Advanced Programming Techniques in Java

COSI 12B

ArrayList

Lecture 16

Class Objectives

- `ArrayList` (section 10.1)

Review: Summary of Features of Actual Classes, Abstract Classes, and Interfaces

Property	Actual Class	Abstract Class	Interface
Instances (objects) of this can be created.	Yes	No	No
This can define instance variables and methods.	Yes	Yes	No
This can define constants.	Yes	Yes	Yes
The number of these a class can extend.	0 or 1	0 or 1	0
The number of these a class can implement.	0	0	Any number
This can extend another class.	Yes	Yes	No
This can declare abstract methods.	No	Yes	Yes
Variables of this type can be declared.	Yes	Yes	Yes

Review: The `ArrayList` class

- An `ArrayList` object uses an array to store its values
- Think of it as an auto-resizing array that can hold any type of object, with many convenient methods
- It maintains most of the benefits of arrays, such as fast random access
- It frees us from some tedious operations on arrays, such as sliding elements and resizing
- To use `ArrayList` remember to import `java.util.*;`
- We can declare arrays of different types e.g., `int[], String[], …` the `ArrayList` class has similar flexibility

Review: Java Generics

- Used to make an object usable for any types, while still preserving the type checking that Java allows
- Normally we must be specific about the type we're passing into an object, but Java allows us to make this variable
- Useful for making data structures, which we want to be applicable for any data we want to insert into them

Review: Java Generics

- We can make this code “generic”

```
public class PointBox{
   private Point p;
   public void put(Point p){
      this.p = p;
   }
   public Point get( ){
      return this.p;
   }
}
```

```
public class Box<T>{
   private T object;
   public void put(T object){
      this.object = object;
   }
   public T get( ){
      return this.object;
   }
}
```

- Now we can put an object of any type “T” into the box

Review: How to use this "Generic" Type

- In the `main` method, you can initialize a `Box` of any type by doing the following:
  `Box<TYPE> name = new Box<TYPE>( );`
  - e.g: `Box<String> stringBox = new Box<String>( );`
  - or: `Box<Point> pointBox = new Box<Point>( );`
- Now our code can be used for any type!

Example Code

```
public class Main{
  public static void main(String[ ] args){
    Point p2 = new Point(0,5);
    System.out.println("Making a box for points:");
    Box<Point> b1 = new Box<Point>( );
    b1.put(p2);
    System.out.println(b1.get().getY());

  }
}
```

Makes a specific version of the `Box` object for points

Java doesn't complain that we do `.getY()` on the object coming out of the box, since we told it that the object was going to be a `Point`

In summary …

- **Generic class** is a type in Java that is written to accept another type as part of itself
  - Generic (or "parameterized") classes were added to Java (after version 5) to improve the type safety of Java's collections
  - A parameterized type has one or more other types' names written between < and >

Why Use Generic Collections?

- Better type-checking: catch more errors, catch them earlier

```
// without Generics
List list = new ArrayList();
list.add("hello");

// With Generics
List<Integer> list = new ArrayList<Integer>();
list.add("hello"); // will not compile
```

- Documents intent
- Avoids the need to downcast from `Object`

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0); // no cast
```

Review: The `ArrayList` class

- An `ArrayList` object uses an array to store its values
- Think of it as an auto-resizing array that can hold any type of object, with many convenient methods
- It maintains most of the benefits of arrays, such as fast random access
- It frees us from some tedious operations on arrays, such as sliding elements and resizing
- To use `ArrayList` remember to import `java.util.*;`
- We can declare arrays of different types e.g., `int[], String[], …` the `ArrayList` class has **similar flexibility**

Wrapper Classes for Primitive Types

- Primitive numeric types are not objects, but sometimes they need to be processed like objects
  - When?
- Java provides *wrapper classes* whose objects contain primitive-type values
  - `Float, Double, Integer, Boolean, Character`
  - They provide constructor methods to create new objects that “wrap” a specified value
  - Also provide methods to “unwrap”

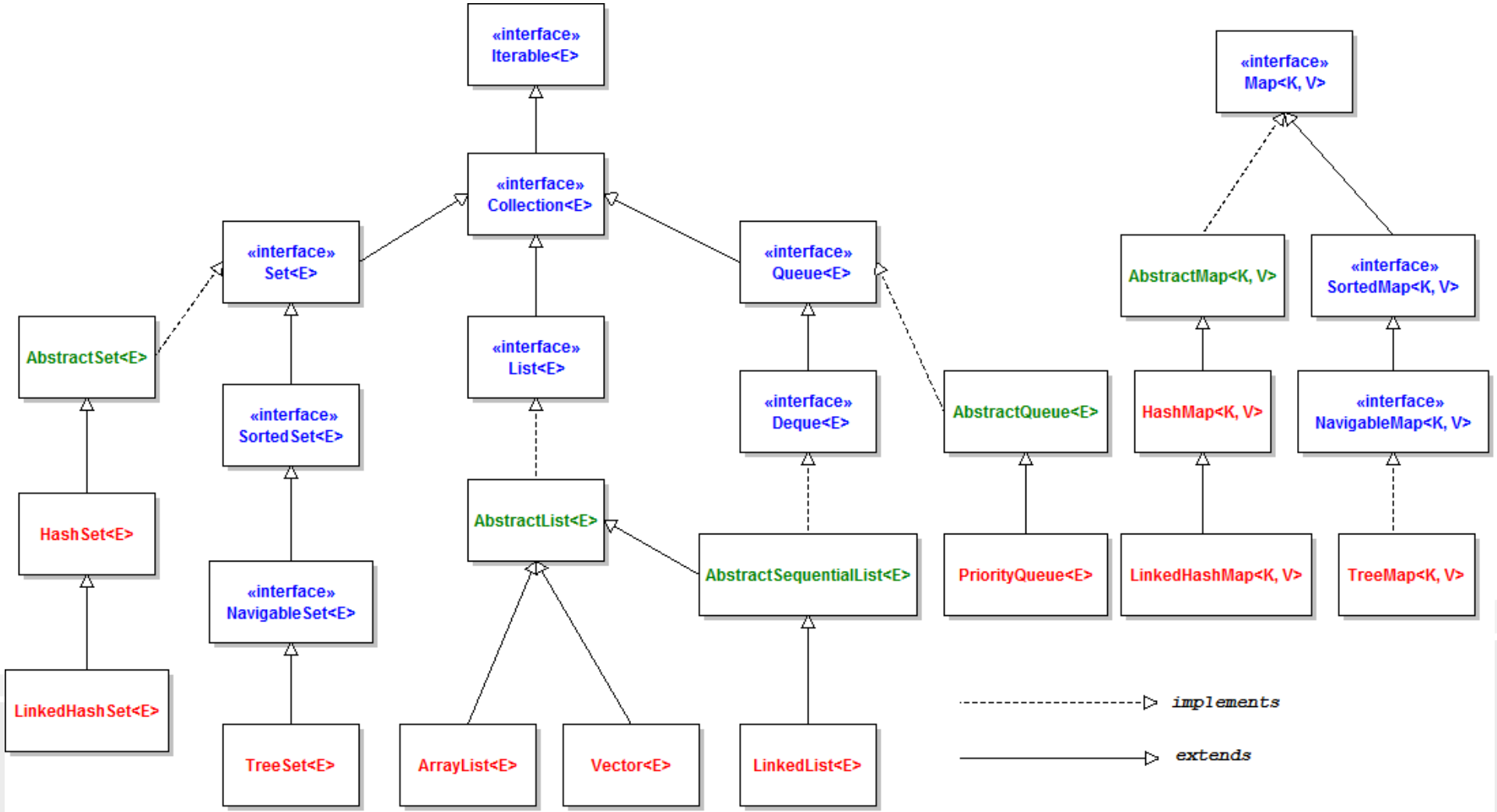
Wrapper classes

Primitive Type	Wrapper Type
`int`	`Integer`
`double`	`Double`
`char`	`Character`
`float`	`Float`
`boolean`	`Boolean`

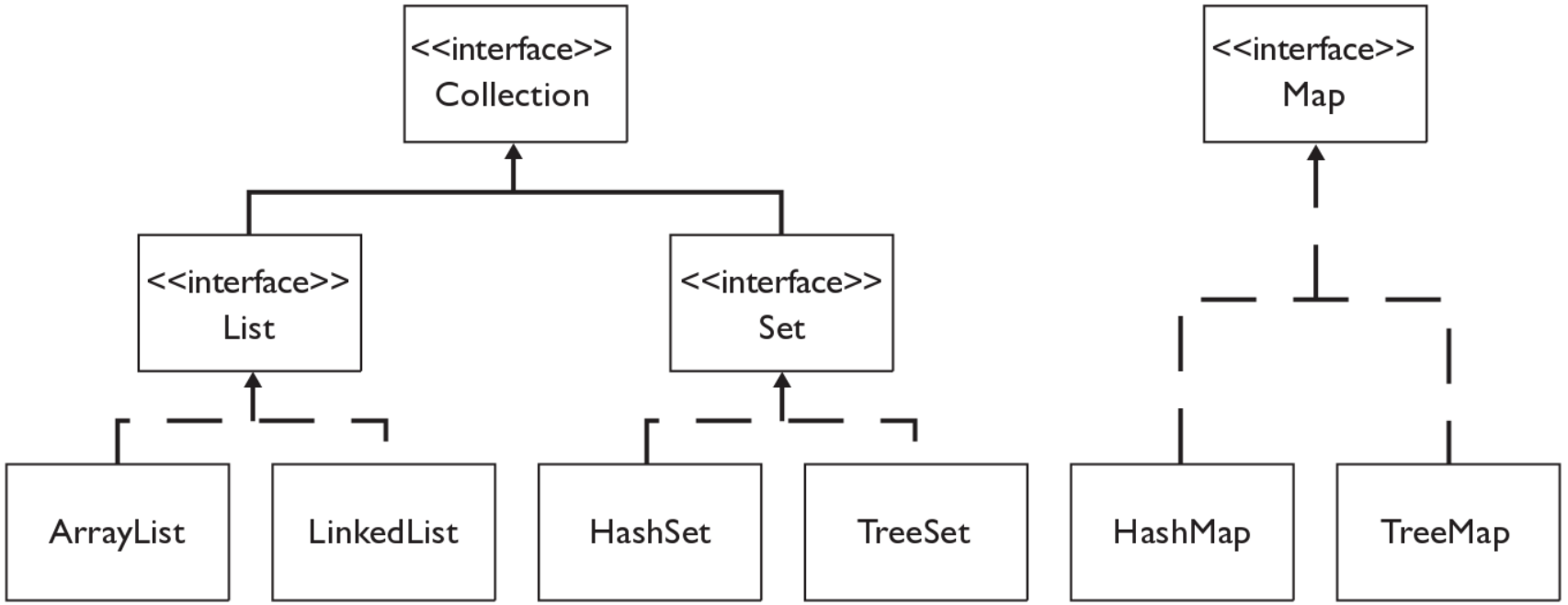
- A wrapper is an object whose sole purpose is to hold a primitive value
- Once you construct the list, use it with primitives as normal

Overview of Java Collections Framework (java.util.*)

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, …
```



Java collections framework



`ArrayList` of any type of objects

- When constructing an `ArrayList`, you must specify the type of elements it will contain between `< >`
  - By making the `ArrayList` class a Generic class, the same `ArrayList` class can store lists of different types

- **Syntax**: `ArrayList<Type> name = new ArrayList<Type>();`

```
ArrayList<String> names = new ArrayList<String>();
```

- Java 7's shorter "diamond operator" syntax

```
ArrayList<String> names = new ArrayList<>();
```

`ArrayList` of any type of objects (cont.)

- You can store any type of object in an `ArrayList` object
- `ArrayList<Point> points = new ArrayList<Point>();`
  - The `points` list will manipulate and return `Points`
- `ArrayList<Color> points = new ArrayList<Color>();`
  - The `points` list will manipulate and return `Colors`

Adding elements

- Elements are added dynamically to the end of the list:

```
ArrayList<String> list = new ArrayList<String>();
list.add("Brandeis");
list.add("Department");
list.add("Computer Science");
```

- What we store after each addition:

```
[]
[Brandeis]
[Brandeis, Department]
[Brandeis, Department, Computer Science]
```

values are always appended at the end of the list

Passing correct object types

- Java makes sure you add values of appropriate object type, otherwise it throws an exception

```
ArrayList<String> list = new ArrayList<String>();
Point p = new Point();
list.add(p);
```

- This does not compile because a `String` object is expected not a `Point`

Printing `ArrayLists`

- Unlike arrays, printing an `ArrayList` is easier since the `ArrayList` class overrides the `toString` method

```
ArrayList<String> list = new ArrayList<String>();
System.out.println("list = " + list);
list.add("Brandeis");
System.out.println("list = " + list);
list.add("Department");
System.out.println("list = " + list);
list.add("Computer Science");
System.out.println("list = " + list);
```

- Output:

```
list = []
list = [Brandeis]
list = [Brandeis, Department]
list = [Brandeis, Department, Computer Science]
```

you can print it even when it is empty

More on adding elements

- You can add a value at particular **index** in the list by using the method **`add(int index, E element)`**
  - It inserts the specified element at the specified position in this list, by shifting values to the right
- Example: `list.add(1, "cs12");`

before: `list = [Brandeis, Department, Computer Science]`

after: `list = [Brandeis, cs12, Department, Computer Science]`

All the values after `cs12` are shifted

Removing elements

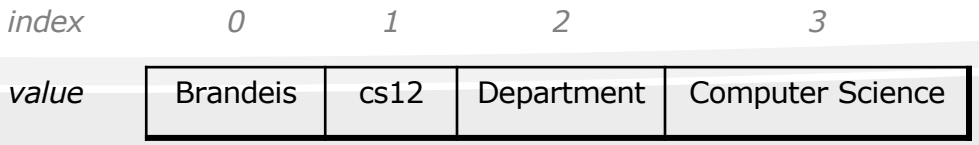
- Elements can also be removed by index:

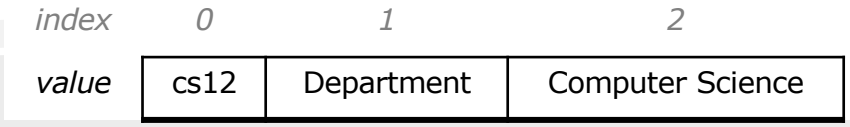
```
System.out.println("before remove list = " + list);
list.remove(0);
list.remove(1);
System.out.println("after remove list = " + list);
```

before : `list = [Brandeis, cs12, Department, Computer Science]`

after: `list = [cs12,Computer Science]`

- Notice that as each element is removed, the others shift downward in position to fill the hole
  - Therefore, the second remove gets rid of `Department`, not `cs12`





....

`size()`

- You can call the **`size()`** method to get the number of elements in the `ArrayList`

Issues with dynamic addition

- Assume you have an `ArrayList words`

```
words = [four, score, and, seven, years, ago]
```

- You want to add '~' before each word
- Solution 1:

```
for (int i=0; i < words.size(); i++) {
      words.add(i,'~');
}
```

- Does this work?

Issues with dynamic addition

- Assume you have an `ArrayList words`

```
words = [four, score, and, seven, years, ago]
```

- You want to add '~' before each word
- Solution 1:

```
for (int i=0; i < words.size(); i++) {
      words.add(i,'~');
}
```

- Does this work?
  - Infinite loop: it will never stop (out of memory error)

```
words = [~, four, score, and, seven, years, ago]
words = [~,~, four, score, and, seven, years, ago]
words = [~, ~, ~, four, score, and, seven, years, ago]
....
```

Solution 1

- The problem was that we ignored the shifting of elements
  - Since we add ‘~’ we want to move 2 positions to the right
- Correct solution:

Solution 1

- The problem was that we ignored the shifting of elements
  - Since we add '~' we want to move 2 positions to the right
- Correct solution:

```
for (int i=0; i < words.size(); i+=2) {
      words.add(i,'~');
}
```

```
words = [~, four, score, and, seven, years, ago]
words = [~, four, ~,score, and, seven, years, ago]
words = [~, four, ~,score, ~, and, seven, years, ago]
....
words = [~, four, ~, score, ~, and, ~,seven, ~, years, ~, ago]
```

“Backwards” solution (solution 2)

- You can visit the elements from right to left
  - Ensures that any changes you make occur on elements you already visited

```
for (int i= words.size()-1; i>=0; i--) {
      words.add(i,'~');
}
```

```
words = [four, score, and, seven, years, ~, ago]
words = [four, score, and, seven, ~, years, ~, ago]
....
words = [~, four, ~, score, ~, and, ~,seven, ~, years, ~, ago]
```

Issues with dynamic removal

- We now want to redo this operation (remove '~')
- Write code that removes every other element starting from the first one

```
words = [~, four, ~, score, ~, and, ~,seven, ~, years, ~, ago]
```

- Does this work? Why?

```
for (int i=0; i < words.size(); i+=2) {
      words.remove(i);
}
```

Issues with dynamic removal

- We now want to redo this operation (remove '~')
- Write code that removes every other element starting from the first one

```
words = [~, four, ~, score, ~, and, ~,seven, ~, years, ~, ago]
```

- Does this work? Why?

```
for (int i=0; i < words.size(); i+=2) {
      words.remove(i);
}
```

- Output

```
words = [four, ~, score, ~, and, ~,seven,~, years, ~, ago]
words = [four, ~, ~, and, ~,seven, ~, years, ~, ago]
words = [four, ~, ~, and, seven, ~, years, ~, ago]
...
```

Solution 1

- Again, dynamic shifting causes the problem
  - Once you remove an element, all the rest are shifted to the left
- Correct solution:

```
for (int i=0; i < words.size(); i++) {
      words.remove(i);
}
```

- Output

```
words = [four, score, and, seven, years, ago]
```