PROFESSIONAL DEVELOPMENT

AP® Computer Science
Advanced Object-Oriented Concepts

Curriculum Module

The College Board

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 College Board is composed of more than 5,700 schools, colleges, universities and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,800 colleges through major programs and services in college readiness, college admission, guidance, assessment, financial aid and enrollment. Among its best-known programs are the SAT®, the PSAT/NMSQT®, the Advanced Placement Program® (AP®), SpringBoard® and ACCUPLACER®. 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.

Contents

Introduction

Paul Tymann
Computer Science Department
Rochester Institute of Technology
Rochester, New York

Inheritance

Inheritance is arguably one of the most powerful features of an object-oriented language. In fact, programming languages that provide all of the functionality associated with object-oriented programming but do not provide support for inheritance are usually classified as object-based languages, as opposed to object-oriented languages.

Inheritance is easy to understand conceptually and is something that we encounter all the time in our lives. For example, there are certain properties and behaviors we expect all automobiles to exhibit. Whether we are driving a sports car, a sedan or a minivan, we would expect those vehicles to have a steering wheel, gas pedal and brakes. The ability to relate different types of objects based on the common behavior they exhibit is extremely powerful. In the case of automobiles, by focusing on their common behavior, it is possible to generalize about the behavior of an automobile. So, once I know how to drive one type of automobile, I can drive any other type of automobile.

Inheritance establishes a relationship between one or more classes, making it possible to share the structure and/or behavior common to the classes. For example, both the `java.util.ArrayList` and `java.util.LinkedList` classes implement the `java.util.List` interface. The `java.util.List` interface defines the behavior that is common to lists. So when I need a list in a Java program, it does not matter if I use a `java.util.ArrayList` or a `java.util.LinkedList` since they both exhibit the same behavior. So, like the automobile example, once I know how to use a list in a program, I can use any list.

In fact, given that instances of a subclass contain all the state and behavior associated with their superclass, this means that an instance of a subclass can mimic the behavior of the superclass. Because a subclass implements all of the behavior of its superclass, I can substitute an instance of a superclass with one of its subclasses in a program. So not only

does inheritance allow me to group classes together based on the behavior they exhibit, but it also allows me to substitute different subclasses in a program with no observable effect. In the list example above, this means that if a program uses a `java.util.LinkedList`, it can be replaced with an instance of a `java.util.ArrayList`, and the program will still work correctly. This gives the programmer the ability to select different implementations of a list in order to satisfy runtime performance constraints of the program by making only trivial changes to the code.

Inheritance can take on a variety of forms in an object-oriented program and can be used in different ways. Five of the most common forms of inheritance are listed below:

Table 1 - Five Forms of Inheritance[1]

Form of Inheritance	Description
Specification	The superclass defines the behavior that is implemented in the subclass. This guarantees that the subclass implements the same behavior as the superclass.
Specialization	The subclass is a specialized form of the superclass but satisfies the specification of the superclass in all aspects.
Extension	The subclass adds new functionality to the superclass but does not change any inherited behavior.
Limitation	The subclass restricts the use of some behavior from the superclass.
Combination	The subclass inherits features from more than one superclass. This form is commonly called multiple inheritance.

Often when inheritance is discussed in a classroom, the focus is on code reuse. While code reuse is an important and practical use of inheritance, the ability to generalize the behavior exhibited by one or more classes is even more powerful. In fact, if you think about the "is a" relationship that is often used as a test to determine if two classes are related by inheritance, you will realize that the term is focusing on the behavior that two classes share rather than their structure. For example, when I say an alarm clock "is a" clock, you are more likely to think about the fact that both of these objects provide the ability for a user to determine the current time, rather than the fact that they both might be made using the same electronic components. Focusing only on code reuse prevents the students from learning about this powerful feature of inheritance.

1. Taken from *Modern Software Development Using Java* by Tymann and Schneider.

Multiple Inheritance

Most instructors do not discuss multiple inheritance (i.e., the combination form of inheritance) in the classroom because it is not supported in Java. While this is a reasonable approach, I think it is important to explain to students *why* multiple inheritance is not supported in Java. The way that I explain this is to ask them to consider a class called "radio alarm clock" that inherits from both the "radio" and "alarm clock" classes. Given the inheritance relationship in this example, there is no need to write methods, such as "set clock," "turn radio on" or "set alarm," because these methods are inherited from one of the two superclasses.

Now, ask your students to consider a method named "turn on" in the "alarm clock radio" class. Clearly, both the radio and alarm clock class have "turn on" methods. If I simply invoke "turn on" in the radio alarm clock class and do not provide an implementation, which "turn on" method (i.e., the one in "alarm clock" or the one in "radio") is executed? This creates an excellent opportunity in the classroom to discuss the fact that you could add syntactic and semantic structures to the language to define what happens in a case such as this, but at what cost? The designers of languages such as C++ decided it was worth making the language more complex and support multiple inheritance, whereas the designers of languages such as Java decided not to complicate the language.

At this point, it is easy to explain why classes can implement multiple interfaces. All that an interface does is to specify the behaviors that an implementing class must provide. An interface does not provide a mechanism to specify the implementation of these behaviors. Thus, in the "radio alarm clock" example, if the "radio alarm clock" implemented the "radio" and "alarm clock" interfaces, it would not matter if both interfaces specified the "turn on" method. As long as the "radio alarm clock" class provides a "turn on" method, it satisfies the requirements of both of the interfaces.

Inheritance and Java

Java provides several mechanisms for programmers to utilize the various forms of inheritance in their code. A class that extends a superclass is usually using either the specialization or extension forms of inheritance. There are many good examples of this type of inheritance in the GridWorld case study. For example, the `Chameleon` critter is an example of the specialization form of inheritance. A `Chameleon` critter "is a" critter, except that it takes on the color of the objects it encounters in the grid. It reuses some of the behavior and state provided by the `Critter` class.

There are at least two ways to utilize the specification form of inheritance in Java. An interface provides a programmer with the ability to specify the behavior that all implementing classes must exhibit. An interface only allows for the sharing of the

specification of behaviors that a class is to implement. Abstract classes also provide a mechanism for utilizing the specification form of inheritance in a program. In an abstract class, unlike an interface, it is also possible to extend or specialize the behavior provided by the superclass.

One interesting point that is worth discussing in the classroom is whether or not inheritance breaks encapsulation. One of the goals of object-oriented programming is to encapsulate the implementation of a class. The argument is that it is not necessary to know how a class works in order to use it. Encapsulation also makes it possible to treat objects like components that can be easily assembled into a working program.

Consider, for example, a superclass that defines an instance variable using protected access. This allows any subclass to directly access the instance variable. What happens if the implementation of the superclass is changed so that this instance variable is no longer needed? It would force all of the subclasses that use the instance variable to be rewritten. Is this a violation of encapsulation? Would a better programming practice be to define all superclass instance variables as private and provide a method that subclasses could use to access the instance variable? Clearly, each approach has its advantages and disadvantages.

This Module

Given that the GridWorld case study provides many excellent examples of the specialization and extension forms of inheritance, this module focuses on the specification form of inheritance. The first component of this module, "Interfaces: Motivation, Creation and Use," introduces interfaces and shows how they can be designed and how they can be used. The second component, "The Standard Java Interfaces: Using the Interfaces," builds upon the idea presented in the first component by showing how interfaces are used in the design of the Java collection framework. The final component, "Simple Abstract Classes," demonstrates the use of abstract classes in a Java program. The appendixes contain embedded class assignments, suggestions and worksheets.

Interfaces: Motivation, Creation and Use

Richard Kick
Newbury Park High School
Thousand Oaks, California

Motivation to Explore Interfaces

Programming is an artistic process that requires analytical techniques and creative talents, combined with the feedback that only experience can bring, in order to generate a useful product. Implementing code that provides functionality and service over a broad range of circumstances is often difficult. An interface is a programming tool that facilitates the production of code, which does provide such a broad purpose under conditions that are anticipated at the time of design, as well as for circumstances that were not anticipated when the code was created. The following examples are provided to illustrate the power and purpose of the interface.

Mathematical Example

Functions

Imagine you are asked to write software that uses a variety of mathematical functions. If you know exactly what functions are to be used, then you can implement classes for each category of function and use those functions effectively. For example, the collection of functions provided with the `Math` class, such as `sin` and `pow`, are easily evaluated by using their static nature and calling `Math.sin(x)` and `Math.pow(base, exponent)`. Linear functions can be evaluated by computing `slope * x + yIntercept`. However, if additional categories of functions need to be added at a later date, it is preferable to minimize the amount of code that needs to be changed in order to accommodate the expansion of functions. Rather than writing each function class as an independent entity, you can create the classes in a way that links them through a common type.

Looking at the generalities that all functions share, we recognize that every function can be evaluated. Considering real valued functions of one variable, we can ensure that each function has the method `evaluate`, which accepts a double value argument and returns a double value. How a particular function implements the evaluation process is impossible to specify without identifying the particular type of function. Yet all functions can be evaluated by the definition of a function.

Instructional Activity

Plan the Lesson

Understanding interfaces and the motivation to use them.

Most introductory students believe that implementing code for specific purposes is the quickest, most efficient way to problem solve through programming. This lesson will help students understand the use of abstraction for effective problem solving.

This lesson will take two class sessions (50 minutes each) of student work.

Teach the Lesson

Have students list all of the types of mathematical functions with which they are familiar. Have students list all common and different capabilities for the listed functions.

Generate a chart listing types of mathematical functions of one variable and their capabilities. Some examples are linear, absolute value, polynomial, rational, trigonometric, and piecewise-defined functions. Identify the differences and commonalities among the functions. Use the chart template that is provided in Appendix 1.

Use the Web to search for various categories of mathematical functions. For example, the square root function found at http://en.wikipedia.org/wiki/Square_root and piecewise functions can be found at http://mathforum.org/library/drmath/view/53255.html.

Find and explore recent Java questions using interfaces (2008 A4, AB 3, AB4, 2007 A4, AB4, 2006 A2, A4, AB2, 2004 AB1).

Interface Creation

The `evaluate` method for any real valued mathematical function is shown below.

```
double evaluate(double argument)
```

In order to create a new type that represents this shared functionality, use the mechanism called an interface.

```
/**
   * Function is the common functionality of real valued functions
*/
public interface Function
{
      /**
      * Provides the range value associated with a given argument
      *
      * @param arg the argument of the function
      * @precondition arg is in the domain of the function
      * @return the range value that is paired with arg
      */
   double evaluate(double arg);
   // Note: no need to specify that the method is public here.
   // All methods declared in an interface are public
}
```

Notice that the code for an interface resembles the code of a traditional class. However, an interface has no constructor, no instance fields and no implementation of methods. The interface is simply a list of the functionality shared by some family of objects.

Instructional Activity

Plan the Lesson

Students should define an interface that represents the most general mathematical function of one variable. The syntax of creating an interface is often confused with the syntax of traditional class creation. Students should take care to not declare instance fields, not implement methods and not use { } after method declarations. They should also be certain to include a semicolon after method declarations.

This lesson will take one class session (50 minutes) of student work.

Teach the Lesson

Provide students with distinct lists of functionality, and have students convert the list into specific interfaces.

Search the Java API (found at http://java.sun.com/javase/6/docs/api/) for real world interfaces.

List the methods for particular interfaces like `ActionListener`, `Comparable`, `Iterator` and `Set`. Describe programming problems for which the use of the interfaces that you listed would be appropriate.

Have students propose new interfaces that could be used to solve particular problems. For example, when writing code for board games, having a common collection of methods such as `move`, `checkForWin` and `calculateScore`, could allow for multiple games to share common code that uses these methods.

Interface Use

The process of wrapping the `evaluate` functionality in an interface creates a type that expresses a shared functionality for all real valued functions. Thus, it is possible to write code that uses `Function` objects to perform calculations without knowing any of the implementation details of how particular functions are evaluated. For example,

```
List<Function> funList = new ArrayList<Function>();
double sum = 0.0;
for (Function f : funList)
{
   sum += f.evaluate(arg);
}
```

and

```
List<Function> funList = new ArrayList<Function>();
for (double x = xmin; x <= xmax; x += xIncrement)
{
   plot(x, funList.get(x).evaluate(arg) );
}
```

Note that the sum of functions can be used for important tasks such as the conversion of analog signals into digits signals and the generation of hard-to-define functions from information about the characteristics of the slopes related to the function. For further information, see

http://en.wikipedia.org/wiki/Fourier_series

and

http://en.wikipedia.org/wiki/Taylor_series

The code above can be written and tested for correctness without having `Function` objects for which the code will be eventually be used. This level of abstraction in the problem-solving process allows for the division of code production without losing code compatibility or thorough testing.

Notice that the collection of `Function` objects is stored in an `ArrayList`, yet the reference used to access and manipulate the collection is a `List` reference and not an

`ArrayList` reference. This illustrates the spirit of interface usage. Write code using the specific type needed to accomplish the task but do not use a type any more specific than necessary. While this example creates an `ArrayList`, another user may create a `LinkedList`, and your code solution should still work with that type of list. Writing your code in terms of the general `List` interface allows for its more flexible use with any class that implements the `List` interface.

Let us examine the collection of code that illustrates this design example.

Instructional Activity

Students will use interfaces to write general solutions to problems using the general type represented by the interfaces given in the previous examples.

Plan the Lesson

Require students to write applications using interfaces. Many students will attempt to declare variables of specific types rather than using the more general interface type. Verify that all solutions use the most general types that can be used to solve the proposed problems.

This lesson will take two class sessions (50 minutes each) of student work.

Teach the Lesson

Have students write solutions to specified problems, given particular interfaces that define types that are related to these problems (see worksheet provided in this document).

Provide examples of code that uses more specific types and discuss the limitations of these code examples in terms of modifiability. It may be helpful to use the `GameCharacter` example in the worksheet found in Appendix 3. Have students design a `Citizen` class instead of the `GameCharacter` interface. After they complete their classes, have students add game characters that include dogs, space aliens and butterflies. Ask them if their `Citizen` class had the instance fields and methods that are needed to accurately represent these new participants in the game. Most, if not all, students will find that their original `Citizen` class is too limited to represent participants that are much more diverse than what was anticipated in the early stages of design. Emphasize to students how powerful interfaces can be for the facilitation of future program enhancement.

Use the Java API to explore interface examples using library interfaces. In particular, have students explore the `Shape` interface and examine the number of known classes that implement it. Ask students if the list of shapes represents all possible shapes that they may use in a program. Ask them to explain how the `Shape` interface can help them write code to solve problems related to shapes. Have students discuss how the use of the `Shape` interface would facilitate the use of new shapes in an application long after the solution code for shape-related problems was written.

Reflect on the Lesson

It is important that students are engaged in conversations that are related to topics in their areas of interest in order for them to effectively connect their knowledge and experience to the abstract concept of the interface. Before talking about interfaces, talk about programming project ideas that students would like to complete before the end of the course. Have students explore their own interests and at the same time help to inspire other students by sharing their ideas in small group and class discussions. Use code examples where interfaces were used to generalize solutions that were shared between projects. The Sudoku programming module (http://en.wikipedia.org/wiki/Taylor_series) is one example. Students will be at different levels of abstraction capability. Some students will be quite good at generating ideas and suggesting solutions that involve interfaces. Others will need to be shown multiple examples and then produce similar examples by modifying what was provided. Allow students to use their strengths and abstraction abilities to make progress toward greater understanding. Appendix 2 presents a messaging example to assess student understanding of the concepts discussed so far.

Adjust the Lesson

For students that struggle with abstract thought, you may wish to provide specific code examples and have them ask questions about each interface implementation. For students that are skilled abstract thinkers, you can provide opportunities to create interfaces for a collection of related software problems and then have them implement the interfaces in order to solve the problems in a common way. For both groups, it is advised that the students use interfaces that are currently available in the Java API to improve their understanding of the appropriate use of interfaces and to recognize how interfaces can be used to generate solutions to related problems, thus minimizing the amount of required code.

Summary

The purpose of this module is to have students i) recognize the usefulness of interfaces in problem solving, ii) design specific interfaces for particular problems and iii) implement interfaces during the problem-solving process. This lesson could be used with the Gridworld case study by asking students to define a set of classes that inherit from `Actor`. Have students discuss the commonality of the classes that are to be created, and then ask them to abstract that commonality into an interface. Have students create an `ArrayList` of objects of the interface type that they created, and then have students call the common functionality of the set of objects in order to solve a particular GridWorld problem. If working under a block schedule, the design and implementation could be accomplished in a single 90-minute block. Under a traditional schedule, students will probably be asked to design an interface during class, implement the design for homework and solve the related GridWorld problem during the next class period.

Bibliography

Java API 6.0 http://java.sun.com/javase/6/docs/api/index.html.

Horstmann, Cay S. 2008. *Java Concepts* (5th Edition), Somerset, NJ: John Wiley & Sons Inc.

Resource List

The Java Tutorials at java.sun.com are filled with examples and discussions from the creators of the Java language. In particular, a general interfaces tutorial can be found at http://java.sun.com/docs/books/tutorial/java/IandI/createinterface.html.

A tutorial on interfaces related to collections written by Josh Bloch can be found at java.sun.com/docs/books/tutorial/collections/index.html.

Rich Pattis' website, www.ics.uci.edu/~pattis/, has an interesting introduction to interfaces. Specifically, you can find the interface material at www.ics.uci.edu/~pattis/ICS-21/lectures/interfaces/index.html.

The Standard Java Interfaces: Using the Interfaces

Laurie White
Mercer University
Macon, Georgia

Introduction

Larry Wall, the author of the Perl programming language, has stated that the three great virtues of a programmer are laziness, impatience and hubris. While this module may not be able to do much about impatience or hubris, the Java standard interfaces can be seen as a way to do a lot of work without much effort, the goal of the lazy everywhere.

In this section, the `Comparable` interface is used to allow for reuse of code for many different types.

As a result of this session, students should be able to:

- write code using objects of type `Comparable`.
- illustrate how one method with `Comparable` parameters can be called with numerous different class types.

The standard Java interfaces are part of the AP® CS curriculum (topic II.C.). Writing code to use interfaces is also part of the AP CS curriculum (topics I.B. and II.B.).

Instructional Activity

Plan the Lesson

This activity can either be used to motivate the study of interfaces, before interfaces have been introduced in the class, or can be used as an example of using interfaces. If it is used to motivate interfaces, the teacher should be sure to review general information about interfaces and how they fit into object-oriented programming in the introduction to this module.

The first part of this activity is designed to be done by the instructor with class input. The second part is designed to be done by the students either individually or in small groups. Ideally, the second part will be done in a closed lab setting with the instructor available to direct students, answer questions and point out similarities to the first part.

Obviously, the teacher should be familiar with interfaces (from the introductory material to this module or from a tutorial like the one in the Resources section). The teacher should also work through the material at least once to see what questions might arise.

The amount of time this activity can take is highly variable. If your students are comfortable with writing methods, you should be able to work through the first few examples quickly before getting to the need for interfaces. Alternately, you can take time to reinforce the technique for building methods. In a college classroom, the instructor part of the activity would take about one 50-minute period, and the student part would take another 50 minutes.

Teach the Lesson

Section 1: Motivation

The focus of this lesson will be writing methods to compute the larger of two values. All of these values will be class instances. Let's start with a class that should be very familiar to the student: the `String` class.

The teacher should introduce the idea of a method to find the larger of two strings, using whatever technique is typically used to present new problems in class. I typically use test-driven development, so I will present a class similar to the following. (All code examples are available at http://apcs.lauriewhite.org/java-interfaces/.)

```
public class LargerOfRunner {
   public static void main(String[] args) {

      // The strings to compare. I like to use humorous values.
      // You should feel free to use whatever string values work
      // best with your students.
      String str1 = "Parrot";
      String str2 = "Spam";

      String larger = largerOf (str1, str2);
      System.out.println ("The larger of " + str1 + " and " + str2
             + " is " + larger);

   }
}
```

I then will start creating the method to find the larger of the two strings. Since I also emphasize Javadoc commenting before the method body is written, I will build the following method shell in the `LargerOfRunner` class.

```
/**
 * Return the larger of two strings.
 * @param first a string to compare
 * @param second a string to compare
 * @return the larger of the two strings
 */
private static String largerOf(String first, String second) {
   String result;

 //insert code here

   return result;
}
```

Of course, your code at this point may vary, depending on what you and your students are comfortable with. Once the basics of the method are done, it shouldn't be too difficult to write the method. The most important point is to remember that `String`s are compared using the `compareTo` method. This is something I will stop and wait for the class to remember on their own since it's such a common error. This should give us a final method like the following:

```
/**
 * Return the larger of two strings.
 * @param first a string to compare
 * @param second a string to compare
 * @return the larger of the two strings
 */
private static String largerOf(String first, String second) {
   String result;
   if (first.compareTo(second) > 0)
      result = first;
   else
      result = second;
   return result;
}
```

At this point, I can compile and test the program if I am doing this on a projected computer. If you are having students work along with you, make sure everyone has the correct code at this point.

Once this method is done, repeat this for a second class type that implements the `Comparable` interface. You can see a list of all classes that implement `Comparable` in the API for `Comparable` (Java APIs are available at http://java.sun.com/javase/6/docs/api). Alternatively, you may have a class definition provided by your textbook that implements `Comparable`. This is also a nice place to bring in GridWorld since the `Location` class implements `Comparable`.

For this example, let us use the `Integer` class. Depending on the level of your class at the time of this activity, you may want to work through all the steps that were done for comparing two `Strings`. You will want to use the same method and parameter names, so that the similarity is as obvious as possible. You should end up with code that looks like:

```
/**
 * Return the larger of two Integers.
 * @param first an Integer to compare
 * @param second an Integer to compare
 * @return the larger of the two Integers
 */
private static Integer largerOf(Integer first, Integer second) {
   Integer result;
   if (first.compareTo(second) > 0)
      result = first;
   else
      result = second;
   return result;
}
```

Hopefully, the similarities between the first code and the second code are fairly obvious at this point. If they're not, do it again (here with GridWorld's `Location` class).

```
/**
 * Return the larger of two Locations.
 * @param first a Location to compare
 * @param second a Location to compare
 * @return the larger of the two Integers
 */
private static Location largerOf(Location first, Location second){
   Location result;
   if (first.compareTo(second) > 0)
      result = first;
   else
      result = second;
   return result;
}
```

Once the class notices the similarities, have them look carefully at what is the same in all of the methods. The basic method structure is given below:

```
private static TYPE largerOf(TYPE first, TYPE second) {
   TYPE result;
   if (first.compareTo(second) > 0)
      result = first;
   else
      result = second;
   return result;
}
```

All you need to create a method to find the larger of two values is to change *TYPE* to the type of those values. While this may satisfy those of you who want an easy answer, it doesn't work for the lazy. Instead of defining more methods, the lazy would like to have one method that works for any type.

Section 2: Interfaces to the Rescue

I often make intentional mistakes in class to let students know that what might seem like an easy solution won't necessarily work. The easy-looking solution here is to just replace *TYPE* with the type `Object`. (If your class doesn't know `Object`, or you don't teach by bad example, you may skip this.) Replace all of the `largerOf` methods by the single method using `Object`.

```
/**
 * Return the larger of two Objects.
 * @param first an Object to compare
 * @param second an Object to compare
 * @return the larger of the two Objects
 */
private static Object largerOf(Object first, Object second) {
   Object result;
   if (first.compareTo(second) > 0)
      result = first;
   else
      result = second;
   return result;
}
```

As nice as this result seems, it won't work, giving a compiler error message similar to:

```
The method compareTo(Object) is undefined for the type
Object
```

At this point, teachers should take a break from the example and discuss the following with the class:

1. Why is `compareTo` not defined for `Object`?

Possible answer: there are some objects for which comparison makes little or no sense. Consider `ArrayLists` and GridWorld's `Bugs`.

2. Could we solve this by adding a class that all of these inherited from that defines `compareTo`?

Practical answer: No, these types are already defined.
More theoretical answer: No, they don't have the "is a" relationship. Instead, they share a similar functionality, they all have a `compareTo` method.

So, we need a new device to say that all of these classes have a similar functionality, the ability to be compared. This is not done with classes. Instead, to indicate what special functionality a class has, we use interfaces.

So, to indicate that a class has the ability to be compared, we have the interface `Comparable`. Since I like my students to see the API as much as possible, I show them the API for `Comparable` interface (http://java.sun.com/javase/6/docs/api/java/lang/Comparable.html). There's a lot of text here, but by this point in the course, they know to skip over the text on first reading and jump to the Method Summary. All it takes for a class to be `Comparable` is that it implement the `compareTo` method. And that's all we need here!

I also look at the APIs for the classes we've used (`String`, `Integer` and GridWorld's `Location`) and point out that all of them implement the `Comparable` interface.

Once we know about the `Comparable` interface, we're almost done. Instead of `Object`, the single `largerOf` method will use `Comparable` as its type.

```
/**
 * Return the larger of two Comparable objects.
 * @param first an item to compare
 * @param second an item to compare
 * @return the larger of the two items
 */
private static Comparable largerOf(Comparable first,Comparable
          second){
   Comparable result;
   if (first.compareTo(second) > 0)
      result = first;
   else
      result = second;
   return result;
}
```

Notice that this solution is not perfect. The simple problem is that because the return type is `Comparable` instead of any specific type, we will need to cast to the appropriate type in the testing code.

A more difficult problem is that `Comparable` is a raw type. The AP CS subset does not currently include generics, so the version `Comparable<T>`, which is preferred by most Java compilers, is not part of the subset. The good news is that this will cause a warning rather than an error, so the code will still run. (Teachers who want to explain how to parameterize `Comparable` can do so at this point, but since it is not part of the AP Java subset, it will not be covered here.)

Since I'm usually projecting my work to the class, I take a few minutes at this point to illustrate you cannot *create* variables of type `Comparable`. You certainly can declare a variable that has the comparable functionality, as is done in the code above. But the variable is just a reference to an object that has those functionalities. I'll typically just tell the class this first, then try a line like:

```
Comparable result = new Comparable(); // WRONG
```

and then, when the code fails and I get the students' attention, I'll tell the class again why this failed.

Section 3: The Students' Turn

Once the `Comparable` interface has been introduced and students have seen how creating one method can handle any class that implements `Comparable`, they can be asked to create another method. Sample assignments that use the `Comparable` interface are included in appendixes 4 through 6. These include:

- `Comparable` Lab: a closed lab assignment that has students write a method with parameters of type `Comparable`. This should be fairly straightforward and will serve as an immediate review of the work presented by the teacher.
- Sorting with the `Arrays` Class: the `Comparable` interface can seem like much ado about very little until students are exposed to the power of code that has already been written for them. This laboratory exercise (which can be done individually or as a class example) introduces students who know the basics of arrays to the sort method in the `Arrays` class. (Similar functionality exists within the `Collections` class as well, if teachers want to repeat this lab with the `ArrayList` to allow students to compare and contrast arrays and `ArrayLists`. I usually will repeat array-based labs with the same lab using `ArrayLists` to help encourage critical thinking on the part of my students.)
- Sorting with `Comparables` Assignment: Once the class has been exposed to arrays and sorting, they can be asked to rewrite one of the sorts covered in class to sort any element that implements `Comparable`.

Section 4: Assessment

Question 7 from page 22 of the AP Computer Science A Course Description is appropriate for this topic:

> Consider the following declarations.
>
> ```
> public interface Comparable
> {
> int compareTo(Object other);
> }
> public class SomeClass implements Comparable
> {
> // … other methods not shown
> }
> ```
>
> Which of the following method signatures of `compareTo` will satisfy the `Comparable` interface requirement?
>
> I. `public int compareTo(Object other)`
> II. `public int compareTo(SomeClass other)`
> III. `public boolean compareTo(Object other)`
>
> (A) I only
> (B) II only
> (C) III only
> (D) I and II only
> (E) I, II, and III

There can be many variations on this theme.

If you are looking for free-response questions, almost any question that asks students to compare values can be adjusted to use the `Comparable` interface. Consider:

- Write a method called `maxOf3` that will return the largest of three `Comparable` parameters sent to it.
- Write a method called `max` which finds the largest value in an array (`ArrayList`) of `Comparable` elements.

Section 5: Reflect on the Lesson

Historically, programmers have a bad case of "Not Invented Here." They resist using code that someone else has written, preferring to reinvent the wheel themselves over and over and over.

I find that lessons like this, especially when we cover how simple it is to sort almost any array, are very useful. Since many of my students are not going on in computer science,

I would much prefer they know how to call a great system sort than to write a mediocre $O(n^2)$ sort when the time comes for them to actually use sorting. Because I emphasize code reuse and reading the API, my students are more than ready to find new ways to make their life easier. If your students resent this or don't see the benefit of `Comparable`, emphasize more of the methods in `Arrays` and `Collections` classes, such as binary search. I often compare learning new techniques to learning shortcuts. The first time you take a shortcut, it may seem longer, since the old way is so familiar, but by the third or fourth time you take it, you can see how the shortcut is saving you time. (I usually also learn two or three new shortcuts in each class, since my students know the town better than I do and like to share something so familiar to them.)

Bibliography

Wall, L., Christiansen, T. & Orwant, J. 2000. *Programming Perl (3rd Edition)*, Sebastapol, CA: O'Reilly Media.

Resource List

Almost all textbooks for this course include a brief (or longer) description of the standard interfaces.

For complete details, see Sun's site at
http://java.sun.com/javase/6/docs/api/java/lang/Comparable.html.

Part 3 of the GridWorld Student Manual has a brief discussion of how the Location class implements the `Comparable` interface.

Simple Abstract Classes

Stacey Armstrong
Cypress Woods High School
Houston, Texas

Introduction

This unit will present abstract classes in a way that will allow students to SEE how they are used and why. When teaching abstract classes, I try to give students topics that are relevant and serve a practical purpose for abstract classes, and I try not to make them seem overly complicated.

Instructional Activity

Plan the Lesson

Abstract classes should be explained and demonstrated before starting the exercises that follow. Students need to see and have an opportunity to experiment with abstract class examples before they are asked to write code that builds on abstract classes.

Students who typically do not see the point of abstract classes have a tough time with the idea of abstract versus concrete. I explain to students that abstract classes are used when you know what you want to do and a little about how you are going to do it. With abstract classes, some of the ways in which you are going to do it is still unknown. Interfaces are used when you know what you want to do but have no idea how you are going to do it. With interfaces, all of the ways in which you are going to do it is still unknown.

The presentation of abstract classes should take approximately 90 minutes.

This lab/exercise should take approximately 90 minutes as well.

Lesson Introduction

To start, bring out a deck of cards. Ask the students to tell you the value of the each card. Show a 9, for instance, and ask the value. Almost every student in the class will say it is worth 9. I tell them they are all wrong. They get very upset and agitated, but they are all very engaged. They ask, "How could it not be worth 9?" My answer is simply that I have not stated what game we are playing, nor have I stated the rules of the game. Cards are worth nothing until placed into a game. The rules of each card game make the cards concrete. I have each student name a few games they play and how the rules change what some cards are worth. I end this intro by stating that we know what we want to do, but we do not have all of the information about how it will happen. We know we want the cards to have value, but we cannot give the value until we know the exact game and the rules that go with that particular game.

Card Game Example

Canasta is a card game that incorporates two decks of 52 cards plus four jokers.

The point values for the cards used in Canasta are as follows :

The Jokers are worth 50 points each.

The Ace and 2 are worth 20 points each.

The King, Queen, Jack, 10, 9 and 8 are all worth 10 points each.

The 7, 6, 5 and 4 are all worth 5 points each.

The value of each card in Canasta is based on the rules of the game and not on the value printed on the actual cards.

Topic Presentation

Present abstract classes and provide relevant examples of abstract classes. Try to provide real world examples of abstract things. I use a mammal as a great example of an abstract thing. You would never see JUST a mammal walking down the hall. Students find this very funny, and I throw in a few jokes here as well. You would see a human, which is a type of mammal, walking down the hall. The mammal example follows up the card intro very well, in that with a mammal, we know what we want to happen but not how it will happen. With a human, the unknown is known, and we can make the abstract concrete. A human will inherit all things from a mammal that all mammals share, and the things specific to a human must be provided. A whale would inherit all things from a mammal that all mammals share, and the things specific to a whale must be provided. Allow students to explore and tinker with the examples provided. Give them time to mess up the examples and ask questions about why the examples do what they do.

Lab Activity

Each student will create an abstract class called `FantasyFootballPlayer`. This class will be the backbone for a project in which students will create different types of player classes to represent each of the player types in a fantasy football game.

The following class is the basic class for this project. Students will be asked to write this class and then extend it to create subclasses that represent all of the player types in fantasy football.

In the game of fantasy football, all players have yards and touchdowns. The score for specific types of players is different based on the position. This structure creates a simple example of using abstract classes. We will put all of the common/known stuff in the `FantasyFootballPlayer` abstract class. We will make the `getScore()` method abstract, as the score for each type of player is unique to its type.

```
public abstract class FantasyFootballPlayer
{
   private double numYards;
   private int numTouchdowns;

   public FantasyFootballPlayer()
   {
      numYards = 0.0;
      numTouchdowns = 0;
   }

   public FantasyFootballPlayer(double numYrds, int numTchs)
   {
      numYards = numYrds;
      numTouchdowns = numTchs;
   }

   public void setNumYards(double yards)
   {
      numYards = yards;
   }

   public void setNumTouchdowns(int touchdowns)
   {
      numTouchdowns = touchdowns;
   }

   public double getNumYards()
   {
      return numYards;
   }
```

```
    public int getNumTouchdowns()
   {
      return numTouchdowns;
   }

      /* Scoring bassed on player type:
      touchdowns - 6 pt for WR – wide receiver,
                   TE – tight end,
                   RB – running back,
                   K - kicker,
                   DEF – team defense
      touchdowns - 4 pt for QB – quarter back
      yards - 1 pt per 10 yards for RB
      yards - 1 pt per 15 yards for WR and TE
      yards - 1 pt per 25 yards for QB */
   public abstract double getScore();

   public String toString()
   {
      return getNumYards() + " - " + getNumTouchdowns()
                                   + " - " + getScore();
   }
}
```

The following class extends the `FantasyFootballPlayer` class and implements the `getScore()` method. All of the subclasses of `FantasyFootballPlayer` will share the same instance variables and methods. Each subclass will have its own specific implementation of `getScore()`.

```
   public class RunningBack extends FantasyFootballPlayer
   {
      public RunningBack(double numYards, int numTouchs)
   {
      super(numYards, numTouchs);
   }

   //have students add more constructors

   public double getScore()
   {
      return getNumYards()/10.0 + getNumTouchdowns()*6;
   }
}
```

The following class extends the `FantasyFootballPlayer` class and implements the `getScore()` method.

```
   public class QuarterBack extends FantasyFootballPlayer
   {
      public QuarterBack(double numYards, int numTouchs)
   {
      super(numYards, numTouchs);
   }

   //have students add more constructors

   public double getScore()
   {
      return getNumYards()/25.0 + getNumTouchdowns()*4;
   }
}
```

Any or all of the other player types can be created.

Student Samples

The following is a student project using the structure setup described in the previous section. Austin Baker of Cypress Woods High School created the following code.

```
   public class FantasyFootballTeam
   {
      private ArrayList<FantasyFootballPlayer> yourTeam;
      private String owner;
      private double teamScore;
      private double teamYardage;
      private int teamTouchdowns;

      public FantasyFootballTeam(String ownerName)throws
           Exception
      {
         yourTeam=new ArrayList<FantasyFootballPlayer>();
         owner=ownerName;
         teamScore=0.0;
         teamYardage=0.0;
         teamTouchdowns=0;
         Scanner file=new Scanner(new File("FantasyPlayers.dat"));
         int range=file.nextInt();
         file.nextLine();

         for(int x=0;x<range;x++)
         {
```

```
            String pos=file.next();
            if(pos.equals("RB"))
            {
               yourTeam.add(new
                  RunningBack(file.nextDouble(),file.nextInt()));
            }
            if(pos.equals("QB"))
            {
               yourTeam.add(new
                  QuarterBack(file.nextDouble(),file.nextInt()));
            }
            //add in more player types
         }

         setScore();
         setYardage();
         setTouchdowns();
   }

   public void setScore()
   {
      double tempScore=0;
      for(FantasyFootballPlayer p:yourTeam)
      {
         tempScore+=p.getScore();
      }
      teamScore=tempScore;
   }

   public void setYardage()
   {
      double tempYards=0;
      for(FantasyFootballPlayer p:yourTeam)
      {
         tempYards+=p.getNumYards();
      }
      teamYardage=tempYards;
   }

   public void setTouchdowns()
   {
      int tempTouchdowns=0;
      for(FantasyFootballPlayer p:yourTeam)
      {
         tempTouchdowns+=p.getNumTouchdowns();
```

```
      }
      teamTouchdowns=tempTouchdowns;
   }

   public String toString()
   {
      String output="";
      output+=owner+"'s Team::\n";
      output+="Team Score = "+teamScore+"\n";
      output+="Team Yardage = "+teamYardage+"\n";
      output+="Team Touchdowns = "+teamTouchdowns+"\n";
      int pCount = 1;
      for(FantasyFootballPlayer p:yourTeam)
      {
        output+="player " + pCount++ + " - " +
        p.toString()+"\n";
      }
   return output;
   }

   public static void main(String[] args)throws Exception
   {
      FantasyFootballTeam a=new FantasyFootballTeam("Austin");
      System.out.println(a);
   }
}
```

Data File

```
2
RB 1000.0 7
QB 4325.2 19
```

Feedback

The student program shown above demonstrates completion of the project. Students were asked to create the abstract class, subclasses and a program that demonstrated clear understanding of the project as a whole. Austin Baker decided to create a stats program, using a data file to house the data for his program. He could easily expand the program to handle complete teams that could be compared and sorted. Each student would have the option to create any type of program to demonstrate that all pieces function and serve some larger purpose. The example provided by Austin is a pretty nice solution.

Reflect on the Lesson

The intent of this lesson is to present abstract classes in a simple and easy-to-understand way so that students can relate to the idea. Many times, students just do not see the point of abstract classes as the examples given are too large and cluttered. If abstract classes are simplified and presented in a fun way, students can grasp and master the concepts very easily.

Adjust the Lesson

Depending on the learning styles of the students, the project could be adjusted to allow students an opportunity to present other examples of abstract classes from their own experiences. Students could be given an opportunity to create videos and podcasts using interactive media.

It could also be very advantageous to use groupings for this project, depending on the dynamics of the class and learning styles of the students.

This lesson could also be expanded to simulate scoring for entire teams and leagues. Students would need to add more constructors and instance variables in order to identify the player and the teams.

The assignment could also be completed graphically, using GridWorld.

Summary:

Goals

The goal of this lesson is to use something most students are familiar with to teach a concept most students are unfamiliar with. Most all students have heard of football and many play fantasy football or know someone that plays. The rules of the game are simple and provide an excellent opportunity to use abstract classes.

Adaptions

This lesson could be expanded to have students make more constructors and to possibly make the assignment graphical, using GridWorld.

Resource List

Armstrong, Stacey. 2008. *A+ Computer Science: Computer Science Curriculum Solutions.* http://apluscompsci.com.

http://www.pagat.com/. *Card Games and Card Game Rules.*

2004 AP CS A Exam, question #2: http://apcentral.collegeboard.com/apc/public/repository/ap04_frq_compsci_a_35988.pdf.

Assessment

Tracy Ishman
Plano West Senior High School
Plano, Texas

Exit Polls

On days when new information is being conveyed, spend the last 5 minutes of class gathering feedback from the students. This is an informal (nongraded) opportunity for students to convey their understanding of the concepts being discussed. Use the feedback to determine areas for reteaching or further discussion the following day. Have the students take out half of a sheet of paper and respond to one of the following prompts. Students hand their responses to you on their way out the door.

Suggested Questions	Hoped-for Response(s)
Describe the difference between an abstract method and a concrete method.	"Abstract methods have a semi-colon and no method body; a concrete method has curly braces with statements inside."
Explain the purpose of using interfaces.	"Allows code to use a different implementation with minimal changes." "Can pass a parameter using the interface name and then give it an object of any implementation."

The X class is implementing the Y interface and is a subclass of the Z class. Write the heading for the X class.	`public class X extends Z implements Y`
	or
	`public class X implements Y extends Z`

Describe a similarity between interfaces and abstract classes.	"An object cannot be instantiated from an interface or abstract class." "They can both contain abstract methods."

Suggested Questions	Hoped-for Response(s)
Describe a difference between interfaces and abstract classes.	"An abstract class could also contain concrete methods." "An abstract class can contain non-public members." "An abstract class can contain instance variables/ constants and static variables."
The X class is implementing 3 out of 5 of the methods defined in the Y interface. Its subclasses are responsible for implementing the remaining methods. Write the heading for the X class.	`public abstract class X implements Y`

Interfaces and Abstract Classes

Seeing design patterns being used in the real world helps students understand their value. The `java.util.ArrayList` class is one of many examples of the extension of an abstract class to finish implementing an interface. The `ArrayList` class extends the `AbstractList` class, which implements the `List` interface. Other examples:

- The `HashMap` class extends the `AbstractMap` class, which implements the `Map` interface.
- The `PriorityQueue` class extends the `AbstractQueue` class, which implements the `Queue` interface.
- In GridWorld, the `BoundedGrid` class extends the `AbstractGrid` class, which implements the `Grid` interface.

Have students research the Java API to find several examples of the interface-abstract class–concrete class relationship. You may want to limit the number of examples that come from the `Collections` hierarchy.

Writing Interfaces

The `Test` interface defines an interface for various types of tests. All classes that implement the `Test` interface need to write the following three methods.

- `getTopic` — returns a string containing the topic of this test; no parameters are needed
- `setPointsPerQuestion` — sets the number of points earned for each correct response and the number of points lost for each incorrect response; requires two parameters for each of these floating-point values; no data is returned

- `calculateScore` — calculates the score based on the number of correct and incorrect responses and the total number of questions; requires three parameters for these integer values; returns the number of points earned as a floating-point number

Write the `Test` interface.

Expected Response:

```
public interface Test
{
   String getTopic();
   void setPointsPerQuestion(double ptsCorr,double ptsIncorr);
   double calculateScore(int numCorrect, int numIncorrect);
}
```

Implementing Interfaces

Write the `CSTest` class that implements the `Test` interface. A `CSTest` score is calculated by earning points for each correct answer and losing points for each incorrect answer but does not earn or lose points if a question is left unanswered. For example, a test may award 3.5 points for each correct answer and deduct 0.5 points for each incorrect answer.

Each `CSTest` object maintains the topic of the test (e.g., Recursion, Loops, etc.), the total number of questions, and the number of points earned for correct answers and lost for incorrect answers. The `CSTest` class has a constructor that takes two parameters: the test topic and the total number of questions. Write the `CSTest` class.

Expected Response:

```
public class CSTest implements Test
{
   private String topic;
   private int numQuestions;
   private double pointsCorrect;
   private double pointsIncorrect;

   public CSTest(String testTopic, double numQ)
   {
      topic = testTopic;
      numQuestions = numQ;
      pointsCorrect = pointsIncorrect = 0;
   }

   public String getTopic()
   {
      return topic;
   }
```

```
    public void setPointsPerQuestion(double ptsCorr,
           double ptsInc)
    {
      pointsCorrect = ptsCorr;
      pointsIncorrect = ptsInc;
    }

    public double calculateScore(int numCorrect, int
           numIncorrect)
    {
      double corr = numCorrect * pointsCorrect;
      double incorrect = numIncorrect * pointsIncorrect;
      return corr - incorrect;
  }
}
```

Appendix 1

Function Chart

Category of Math Function	Capability
Linear	Linear functions can be used to determine the slope and the intercepts of a line. They can also be used to determine the coordinates of points on a line; specifically, they can be used to determine the y-coordinate of a point on a line, given the x-coordinate.
Absolute value	Absolute value functions can be used to determine the distance a value lies from the origin. Given an x-value, the absolute value function evaluates to a y-value that is always positive.
Polynomial	Polynomial functions are defined as the sum and difference of terms. A term is the product of variables and constants.
Rational	Rational functions are defined as the ratio of two polynomial functions.
Piecewise-defined	Piecewise-defined functions are defined using different function definitions over subsets of the function domain.

Category of Math Function	Capability

Are there differences in the types of functions that you listed? If so, what are they?

Are there any common capabilities among the functions you have chosen? If so, what are they?

Appendix 2

Problem Solving with Interfaces

Often, students write code that is designed to solve narrowly defined problems without taking into account generalizations that would allow for more robust solutions. Leading students through the process of abstraction can facilitate an enhanced understanding of key programming concepts. This understanding can enable them to create tools that are useable not only for solving problems that are under consideration but also for developing code that solves related problems not necessarily currently defined.

Formative Assessment

After completing an introductory programming course, students should be familiar with the process of modeling concepts through class creation. It is common that the resulting classes are limited in terms of the contexts in which they can be effectively used. The ability to see beyond the concrete and to grasp abstractions from multiple problem contexts is developed through participation in programming activities over an extended period of time. The following formative assessment is designed to provide experiences in abstraction and problem solving using the concept of an interface.

Messaging

Many electronic devices used today have the ability to send and receive text messages. Cell phones commonly use the short message service (SMS) to exchange these messages, while most computers are equipped to send and receive instant messages (IM) and e-mail. Writing classes that handle each of these types of messages can leave programmers bogged down in a pool of details that require considerable time and attention. In the past, programming teams would often have to wait for details to be implemented before building upon the results of the message handling software. However, if the problem is examined in terms of common functionality, interfaces can be developed that list this functionality and allow for the effective creation of software that uses the method list of the interface before specific classes that implement the messaging details are completed. This embedded classroom activity will lead students through a series of steps that help them better understand and effectively apply the process of concept abstraction using interfaces.

Illustrative Interpretive Framework

A set of questions is provided that could be used to prompt the students to reveal their thinking during each of the segments of the scenario.

1. Assessment of Basic Understanding of Problem Definition

Possible student-teacher exchanges: Messaging exists in several forms, but these forms are related by common functionality.

Q: What are some of the types of messaging systems that exist today?

Q: Of these types of messaging, which are electronic?

Q: What functionality is provided by text messaging systems?

Q: What functionality is common to all text messaging systems?

2. Modeling

Possible student-teacher exchanges: Messaging systems can be implemented using object-oriented programming techniques.

Q: What are typical scenarios that occur daily among users of electronic text messaging systems?

Q: What are the nouns that commonly appear in the descriptions of these scenarios?

Q: What are the verbs that commonly appear in the descriptions of these scenarios?

Q: From the lists of nouns and verbs, what are likely classes and methods that would be created in the development of a text messaging system program?

Q: Are there any methods that are common to multiple classes?

Q: How can you abstract these common methods so they can be effectively accessed by classes that use this functionality?

Q: What are the advantages and disadvantages of using both classes and interfaces versus using classes only?

3. Implementation and Testing

Possible student-teacher exchanges: Implementing large systems often requires teams of programmers independently developing portions of the project code.

Q: What should the interface be for any class that uses electronic text messaging?

Q: How can you write code that uses objects that implement an interface before the actual details of individual classes are completed?

Q: How can you test software that uses objects which implement particular interfaces?

Appendix 3

Coding Problems

Student Worksheet

1. You are designing a video game in which game characters move throughout the game world, move toward each other, greet each other and then travel in a new location.

    a. Although specific game character classes have not yet been defined, create a Java interface called `GameCharacter` that declares this functionality that is associated with the characters. In particular, include the following methods:

moveTo –	takes a `GameCharacter` parameter and moves this `GameCharacter` to the character represented by the parameter.
greet –	takes a `GameCharacter` parameter and greets the `GameCharacter` character represented by the parameter.
travelNorth –	takes no parameter and moves this `GameCharacter` to the north.

    b. Write Java statements that take two objects of type `GameCharacter` called `friend1` and `friend2`, moves `friend1` to `friend2`, and then has the friends greet each other.

    c. Write a method `interaction` that takes an `ArrayList` of `GameCharacter` objects as a parameter, randomly selects a `GameCharacter` as a greeter, randomly selects a `GameCharacter` to be greeted and then performs the greeting.

2. Shapes are to be used to create geometric designs on a graphics screen for geometry students. Students using the shapes software are to be given exercises related to the area of the shapes.

    a. Create a Java interface called `GeometricShape` that declares the functionality described below.

draw –	displays this `GeometricShape` on a graphics screen with which it is associated.
area –	returns the area of this `Shape` in the form of a `double` value.

b. Write a method `displayAll` that takes an `ArrayList` of `GeometricShape` objects as a parameter and draws all shapes on a graphics screen with which all objects are associated.

c. Write a method `totalArea` that takes an `ArrayList` of `GeometricShape` objects as a parameter and returns the sum of all areas of all shapes in the form of a `double` value.

Student Worksheet — Answer Key

1. You are designing a video game in which game characters move throughout the game world, move toward each other, greet each other and then travel in a new location.

    a. Although specific game character classes have not yet been defined, create a Java interface called `GameCharacter` that declares this functionality that is associated with the characters. In particular, include the following methods:

moveTo –	takes a `GameCharacter` parameter and moves this `GameCharacter` to the character represented by the parameter.
greet –	takes a `GameCharacter` parameter and greets the `GameCharacter` character represented by the parameter.
travelNorth –	takes no parameter and moves this `GameCharacter` to the north.

```
public interface GameCharacter
{
   /*
    * moves to GameCharacter represented by parameter
    * @param citizen GameCharacter to which this object moves
    */
   void moveTo(GameCharacter citizen);

   /*
    * greets the GameCharacter represented by the parameter
    * @param citizen GameCharacter that is greeted
    */
   void greet(GameCharacter citizen);

   /*
    * moves this GameCharacter in the northerly direction
    */
   void travelNorth();
}
```

    b. Write Java statements that take two objects of type `GameCharacter` called `friend1` and `friend2`, moves `friend1` to `friend2` and then has the friends greet each other.

```
friend1.moveTo(friend2);
friend1.greet(friend2);
friend2.greet(friend1);
```

c. Write a method interaction that takes an `ArrayList` of `GameCharacter` objects as a parameter, randomly selects a `GameCharacter` as a greeter, selects the `GameCharacter` to be greeted as the next object in the `ArrayList` (use the first object if the last object in the list is chosen as greeter), performs the greeting, and then travels toward the north.

```
public void interaction(ArrayList<GameCharacter> allCharacters)
{
   int size = allCharacters.size();
   int greeterIndex = (int) ( size * Math.random() );
   int greetedIndex;

   if (greeterIndex == size - 1)
   {
      greetedIndex = 0;
   }
   else
   {
      greetedIndex = greeterIndex + 1;
   }

   GameCharacter greeter = allCharacters.get(greeterIndex);
   GameCharacter greeted = allCharacters.get(greetedIndex);
   greeter.moveTo(greeted);
   greeter.greet(greeted);
   greeter.travelNorth();
}
```

2. Shapes are to be used to create geometric designs on a graphics screen for geometry students. Students using the shapes software are to be given exercises related to the area of the shapes.

   a. Create a Java interface called `GeometricShape` that declares the functionality described below.

      draw – displays this `GeometricShape` on a graphics screen with which it is associated.

      area – returns the area of this `Shape` in the form of a `double` value.

```
public interface GeometricShape
{
   /*
    * draws this GeometricShape on the default graphics screen
    */
   void draw();
```

```
   /*
    * Provides access to the area of the GeometricShape
    * @return the area of this GeometricShape
    */
    double area();
}
```

b. Write a method `displayAll` that takes an `ArrayList` of `GeometricShape` objects as a parameter and draws all shapes on a graphics screen with which all objects are associated.

```
public void displayAll(ArrayList<GeometricShape> allShapes)
{
   for (GeometricShape shape : allShapes)
   {
      shape.draw();
   }
}
```

c. Write a method `totalArea` that takes an `ArrayList` of `GeometricShape` objects as a parameter and returns the sum of all areas of all shapes in the form of a `double` value.

```
public double totalArea(ArrayList<GeometricShape> allShapes)
{
   double sum = 0.0;
   for (GeometricShape shape : allShapes)
   {
      sum += shape.area();
   }
   return sum;
}
```

Appendix 4

Comparable Lab

APCS Lab: Working with Comparables

This lab builds on our class on the Java standard interface `Comparable`. You'll have the chance to create your own method of dealing with `Comparable` objects.

Lab Preparation

Read through this lab carefully and review the material on `Comparable` in your textbook.

Materials Needed

This sheet, your course text and your course notebook.

Lab Setup

Note to instructor: Customize the instructions here to match your lab environment. There is just one file that students will need to be able to edit. Import the files from the `Comparable` Lab directory to a new project.

Getting Started

Look at the program in `CompareItemRunner`. You will need to complete the method `getComparison` to return the correct string, based on the values of its parameters.

```
public class CompareItemRunner {

   public static void main(String[] args) {

      // The string items to compare.
      String str1 = "Parrot";
      String str2 = "Spam";

      String relationship = getComparison (str1, str2);
      System.out.println(str1 + " is " + relationship + " " + str2);

      relationship = getComparison (str2, str1);
      System.out.println(str2 + " is " + relationship + " " + str1);
      relationship = getComparison (str1, str1);
```

```
      System.out.println (str1 + " is " + relationship + " " + str1);
   }
   /**
    * Return a string which expresses the relationship between
               two strings.
    * @param first a string to compare
    * @param second a string to compare
    * @return
    *      "equal to" if the strings are equal
    *      "greater than" if the first string is greater than
               the second string
    *      "less than" if the first string is less than the
               second string
    */
   private static String getComparison(String str1, String str2) {

      // insert code here

   }
}
```

To ensure you understand what you are being asked to do, give all the expected output of the final program below:

Note to instructor: I usually ask my students to show me this before going on.

Once you are sure you understand the problem, insert the appropriate code into `getComparison`, and get your program to compile and run correctly.

Adding Another Type

The program works for `Strings`, but what if you want to compare another type, for instance, integers? Add the following lines to the main program and add a new method to return the comparison string for `Integers`.

```
Integer int1 = new Integer (42);
Integer int2 = new Integer (-321);

relationship = getComparison (int1, int2);
System.out.println (int1 + " is " + relationship + " " + int2);

relationship = getComparison (int2, int1);
System.out.println (int2 + " is " + relationship + " " + int1);

relationship = getComparison (int1, int1);
System.out.println (int1 + " is " + relationship + " " + int1);
```

First, what output do you expect from the revised main program once you've added the method?

Now, add the new method, compile it and ensure it works correctly.

We could add more types, one by one, but the lazy approach is to handle all `Comparable` items at one time. Replace the methods you've created for `getComparison` with `String` and `Integer` parameters with just one `getComparison` method that uses parameters of type `Comparable`.

Add test code to the main program to compare the `Locations` (2, 4) and (3, 1). You should not have to change anything else in the program to use `getComparison` with `Comparable` parameters.

Appendix 5

Sorting with the Arrays Class Lab

APCS Assignment: This lab builds on our class on the Java standard interface `Comparable`. You'll have the chance to use code that will sort `Comparable` objects.

Lab Preparation

Read through this lab carefully and review the material on `Comparable` in your textbook.

Materials Needed

This sheet, your course text and your course notebook.

Lab Setup

Note to instructor: Customize the instructions here to match your lab environment. There is just one file that students will need to be able to edit. Import the files from the Sorting with the Arrays Class Lab directory to a new project.

Getting Started

Look at the main program in `SortRunner`. It calls a method to read in an array of `String`s and prints out the strings in the array.

```
public static void main(String[] args) {
   String [] things = getStrings();

   // Add code here

   for (String thing: things)
      System.out.print(thing + " ");
}
```

Run the program and be sure you understand its basic function. Don't worry, there's not a lot going on!

Sorting the Array

It would be nice if the strings in the array were printed in alphabetical order. To do this, we need to sort the array. Sorting is an extremely common function in computer programs.

Give four different places you expect to see output sorted by some value in programs you use:

_____

_____

_____

_____

Since sorting is so common (and some data sets that are sorted can hold millions of records, if not more), it's important to understand efficient methods to sort arrays. Fortunately though, since sorting is so common, it's a problem that has already been solved. The `Array` class in Java has methods to perform common operations on arrays, including sorting them.

Take a look at the API for the `Arrays` class (http://java.sun.com/javase/6/docs/api/java/util/Arrays.html#sort%28java.lang.Object[]%29). There are a number of different sorts defined. The one we are interested in has the description:

sort
`public static void sort (Object [] a)`

> Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the `Comparable` interface. Furthermore, all elements in the array must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).
>
> This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.
>
> The sorting algorithm is a modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed n*log(n) performance.
>
> **Parameters:**
> a - the array to be sorted
>
> **Throws:**
> `ClassCastException` — if the array contains elements that are not *mutually comparable* (for example, strings and integers).

Instead of calling it a method of an instance of the class, we call it with the name of the class (much like the `Math` class's methods) since this is a static method. So, to use the `Arrays` sort to sort the array things, the call is:

```
Arrays.sort (things);
```

Try this. Describe how adding this one line changes your program.

Sorting Other Items

The `Arrays.sort` method does not just sort strings. It can sort any object which implements the `Comparable` interface. Let's try it out on a class that represents a `Song`.

Examine the class in the file Song.java.

```
/**
 * A class representing a song.
 * @author Laurie White
 *
 */

public class Song {

   /**
    * Create a new song
    * @param title The title of the song
    * @param artist The artist of the song
    * @param length The length of the song in seconds
    */
   public Song (String title, String artist, int length)
   {
   this.title = title;
   this.artist = artist;
   this.length = length;
}

   /**
    * Create a string with the information about this song.
    * @return a string representing the song
    */
   public String toString()
{
      return title + " (" + artist + "), " + length + " seconds";
}

private String title;
private String artist;
```

```
   private int length; // In seconds
}
```

Look at the fields and methods of the class.

How would you use the class to create a variable named `favoriteSong` representing your favorite song?

The file `SongSorter` will create and sort an array of `Song` objects. It is reproduced below.

```
import java.util.Arrays;
/**
 * A program to illustrate the use of the Arrays class with a
       user defined
 * class.
 * @author Laurie White
 *
 */
public class SongSorter {

   public static void main(String[] args) {
      Song [] songs = createSongs();

      Arrays.sort(songs);

      for (Song song: songs)
        System.out.println (song);

   }
   // This is a method to build an array.
   private static Song[] createSongs() {
      Song [] result = new Song [4];
      result[0] = new Song ("There's Hope", "India.Arie", 201);
      result[1] = new Song ("I Want to Hold Your Hand", "The
             Beatles", 144);
      result[2] = new Song ("American Pie", "Don McLean", 513);
      result[3] = new Song ("Fernando", "Abba", 254);
      return result;
   }
}
```

Add your favorite song to the array built in `createSongs`. Don't forget to change the array's size!

Try compiling the program `SongSorter`. What error message do you get?

_____

_____

_____

_____

You should have received an error message telling you that you can't sort items that are not comparable. This should make sense … how would you compare a swimming pool and a puppy, for example? But you can compare objects of type `Song`; you just need to indicate they are `Comparable` (by adding the line "`implements Comparable`" to the class header and including a `compareTo` method).

Do this:

- Add the line "`implements Comparable`" to the class header
- Create a `compareTo` method (see the API for `Comparable` if you don't remember what its header should be) that will compare two songs by comparing their names.
- Get your program to compile and run. Make sure you get the output you expect.

Just a Little Bit More

Try changing your `compareTo` method to compare two songs based on length. How would you have to change `compareTo` so that, when sorted, the *longest* song would appear first? (You can have `Arrays.sort` use different `Comparators` in the same program, but this is a bit more advanced than we want to get here.)

Without understanding anything about the mechanics of sorting, by making your class `Comparable` and using `Arrays.sort`, you can have Java do the hard work of sorting. You just have to worry about how items should be sorted when you create the `compareTo` method.

Appendix 6

APCS Assignment: Sorting with Comparables

This assignment can be customized to any type of sort covered in class. Replace "fun sort" with the name of the sort you want students to use (such as "selection sort" or "merge sort").

This lab builds on our class on the Java standard interface `Comparable` and how to sort arrays using the fun sort. You'll have the chance to code your own *fun sort* to sort `Comparable` objects.

The easiest version of this assignment will give students the framework below (or something with a `getThings` similar to what is used in class) and ask them to complete the `funSort` method.

```
public class FunSortRunner {
   public static void main(String[] args) {
      Comparable [] things = getThings();

      funSort(things);

      for (Comparable thing: things)
         System.out.print(thing + " ");
   }


   private static void funSort(Comparable[] things) {


   }
   // This method is just being used to get some values in an array.
   // It should be adjusted for whatever the problem domain
         is. It could be
   // as simple as Strings or as complicated as reading in
complex data about
 // songs, football players, or shoes.
 private static Comparable[] getThings() {
      String[] result = new String[] {"January", "February",
"March", "April"};
      return result;
   }
}
```

I prefer to have programming assignments that ask students to tie together numerous items, so I might instead describe a `Song` class (similar to that in Sorting with the Arrays Class lab), an input format for songs, and then ask students to read a list of songs and print them in order based on one of the fields. While they could do this without the `Comparable` interface, I require they use `Comparables` (and often will follow up with similar assignments where they can reuse their work if they do use `Comparables`).

About the Contributors

Paul Tymann is professor and chair of the Computer Science Department at the Rochester Institute of Technology in Rochester, N.Y. He has taught both basic and advanced computer science courses. His research interests include bioinformatics, operating systems, networking, parallel computing, and object-oriented programming and design. He is a current member of the 2010-11 AP® Computer Science Development Committee and has served as an AP Reader and Exam Leader for the AP Computer Science Exams.

Richard Kick is a computer science and mathematics instructor at Newbury Park High School in Thousand Oaks, Calif. He has taught and developed curriculum for multiple mathematics and computer science courses and has written and presented several papers on mathematics and technology in secondary education. He is a former AP Computer Science Development Committee member (2000–2004), an AP Computer Science Consultant, and has served as an AP Reader and Exam Leader for the AP Computer Science Exams.

Laurie White is a professor of Computer Science at Mercer University in Macon, Ga. She is responsible for the content, instruction and evaluation of classes in an ABET-accredited computer science degree program. White is the current chair of the 2010-11 AP Computer Science Development Committee and has served as an AP Reader and Exam Leader for the AP Computer Science Exams.

Stacey Armstrong is a computer science teacher at Cypress Woods High School in Houston, Texas. He serves on the ACM Education Policy Committee. He is also a member of the AP Computer Science Principles Commission and serves on the Texas University Interscholastic League Computer Science Advisory Committee. Armstrong is an AP Computer Science Consultant and has served as an AP Reader and Exam Leader for the AP Computer Science Exams.

Tracy Ishman is a computer science teacher at Plano West Senior High School in Plano, Texas. She established computer science courses at a new school, teaches regular and Advanced Placement® sections of Computer Science and develops curricula that include group activities, lab assignments, quizzes and multiple-choice/free-response questions designed to prepare students for the AP Computer Science Exam. Ishman is a former member of the AP Computer Science Development Committee (2006–2010) and has served as an AP Reader and Exam Leader for the AP Computer Science Exams.

CM10COMHB00800