Announcements

1. HW2 posted tomorrow. When it is, DON’T WAIT to read it, though you’ll need Wednesday’s material to really dig in.
Today’s Topics

1. The List ADT
2. Using Generics in Lists
3. Iterators
public interface Collection<E>
extends Iterable<E>

- What does the <E> mean in the above code?
  A. That this collection can only be used with objects of a built-in Java type called E
  B. That the object can be instantiated to work with any object type
  C. That a single collection can hold objects of different types
Java generics: Using parameterized types in class definitions

```java
public class RecentRememberer<T> {
    private T lastElement;
    private int numElements;

    public RecentRememberer() {
        numElements = 0;
        lastElement = null;
    }

    public void add(T element) {
        lastElement = element;
        numElements++;
    }

    ...
}
```
Java generics: Using parameterized types in class definitions

```java
public class RecentRememberer<T> {
    private T lastElement;
    private int numElements;

    public RecentRememberer() {
        numElements = 0;
        lastElement = null;
    }

    public void add(T element) {
        T prevLast = lastElement;
        lastElement = element;
        numElements++;
    }
    ...
```

Is this line legal Java code?
A. Yes
B. No
Java generics: Using parameterized types in class definitions

```java
public class RecentRememberer<T> {
    private T lastElement;
    private int numElements;

    public RecentRememberer() {
        numElements = 0;
        lastElement = null;
    }

    public T add(T element) {
        T prevLast = lastElement;
        lastElement = element;
        numElements++;
        return prevLast;
    }
}
```

T can be used to stand for a type (to be specified later anywhere in this class (and its inner classes!))
Exceptions

```java
public class RecentRememberer<T> {
    ...  
    public void add(T element) {
        if (element == null)
            // << code goes here >>
            lastElement = element;
            numElements++;
    }

    // What code could I add if I want to prevent null elements from being added to this class?
    A. throw new NullPointerException();
    B. return -1;
    C. Either A or B
    D. None of the above
```
Exceptions

public class RecentRememberer<T> {
...
    public void add(T element) {
        if (element == null)
            throw new NullPointerException();
        lastElement = element;
        numElements++;
    }
...

Will this code compile even though it does not declare the NullPointerException (potentially) being thrown?
A. Yes, it will compile (why?)
B. No, it will not compile (why not?)
Exceptions

public class RecentRememberer<T> {
    ...
    public void add(T element) throws NullPointerException {
        if (element == null)
            throw new NullPointerException();
        lastElement = element;
        numElements++;
    }
    ...

    Will this code compile now?
    A. Yes, it will compile (why?)
    B. No, it will not compile (why not?)
Lists and Collections

A list is a collection. The Java List interface extends the Java Collection interface.

interface List<E> extends Collection<E>
Which is legal?

public class Base {
    protected int x;
}

public class Derived extends Base {
    protected int y;
}

A: Base b=new Base(); Derived d=b;
B: Derived d=new Derived(); Base b=d;
C: Base b=new Derived();
D: Derived d=new Base();
E: Other/none/more
Lists and Collections

Collection declares an add method:
Ensures that this collection contains the specified element.

List inherits the add method... and changes its contract:
Appends the specified element to the end of this list
Lists have order

*Note: ls is some object of a type that implements List<MyObjs>*

```
int i=0;
while (i<c.size()) {
    MyObjs x = ls.get(i);
    //do something with x
    i++;
}
```
Iterators

Next!
The Iterator Software Design Pattern

- A common situation: A client needs to inspect the data elements in a collection, without wanting to know details of how the collection structures its data internally

- Solution:
  - Define an interface that specifies how an iterator will behave
  - Design the collection to be able to supply an object that implements that iterator interface
  - A client then can ask the collection for an iterator object, and use that iterator to inspect the collection’s elements, without having to know how the collection is implemented
Iterable\(<E>\) Interface

- The Collection\(<E>\) interface extends the Iterable\(<E>\) interface, which is defined as follows:
  ```java
  public interface Iterable\(<E>\) {
    public Iterator\(<E>\) iterator();
  }
  ```
- So any class that implements Collection\(<E>\) must define an instance method iterator() that returns an Iterator\(<E>\) object for that instance
- And Iterator\(<E>\) is also an interface in the JCF…
The `Iterator<E>` interface is defined as follows:

```java
public interface Iterator<E> {
    public E next();
    public boolean hasNext();
    public void remove();
}
```

So, any object that is-a `Iterator<E>` will have those operations as part of its API.

But what are these methods supposed to do? One example
Traversal using an Iterator

Note: ls is some object of a type that implements List<MyObjs>

```java
Iterator<MyObjs> i =
    ls.iterator();
while (i.hasNext()) {
    MyObjs x = i.next();
    //do something with x
}
```
WHY have iterators? What is the difference between these codes?

Note: `ls` is some object of a type that implements `List<MyObjs>`

**Iterator version**

```java
Iterator<MyObjs> i = ls.iterator();
while (i.hasNext()) {
    MyObjs x = i.next();
    //do something with x
}
```

**Increment version**

```java
int i=0;
while (i<ls.size()) {
    MyObjs x = ls.get(i);
    //do something with x
    i++;
}
```

A. No difference: they are totally equivalent  
B. Difference: Increment will run faster  
C. Difference: Iterator will run faster  
D. No difference: Java just likes having iterators to make students’ lives harder  
E. Other/none/more
WHY have iterators?

- **Linked List**
  - To go to the n\textsuperscript{th} element, you must start at the beginning and go through one at a time until you get to the n\textsuperscript{th} element.
  - A pain to start over vs just having the iterator keep your place like a bookmark.

- **Trees**
  - To go to the “n\textsuperscript{th}” element, you must travel around the tree in some planned way.
  - A pain to start over vs just having the iterator keep your place in the journey.
Next time

- Developing an object that implements the List ADT: Linked Lists