lecture 22

Java break : interfaces

- List
- Comparable
- Iterator / Iterable
- enhanced for loop

ADTs

- list
- stack
- queue
- tree
- binary search tree
- .....

Useful because they allow us to ignore (arbitrary) implementation level details.

"Java API"

(application program interface)

- You are not given "the" source code for Java classes such as LinkedList, Math, etc,
- You are only given the class names and their method signatures and a description of what they do.

Java "interface"

- reserved word in Java language
- like a class definition BUT
  - only the method signature is given
  - fields are not given (except for some constants eg. π)

```
interface  List<T> {
    void     add (T)
    void     add (int, T)
    T        remove (int)
    boolean  isEmpty()
    T        get (int)
    int      size()
        :
}
```

```
class  ArrayList<T>
        implements  List {

        void  add(T t) {
              :
        }

    // each of the methods of
       List must be implemented
}
```

```
class LinkedList<T>
        implements List {
    void add(T t) {
        :
    }

    // each of the methods of
       List must be implemented
}
```

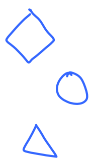
Why is this useful?

Sometimes you don't care whether you have a LinkedList vs. ArrayList.

Example

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

Another Example



```
interface Shape {
    float getArea();
    float getPerimeter();
}

class Rectangle implements Shape {
    :
}

class Circle implements Shape {
    :
}
```

```
class Rectangle implements Shape {
    float height, width
    Rectangle(float h, float w) {
        height = h; weight = w;
    }
    float getArea() {
        return height * width;
    }
    float getPerimeter() {
        return 2 * (height + width);
    }
}
```

```
class Circle implements Shape {
    float radius;
    Circle(float rad) {
        radius = rad;
    }
    float getArea() {
        return Math.PI * radius *
                          radius;
    }
    float getPerimeter() {
        return 2 * Math.PI * radius;
    }
}
```

```
Shape s = new Rectangle(3.0, 4.0)
          :
      s = new Circle(2.51));
          :
```

```
interface Comparable <T> {
    int compareTo ( T t )
}
```

a1. compareTo (a2)

returns { 0, a1.equals (a2) ; +, a1 > a2 ; −, a1 < a2 }

```
class Rectangle implements Shape, Comparable {
    float height, width
    Rectangle (float h, float w ){
        height = h; weight = w;
    }
    float getArea() {
        return height * width;
    }
    float getPerimeter() {
        return 2* (height + width);
    }
```

```
int compareTo(Rectangle r) {
    float diff = this.getArea()
                 - r.getArea();

    if (diff > 0)
        return 1;
    else if (diff == 0.0)
        return 0;
    else return -1;
  }
}
```

We could have compared Rectangles instead by:

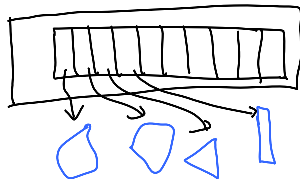
- perimeter
- length of longest side
- etc

Each of these would require a different implementation of compareTo().

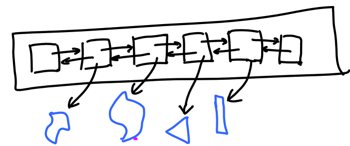
Java interface : Iterator

Motivation: often we want to visit all objects in a collection

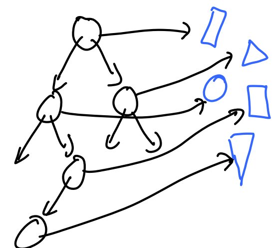
```
ArrayList<Shape> list
for (i=0; i< list.size(); i++ ) {
    list.get(i);
}      :
```



```
LinkedList<Shape> list
for (i=0; i< list.size(); i++) {
    list.get(i);
}      :
```



Similarly, we have seen tree traversal algorithms.



```
interface Iterator<T> {
    boolean hasNext()
    T next()
}
```

See example code

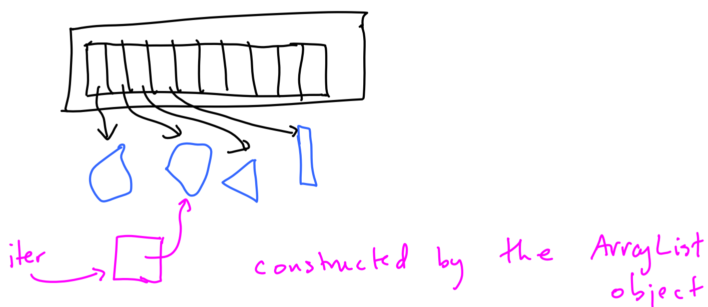
- lecture 4: singly linked lists
- lecture 21: binary search trees

- What do Iterator objects do?
- Where do Iterator objects come from? Which object constructs them?

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

Classes that implement Iterable have a method, iterator(), that constructs Iterator objects.

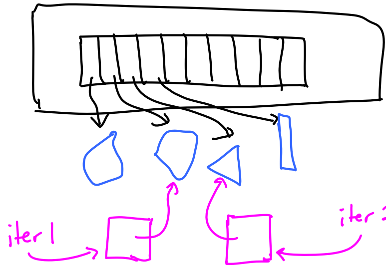
```
ArrayList<T> list = new ArrayList<T>()
Iterator<T> iter = list.iterator();
// for ArrayLists, the implementation is hidden!
```



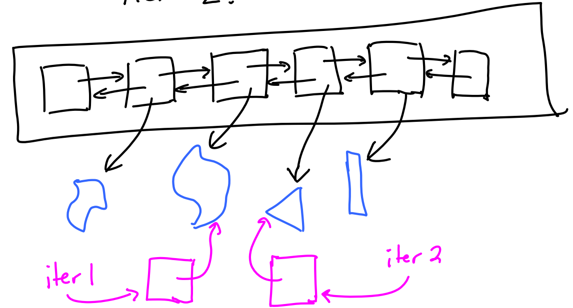
Why are iterators useful?

- you may need several iterators at same time
- a 'for' or 'while' loop might not be appropriate (you cannot break out, and then return back inside)

```
ArrayList<T> list = new ArrayList<T>.
Iterator<T> iter1 = list.iterator( );
         iter1. next()
Iterator<T> iter 2 = list. iterator ()
         iter 2. next()
         iter 2. next()
```




```
LinkedList<T> list = new LinkedList<T>
Iterator<T>   iter1 = list. iterator()
              iter 1. next()
Iterator<T>   iter 2 = list. iterator ()
              iter 2. next()
              iter 2. next()
```



Java "enhanced for loop"

Java language feature that allows you to step through any object whose class implements Iterable.

```
List< String > list;
   :
for ( String s : list) { ... }

// does the same as:

for (int i=0; i< list.size(); i++ ) {
    s = list. get(i);
        :
}
```

```
BST< Integer > tree ;
   :
for (Integer i : tree) {
  // Traverses the tree
     Uses the iterator() method
     of BST class to
     construct an Iterator object

}
```