INTERFACES AND ABSTRACT CLASSES

Interfaces for defining ADTs

While studying the Queue ADT, we saw:

- How to define an interface using the keyword `interface`.
- How to write a class that implements an interface using the keyword `implements`.

But we could have just written classes `ArrayQueue` and `CircularQueue`, without defining interface `Queue`.

So why bother to have the interface in addition to the classes?

Advantages of separating an ADT's interface from its implementation

$\boxed{1.}$ Easier to change implementations.

Recall our interface `Queue` for the Queue ADT, and our two implementations of it: `ArrayQueue` and `CircularQueue`.

Change this program to use a `CircularQueue`:

```
public class QueueTest {
    public static void fill(Queue q, int n) {
        for (int i=0; i != n; i++) {
            q.enqueue(new Integer(i));
        }
    }

    public static void main(String[] args) {
        Queue q = new ArrayQueue(15);
        fill(q, 15);
    }
}
```


How hard would it be if instead we had classes `ArrayQueue` and `CircularQueue` but no interface `Queue` above them?

The difference would be more dramatic with a large program.

It would be even worse if classes `ArrayQueue` and `CircularQueue` didn't use exactly the same method headers. The interface enforces that.

$\boxed{2.}$ Easier to use two implementations simultaneously.

Example:

```
Queue q1 = new ArrayQueue(15);
Queue q2 = new CircularQueue(22);
fill(q1, 15);
fill(q2, 15);
```

Some rules about interfaces

- A Java `interface` may only contain
  - instance method headers, and
  - constants (implicitly `static` and `final`).
- All these members are implicitly `public`.
- You cannot instantiate (construct instances of) an interface.
- If a class implements an interface, it must provide bodies for all methods — otherwise it must be an "abstract" class (more later).
- An interface can extend other interfaces.
- A class can extend only one class, but it can implement many interfaces.

Questions

Why would it be pointless to:

- Allow members of an interface to be private?
- Create an instance of an interface?

Would interfaces be useful if variables in Java didn't need to have their types declared?

Interfaces for general code

Suppose we want to be able to sort arrays of fractions, students, integers, cars, etc.

Once we've chosen a sorting algorithm, the logic for sorting is the same no matter what the objects are. So we certainly don't want to write a separate sort method for each kind of object!

We want a general method to sort any array of `Object`s.

```
public static void sort(Object[] a) {
    // Loop around and put things in order.
}
```

But now we're stuck: To put things in order, we have to be able to compare any two and decide which goes first. `Object` doesn't provide any method for doing this.

The problem

We want to provide a sorting service, but we need our subscribers to ensure that they give us only objects that have a particular property: they must be comparable.

An interface can help express this.

The comparable idea is so useful that it is in the Java API.

Interface `Comparable`

`Comparable` is part of the Java API. It contains only one method:

```
public interface Comparable {
    /**
     * Compares this object with o for order. Returns
     * a negative integer, zero, or a positive integer
     * as this object is less than, equal to, or
     * greater than o.
     */
    int compareTo(Object o);
}
```

A service that requires a `Comparable`

We can now write `sort()`:

```
public static void sort(Comparable[] a) {
    // Loop around and put things in order.

    // Use compareTo() to decide on the order:
    if (a[i].compareTo(a[j]) < 0) ...

    // Other details are unimportant here.
}
```

Expecting a class to `implement Comparable` (so that it can be sorted) is not onerous because:

- This requires only that it have a `compareTo()` method, the weakest requirement that will make sorting possible.
- As we mentioned, a class can `implement` as many interfaces as needed (unlike extending classes), so a class can subscribe to many services.

Using the sort service

`String implements Comparable`, so we can sort `String`s with our method:

```
String[] list = new String[10];
// Fill in list...
    :
sort(list);
// list is sorted!
```

Two nice features:

- Method `sort` is general: it can handle *anything* that is sortable, not just `String`s. Many Java API classes implement `Comparable`, and any class of your own design can be made comparable. (How?)
- We can write `sort()` without knowing anything about the particular `Comparable` objects that will be passed in.

We can write lots of other methods that use the `Comparable` interface. Ideas?

Properties of Objects vs of Classes

Consider the following code fragment:

```
Comparable[] list = new Comparable[2];
list[0] = "Hello";
list[1] = new Integer(57);
sort(list);
```

What do you think happens?

Being comparable is really a property of a *set* of objects. `String`s are `Comparable` *to each other*. This is called “mutually comparable” in the Java API.

There’s no completely safe way to specify this in Java (but there will be in an upcoming version of Java).

Our convention in this course is to assume that objects specified to be `Comparable` are mutually comparable, unless we say otherwise

Question: Does `equals(Object)` make sense for objects of two different classes?

A Design Pattern

Suppose we want to go through the elements contained in an object, doing something with each one.

Your first thought: go through by index.

```
// LinkedLists have elements, like Vector or an array.
LinkedList list = new LinkedList();
list.add(...);
...
for (int i = 0; i != list.size(); ++i) {
    ... do something with list.get(i) ...
}
```

For many data structures (`LinkedList` is one of them), getting the element at a particular index isn’t very fast, but moving from one element to the next is fast.

An Index with Memory

To get the ith element of a `LinkedList`, the list has to look through all the elements before it.

So, for example, `list.get(20)` takes twice as long as `list.get(10)`.

But if the `LinkedList` could "remember where" the ith element is it could get the $i + 1$st very quickly.

We'll learn more about this in the "Linked Data Structures" section of the course.

For reasons we discuss a bit later, instead of having the collection remember where the last thing we asked for is, we use another object as a kind of "smart index". This is a common approach, called an "Iterator".

An iterator is an example of a "design pattern".

Design Patterns

A design pattern is a solution to a common software design problem. It is a general, flexible template that can be applied in many different situations, and so it supports software reuse.

The classic reference on this topic is "Design Patterns", by Gamma, Helm, Johnson and Vlissides.

Iterators are such a good idea that the Java API already has an interface for them.

Interface Example: `Iterator`

It contains these methods:

```
/** Returns whether the iteration has more elements. */
boolean hasNext()

/**
 * Returns the next element in the iteration.
 * Throws:
 *     NoSuchElementException if the iteration has
 *         no more elements.
 */
Object next()

/**
 * Remove from the underlying collection the last
 * element returned by the iterator.  This method can
 * be called only once per call to next.
 * Throws:
 *    UnsupportedOperationException - if the remove
 *        operation is not supported by this Iterator.
 *    IllegalStateException - if the next method has
 *        not yet been called, or the remove method
 *        has already been called after the last call
 *        to the next method.
 */
void remove()
```


`Vector`s, `LinkedList`s, `ArrayList`s, and other collections have a method:

```
Iterator iterator()
```

which gives us an iterator that goes through the collection.

So instead of our previous `for` loop, we can write:

```
Iterator i = list.iterator();
while(i.hasNext()) {
    ... do something with i.next() ...
}
```

We're now asking the iterator `i` for the elements; it fetches them from `list`, but more efficiently than if we asked for them by number.

Notice that the size of the collection isn't needed. The iterator approach can even work with collections whose size is only known by going through them, for example the collection of lines in a file.

Why a Separate Object?

Couldn't we have LinkedList implement something like Iterator, and write code like:

```
list.startIteration();
while(list.hasNext()) {
    ... do something with list.next() ...
}
```

This would work, but using a separate object has two advantages:

1. We can have more than one traversal going *at the same time*.
2. There can be iterators that traverse the same collection in different ways. For example, Vector could have a method
   `Iterator reverseIterator()`
   giving us an iterator that goes through the Vector backwards.

Question: Is iterable best thought of as a property of some classes, or of some objects?

A service that requires an Iterator

```
/**
 * Remove from the collection underlying itr every
 * element that is less than c.
 * Requires: itr contains only Comparable objects.
 *           c != null.
 * Throws:
 *     UnsupportedOperationException if an element in
 *         the collection is not Comparable.
 */
public static void removeLess(Iterator itr,
                              Comparable c)
               throws UnsupportedOperationException {
    while (itr.hasNext()) {
        if (c.compareTo(itr.next()) > 0) {
            itr.remove();
        }
    }
}
```

removeLess() provides a service to clients, and requires that subscribers to it implement Iterator.

Question: What, because of a convention mentioned earlier, haven't we mentioned in the precondition?

Using the removeLess service

```
// Add Strings to a Vector and print its contents.
Vector v = new Vector();
v.addElement("hello");    v.addElement("apple");
v.addElement("sniffle");  v.addElement("bleep");
System.out.println("Before: " + v);
// Remove all Strings < "great" and reprint.
removeLess(v.iterator(), "great");
System.out.println("After: " + v);
```

Output:

```
Before: [hello, apple, sniffle, bleep]
After: [hello, sniffle]
```

Two nice features:

- removeLess() is very general: it can handle *any* Iterator, not just one from a Vector.
- We can write removeLess() without knowing anything about the particular Iterator object that will be passed in.

Question: iterator() returns a new Iterator, but since Iterator is an interface it can't be instantiated. What must iterator() be doing?

When to implement an interface

If you want to subscribe to a service that requires an interface, that interface must be implemented.

Eg 1: Sort

- To call sort, Comparable had to be implemented for the items in the array.
- We wanted to sort Strings, which already are Comparable.
- But if we wanted to sort, say, Fractions, we'd have to make class Fraction implement Comparable.

Here's how to do it:

```
public class Fraction implements Comparable {
    public int numerator, denominator;

    // Fraction methods go here -- multiply(),
    // divide(), add(), and so on.

    // We're obliged to implement this:
    public int compareTo(Object o) {
        Fraction f = (Fraction)o;
        // Multiply the relation
        //   n1/d1  >, =, or <  n2/d2
        // by the positive number d1 * d2 * sgn(d1 * d2)
        // (thus preserving the relation)
        // and subtract, to get a number whose sign
        // reflects the original relation.
        return
            MathHelpers.sgn(denominator * f.denominator)
            * (numerator * f.denominator -
               f.numerator * denominator);
    }
}

class MathHelpers {
    /** Return the signum of l: -1, 0 or 1
      * as l is negative, zero or positive. */
    public static int sgn(long l) {
        return (int)(Math.abs(l) / l);
    }
}
```


Eg 2: removeLess

- To call `removeLess`, `Iterator` had to be implemented for the first argument.
- We wanted to remove smaller things from a `Vector`. `Vector`s aren't themselves "iterable" — the class `Vector` doesn't implement `Iterator`. But it has a method that will give you an `Iterator` over its elements.

When to define your own interface

When you want to provide a service that will need to assume something, you should express that with interfaces: make the type(s) of the relevant parameter(s) be the appropriate interface(s).

If the Java API doesn't have an appropriate interface, you should define your own.

Always impose the weakest possible constraints on your subscribers. What does this mean you should do?

Question: Why is it important to try to use an existing `interface`?

Exercise: In Java, `extend`ing lets you *build* a class by adding to another. Implementing an interface lets you *use* a class in certain ways. You can only `extend` from one class, but `implement` many interfaces. If you could change the language to reverse this (build from many classes, conform to at most one interface), would you?

Partially Implementing an Interface or Class

Consider our stack datatype, with: push, pop, isFull and isEmpty.

Suppose users often ask for the head (without removing it), and we don't want them to have to push and pop to get it. It would be nice if we could write something like:

```
... Stack {
    void push(Object o);
    Object pop();
    Object head() {
        Object result = pop();
        push(result);
        return result;
    }
    boolean isFull();
    boolean isEmpty();
}
```

and if stack implementers could `extend` / `implement` this `Stack`.

Java allows such things, in an "abstract class".

Abstract Classes

An abstract class is halfway between a class and an interface.

It's declared like a class, but with the keyword `abstract`:

```
public abstract class Stack { ... }
```

But like an interface, it can have unimplemented `public` instance methods. These methods are also marked with the keyword `abstract`:

```
public abstract void push(Object o);
```

Instantiating

- You can't instantiate an abstract class.
- If a descendant does not fill in all the missing method bodies, it must be declared `abstract` too.
- If a descendant fills in all method bodies, it does not need to be abstract.

A Tradeoff

We have given a simple rule: Always use an `interface` to represent an ADT.

But another option is to use an `abstract class`. It's not obvious which choice is better.

Reason to prefer an `interface`:

- A class implementing the interface can still extend something else.

Reason to prefer an `abstract class`:

- Can provide code for an operation defined in terms of other ones. A subclass can still override this code if it wants.

Revised rule: Use an `interface` unless you want to provide default method implementation(s).

Question: Why might a subclass override `head()`?