NC STATE UNIVERSITY

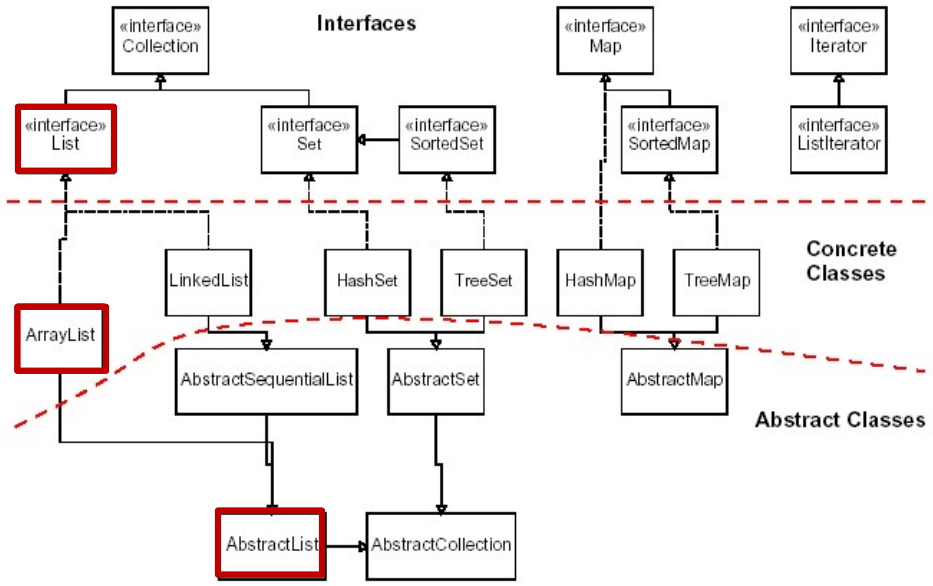
Generic ArrayLists

Reading
Zybooks 13.3 & 13.4

NC STATE UNIVERSITY

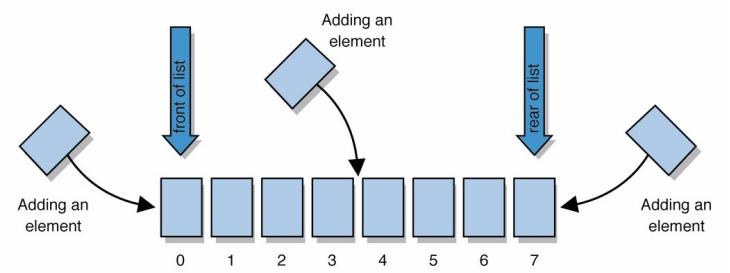
Collections

- Data structures – stores elements in a manner that makes it easy for a client to work with the elements
- Specific collections are specialized for particular types of operations
  - Ordering
  - Duplicates
  - Add/remove
- Common set of operations: add, remove, get, contains, size



Lists

- List: an ordered collection of elements
  - Accessible by a zero-based index
  - Size: number of elements in the list
  - Elements may be added to *an empty list*, at the *front*, *middle*, and *back*



NC STATE UNIVERSITY

Scenario

- Suppose a Course has a list of enrolled students
- We want to enroll, drop, email all the students, etc.

- ArrayIntList won't work for Courses!

- Options:
  - Create an ArrayCourseList
    - But then we have to create a custom list for every object type
  - Create an ArrayList of Objects
    - But then we have to cast when we get objects out
  - Create an ArrayList of *generic objects!*

NC STATE UNIVERSITY

Working with Generic ArrayLists

- Constructing

```
ArrayList<Course> list = new ArrayList<Course>();
```

- Adding an element to the end of the list

```
list.add(new Course(…));
```

- Adding an element at an index

```
list.add(3, new Course(…)); //(index, element)
```

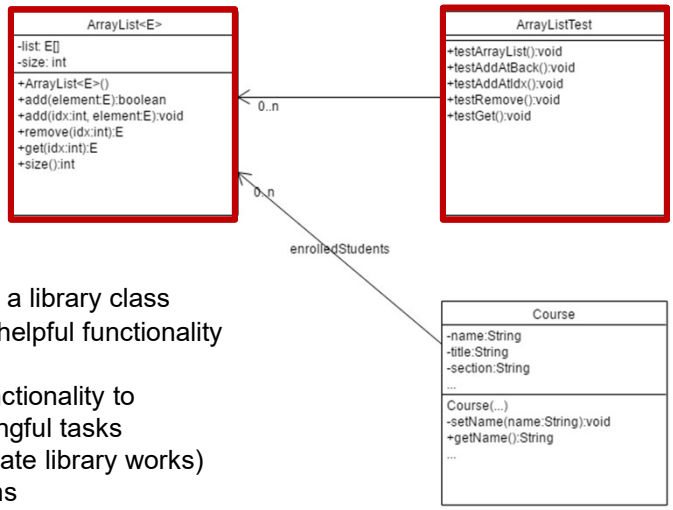
- Removing an element

```
Course c = list.remove(3); //(index)
```

- Getting an element from an index

```
Course c = list.get(3); //(index)
```

ArrayList Design



- **`ArrayList<E>`** is a library class
  - Encapsulates helpful functionality
- Client uses the functionality to accomplish meaningful tasks
  - Tests (to evaluate library works)
  - Client programs

Testing Lists

- Test each List method
  - Empty list
  - Front of the list
  - Middle of the list
  - Back of the list
- After each operation
  - Length is correct
  - Contents are correct (are the elements in the right order?)
- Then test the methods in combination
  - maybe `add` usually works, but fails after you call `remove`
  - what happens if I call `add` then `size`? `remove` then `toString`?
  - make multiple calls; maybe `size` fails the second time only

Testing `ArrayList()`

- Start by testing the constructor
  - Should create an empty list of strings

We can start with our **`ArrayIntList`** tests and refactor for Strings!

```
public class ArrayListTest {
    @Test
    public void testArrayList() {
        ArrayList<String> l = new ArrayList<String>();
        assertEquals(0, l.size());
    }
}
```

Why aren't we testing with **`Course`** objects?

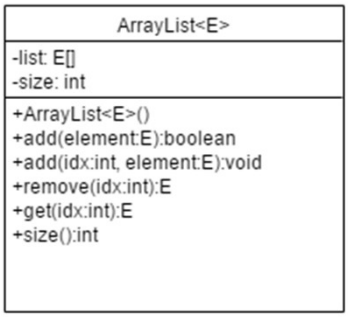
If the implementation is generic, we can test with whatever object type we want!

Generics in Java

- Provide a ***type*** for ***elements*** stored in collection classes
  - Ex: **`ArrayList`** of **`String`**s
- Our own classes can provide a generic type for the client to define
  - Use **`E`** to represent the element/type the user provides
  - **`E`** is a *variable* for the *type*! And really it can be named anything b/c it's a variable – just for a type!
- Class name with generics
  - **`ArrayIntList`** to **`ArrayList<E>`**
  - **`int`** to **`E`** when referring to an *element* of the list, not the size, indexes, and capacity

`ArrayList` State

- The state for a generic **`ArrayList`** is still an array and the size
  - Define the generic **`ArrayList`** class
  - Define the state for the generic **`list`** and **`size`**



`ArrayList` Constructor

- Constructor header does NOT include the generic type!
  - Even though clients include it when constructing an object.
  - The need for the generic type is assumed from the class' header

```
public ArrayList() {
   this(CAPACITY);
}
public ArrayList(int capacity) {
   if (capacity < 0) throw new IAE();
   //initialize list and size
}
```

NC STATE UNIVERSITY

Constructing an Array of a Generic Type

- Our `list` is an array of a generic type, `E`
- `list` is only a reference to an array
  - An array needs to know it's type and size to be constructed
  - For the class with the generic type, the compiler doesn't understand how to create an array of an unknown type
  - It also doesn't know how to create an object of an unknown type
    - The actual object type may not actually have a default constructor
    - Defining parameters doesn't make the type generic!

NC STATE UNIVERSITY

Constructing Generic Objects

- You can't construct objects of a generic type

```
public class ArrayList<E> {

   private E[] list;

   public ArrayList() {
      E item = new E();          //illegal!
      list = new E[10];          //illegal!
   }
}
```

Constructing Generic Objects

- We can work with the generic type when it's a reference
  - This includes parameters and return types

- But we need to construct an array of something to have a list!
  - What is the most generic concrete object in Java?

Constructing Generic Objects

- Create an `Object` or `Object[]`
- Cast to the generic type
- Ignore the Cast warning using an `@SuppressWarnings` annotation

```
public class ArrayList<E> {
   private E[] list;

   @SuppressWarning("unchecked")
   public ArrayList() {
      E item = (E) new Object();
      list = (E[]) new Object[10];
   }
}
```

Implementing `ArrayList`

- Implementing **`ArrayList`** is *almost* exactly the same as **`ArrayIntList`**
  - Update the type for ***elements*** from **`int`** to **`E`**
  - Update primitive comparisons on elements to use **`equals()`**

```
public int indexOf(E value) {
   for (int i = 0; i < size; i++) {
      //if (list[i] == value) {
      if (list[i].equals(value)) {
          return i;
      }
   }
   return -1;
}
```


Memory and Garbage Collection

- If you remove an element, there is still data in array locations outside the size of the array (but within the capacity)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	**7**	**5**	**12**	14	n	n	n	n
size	**5**									

- Now that we're working with objects, that means each element is a reference to an object!
- Java's garbage collector finds objects that are no longer used (referenced) and destroys them
  - Our array list may interfere with this. The object referenced by index 5 is no longer used, but still referenced. Therefore, no garbage collection.
  - Need to explicitly set the array element back to null (removes the reference)
  - When could this be a problem?

NC STATE UNIVERSITY

Removing the Reference

- Set the element at index `size-1` to null (before decrementing size!)

```
public E remove(int idx) {
   checkIndex(idx);
   E v = list[idx];
   //left shift elements

   list[size – 1] = null;
   size--;
   return v;
}
```


NC STATE UNIVERSITY

Generic Interface

```
public interface List<E> {
   public boolean add(E element);
   public void add(int idx, E element);
   public E get(int idx);
   public int indexOf(E value);
   public boolean isEmpty();
   public E remove(int idx);
   public E set(int idx, E value);
   public int size();
   …
}
public class ArrayList<E> implements List<E> {…}
```

Abstract*

- The **Abstract*** classes in the Java Collections Framework provide a partial implementation of the list
- It's still up to the user to define the list storage and size

- **Abstract*** classes define behaviors that utilize other list methods

```
public boolean add(E element) { // adding at the end of the list
    add(size, E element);  // is really adding at index size
    return true;
}
```


AbstractList

You'll implement a custom list that extends **AbstractList** in Lab 7!

- Intended for use for a random access data store (an array)
- *Unmodifiable list* – create the list and it doesn't change
  - Implement **get(int)** & **size()**
- *Modifiable list* – create list and change contents within existing elements
  - Implement **set(int, E)**
- *Variable-size list* – create list and change contents and size
  - Implement **add(int, E)** and **remove(int)**
- Provide a default constructor
- May override any other methods as needed

Customizing Lists

- But we have **`ArrayList`**, do we really need custom lists?
  - Yes!
- **`ArrayList`**
  - Allows **`null`** elements
  - Allows duplicates
  - Grows "forever" (until memory runs out)
  - Doesn't maintain elements in sorted order
- Custom lists can help us create lists specific for a set of requirements

`SortedList`

- **`SortedList`** keeps elements in sorted order
- What is the algorithm for adding an element to a sorted list such that the list remains sorted?
- Do any other list methods need additional logic to maintain sorted order? If so, which?

NC STATE UNIVERSITY

Comparing Objects

- With primitives, we can compare values using >, >=, <, and <=, ==, !=
- We have a way of comparing objects for equality
  - `.equals()` method
- To compare objects for order, we use the **Comparable** interface
  - The `compareTo()` method defines the natural ordering of `this` object to the parameter object

NC STATE UNIVERSITY

Comparable Interface

```
public interface Comparable<E> {
   public int compareTo(E other);
}
```

- `A.compareTo(B)` returns
  - Value < 0, if A comes “before” B in the natural ordering
  - Value > 0, if A comes “after” B in the natural ordering
  - Value = 0, if A and B are “equal” in the natural ordering

Using `Comparable.compareTo()`

- Use a call to `compareTo()` in if statements

Primitives	Objects
`if (a < b) { … }`	`if (a.compareTo(b) < 0) { … }`
`if (a <= b) { … }`	`if (a.compareTo(b) <= 0) { … }`
`if (a == b) { … }`	`if (a.compareTo(b) == 0) { … }`
`if (a != b) { … }`	`if (a.compareTo(b) != 0) { … }`
`if (a >= b) { … }`	`if (a.compareTo(b) >= 0) { … }`
`if (a > b) { … }`	`if (a.compareTo(b) > 0) { … }`

From Reges and Stepp slides

Using `Comparable`

```
public class MyClass implements Comparable<MyClass> {

   public int compareTo(MyClass other) {
      …
   }

}
```

`Comparable` Strategies

- When comparing the fields of an object to determine natural ordering
  - If you're comparing integers, you can subject related numeric values
    - `return x - other.x;`
  - If you're comparing Object fields that implement `Comparable`, delegate to the Object's `compareTo()`
    - `return object.compareTo(other);`
  - Compare the Object's `toString()` representations
    - `return toString().compareTo(other.toString());`
- Using these strategies depends on the requirements!

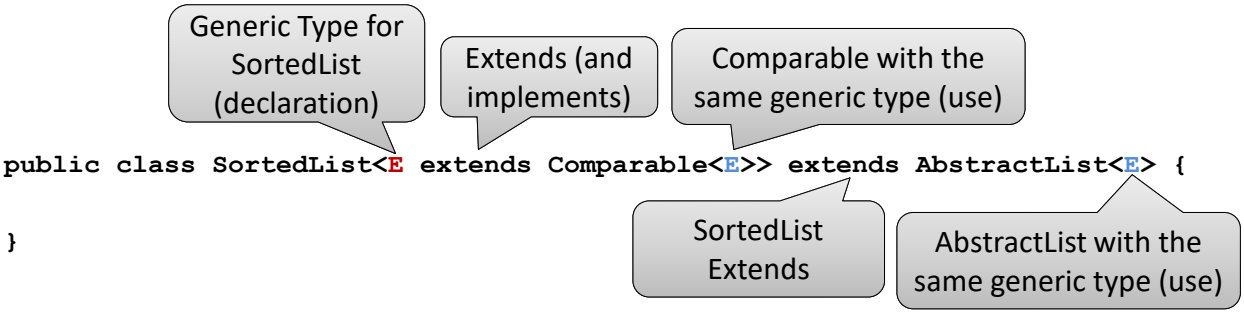
`SortedList` and `Comparable`

- To compare objects, the object should implement the `Comparable` interface
- But what if a client tries to use a `SortedList` with an element type that doesn't implement `Comparable`?

NC STATE UNIVERSITY

`SortedList` and `Comparable`

- We need a way to guarantee that the element type has the **`compareTo()`** method
- We can do that in the generic type definition



NC STATE UNIVERSITY

Generic Wildcards (?)

- Wildcards are used on ***references*** to ***generic objects*** when the type isn't known
  - When the client of a class with a generic type, doesn't know what type their client will use
  - Represents an unknown type
  - Used as a parameter, return type, field, or local variable of a client of a generic class

- See https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html for more details

NC STATE UNIVERSITY

Upper Bounded Wildcards

- Relax restrictions on a variable
- When used with Lists, makes the List read-only in the context of the method using the wildcard

```
public static double sumOfList(List<? extends Number> list) {
   double sum = 0;
   for (Number n : list) {
      s += n.doubleValue();
   }
   return sum;
}
```

Works for **Integer**, **Double**, and **Number**!

NC STATE UNIVERSITY

Unbounded Wildcards

- Used when
  - Writing a method that can be implemented with Object functionality
  - Restricting to object functionality that doesn't rely on the generic type
  - When working with lists, can only add null

```
public static void printList(List<?> list) {
   for (Object e : list) {
      System.out.print(e + " ");
   }
   System.out.println();
}
```

Lower Bounded Wildcards

- Restricts the unknown type to be a specific type or a super type of that type
- Example would work with `List<Integer>`, `List<Number>`, and `List<Object>`

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```