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 any 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 sometimes want to write and analyze algorithms in pseudocode without having to commit ourselves to any particular programming language.

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 signatures1 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 what is the method signature. 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. 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. These two classes share many list-like methods which have the same signatures. Of course, the underlying implementations of the methods are very different since arrays and linked lists 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 new list which doesn’t depend on whether the original list was implemented with an array or with a linked list.

---

1 By “signature”, we mean the return type, method name, and parameters with their types.
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 want your code to be general enough to allow either to be used. In this case, you can use the generic Java interface List<T>. For example,

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

The Java compiler will see the declared List<String> 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, with generic type String. As long as list only uses methods from the List interface, the compiler will not complain. We will say more about this in the next few lectures.

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 tree, the elements that we are collecting in the tree 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. Any class that implements the interface Comparable<T> must, by definition, have a method compareTo( T ).

The compareTo() method returns an integer and it is defined as follows. Suppose we have variables
Then the Java API recommends that `t1.compareTo(t2)` return:

- a negative integer, if the object referenced by `t1` is “less than” the object referenced by `t2` in the ordering,
- 0, if `t1.equals(t2)` is true (more on this below)
- a positive integer, if the object referenced by `t1` is “greater than” the object referenced by `t2`.

Notice that this definition depends on the `equals()` method. The default meaning of the `equals()` method is that `t1.equals(t2)` is true if and only if `t1` and `t2` reference the same object. But a class can and often does override this default method. We will revisit this issue (and specify what “default” means) in the next few lectures.

**Example: Rectangle**

Let’s define a Java class `Rectangle` that implements the `Comparable` interface. `Rectangle` has two fields `height` and `width`, two methods `getArea()` and `getPerimeter()`, and a `compareTo()` method. How should the `compareTo()` be defined for `Rectangle`? We could compare two rectangles by their areas, perimeters, widths, or heights, etc. In the code below, we compare by area.

```java
public class Rectangle implements Comparable<Rectangle>{
    private double width;
    private double 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;  // Note this will return 0 for any two
                        // two Rectangles with the same area.
    }

    public double getPerimeter(){
        return 2*(width + height);
    }
}
```
Does this definition of the `compareTo()` method satisfy the Java recommendations? Not really. The Java API recommends\(^2\) that `r1.compareTo(r2)` is 0 if and only if `r1.equals(r2)` returns `true`. This recommendation is ignored in the above code. In particular, there is no `equals` method in the `Rectangle` class and so the default `equals` method will be used here – namely two Rectangles are considered equal if and only if they are the same object (as opposed to them having the same area, or the same width and height, etc). This could potentially produce weird behavior i.e. the `compareTo` and `equals` method could disagree on whether two objects are “equal”. Here is a simple example:

```java
public class Test{
    public static void main(String[] args){
        Rectangle r1 = new Rectangle(4.0,5.0);
        Rectangle r2 = new Rectangle(2.0,10.0);
        System.out.println("result: "+ r1.compareTo(r2))

        // Would print "result: 0" since 20.0 == 20.0
        System.out.println("equals result: "+ r1.equals(r2))

        // Would print "result: false" since r1 references a different object.
    }
}
```

[Added Dec. 12, 2012] Following the Java recommendation, one should re-define the `equals()` method so that it is consistent with the `compareTo()` method.

```java
    public boolean equals(Object r){
        if (this.compareTo( (Rectangle) r) == 0 )
            return true;
        else
            return false
    }
```

Here I am overriding the Object class’es `equals( Object)` method. Now when I do this, the above test code prints ‘‘result: true’’.

**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, though, Java defines an interface `Iterator<E>` that makes this a bit easier to do.

[^2]: [http://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html](http://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html)
interface Iterator<E>
{
    boolean hasNext();
    E next(); // Returns the next element and advances.
    void remove(); // I did not mention this one in the slides.
}

Note that an Iterator object is distinct from the collection that it is iterating over. Moreover, you might have several different Iterators defined for a given collection.

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

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

**Iterable interface**

Collections such \texttt{ArrayList} and \texttt{LinkedList} do not themselves implement the \texttt{Iterator} interface, since they are not Iterators. Rather, they construct Iterator objects that implement the \texttt{Iterator} interface. How is this done? Iterator objects are constructed by a method \texttt{iterator()} which is the one method in a interface called \texttt{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). Because the collection has this property, you would say that the collection is “iterable”. So, Java has an interface:

\begin{verbatim}
interface Iterable<T>
{
    Iterator<T> iterator();
}
\end{verbatim}

Any class (such as \texttt{LinkedList} or \texttt{BST}) for which it makes sense to have Iterator objects should also have a method called \texttt{iterator} that returns an object (an Iterator) whose class implements the \texttt{Iterator} interface.

[ASIDE: For many people, the above interfaces are very confusing. But let me assure you that eventually they do make sense and will seem natural.]

**Example: list of rectangles**

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

\begin{verbatim}
LinkedList<Rectangle> list = new LinkedList<Rectangle>();
Iterator<Rectangle> iter = list.iterator();
\end{verbatim}

\footnote{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.}
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). Instead the usual way to do it is:

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

Using an iterator instead, we could write:

```java
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 your program (client) which uses this class 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.

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

Similarly, suppose you have a binary search tree holding objects of type `Integer`.

```java
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 very easy to traverse the tree and visit the elements in order.