

## 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 data and operations that one performs on the data. These operations are defined independently of the implementation in some programming language. It is useful to keep the implementation details “hidden”, so as not to confuse *what* is computed from *how* it is computed, namely we can write and analyze algorithms in pseudocode without worrying about the nitty gritty implementation details which might change from one language to the next.

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 (public) classes and methods (and possibly fields).

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 or you hire someone to implement them.

A user (client) of the class should not need to see the implementation of a method to be able to use it. If the design is good, then all the client needs is a description of what the method does, and the method signature (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 has each method that is defined in the interface, in particular, the signatures of the method must be the same. That is, if a class `C` implements an interface `I`, then `C` must implement all the methods from interface `I`, meaning that `C` must specify the body of these methods. (In addition, `C` can have other methods.)

## 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 new list. 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 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 uses methods from the `List` interface, the compiler will not complain. *We will say more about this at the end of the course when I discuss polymorphism.*

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.

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

## Comparable interface

Recall that to define a binary search trees, 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 and you can use the "<" operator. However, 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)` is true<sup>1</sup>
- a positive integer, if the object referenced by `a1` is “greater than” the object referenced by `a2`.

During the lecture, there were a few questions about what exactly the `equals()` method does. The answer is: it depends. The default meaning is that `a1.equals(a2)` is true if and only if `a1` and `a2` reference the same object. However, a class can override this default method. We will revisit this issue (and specify what “default” means) at the end of the course, when we discuss the topic of inheritance.

### Example: Rectangle

Let’s define a Java class that implements the `Comparable` interface, namely `Rectangle`. Let `Rectangle` have two fields `height` and `width`, along with getters and setters and two other methods `getArea()` and `getPerimeter()`.

Note that there is only a single `compareTo()` method with the above signature in a class, and so we have to decide how to implement this method in `Rectangle`. We could compare rectangles by the values of their widths, or their heights, their areas or perimeters, etc. In the code below, we compare by area.

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

    public Rectangle(double width, double height){
        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;
    }
}
```

---

<sup>1</sup> Have a look at <http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Comparable.html>. You will see that there is some flexibility about this condition on the `equals` method.

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

A few points to note. First, the `Rectangle` class also includes a method `getPerimeter()`. We could have used this method to define the `compareTo()` function, instead of using the `getArea()` method. Or we could have used yet a different method, for example, which compares two rectangles based on the longer of the two sides.

Second, as was observed by one student during the lecture, this code ignores the requirement that `r1.compareTo(r2)` is 0 if and only if `r1.equals(r2)`. In particular, there is no `equals` method in this class and so the default `equals` method will be used here (see discussion above). This could potentially produce weird behavior i.e. the `compareTo` and `equals` method could disagree on whether two objects are “equal”. Again, we will return to the issue later when we discuss inheritance.

Here is a simple example:

```
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
    }
}
```

## Iterator interface

We have seen many data structures for representing collections of objects (including lists, trees) and we will see more (graphs, hashtables, etc). Often it happens that we would like to visit all the objects in a collection. We have seen how to do this for lists and trees (traversal).

Because stepping 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
}
```

An iterator is an object which is distinct from the collection that it is iterating over. Moreover, you might have several different iterators defined for a given collection. (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.)

Consider how the iterator might be implemented for a singly linked list class. There would need to be a private field `cur` which would be initialized to point to the first element in the list, namely `head`. The constructor of the iterator would set this field. The `hasNext()` method would then check if `cur == null`. The `next()` method would advance to the next element in the list if `iterator.hasNext()` returns `true`. I have implemented an iterator for the `SLinkedList` class – see online code for lecture 4. Note that `Iterator` is an interface, not a class, so I had to implement a class `SLLIterator` which implements the `Iterator` interface.

I also implemented an iterator for binary search trees. See code online for lecture 21.

## Iterable interface

Objects in a collection do not themselves implement the `Iterator` interface. Rather, they construct iterator objects that implement the `Iterator` interface. For example, `ArrayList` and `LinkedList` do not implement the `Iterator` interface. Rather, they construct iterator objects. How is this done?

Iterator objects are constructed by a method `iterator()` which is a method in a interface called `Iterable`. 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”. So,

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

This means that the class that defines the collection has a method called `iterator` that returns an object (an iterator) whose class implements the `Iterator` interface. [Aside: it takes a while to get used to these definition.]

### 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
}
```

Admittedly, for this LinkedList example, the latter seems no better than the former. To see the advantage of iterators, you need to go beyond this example. In the slides, I give a simple example of a linked list with multiple iterators. (Recall the exam grader analogy from earlier in the lecture.)

For other collections, such as trees, using an iterator would allow you to avoid having to write traversal code over and over again. You would just write the traversal code once (in the class that implements Iterator). Then you would use the `next()` method to step through the collection.

## 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
}
```

Similarly, suppose you have a binary search tree holding objects of type `Integer`. (`Integer` is different from class `int`. See the Java API.)

```
BST<Integer>    tree;

for ( Integer i : tree ){
    // do something with element i in the tree
}
```

For this to work, you need to have an `iterator()` method in the `BST` class. (See code provided with lecture 21.) But once you define this method, the enhanced for loop makes it is very easy to traverse the tree and visit the elements in order.