine if the calling set is a subset of the argument.
   - `boolean superset(Set s)`—This method will determine if the calling set is a superset of the argument (in other words, the argument is a subset of the calling set).

Map Lesson: Teacher Notes

After students demonstrate an understanding of `Set`, I begin the lesson on `Map`. I use the same format as above with the PowerPoint presentation, worksheet, and a quiz. I like to use the sample free-response question from the *AP® Computer Science Course Description* (number 2 from page 111) as one lab, and for another lab using sets and maps I use the Schedule Builder program (see below). Another idea for a lab assignment that incorporates maps is to simulate a foreign language dictionary by pairing a word or phrase in another language with a word or phrase in the English language.

Included in this lesson:

- PowerPoint presentation on `Map` that can be found on AP Central at http://apcentral.collegeboard.com/workshop/csfiles
- `Map` Worksheet (below)
- Answers to `Map` Worksheet (below)
- Schedule Builder Lab Assignment (below)

`Set` Worksheet

Name:_____

1. Which of the following correctly defines a `Set` object?
   a. `Set A = new Set();`
   b. `Set B = new HashSet();`
   c. `HashSet C = new Set();`
   d. Both a and b

2. What is a possible output for the following?

```
Set s1 = new HashSet();
s1.add("one");
s1.add("two");
s1.add("three");
s1.add("two");

System.out.print(s1);
```

   I. `[one, two, three]`
   II. `[one, two, three, two]`
   III. `[three, two, one]`
   IV. `[one, three, two]`

   a. I only
   b. II only
   c. IV only
   d. I, III, and IV only
   e. I, II, III, and IV

3. What is a possible output for the following?

```
Set s1 = new TreeSet();
s1.add("one");
s1.add("two");
s1.add("three");
s1.add("two");

System.out.print(s1);
```

I. `[one, two, three]`
II. `[one, two, three, two]`
III. `[three, two, one]`
IV. `[one, three, two]`

a. I only
b. II only
c. IV only
d. I, III, and IV only
e. I, II, III, and IV

4. What is a possible output for the following code segment?

```
Set s2 = new HashSet();
for(int i = 1; i<10; i+=2)
   s2.add(new Integer(i));
System.out.print(s2);
```

a. `[9, 1, 3, 7, 5]`
b. `[1, 3, 7, 5, 9]`
c. `[1, 3, 5, 7, 9]`
d. All of the above

5. Given that `s3` is a `HashSet` and contains the following elements, what is the possible output?

```
//s3 = [4, 8, 9, 1, 16, 18, 12]
s3.add(new Integer(s3.size()));
System.out.println(s3);
```

a. `[4, 8, 9, 1, 16, 18, 12, 6]`
b. `[4, 8, 9, 1, 16, 18, 12, size]`
c. `[4, 8, 9, 1, 16, 18, 12, 7]`
d. `[4, 8, 9, 1, 16, 18, 7, 12]`
e. Both c and d

6. Given the following declarations, which of the following would successfully remove value 12 from the set `s3`?

```
Set s3 = new HashSet();// assume elements are added to s3
Iterator it = s3.iterator();
Integer x = new Integer(12);
```

I.
```
boolean takeOut = false;
while(it.hasNext())
{
  if(it.next().equals(x))
takeOut = true;
}
if(takeOut)
  s3.remove(x);
```

II.
```
while(it.hasNext())
 {
   if(s3.contains(x))
      it.remove(x);
   it.next();
}
```

III.
```
if( s3.contains(x))
      s3.remove(x);
```

IV.
```
s3.remove(x);
```

a. I only
b. II only
c. I, II, and III only
d. I, III, and IV only
e. I, II, III, and IV

7. Which of the following would you check to see if `s1` is a subset of `s2`?
   a. `s1.subset(s2);`
   b. `s2.subset(s2);`
   c. `s1.containsAll(s2);`
   d. `s2.containsAll(s1);`

8. Write the code to find the union of sets `s1` and `s2`.

9. List and describe five methods in the `Set` interface.

10. List the similarities and differences between `TreeSet` and `HashSet`.

Answers to `Set` Worksheet

1. Which of the following correctly defines a `Set` object?

   b. **`Set B = new HashSet();`**

2. What is a possible output for the following?

```
Set s1 = new HashSet();
s1.add("one");
s1.add("two");
s1.add("three");
s1.add("two");

System.out.print(s1);
```

I. `[one, two, three]`
II. `[one, two, three, two]`
III. `[three, two, one]`
IV. `[one, three, two]`

   d. I, III, and IV only

3. What is a possible output for the following?

```
Set s1 = new TreeSet();
s1.add("one");
s1.add("two");
s1.add("three");
s1.add("two");

System.out.print(s1);
```

I. `[one, two, three]`
II. `[one, two, three, two]`
III. `[three, two, one]`
IV. `[one, three, two]`

   c. IV only

4. What is a possible output for the following code segment?

```
Set s2 = new HashSet();
for(int i = 1; i<10; i+=2)
   s2.add(new Integer(i));
System.out.print(s2);
```

d. All of the above

5. Given that `s3` is a `HashSet` and contains the following elements, what is the possible output?

```
//s3 = [4, 8, 9, 1, 16, 18, 12]
s3.add(new Integer(s3.size()));
System.out.println(s3);
```

e. Both c and d

6. Given the following declarations, which of the following would successfully remove value 12 from the set `s3`?

```
Set s3 = new HashSet();// assume elements are added to s3
Iterator it = s3.iterator();
Integer x = new Integer(12);
```

I.
`boolean takeOut = false;`
`while(it.hasNext())`
`{`
`if(it.next().equals(x))`
`takeOut = true;`
`}`
`if(takeOut)`
`s3.remove(x);`
	II.
	`while(it.hasNext())`
	`{`
	`if(s3.contains(x))`
	`it.remove(x);`
	`it.next();`
	`}`

III.
`if( s3.contains(x))`
`s3.remove(x);`
	IV.
	`s3.remove(x);`

d. I, III, and IV only

7. Which of the following would you check to see if `s1` is a subset of `s2`?
   d. `s2.containsAll(s1);`

8. Write the code to find the union of sets `s1` and `s2`.

```
Set s3 = new HashSet();
s3.addAll(s1);
s3.addAll(s2);
```

9. List and describe five methods in the `Set` interface.
   Answers will vary.

10. List the similarities and differences between `TreeSet` and `HashSet`.
    Answers will vary (most should say that `TreeSet` is ordered and `HashSet` is not).

Map Worksheet

Name:_____

1. What is the output of the following?

```
Map colors = new TreeMap();
String[] words = {"blue", "red", "green", "yellow",
"black"};
int j = 0;
for(int i = 1; i<10; i+=2)
   {colors.put(new Integer(i), words[j]);
 j++;}
System.out.println(colors);
```

   a. `{1=blue, 3=red, 5=green, 7=yellow, 9=black}`
   b. `{blue=1, red=3, green=5, yellow=7, black=9}`
   c. `{blue, red, green, yellow, black}`
   d. `[1, 3, 5, 7, 9]`

2. Using colors as declared above, what is the output of the following?

```
System.out.println(colors.put(new Integer(colors.size()),
"magenta"));
```

   a. `true`
   b. `false`
   c. `green`
   d. `magenta`

3. What will the contents of colors now be after the statement in question 2?
   a. `{1=blue, 3=red, 5=green, 7=yellow, 9=black}`
   b. `{1=blue, 3=red, 5=magenta, 7=yellow, 9=black}`
   c. `{1=blue, 3=red, 5=green=magenta, 7=yellow, 9=black}`
   d. `{blue=1, red=3, magenta=green=5, yellow=7, black=9}`
   e. `{blue, red, magenta, green, yellow, black}`

4. What is outputted by the following (using colors from question 1)?

```
System.out.println(colors.keySet());
```

   a. `[blue, red, green, yellow, black]`
   b. `[1, 3, 5, 7, 9]`
   c. `{1=blue, 3=red, 5=green, 7=yellow, 9=black}`
   d. `true`

5. What is the difference between a `Map` and a `Set`?

6. What is the difference between a `TreeMap` and a `HashMap`?

7. Which of these would return the value to which this map maps the specified key?
   a. `keySet`
   b. `values`
   c. `put`
   d. `get`

8. Describe the following methods:
   a. `clear`

   b. `containsKey`

   c. `containsValue`

   d. `get`

   e. `put`

   f. `remove`

9. Given this code from question 1:

```
   Map colors = new TreeMap();
   String[] words = {"blue", "red", "green", "yellow",
"black"};
   int j = 0;
   for(int i = 1; i<10; i+=2)
         {colors.put(new Integer(i), words[j]);
   j++;}
```

   Write the code needed to output the map as follows:

Color	Number
blue	1
red	3
...	

Answers to `Map` Worksheet

1. What is the output of the following?

```
Map colors = new TreeMap();
String[] words = {"blue", "red", "green", "yellow",
"black"};
int j = 0;
for(int i = 1; i<10; i+=2)
   {colors.put(new Integer(i), words[j]);
 j++;}
System.out.println(colors);
```

   a. `{1=blue, 3=red, 5=green, 7=yellow, 9=black}`

2. Using colors as declared above, what is the output of the following?

```
System.out.println(colors.put(new Integer(colors.size()),
"magenta"));
```

   c. `green`

3. What will the contents of colors now be after the statement in question 2?
   b. `{1=blue, 3=red, 5=magenta, 7=yellow, 9=black}`

4. What is outputted by the following (using colors from question 1)?

```
System.out.println(colors.keySet());
```

   b. `[1, 3, 5, 7, 9]`

5. What is the difference between a `Map` and a `Set`?
   Answers will vary, but look for something like: a `Map` uses keys, a `Set` does not.

6. What is the difference between a `TreeMap` and a `HashMap`?
   `TreeMap` is ordered by keys.

7. Which of these would return the value to which this map maps the specified key?
   d. `get`

8. Describe the following methods:
   a. `clear`
      Removes all mappings from this map
   b. `containsKey`
      Returns `true` if this map contains a mapping for the specified key
   c. `containsValue`
      Returns `true` if this map maps one or more keys to the specified value
   d. `get`
      Returns the value to which this map maps the specified key
   e. `put`
      Rates the specified value with the specified key in this map
   f. `remove`
      Removes the mapping for this key from this map if it is present

9. Given this code from question 1:

```
Map colors = new TreeMap();
String[] words = {"blue", "red", "green", "yellow",
"black"};
int j = 0;
for(int i = 1; i<10; i+=2)
      {colors.put(new Integer(i), words[j]);
j++;}
```

   Write the code needed to output the map as follows:

Color	Number
blue	1
red	3
...	

```
Set keys = colors.keySet();
Iterator it = keys.iterator();
System.out.println("Color \tNumber");
while(it.hasNext())
{Integer x = (Integer)it.next();
     String c = (String)colors.get(x);
     System.out.println(c + "\t" + x);}
```

Special Focus: Using the Java Collections Hierarchy

Schedule Builder Lab Assignment

Consider the following `StudentInfo` interface that will represent a student's name and the course that student wishes to add to his or her schedule:

```
public interface StudentInfo
{
String name();
String course();
}
```

The following class, `Schedules`, will be used to store students' names and their schedules. Information from `StudentInfo` objects will be stored in this class as a `TreeMap`. In the `TreeMap`, the keys are the student names, and for each key the corresponding value is a `Set` of the classes the student has signed up for (since no one can sign up for the same class twice, `Set` is used).

```
public class Schedules
{
private Map theSchedules;

public Schedules()
{
theSchedules = new TreeMap();
}

// postcondition: Information from theStudent
// has been added to theSchedules
public void addClassToSchedule(StudentInfo theClass)
{ /* to be implemented in part (a) */ }

public void printSchedule(String studentName)
{ /* to be implemented in part (b) */ }

public void printRoster()
{ /* to be implemented in part (c) */ }

private Set courseListing()
//implementation not shown

// ... other methods not shown
}
```

For example, assume that a `Schedules` object has been initialized with the following:

```
("George", "Computer Science")
("George", "Math")
("Smith", "Math")
("Thompson", "Computer Science")
("Thompson", "English")
```

The following table represents the entries in `theSchedules`:

Key	**Value**
George	[Computer Science, Math]
Smith	[Math]
Thompson	[Computer Science, English]

a. Write the `States` method addClassToSchedule, which is described as follows. Method `addClassToSchedule` takes one parameter, a `StudentInfo` object, and updates `theSchedules` to include the information encapsulated in the `StudentInfo` object.

   Complete the method `addClassToSchedule` below:

```
// postcondition: Information from theStudent
// has been added to theSchedules
public void addClassToSchedule(StudentInfo theClass)
{
```

b. Write the method `printSchedule`, which is described as follows. Method `printSchedule` takes a String representing a name of a student. It prints the name of the student and a list of the classes he or she is taking. The output should not include `[]`, and the classes should each be separated by a blank space.

   Complete the method `printSchedule` below. A solution that creates an unnecessary instance of any `Collection` class will not receive full credit.

```
public void printSchedule(String studentName)
{
```

c. Write the method `printRoster`, which is described as follows. Method `printRoster` outputs each course followed by the students enrolled in each course. Use the helper method `courseListing()` to retrieve a `Set` of all courses. Example output for a call to `printRoster` could be:

```
Course: Computer Science
Students: George Thompson

Course: Math
Students: George Smith

Course: English
Students: Thompson

public void printRoster()
{
```

StudentInfo (Java)

```
/*public interface StudentInfo
{
String name();
String course();
} */
public class StudentInfo
{
  private String name;
  private String course;

  public StudentInfo(String n, String c)
  {

     name = n;
     course = c;
  }

  public String name()
  {
     return name;
  }

  public String course()
  {
     return course;
  }

}
```

Schedules (Java)

```
import java.util.*;

public class Schedules
{
   private Map theSchedules;

   public Schedules()
   {
         theSchedules = new TreeMap();
   }

   // postcondition: Information from theStudent
   // has been added to theSchedules
   public void addClassToSchedule(StudentInfo theClass)
   {
         Set students;
         if(theSchedules.containsKey(theClass.name()))
           students =(Set) theSchedules.get(theClass.name());
         else
         {
               students = new HashSet();
               theSchedules.put(theClass.name(),students);
         }
         students.add(theClass.course());
   }

   public void printSchedule(String studentName)
   {
         Set s = (Set) theSchedules.get(studentName);
         System.out.print(studentName + ": ");
         Iterator it = s.iterator();
         while(it.hasNext())
         {
               System.out.print("\t" + it.next());
         }
         System.out.println();
   }
```

```
public void printRoster()
{
      Set classes = courseListing();
      Iterator it = classes.iterator();

      while(it.hasNext())
      {
            String course = (String)it.next();
            System.out.println("Course: " + course);
            Set students = theSchedules.keySet();
            Iterator it2 = students.iterator();

            System.out.print("Students: ");
            while(it2.hasNext())
            {
                  String stuName = (String)it2.next();
                  Set s = (Set)theSchedules.get(stuName);
                  if(s.contains(course))
                        System.out.print(stuName + "  ");
            }
            System.out.println();
            System.out.println();
      }
}

private Set courseListing()
{
      Set s = new TreeSet();
      Set students = theSchedules.keySet();
      Iterator it = students.iterator();

      while(it.hasNext())
            {
                  String stuName = (String)it.next();
                  s.addAll((Set)theSchedules.get(stuName));

            }
```

```
        return s;
    }



    public static void main(String[] args)
    {
        System.out.println();
        Schedules stuSched = new Schedules();
        stuSched.addClassToSchedule(new StudentInfo("George","CS"));
        stuSched.addClassToSchedule(new StudentInfo("Scearce","Math"))
;
        stuSched.addClassToSchedule(new StudentInfo("George",
"Math"));
        stuSched.printSchedule("George");

        System.out.println("\n\n\n==============Rosters===============
");
        stuSched.printRoster();



    }
}
```

Collections API Activity

Cody Henrichsen
Granger High School
West Valley City, Utah

API (Application Programming Interface) Worksheet

The purpose of this worksheet is to familiarize you with the various aspects of the Java API. The API contains tons of explanations of interfaces, classes and methods, and implementation details. Though most of these details are not tested on the AP Exam, navigating the API is an important skill in programming. You should use the Java API in answering these questions.

`ArrayLists`

1. When an `ArrayList` is constructed, what is its initial capacity?

2. How many constructors does an `ArrayList` have?

3. From where is the `ArrayList` method `iterator` inherited?

4. What is the Big-Oh running efficiency of adding an item to the beginning of an `ArrayList`? Explain.

5. What is the Big-Oh running efficiency of adding an item to the end of an `ArrayList`? Explain.

6. What happens when you attempt to add items beyond the current capacity of the `ArrayList`?

7. What is the result of calling the `get` method in an `ArrayList` with an argument that is not in the range of existing elements?

8. When does the `hasNext` method of an iterator return false?

9. Give an example that would cause a `NoSuchElementException` when using an iterator with an `ArrayList`?

10. What is the difference between an `Iterator` and a `ListIterator`?

Special Focus: Using the Java Collections Hierarchy

LinkedLists

11. When must a `LinkedList` be externally synchronized?

12. What method returns an in-order array of all the elements in a `LinkedList`?

13. When is `-1` returned from the method `lastIndexOf(Object o)`?

14. What type of value is returned when using `removeFirst()`?

15. What is the difference between the `LinkedList` methods `getLast()` and `removeLast()`?

Sets

16. What is the purpose of a `hashCode` method and from where is it inherited?

17. What should be confirmed before a `HashSet` is used with objects of user-defined classes?

18. What should be confirmed before a `TreeSet` is used with objects of user-defined classes?

19. In what order are elements returned with an iterator in a `HashSet`?

20. What is the Big-Oh efficiency for the `remove()` method of a `HashSet`? Explain why this is the case.

21. What is one major difference between the `add(Object obj)` method in `ArrayList` and `HashSet`?

22. What is the difference between a `TreeSet` and a `HashSet`?

23. In what order are elements returned with an iterator in a `TreeSet`?

24. What is the Big-Oh efficiency of accessing the first element in a `TreeSet`?

25. What is the default size of a `HashSet` or a `TreeSet`?

26. How do you construct a `HashSet` of default size?

27. How do you construct a `TreeSet` of default size?

28. Are there any restrictions in calling a set's iterator's `remove()` method?

Maps

29. How many constructors does a `Map` have?

30. In what order does an iterator traverse the elements of a `HashMap`?

31. In what order does an iterator traverse the elements of a `TreeMap`?

32. List a few possible applications that might choose to use a class that implements the `Map` interface.

33. What happens when the `Map` method `get` is called with a parameter `key` that is not found in the `Map`?

34. What is the difference in running time between the `TreeMap`'s `get` method and the `HashMap`'s `get` method?

Collections

35. The `Collections` class consists entirely of **static** methods. What does this mean?

36. Consider the following statement:

```
Collections.binarySearch(someCollection, key);
```

What must be true of the parameter `someCollection`?

37. Is there a `Collections` method that would randomly order the elements of an `ArrayList`? If so, give an example of a call to this method.

38. The `Collections` class defines two `sort` methods. Explain the difference between these two methods.

39. What is the Big-Oh efficiency of these `sort` methods?

40. How is the `swap` method of the `Collections` class called? Explain the result.

API (Application Programming Interface) Worksheet with Answers

The purpose of this worksheet is to familiarize you with the various aspects of the Java API. The API contains tons of explanations of interfaces, classes and methods, and implementation details. Though most of these details are not tested on the AP Exam, navigating the API is an important skill in programming. You should use the Java API in answering these questions.

`ArrayLists`

1. **When an `ArrayList` is constructed, what is its initial capacity?**
   10

2. **How many constructors does an `ArrayList` have?**
   3

3. **From where is the `ArrayList` method `iterator` inherited?**
   `ArrayList` is a subclass of `AbstractList`, which implements the `List` interface. `List` contains the method `iterator`, which is implemented by `AbstractList` and then inherited by `ArrayList`.

4. **What is the Big-Oh running efficiency of adding an item to the beginning of an `ArrayList`? Explain.**
   O(n), because of the shifting that takes place.

5. **What is the Big-Oh running efficiency of adding an item to the end of an `ArrayList`? Explain**.
   O(1), because no shifting is necessary.

6. **What happens when you attempt to add items beyond the current capacity of the `ArrayList`?**
   As elements are added to an `ArrayList`, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

7. **What is the result of calling the `get` method in an `ArrayList` with an argument that is not in the range of existing elements?**
   `IndexOutOfBoundsException`

8. **When does the `hasNext` method of an iterator return false?**
   It returns false in the case where calling `next()` would result in an exception being thrown (if there is not an element for the iterator to return when `next()` is called).

9. **Give an example that would cause a `NoSuchElementException` when using an iterator with an `ArrayList`?**
   Calling `next` when there are no more elements in the list:

```
ArrayList a = new ArrayList()
a.add("one");
Iterator itr = a.iterator();
Object o = itr.next();
o = itr.next(); ← Exception thrown
```

10. **What is the difference between an `Iterator` and a `ListIterator`?**
    An `Iterator` allows forward traversal through the list and removal of elements. A `ListIterator` allows for traversal in both directions and addition, modification, and removal of elements.

`LinkedLists`

11. **When must a `LinkedList` be externally synchronized?**
    A linked list needs to be synchronized when a thread-based structure modification occurs, this would be an addition or deletion from the linked list.

12. What method returns an in-order array of all the elements in a `LinkedList`?
    The `toArray()` method returns the array of all elements within the `LinkedList`.

13. When is `-1` returned from the method `lastIndexOf(Object o)`?
    The value of `-1` is returned only if the object `o` in question is not located within the `LinkedList`.

14. What type of value is returned when using `removeFirst()`?
    The return value is of type `Object`.

15. What is the difference between the `LinkedList` methods `getLast()` **and `removeLast()`?**
    The difference between `removeLast()` and `getLast()` is that the `removeLast()` method returns and deletes the last item stored in the `LinkedList`; `getLast()` returns the last item but does not alter the list contents.

Sets

16. **What is the purpose of a `hashCode` method and from where is it inherited?**
    The `hashCode` method is inherited from `Object`. It is the specific `hashCode` value for the `Object` within the collection. It allows for a more efficient search because of the efficiency of the hash search functions.

17. What should be confirmed before a `HashSet` is used with objects of **user-defined classes?**
    `hashCode` should be defined in the user-defined class.

18. What should be confirmed before a `TreeSet` is used with objects of **user-defined classes?**
    `compareTo` should be defined in the user-defined class.

19. **In what order are elements returned with an iterator in a `HashSet`?**
    No particular order.

20. **What is the Big-Oh efficiency for the `remove()` method of a `HashSet`? Explain why this is the case.**
    `O(1)`. All simple algorithms based on hashing have a constant order of efficiency.

21. **What is one major difference between the `add(Object obj)` method in `ArrayList` and `HashSet`?**
    `HashSet` will not add duplicate elements.

22. **What is the difference between a `TreeSet` and a `HashSet`?**
    `TreeSet` is ordered; `HashSet` is not.

23. **In what order are elements returned with an iterator in a `TreeSet`?**
    The elements are returned in ascending order, as determined by the `compareTo` method within the class.

24. What is the Big-Oh efficiency of accessing the first element in a `TreeSet`?
O(log n)

25. What is the default size of a `HashSet` or a `TreeSet`?
0

26. How do you construct a `HashSet` of default size?
```
HashSet h = new HashSet();
```

27. How do you construct a `TreeSet` of default size?
```
TreeSet t = new TreeSet();
```

28. Are there any restrictions in calling a set's iterator's `remove()` method?
The method can be called only once per call to `next`.

Maps

29. How many constructors does a `Map` have?
None. `Map` is an interface.

30. In what order does an iterator traverse the elements of a `HashMap`?
No particular order.

31. In what order does an iterator traverse the elements of a `TreeMap`?
The elements are returned in ascending order, as determined by the `compareTo` method defined within the class of the key.

32. List a few possible applications that might choose to use a class that implements the `Map` interface.
Various answers; for example, people mapped to phone numbers.

33. What happens when the `Map` method `get` is called with a parameter `key` that is not found in the `Map`?
`null` is returned.

34. What is the difference in running time between the `TreeMap`'s `get` method and the `HashMap`'s `get` method?
`TreeMap` = *O(log n)*
`HashMap` = *O(1)*

Special Focus: Using the Java Collections Hierarchy

`Collections`

35. The `Collections` class consists entirely of static methods. What does **this mean?**
    No object of type `Collections` is needed to use the methods.

36. **Consider the following statement:**
    ```
    Collections.binarySearch(someCollection, key);
    ```
    What must be true of the parameter `someCollection`?
    `someCollection` must implement the `List` interface.

37. **Is there a `Collections` method that would randomly order the elements of an `ArrayList`? If so, give an example of a call to this method.**
    ```
    Collections.shuffle(arrList);
    ```

38. **The `Collections` class defines two `sort` methods. Explain the difference between these two methods.**
    One has one parameter and will sort the list using `compareTo`. The other passes a `Comparator` as a second parameter and will use the `Comparator` to sort the list.

39. **What is the Big-Oh efficiency of these `sort` methods?**
    `O(log n)`

40. **How is the `swap` method of the `Collections` class called? Explain the result.**
    `swap` has three parameters: the list and two indices. It swaps the elements in the indices specified in the list. The list is altered.

Teaching with Tiger: Using Java 5.0 Features in AP Computer Science Courses

Cay S. Horstmann
Department of Mathematics and Computer Science
San Jose State University
San Jose, California

This article reviews the features that were added to the Java language in the 5.0 release (aka 1.5, aka Tiger) and gives suggestions on how to use them in an AP Computer Science course. The focus of this article is on teaching an introductory course, not on the AP CS Exam. As this article is written, the Development Committee has not yet decided which new features (if any) will be included in the AP CS Java subset.

Generic Collections

Probably the most useful new feature in Java 5.0 is the ability to specify element types of collections. You can now form an

```
ArrayList<Fish> fishList;
```

or a

```
Map<Location, Fish> fishLocator;
```

The benefit is clear: there is no more guesswork about what kind of `Object` is in a given collection.

Generic collections are easy to use. All collection classes and interfaces in the standard Java library support type parameters. Simply specify the element type for collections or the key and value types for maps, enclosed in angle brackets.

Because the compiler keeps track of the element types, casts are no longer required:

```
Fish firstFish = fishList.get(0); // no (Fish) cast needed
```

Generic collections cannot hold primitive types. For example, there is no `ArrayList<int>`. The remedy is to use wrapper types such as `ArrayList<Integer>`. The new autoboxing feature, discussed later in this article, makes it easy to convert between primitive types and wrapper classes.

You are not forced to use generic collections—if you omit the type parameters, you simply get "raw" collections that hold elements of type `Object`. You can even mix generic and raw collections in the same program, but then you may get unsightly compiler

warnings when the compiler loses track of type information.
Of course, C++ veterans will recognize generic types as the equivalent of **templates** such as `vector<Fish>`. As with C++ templates, Java generic types are easy to use if you stick to the basics and just use classes that other programmers have defined, such as collections. Defining your own generic types is considerably more complex, as you will see later in this article.

I recommend that you teach generic collections instead of the raw collections. The resulting code is clearer to read, and you don't need casts. In particular, you can use `ArrayList<T>` as early as you like, without having to mention the `Object` class or inheritance.

It is likely that the AP CS Exam will use generic collections. Students should at least be able to read code that uses generic collections, and they should understand that no cast is required when retrieving elements from generic collections.

The "For Each" Loop

The "for each" loop (aka "enhanced for loop") is a new looping construct that **iterates over all elements** in an array or a collection. For example:

```
Fish[] fishes = . . .;

for (Fish f : fishes)
    f.act();
```

The loop iterates over the elements of `fishes`. In each iteration, the variable `f` is set to the next element, and the loop body is executed.

The benefit is simpler code. The "for each" loop is much easier to read than a traditional loop:

```
for (int i = 0; i < fishes.length; i++)
{
    Fish f = fishes[i];
    f.act();
}
```

There is also less room for indexing errors.

You can use the same loop to visit each element in a collection:

```
ArrayList<Fish> fishList = . . .;

for (Fish f : fishList)
    f.act();
```

Technically, this loop is a shortcut for the following traditional loop:

```
for (Iterator<Fish> iter = fishList.iterator(); iter.hasNext(); )
{
    Fish f = f.next();
    f.act();
}
```

In fact, you can use the "for each" loop to iterate through objects of **any** class that implements the `Iterable` interface. That interface has a single method, `iterator`. In Java 5.0, all collection classes implement the `Iterable` interface. Therefore, you can use the "for each" loop to iterate through linked lists and sets, not just array lists.

```
Set<Fish> fishSet = . . .;

for (Fish f : fishSet)
    f.act();
```

To visit all entries in a map, you iterate through the key set, like this:

```
Map<Location, Fish> fishLocator = . . .;
for (Location loc : fishLocator.keySet())
{
    Fish f = fishLocator.get(loc);
    . . .
}
```

Of course, there are many loops that cannot be expressed as a "for each" loop. For example, if you want to skip the first or last element, or if you need the index variable in the loop body, or if you want to remove elements during the iteration, then you still need a traditional loop.

Should you teach the "for each" loop, or should you simply stick with traditional loops? In my opinion, it doesn't much matter either way. However, most programmers find the "for each" loop seductive, and you too may fall under its spell.

It is possible that future versions of the AP CS Exam will use the "for each" loop to simplify code examples, but no final decision has been made.

Autoboxing

Autoboxing refers to the automatic conversion between primitive types and their corresponding wrapper types. (Autowrapping would have been a better term, but the Java designers took this feature, including the name, from C#.) For example:

```
Integer integerObj = 1729; // automatically calls the constructor new
Integer(1729)
```

The converse (sometimes called auto-unboxing) is also automatic:

```
int n = integerObj; // automatically calls
integerObj.intValue()
```

Automatic boxing and unboxing also happens in arithmetic expressions that involve wrapper objects:

```
integerObj--; // same as integerObj = new Integer(integerObj.intValue()
- 1);
```

Autoboxing is useful when primitive type values are stored in collections. For example:

```
ArrayList<Integer> luckyNumbers = new ArrayList<Integer>();
luckyNumbers.add(1729); // same as luckyNumbers.add(new Integer(1729));
int n = luckyNumbers.get(0);
```

For professional programming, the opportunity for autoboxing does not occur very often. However, when teaching introductory computer science courses, collections that hold numbers are commonly used as examples, and autoboxing is convenient.

Note that it is *not* a good idea to replace all primitive types with wrappers. Even though the resulting code will compile and run in most cases, the code is significantly less efficient and does not resemble "real world" code.

The downside of autoboxing is that the exact rules are rather complex. For example, consider the comparison:

```
integerObject == n
```

Does this code unbox `integerObject` and compare two integer values, or does it box `n` and compare two object references? As it turns out, it does the former. But you probably don't want to spend valuable class time discussing syntax trivia. If you decide to cover autoboxing in your class, it would seem best to stick to simple situations—in particular, getting and setting elements in wrapper collections.

It is possible that future versions of the AP CS Exam will use autoboxing to simplify code examples. But there would be no "trick questions" about syntax trivia, and autoboxing would only be used in simple and unambiguous situations.

Implementing Generic Types

AP Computer Science students learn to implement linked lists, hash tables, and binary trees. It seems reasonable to expect students to implement generic classes that exactly mimic the behavior of the standard collection classes.

In simple cases, this is indeed easy in Java 5.0. For example, a generic list node class can be defined like this:

```
public class ListNode<E>
{
    public ListNode(E value, ListNode next) { . . . }
    public E getValue() { . . . }
    public ListNode getNext() { . . . }
    private E value;
    private ListNode next;
}
```

A `ListNode<E>` holds a value of type `E`. A linked list with element type `E` is composed of `ListNode<E>` objects:

```
public class LinkedListImpl<E>
{
     . . .
     private ListNode<E> link;
}
```

However, with other data structures, the situation is more complex. Consider a binary search tree. The tree node values should implement the `Comparable` interface so that we can compare them. The syntax for expressing this constraint is:

```
public class TreeNode<E extends Comparable<E>>
{
     . . .
}
```

Note that `Comparable` is a generic interface in Java 5.0, defined like this:

```
public interface Comparable<T>
{
     int compareTo(T other);
}
```

The type parameter specifies the type of the `other` parameter of the `compareTo` method. This is an improvement over the "raw" `Comparable` interface that required a cast in implementations of `compareTo`. For example:

```
public class Person implements Comparable<Person>
{
     public int compareTo(Person other)
     {
          return id - other.id;
               // no need to cast other
     }
     . . .
     private int id;
}
```

Because `Person` implements the `Comparable` interface, we can form a `TreeNode<Person>`. But if we tried to form a `TreeNode<Rectangle>`, then the compiler would complain that `Rectangle` does not implement `Comparable<Rectangle>`. That is good.

But now something very unpleasant happens. Suppose we want to form a subclass of `Person`, say `Student`:

```
public class Student extends Person { . . . }
```

Can we form a `TreeNode<Student>`? No—`Student` doesn't implement `Comparable<Student>`, only `Comparable<Person>`. This is an unreasonable restriction since we can obviously compare two `Student` objects. To overcome this restriction, you have to relax the constraint on the generic type, like this:

```
public class TreeNode<E extends Comparable<? super E>>
{
     . . ..
}
```

This means "`E` is a type that implements `Comparable<?>`, where `?` is an anonymous type that is a supertype of `E`." It may be possible to explain this to a beginning student, but it is far removed from the material that we want to study, namely the implementation of binary search trees.

There are other pitfalls. Suppose we want to implement a dynamic array:

```
public class ArrayListImpl<E>
{
     public ArrayListImpl(int capacity)
     {
           elements = new E[capacity]; // ERROR
     }

     private E[] elements;
}
```

Unfortunately, it is not legal to construct an array of a generic type. This restriction is due to the implementation of generics through "type erasure," something that you probably

don't want to explain to your students. There are workarounds (after all, the Java library designers managed to implement `ArrayList<E>`), but they are not student friendly.

I think that data structures are best covered in the traditional way, using collections of type `Object` or the "raw" `Comparable` type without a type parameter. However, it is not clear how this issue will be tackled in college-level courses, and the Development Committee is tracking this issue.

Conclusion

The Java 5.0 release has more new language features than any Java release since 1.0. Many of these features are of marginal interest to beginning students. However, several features are compelling because they make programs easier to read, in particular:

- Generic collections
- The "for each" loop
- Autoboxing

Many college-level texts have embraced these features, and a future version of the AP CS Java subset may include some or all of them. Finally, Java 5.0 provides convenient classes for console input and formatted output.

Web Resources for Collection Classes

Debbie Carter
Lancaster Country Day School
Lancaster, Pennsylvania

Teaching Gems

The Card Game Assignment
(Nifty Assignments 2004)
John K. Estell
A GUI-based assignment that uses `Lists`, distributable card images (made available under the GNU General Public License), and step-by-step instructions for developing the classes. (You fill in the details of the card game that you choose; these resources can be used to develop any card game that uses standard playing cards.)
http://nifty.stanford.edu/2004/EstellCardGame

Computer Science Labs (Revised)
Roger Frank
A diverse group of student assignments, many of which deal specifically with collection classes. (With choices like "Fish Tree," "Tropical Fruits," and "Superstition," you're certain to find something fun for your students.)
www.rfrank.net/cslabs/cslabs.htm

LJV: Lightweight Java Visualizer
John Hamer, Department of Computer Science, University of Auckland, New Zealand
A tool for visualizing Java data structures, LJV (available under the GNU General Public License) uses *Graphviz* (open-source licensed software from AT&T Labs). The Web site shows sample diagrams of an `ArrayList` and a `HashMap`.
www.cs.auckland.ac.nz/~j-hamer/LJV/TeacherIntro.html

Reference Materials

Abstract Data Types in Java
(Portion of "Lecture Notes for COMPSCI.220FT")
Georgy Gimel'farb
Section 2.2, "ADTs and Java Classes," discusses the Java 1.2 `Collections` framework. Page 43 has a detailed diagram of the `Collections` framework hierarchy.
www.citr.auckland.ac.nz/~georgy/teaching/2001/220FT/pdf-files/220cha02.pdf

Special Focus: Using the Java Collections Hierarchy

Introductory Java Programming Tutorial
Richard G. Baldwin
Links to individual lessons and articles on various topics. Of special note are the "Data Structures in Java" tutorials, parts 1 to 8. From a review by Leigh Ann Sudol in AP Central's Teachers' Resources Area:

> The tutorials on data structures take a reusability approach to the collection classes they describe. The author emphasizes reuse versus reinvention. Since many of the methods that must normally be created by students studying different data structures are already included in the collection classes (such as `add()` to a `Tree`), he stresses that curriculums should shift their focus to application of these data structures, not reimplementation of them. The differences between the collection classes are emphasized by consistent coding (very similar programs, just a change in data structure) with the same test data. The tutorials analyze and explain the output in terms of how the chosen data structure affects what the data does.

www.dickbaldwin.com/tocint.htm

Collection Classes in Java: Part 2
Department of Computer Science, University of Waikato
A slide lecture that covers the following topics: containers; `Collections` and `Maps`; container taxonomy diagrams (showing the relationships between the various interfaces and classes); `Iterators`; functionality of `List`, `Set`, and `Map`; and hashing and hash codes. Includes an example of a concordance using a `HashMap`. (The PDF file has four slides per page, so it's not appropriate for classroom projection, but it's a great reference.)
www.cs.waikato.ac.nz/Teaching/COMP209B/Collections2.pdf

Java Tutorial: "Trail: Collections"
Joshua Bloch
Seven lessons on the `Collections` framework.
http://java.sun.com/docs/books/tutorial/collections/index.html

Sun Developer Network: Technical Articles and Tips
"Choosing a `Collections` Framework Implementation" (February 20, 2003)
John Zukowski
http://java.sun.com/developer/JDCTechTips/2003/tt0220.html#1

"Using `HashSet`, `LinkedHashSet`, and `TreeSet`" (November 5, 2002)
Glen McCluskey
http://java.sun.com/developer/JDCTechTips/2002/tt1105.html#1

The Collections Framework
Sun Microsystems
Includes Java™ 2 SDK, standard edition documentation (version 1.4.2).
http://java.sun.com/j2se/1.4.2/docs/guide/collections

Planet Java Tutorial: "The Collection API"
John Hunt
Explains each interface and class, and includes sections on `Iterators` and choosing a collection class.
www.jaydeetechnology.co.uk/planetjava/tutorials/language/Collections.PDF

Collection Class Excerpts from Online Textbooks

Note: Free, downloadable source code is available for all texts in this list.

***Introduction to Programming Using Java*, version 4.1, June 2004**
David J. Eck
http://math.hws.edu/javanotes/index.html

- Section 8.3: "Dynamic Arrays, `ArrayLists`, and Vectors"
  `ArrayLists`
  http://math.hws.edu/javanotes/c8/s3.html
- Section 12.2: "`List` and `Set` Classes"
  `List` and `Set` interfaces, `Iterators`, `ArrayList`, `LinkedList`, and `TreeSet`. (`HashSet`,briefly)
  http://math.hws.edu/javanotes/c12/s2.html
- Section 12.3: "Map Classes"
  `Map` interface, hash tables, `TreeMap`, and `HashMap`
  http://math.hws.edu/javanotes/c12/s3.html
- Section 12.4: "Programming with Collection Classes"
  http://math.hws.edu/javanotes/c12/s4.html

Special Focus: Using the Java Collections Hierarchy

***Thinking in Java*, 3rd ed.**
(Online or downloadable version)
Bruce Eckel
www.mindview.net/Books/TIJ/

- **Chapter 11: "Collections of Objects"**
  Look for the section called "Introduction to Containers."

***Java Au Naturel*, 4th ed., May 2004**
William C. Jones, Jr., Central Connecticut State University
www.cs.ccsu.edu/~jones/book.htm
This text's copyrighted material is available free of charge for teaching, provided you fill out and submit a five-minute questionnaire. PDF files, source code, and syllabi are provided.

- Chapter 7, section 7.11: "Implementing `Queue` as a Subclass of `ArrayList`"
  www.cs.ccsu.edu/~jones/chap07.pdf
- Chapter 15: "Collections and Linked Lists"
  `Iterator`, `ListIterator`, `LinkedList`
  www.cs.ccsu.edu/~jones/chap15.pdf
- Chapter 16: "Maps and Linked Lists"
  `Map` interface, `HashMap`, and `TreeMap`
  www.cs.ccsu.edu/~jones/chap16.pdf

Contributors

Information current as of original publish date of September 2005.

About the Editor

Fran Trees taught AP Computer Science from 1983 to 2001 in Westfield, New Jersey. She presently teaches CS1/CS2 at Drew University in Madison, New Jersey. Fran is a College Board consultant for AP CS, an Exam Leader, and AP Central's content adviser for computer science.

Debbie Carter is a computer coordinator at Lancaster Country Day School in Lancaster, Pennsylvania, where she teaches computer science and assists faculty with technology integration. She is a Question Leader for the AP Exam and a College Board consultant.

Christian Day has been teaching computer science at Emma Willard School since 2001 and was an instructor in computer science at Phillips Exeter Academy from 1996 to 2001. Christian has been an AP Exam Reader since 2001.

Bekki George teaches computer science and math at James E. Taylor High School in Katy, Texas. She also coaches the Academic Decathlon team at Taylor. Bekki is a College Board consultant for AP Computer Science and a Reader for the AP CS Exam.

Cody Henrichsen teaches computer science at Granger High School in West Valley City, Utah, and is a Reader for the AP Exam. Cody also teaches American history and coaches the debate team.

Cay S. Horstmann is a professor of computer science in the Department of Mathematics and Computer Science at San Jose State University, California, and author of many popular computer science text books. Cay is presently a member of the AP Computer Science Development Committee.

Pat Phillips taught computer science at Craig High School in Janesville, Wisconsin, for 25 years and served as instructional manager. Pat also works with the Microsoft Faculty Advisory Board and advises the Dreams Club at Craig, a club for girls interested in math, science, and technology.

Leigh Ann Sudol is a mathematics and computer science teacher at Fox Lane High School in Bedford, New York. Leigh Ann is also a College Board consultant and coauthor of *Java Software Structures for AP® Computer Science (for the AB Exam)*.

Contact Us

College Board Regional Offices

National Office
Advanced Placement Program
45 Columbus Avenue
New York, NY 10023-6992
212 713-8066
Email: ap@collegeboard.org

AP Services
P.O. Box 6671
Princeton, NJ 08541-6671
609 771-7300
877 274-6474 (toll free in the U.S. and Canada)
Email: apexams@info.collegeboard.org

AP Canada Office
1708 Dolphin Avenue, Suite 406
Kelowna, BC, Canada V1Y 9S4
250 861-9050
800 667-4548 (toll free in Canada only)
Email: gewonus@ap.ca

AP International Office
Serving all countries outside the U.S. and Canada
45 Columbus Avenue
New York, NY 10023-6992
212 373-8738
Email: apintl@collegeboard.org

Middle States Regional Office
Serving Delaware, District of Columbia, Maryland, New Jersey, New York, Pennsylvania, Puerto Rico, and the U.S. Virgin Islands
2 Bala Plaza, Suite 900
Bala Cynwyd, PA 19004-1501
866 392-3019
Email: msro@collegeboard.org

Midwestern Regional Office
Serving Illinois, Indiana, Iowa, Kansas, Michigan, Minnesota, Missouri, Nebraska, North Dakota, Ohio, South Dakota, West Virginia, and Wisconsin
1560 Sherman Avenue, Suite 1001
Evanston, IL 60201-4805
866 392-4086
Email: mro@collegeboard.org

New England Regional Office
Serving Connecticut, Maine, Massachusetts, New Hampshire, Rhode Island, and Vermont
470 Totten Pond Road
Waltham, MA 02451-1982
866 392-4089
Email: nero@collegeboard.org

Southern Regional Office
Serving Alabama, Florida, Georgia, Kentucky, Louisiana, Mississippi, North Carolina, South Carolina, Tennessee, and Virginia
3700 Crestwood Parkway NW, Suite 700
Duluth, GA 30096-7155
866 392-4088
Email: sro@collegeboard.org

Southwestern Regional Office
Serving Arkansas, New Mexico, Oklahoma, and Texas
4330 South MoPac Expressway, Suite 200
Austin, TX 78735-6735
866 392-3017
Email: swro@collegeboard.org

Western Regional Office
Serving Alaska, Arizona, California, Colorado, Hawaii, Idaho, Montana, Nevada, Oregon, Utah, Washington, and Wyoming
2099 Gateway Place, Suite 550
San Jose, CA 95110-1051
866 392-4078
Email: wro@collegeboard.org