Outline

- List interface
- Array lists

Lists

- **A List is similar to an array, but may:**
  - grow or shrink in size
  - insert or delete elements at a given position
  - in an array, can only **set** the value at a given position, inserting or deleting requires a shift
- **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, pointers) to objects
- **All lists extend the abstract class AbstractList, and implement the List interface**
  - AbstractList implements the List interface, so all its subclasses do too
- **lists also have operations to search for elements, and to do something with every element of the list**

Generic Interfaces

- **a list can store objects of any one object type:**
  - a list of strings: `List<String>`
  - a list of integers: `List<Integer>`
    - this is the `Integer` object, and not the `int` basic type
  - a list of objects: `List<Object>`
- **the notation "`interface 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, we have a collection of interfaces, one for each possible class
  - E can only be instantiated to an object type (not a basic type)

Generic Classes

- **Java provides generic classes and generic interfaces**
- **generic classes and generic interfaces use the same notation**

  `public class ThisClass<E> { ...`

  - 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 position
  E set(int index, E element); // returns old value
  int indexOf(Object o);// returns index of object
  int size(); // returns # of elements
  boolean add(E element); // add at the end of list
  void add(int index, E element); // add at position
  E remove(int index); // removes object at position
  ...
}
```

AbstractList

- **AbstractList<E> is a generic abstract class**
- **AbstractList<E> provides all the methods specified in the List<E> interface**
  - the subclass must provide the methods add, get, remove, set, size
- **the method implementations in AbstractList<E> are functional but may not be optimal**

ArrayList<E>

- **an <u>array list</u> uses an array to store the objects in the list**
- **the object at position i in the list is stored at array index i**
- **if the array needs to grow, a bigger array is allocated, and data is copied from the old array to the new array**
- **reallocating is an expensive operation: it takes time proportional to the *total size* of the collection, O(n) where n is the size of the list**
- **in general, the underlying array may have more elements than the collection**
  - for example, a collection with 22 elements may be stored in an array with 39 elements
  - an array list always has a <u>`capacity`</u> (39 in this example) >= to its <u>`size`</u> (22)

Implementation of ArrayList<E>

- **an object of type ArrayList<E> has two instance variables:**
  - one that is an array of objects of type E
  - another that keeps track of the size
- **the capacity is simply the array length**

```
public class ArrayList<E> {
  protected E [] data;
  protected int size;
  public int capacity() {
    return data.length;
  }
```

ArrayList<E> constructor

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

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

Alternative implementation of ArrayList<E>

- the last example used an array of elements of type `E`
- we could instead use an array of elements of type `Object`
- and only cast the value when `get` is called:

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

Type safety of this alternative implementation of ArrayList<E>

- **if the object in the array does not have type `E`, the cast in the code of `get` will fail**
- **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` or `set`**
- **the compiler doesn't know that the objects are of type E, so we suppress the warning**
  - the cast is safe, even though the compiler doesn't know that it is safe
- **a useful property of a program that the programmer knows and that is always true, is an <u>invariant</u>**

Invariants

- **a useful property of a program that the programmer knows and that is always true, is an *invariant***
  - here the invariant is that: non-null elements of `data` from 0..size-1 are all of type E
  - the invariant must be true whenever a public method is called and whenever a public method returns, but may not be true in the middle of a method body
  - constructors must establish all invariants
  - for example, the ArrayList class has the invariant that size <= data.length
  - within the ArrayList `add`, we may change size before we assign the new value: the invariant is temporarily broken when size is changed, then restored before the method completes

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;
}
```

- doubling the size of the array means we can add data.length more elements in constant time before needing to resize again – resizing takes O(data.length)

ArrayList<E> adding in the middle

- In-class exercise: (groups of 2-4), 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 the Array list methods:**
  - add
    - add at the end
    - add in the middle or at the front
  - remove:
    - remove from the end
    - remove from the middle or from the front