Outline

- Big-O analysis in practice
- List interface
- Array lists

Big-O Analysis in Practice

```
n = 100000;  // one hundred thousand
startTimer();
for (int i = 0; i < n; i++) {
    count++;
}
stopTimerAndPrint("example 1", n);
```

- how much longer does the second loop take than the first?

```
n = 10000000;  // ten million
startTimer();
for (int i = 0; i < n; i++) {
    count++;
}
stopTimerAndPrint("example 1", n);
```

- how much longer does the third loop take than the first?

```
n = 100000000; // one hundred million
startTimer();
for (int i = 0; i < n; i++) {
    count++;
}
stopTimerAndPrint("example 1", n);
```

Big-O Analysis in Practice

```
n = 1000;
startTimer();
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      count++;
    }
}
stopTimerAndPrint("example 2", n);
n = 10000;
startTimer();
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      count++;
    }
}
stopTimerAndPrint("example 2", n);
```

- how much longer does the second loop take than the first?

Lists

- A List is similar to an array, but may:
  - grow or shrink in size
  - insert or delete elements at a given position
- there are many lists, usually categorized by how they are implemented:
  - ArrayLists are implemented using arrays (Vectors are similar)
  - LinkedLists are implemented using links (references) to objects
- All lists are derived from the abstract class AbstractList, and implement the List interface (even AbstractList implements the List interface)
- lists also have operations to search for elements, and do something with every element of the list

Generic Interfaces

- a list can store object of any one type:
  - a list of strings: `List<String>`
  - a list of integers: `List<Integer>`
  - a list of objects: `List<Object>`
- the notation "`List<E>`" indicates that the list interface is parametrized over the type E of objects it can store
- in this case, E is a type parameter -- logically, it provides a collection of interfaces, one for each possible class

Generic Classes

- generic classes can use the same notation as generic interfaces
- collection classes, which can store objects of any type, are often generic
- the Java compiler can check that the type parameter is the same for every use of a variable: for example, that all operations involving a List<String> actually store and retrieve strings

`List` Interface

```
public interface list<E> extends Collection<E> {
    E get(int index);      // returns object at pos.
    E set(int index, E element); // returns old value
    int indexOf(Object o);// returns index of object
    E remove(int index);   // removes object at pos.
    int size();            // returns # of elements
    boolean add(E element); // add at the end of list
    void add(int index, E element); // add at pos.
    ...
}
```

AbstractList

- AbstractList<E> essentially provides the same methods as List<E>
- the methods implemented by AbstractList provide very basic functionality for lists

ArrayList<E> class

- an array list uses an array to store the objects in the list
- the object at position *i* in the list are found at array index *i*
- when the array needs to grow, a bigger array is allocated, and data is copied from the old array to the new array
- this is an expensive operation: it takes time proportional to the **total size** of the collection
- in general, the underlying array may have more elements than the collection
- for example, the array may have 39 elements, but the collection may only have 22 elements
- an array list always has a `capacity` (39) greater than or equal to its `size` (22)

ArrayList<E> implementation

- must have an actual array of objects of type E
- must keep track of the size
- the capacity is the same as the array length

```
public class ArrayList<E> {
    protected E [] data;
    protected int size;
```

ArrayList<E> constructor

```
@SuppressWarnings("unchecked")
public ArrayList() {
    data = (E []) new Object [16];
}
```

- Java will not allocate an array with a type that is not know at compile time
- @SuppressWarnings("unchecked") is used to suppress warnings about the type conversion not being checked

Alternative implementation of ArrayList<E>

```
protected Object[] data;
. . .
@SuppressWarnings("unchecked")
public E get(int index) {
    if (index < size) {
        return (E)data[index];
    }  // else throw exception
}
```

- all the objects added to the array are necessarily of type E, because that is the only type that can be a parameter to `add`
- the compiler doesn't know this, so we suppress the warning
- the cast is safe, even though the compiler doesn't know that
- a useful property of a program that the programmer knows and that is always true, is an **invariant**
  - in this case, the invariant is that the elements of `data` from 0..`size`-1 are all of type E
  - the invariant must be true whenever a public method is called and when a public method returns, but may not be true in the middle of a method body – for example, in `add`, we may change `size` before we assign the new value

ArrayList<E> adding at the end simple case

- if there is room, adding at the end is easy:

```
public boolean add(E value) {
    data [size] = value;
    size++;
    return true;
}
```

ArrayList<E> adding at the end making room

- we may need to make room by reallocating the array:

```
public boolean add(E value) {
  if (size == data.length) {
    data = Arrays.copyOf(data, data.length * 2);
  }
  data [size] = value;
  size++;
  return true;
}
```

ArrayList<E> adding in the middle

- In-class exercise: (groups of 2 or 3), implement this method (below) to add a value somewhere in the middle of the array. Assume that the index is valid and that there is room in the array, i.e. index >= 0 and index < data.length - 1.
- Your code must shift all the data that is at or after index, so there is room for one new element
- only use a for loop, not methods from other classes

```
public void add(int index, E value) {
```

In-class exercise on big O

- what is the big O for Loops.java?
- what is the big O for the Array list methods:
  - add (add at the end and add in the middle)
  - remove (remove from the end and remove from the middle)