ADTs versus APIs

We have seen many abstract data types (ADTs): lists, stacks, queues, trees, binary search trees. Each ADT consists of a set of operations that one performs on the data. These operations are defined independently of the implementation in some programming language. It is sometimes useful to keep the implementation details hidden, since they may just be distracting and irrelevant.

Although ADT's are meant to be independent of any particular programming language, in fact they are similar to concrete quantities in programming, namely interfaces that are given to a programmer. In Java, for example, there is the *Java API*, which you are familiar with. The API (*application program interface*) gives a user many predefined classes with implemented methods. What makes this an "interface" is that the implementation is hidden. You are only given the names of classes and methods (and possibly fields) and comments on what these methods do.

The word "interface" within "Java API" should not be confused with related but different usage of the word, namely the Java reserved word `interface`, which is what this lecture is about.

Java `interface`

A typical first step in designing an object oriented program is to define the classes and the method signatures within each class and to specify (as comments) what operations the methods should perform. Eventually, you implement the methods.

A user (client) of the class should not need to see the implementation of a method to be able to use it, however. The user only sees the API. If the design is good, then all the client needs is a description of what the method does, and the method signatures: the return type, method name, and parameters with their types. (This hiding of the implementation is called *encapsulation*.)

In Java, if we write *only* the signatures of a set of methods in some class, then technically we don't have a class. What we have instead is an `interface`. So, an `interface` is a Java program component that declares the method signatures but does not provide the method bodies.

We say that a class `implements` an interface if the class implements each method that is defined in the interface. In particular, the method signatures must be the same as in the interface. So, if we say a class `C` implements an interface `I`, then `C` must implement all the methods from interface `I`, which means that `C` specifies the body of these methods. In addition, the class `C` can have other methods.

e.g. Java `List` interface

Consider the two classes `ArrayList<T>` and `LinkedList<T>` which are used to implement lists. As such, these two classes share many method signatures. Of course, the underlying implementations of the methods are very different, but the result of the methods are the same, in the sense of maintaining a list. For example, if you have a list and then you remove the 3rd item, you get a well defined result. This new list should not depend on whether the original list was implemented with a linked list or with an array.

The `List<T>` interface includes familiar method signatures such as:

- `void add(T o)`
- `void add(int index, T element)`

- `boolean isEmpty()`
- `T get(int index)`
- `T remove(int index)`
- `int size()`

Both `ArrayList<T>` and `LinkedList<T>` implement this interface, namely they implement all the methods in this interface.

Why is the `List` interface useful? Sometimes you may wish to write a program that uses either an `ArrayList<T>` or a `LinkedList<T>` but you may not care which. You want to be flexible, so that the code will work regardless of which type is used. In this case, you can use the generic Java interface `List<T>`. For example,

```
void myMethod( List<String> list ){
       :
   list.add("hello");
       :
}
```

Java allows you to do this. The compiler will see the `List` type in the parameter, and it will infer that the argument passed to this method will be an object of some class that implements the `List` interface. As long as `list` only invokes methods from the `List` interface, the compiler will not complain. I will say more about this a few lectures from now.

See the lecture slides for a similar example. I defined a `Shape` interface which has two methods: `getArea()` and `getPerimeter`, where the latter is the length of the boundary of the shape. I then defined two classes `Rectangle` and `Circle` that implement the `Shape` interface, namely they provide method bodies. A similar example comes up in Assignment 4 (although there `Shape` is an abstract class rather than an interface. We will discuss abstract classes in an upcoming lecture).

Let's now turn to a few other commonly used Java interfaces.

Java `Comparable` interface

Recall that to define a binary search tree, the elements that we are considering must be comparable to each other, meaning that there must be a well-defined ordering. For strings and numbers, there is a natural ordering, but for more general classes, you need to define the ordering yourself. How?

Java has an interface called `Comparable<T>` which has one method `compareTo(T)` that allows an object of type `T` to compare itself to another object of type `T`. By definition, any class that implements the interface `Comparable<T>` must have a method `compareTo( T )`.

The `compareTo()` method returns an integer and it is defined as follows. Suppose we have variables

```
  T  a1, a2;
```

Then the Java API specifies that `a1.compareTo(a2)` must return:

- a negative integer, if the object referenced by `a1` is "less than" the object referenced by `a2`,
- 0, if `a1.equals(a2)` returns true
- a positive integer, if the object referenced by `a1` is "greater than" the object referenced by `a2`.

By "less than" or "greater than" here, I just mean the desired ordering – what you want if the elements are to be sorted or put in a binary search tree or priority queue, etc.

I will discuss the `equals()` method in more detail next lecture. For now, I will just note that the default meaning is that `a1.equals(a2)` is true if and only if `a1` and `a2` reference the same object, that is `a1 == a2`. However, a class can override this default method.

Example: `Rectangle`

The usage of the `Comparable` interface sometimes can be a bit subtle. To get a flavour for this, let's define a Java class `Rectangle` that implements the `Comparable` interface. (Earlier we let `Rectangle` implement a `Shape` interface, but here we ignore this aspect.) Let `Rectangle` have two fields `height` and `width`, along with getters and setters and two other methods `getArea()` and `getPerimeter()`.

How should we implement the `compareTo()` method for the class `Rectangle`? We could compare rectangles by the values of their widths, or their heights, their areas or perimeters, etc. In the code below, we arbitrarily choose to compare by area.

```
public class Rectangle implements Comparable<Rectangle>{  // ignore Shape
   private width;
   private height;

   public Rectangle(double width, double height){    // constructor
      this.width = width;
      this.height = height;
   }

   public double getArea(){
       return width * height;
   }

   public int  compareTo(Rectangle r) {
      if (getArea() > r.getArea() )
         return 1;
      else if (getArea() < r.getArea() )
         return -1;
      else return 0;
   }

   public double getPerimeter(){
       return  2*(width + height);
   }
 }
```

Here is a simple example, which I didn't show in the slides:

```
public class Test{
  public static void main(String() args){
    Rectangle  r1 = Rectangle(4.0,5.0);
    Rectangle  r2 = Rectangle(2.0,9.0);
    System.out.println("result: " + r1.compareTo(r2))

    // would print "result: 1"  since 20.0 > 18.0

    Rectangle  r1 = Rectangle(4.0,5.0);
    Rectangle  r2 = Rectangle(2.0,10.0);
    System.out.println("result: " + r1.compareTo(r2))

     // would print "result: 0"  since 20.0 = 20.0
  }
}
```

`compareTo()` and `equals()`

The Java API for `Comparable` recommends that `r1.compareTo(r2)` is 0 if and only if `r1.equals(r2)` returns true. This imples that the `Rectangle` class also should implement the `equals` method, and it should do so as follows:

```
public  boolean  equals(Rectangle r){
   return (this.getArea() == r.getArea());
}
```

This definition of `equals()` may seem strange, since one naturally might prefer to define the `equals` method as follows:

```
public  boolean  equals(Rectangle r){
    return (this.height == r.height) &  (this.width == r.width);
}
```

However, by doing so, one potentially could get you into trouble since the `compareTo()` and `equals()` method would not be consistent. The problem is that clients (users) of the class might assume that these methods are equivalent for checking equality of two objects, and might use the two methods interchangably for that purpose.

`Iterator` interface

We have seen many data structures for representing collections of objects including lists, trees, hashtables, graphs. Often we would like to visit all the objects in a collection. We have seen algorithms for doing this, and implementaions for some data structures such as linked lists.

Because iterating through a collection is so common, Java defines an interface `Iterator<E>` that makes this a bit easier to do.

```
interface Iterator<E>{
  boolean   hasNext( E )
  E         next();    //  returns the next element and advances
}
```

In fact there are a few different motivations for using iterators. One is that you sometimes are writing methods that do certain things to all of the objects of a collections, but the details of how the collection is organized are just not relevant. For example, you might want to set the value of a particular boolean field of element object to be true. To do so, you don't want to have to write out the details for how to iterate through.

A related motivation is that for some classes, like the Java LinkedList class, if you iterate through by using the `get(i)` method for all `i`, then this is very inefficient namely it would be $O(n^2)$ instead of $O(n)$. You can use an enhanced for loop for this, although as I'll mention again below, the enhanced for loop in fact is implemented "under the hood" with an `Iterator` object.

Another motivation is that you might want to use several different iterators at the same time. This would be awkward to do with a loop, whether it is an enhanced for loop or some other loop. As an analogy, consider a collection of quizzes that need to be marked. Each quiz is an object. Suppose each quiz consisted of four questions and suppose there were four T.A.'s marking the quizzes, namely each T.A. marks one question. Think of each T.A. as an iterator that steps through the quizzes. Different TAs might be grading different quizzes at any time.

Consider how the `Iterator` interface might be implemented for a singly linked list class. There would need to be a private field (say `cur`) which would be initialized to reference the same node as `head`. The constructor of the iterator would do this. The `hasNext()` method checks if `cur == null`. The `next()` method advances to the next element in the list. Naturally, you would only call `next()` if `iterator.hasNext()` returns `true`.

`Iterable` interface

A collection (e.g. a `LinkedList` object) does not implement the `Iterator` interface. Rather, the collection construct objects that implement the `Iterator` interface. How is this done? Iterator objects are constructed by a method `iterator()` which is a method in a interface called `Iterable`.

```
interface Iterable<T>{
    Iterator<T>  iterator();
}
```

This means that the class that defines the collection has a method called `iterator()` that returns an object whose class type implements the `Iterator` interface. The idea here is that if you have a collection such that it makes sense to step through all the objects in the collection, then you can define an iterator object to do this stepping (if you want) and, because you can do this, you would say that the collection is "iterable".

Think of the `iterator()` method as a "constructor" for `Iterator` objects. Note that `Iterator` is not a class it is an interface. Different `Iterable` classes require quite different iterators, since the underlying data structures are so different.

Example: list of rectangles

Let's look at an example of how iterator works. We make a linked list of rectangles.

```
   LinkedList<Rectangle>  list = new LinkedList<Rectangle>();
   Iterator<Rectangle>    iter = list.iterator();
```

Suppose we build up a list of rectangles. Then we want to visit all the rectangles in the list and do something with them. We don't have access to the underlying data structure of the linked list (nor do we want to have access – its messy!) Instead the usual way to do it is:

```
for (int i=0;  i < list.size();  i++){
   list.get(i)   //  do something with i-th rectangle
    :
}
```

Using an iterator instead, we would write:

```
   while (iter.hasNext() ){
      r = iter.next();
      //   do something with rectangle pointed to by r
   }
```

This avoids the problem with `get(i)` method that it always starts at the beginning of the list.

See the slides for a simple example of a linked list with multiple iterators. (Recall the exam grader analogy from earlier in the lecture.)

Java "enhanced for loop"

Java allows you to replace the `while` loop above by an *enhanced for loop*, which does the same thing.

```
   for (Rectangle r: myRectangleList){
       //  do something with rectangle r
   }
```

The enhanced for loop will work for any class that implements `Iterable` interface. Indeed, this enhanced for loop just gets compiled into equivalent statements that use iterators.