Abstract Data Type
List

COMP128 Data Structures

The List Interface

`Boolean add(E e)`	Appends the specified element to the end of this list.
`void add(int index, E element)`	Inserts the specified element at the specified position in this list.
`E get(int index)`	Returns the element at the specified position in this list.
`Int indexOf(Object o)`	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`E remove(int index)`	Removes the element at the specified position in this list.
`Boolean remove(Object o)`	Removes the first occurrence of the specified element from this list, if it is present.

Iterating over Lists

- There are several ways to traverse every element of a List class in Java
- Let’s look at some of them

Assume We Have Created a List

```
// make some new student instances
Student sam = new Student("Gilbert", "Sam", 2220390);
Student john = new Student("Smith", "John", 333333);
Student libby = new Student("Shoop", "Libby", 4444444);

List<Student> studentList = new ArrayList<Student>();

studentList.add(sam);
studentList.add(john);
studentList.add(libby);
```

Basic For Loop (least desirable option)

```
// 1. While a familiar approach, this for loop example
// is not typically what we use to access every element
// in a List

for (int i = 0; i < studentList.size(); i++) {
    System.out.println(studentList.get(i));
}
```

For Each Loop (very serviceable option)

```
// 2. This style is preferred for both efficiency and
// elegance of the code itself.
// It is ideal for traversal of all elements  or some
// elements until a given one is found

for (Student stu : studentList) {
    System.out.println(stu);
}
```

For Each Loop (repeat until sentinel pattern)

```
// 2a. Using the colon style and repeat until sentinel
// to find a desired element.

for (Student stu : studentList) {
    if (stu.getID() == 333333) {
        System.out.println(stu);
        break;
    }
}
```

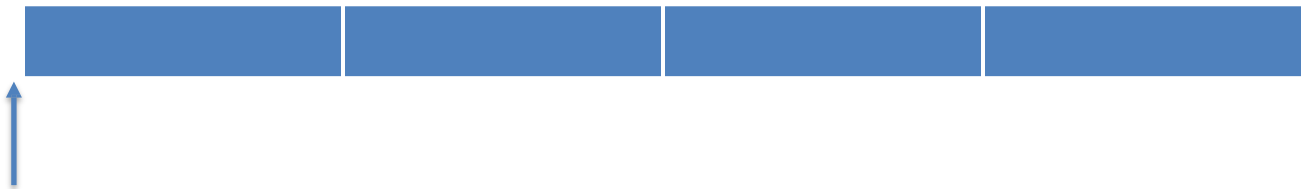
Streams

```
studentList.stream().forEach(System.out::println);
```

While Loop using an Iterator

```
// 3. Using an Iterator.
// Must include java.util.Iterator

Iterator iter = studentList.iterator();
while (iter.hasNext()) {
    Student stu = (Student) iter.next();
    System.out.println(stu);
}
```



Safe to Change the List (using an iterator)

```
// 3a. Using an Iterator.
// This enables us to change the list while iterating over it.

Iterator iter2 = studentList.iterator();
while (iter2.hasNext()) {
    Student stu = (Student) iter2.next();
    if (stu.getID() == 333333) {
        //  note that using remove on Iterator object is a
        //  safe way to remove the element and keep going
        iter2.remove();
    }
}
System.out.println("List after removing one student:");
System.out.println(studentList);
```

In-class Activity

Nim Activity

CircularArrayQueue

```
public class CircularArrayQueue<T> implements QueueADT<T> {
    private final static int DEFAULT_CAPACITY = 100;
    private int front, rear, count;
    private T[] queue;
…

   public void enqueue(T element){
        if (size() == queue.length)
            expandCapacity();

        queue[rear] = element;
        rear = (rear+1) % queue.length;
        count++;
    }
```

CircularArrayQueue

```
public T dequeue() throws EmptyCollectionException{
    if (isEmpty())
        throw new EmptyCollectionException("queue");

    T result = queue[front];
    queue[front] = null;
    front = (front+1) % queue.length;

    count--;

    return result;
}
```