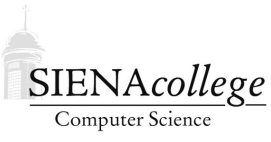

Computer Science 225
Advanced Programming
Siena College
Spring 2021

Topic Notes: Interfaces

You have worked in your earlier courses and at the start of this one with Java *interfaces*. A Java interface defines a set of methods that must exist in any Java class that *implements* that interface. This is a key tool in good *object-oriented design*.

A class that implements an interface in Java specifies this in its class header. For example:

```
public class ArrayList<E> implements List<E>
```

This topic is also only tangentially related to an upcoming topic: Java's Graphical User Interface mechanisms. That is, while this topic "Interfaces" and the upcoming topic "User Interfaces" share part of their name, the connection between the two topics is not very strong.

The advantage as we have seen so far is that objects of any class that implements an interface can be used interchangeably as long as the only methods used are those defined in the interface. This week in lab, you will see one way that this helps us to write highly *reusable* code when we wrote a method that operates on a `java.util.List` that can work with any type list that implements the interface, including `ArrayList`, `LinkedList`, and `Vector`.

We can do something similar with the `Iterator` interface. The following method can find the maximum value returned by an `Iterator<Integer>` no matter where that `Iterator` was created:

```
    public static int max(Iterator<Integer> i) {

        // we require at least one element for this to make sense
        int result = i.next();
        while (i.hasNext()) {
            int x = i.next();
            if (x > result) {
                result = x;
            }
        }
        return result;
    }
]
```

This is demonstrated in the `IteratorMethod` example.

In your study of custom classes and abstract data types, you saw that an important goal was to hide the details of how the behavior of a class was implemented by keeping its instance variables `private` and then defining `public` methods that provided the ability to change or determine a limited set of aspects of the object's state. This permits the author of such a class to make internal changes without breaking existing code that makes use of that class, as long as the *public interface* of the class does not change. For example, suppose you have an implementation of a linked list that did not maintain a variable that tracks the number of elements it contains (so its `size` method needs to walk down the list to count the number of elements it contains). You could add a variable to track this, make sure it is updated by all methods that modify the number of elements in the list, and replace the `size` method with a much more efficient implementation that just returns the value of the variable. Users of the class would not need to make changes to their code, but would benefit from the new, more efficient, `size` method (however we should be mindful that the structure would increase slightly in size and the additional code to maintain this variable would need to be executed on every list modification).

Other interfaces you have seen so far include `Comparator` and in lab you will work also with `Comparator`. We will soon use other interfaces from the standard Java API.

Writing Interfaces

Java allows us to define and use our own interfaces. The syntax is demonstrated in the `AnimalInterface` example.

- Like a class definition, an interface specification is constructed out of a header followed by a body enclosed within curly braces.
- The body consists of a list of method headers. Each header is terminated by a semicolon (rather than being followed by a method body).

A few points to note from this example:

- Each class includes the `implements Animal` qualifier in its class header.
- By doing this, it is required to provide the four methods listed in the interface definition.
- When implementing the methods required by an interface, it is good programming practice to include an `@Override` annotation before the method header. It is not an error to leave them out, but would be an error to include the annotation if you are not actually overriding a method. We will talk much more about overriding and related topics soon as we study inheritance.
- The classes that implement the interface can define any other methods they wish. In each case, there is also a `toString` method, and in the `Dog` class, we also add a dog-specific `bark` method.

- In the `Zoo` class, we take advantage of the interface by constructing an `ArrayList<Animal>` which can then accept instances of any class that `implements Animal`.
- When we loop over and print out each `Animal`, we see that it chooses the appropriate `toString` method and `numLegs` method.
- When we retrieve the elements from the `ArrayList<Animal>`, they are known to be of type `Animal`. If we want to check to see if an `Animal` is actually one of the specific animal types, we can use the `instanceof` operator. We do this to determine which `Animal` objects which are of type `Dog`, and for ones that are, we *cast* the reference to one of type `Dog`, at which point we can call the `Dog`-specific method `bark`.
- The cast in the example is guaranteed to be successful, since we just checked to make sure the `Animal` is an instance of `Dog` before casting. However, Java still needs to check, and if a cast is encountered where the object referenced is not actually an instance of the type in the cast, Java will throw a `ClassCastFailedException`.

Anonymous Classes

You might have noticed that many classes that implement simple interfaces, like `Iterator` or `Comparator`, tend to be short and in many cases are used only once. There are a few options about how and where such implementations can be defined.

- The class can be defined as `public` class in its own `.java` file. This is what you did for the `Comparator` in Lab 2: `IntegerAbsComparator`.
- The class can be defined as a non-`public` class in the same file where it is used. This is what was done for the existing `Iterator` and the one you added in the `SimpleLinkedList` class also in Lab 2. The details of the `Iterator` are visible only within the same `.java` file. It is returned as an `Iterator` from the list's `iterator` method. The same is done for the `LegsComparator` in the `ZooArray` program in the `AnimalInterface` example.
- An *anonymous class* can be used for this purpose. Here, the object is constructed and its required methods defined exactly where needed, without introducing a new class name or any variables or methods. This is done for the `Comparator` that compares `Animals` by weight in the `ZooArray` program in the `AnimalInterface` example.

Implementing Multiple Interfaces

- A class can implement zero or more interfaces.
- A class that implements multiple interfaces separates the intefaces that it implements by commas.

- The fact that Java allows a class to implement multiple interfaces introduces the potential for name collisions when more than one interface includes methods of the same name.
  - If the methods have different signatures, the methods are overloaded.
  - If they have the same signature and return type, the methods are collapsed into one. That is, a single matching implementation satisfies all such interfaces.
  - If they have the same signature but different return types, it will produce a compilation error.

We will see more about interfaces as we continue to study Java's support for object-oriented design.