ArrayList

Class that:
- works like a resizable array
- Has a current capacity: if you add more elements, it’ll allocate more space (including spare capacity)

Methods
- void add(int index, Object element)
- void add(Object element); // adds to the end
- Object get(int index)
- void set(int index, Object element)
- void trimToSize // reduces capacity “shrink to fit”

Methods
- To use, import java.util.ArrayList

Problems with standard ArrayList

Doesn’t know the types of objects in it
- For example, an array of Strings:
  - ArrayList a = new ArrayList();
  - String s = (String) a.get(i);
  - System.out.println(s);
  - Cast is necessary because compiler doesn’t know that the ArrayList contains only Strings.
  - Cast can fail at runtime if somehow a non-String was added to the ArrayList.

Aside: reminder of new for-each loop

Instead of:
- Object array[] …;
  - for (int i = 0; i < array.size(); i++)
    - Object o = array[i];
    - doSomething(o);
  - Can do instead:
  - for (Object o: array)
    - doSomething(o);

Redo example from last slide:
- for (Object o: a)
  - System.out.println((String) o);
Solution: Generic ArrayList

Declaration of generic ArrayList includes the type as well:

- `ArrayList<String> a = new ArrayList<String>();`
- `while (in.hasNextLine())`
  - `a.add(in.nextLine());`
- `for (String s: a)`
  - `System.out.println(s);`

- Compiler won’t allow non-String to be added:
  - `a.add(new Critter());`
- Compiler knows that the list contains only Strings
  - `a.get(i)` will return a String

Is it OK to do:

- `as = ao;`  
  - no, because, then `as.get()` might return non-string
- `ao = as;`  
  - no, because then `ao.add(aCritter)` could add a critter to an ArrayList that’s supposed to contain only Strings

Even though String is a subclass of Object, 
ArrayList<String> is not a subclass of ArrayList<Object>

Generic Classes

What if you have:

- `ArrayList<Object> ao = new ArrayList<Object>();`
  - `ao.add(new Integer(3));`
- `ArrayList<String> as = new ArrayList<String>();`

Is it OK to do:

- `as = ao;`  
  - no, because, then `as.get()` might return non-string
- `ao = as;`  
  - no, because then `ao.add(aCritter)` could add a critter to an ArrayList that’s supposed to contain only Strings

Even though String is a subclass of Object, 
ArrayList<String> is not a subclass of ArrayList<Object>

Wildcards

What if you want to write a method that prints all elements of an ArrayList?

- `void printArrayList(ArrayList<Object> a) {`
  - `for (Object o: a)`
    - `System.out.println(o);`
  - `}`

This can’t be called with:

- `ArrayList<String> as;`
  - `printArrayList(as);`
  - because ArrayList<String> isn’t subclass of ArrayList<Object>

Solution:

- `void printArrayList(ArrayList<?> a) {`
  - `for (Object o: a)`
    - `System.out.println(o);`
  - `}`
  - `?` is unknown type. Can call this routine with any ArrayList<…>
  - Return type of get is unknown, but must be a subclass of Object

Bounded wildcards

What if you wanted to deal with just critters?

- `void oneTimeStep(ArrayList<Critter> a) {
  for (Critter c: a)
    c.oneTimeStep();
  }
- No good, can’t call with:
  - `oneTimeStep(new ArrayList<Doodlebug>);`
  - `ArrayList<?> no good, would allow calling with ArrayList<String>`
  - `? extends Critter`

- Bounded wildcard. `?` is a Critter or some subclass

- `void oneTimeStep(ArrayList<? extends Critter> a) {
  for (Critter c: a)
    c.oneTimeStep();
  }
-`
How to specify multiple extensions

You can specify one class and multiple interfaces
- Separate each one with &
  - ArrayList<? extends Critter & Serializable & Cloneable>

Limitations with wildcards

Can't add into ArrayList<? extends Critter>
- public void doSomething(ArrayList<? extends Critter> a)
  a.add(new Ant());
- We don't know what the type of ? is. But, it might be a list of Doodlebug or Critter.

ArrayList<E> vs. array

Advantages of ArrayList
- Can regrow as necessary
- Can add new methods

Disadvantages vs. Array
- Must use method syntax rather than []
- Less efficient
- Can't store primitive types (like int)
  - Although, can use Integer, (automatic boxing/unboxing makes it not-too-bad)

ArrayList<E> versus Vector<E>

Vector<E> offers similar functionality to ArrayList<E>, however is older.

Recommendation
- For new code, always use ArrayList<E>
  - Technical note, feel free to ignore: Vector<E> is synchronized, but you can create a synchronized ArrayList<E> with:
    - new Collections.synchronizedList(new ArrayList<E>)
Generics
Allows specifying compile-time type parameters for classes and methods
- Class definition has a T parameter specifying the type
  - public class ArrayList<T> { ... }
- When using the class, provide a type:
  - ArrayList<String> = new ArrayList<String>();

Creating your own generics
When specifying your class, specify a type parameter(s) in angle brackets:
- public class Sample<T> {
  T data;
  public void setData(T d) {
    data = d;
  }
  public T getData() {
    return data;
  }
}
To use:
- Sample<String> s = new Sample<String>();
  s.setData("hello");
  System.out.println(s.getData());

Automatic boxing-unboxing
Auto-conversion from primitive types to/from boxed types:
- Sample<Integer> si = new Sample<Integer>();
  si.setData(3);
  int i = si.getData();

Generic classes can have >1 type
Store a pair of elements of (possibly) different types
- public class Pair<S, T> {
  S data1;
  T data2;
  public Pair(S s, T t) {
    data1 = s;
    data2 = t;
  }
  public void setData1(S d) {
    data1 = d;
  }
  public void setData2(T d) {
    data2 = d;
  }
  public S getFirst() {
    return data1;
  }
  public T getSecond() {
    return data2;
  }
}
Generic methods

Allow you to specify a type parameter for a method

- Just as generic classes allow you to specify a type parameter for a class

Syntax:

- `<T> return_type method_name(parameter list) {

  }

To use:

- Either just:
  - result = method_name(args)
  - Compiler calculates `T` from the arguments
- Or:
  - `t = object.<class>method_name(args)`

Copy an array to an ordered list

```java
public static <T> ArrayList<T> copyToArrayList(T[] objects) {
    ArrayList<T> al = new ArrayList<T>(objects.length);
    for (T e: objects)
        al.add(e);
    return al;
}
```

```java
ArrayList<String> a = copyToArrayList(array);
System.out.println(a);
```
or

```java
ArrayList<String> a = TestGeneric.<String>copyToArraylist(array);
```

Return minimum element in an array

```java
static <T extends Comparable<T>> T min(T[] array) {
    T min = array[0];
    for (T e: array)
        if (e.compareTo(min) < 0)
            min = e;
    return min;
}
```