CSE 143
Lecture 2

More `ArrayList`; classes and objects

reading: 10.1; 8.1 - 8.7

slides created by Marty Stepp and Hélène Martin
http://www.cs.washington.edu/143/

Collections

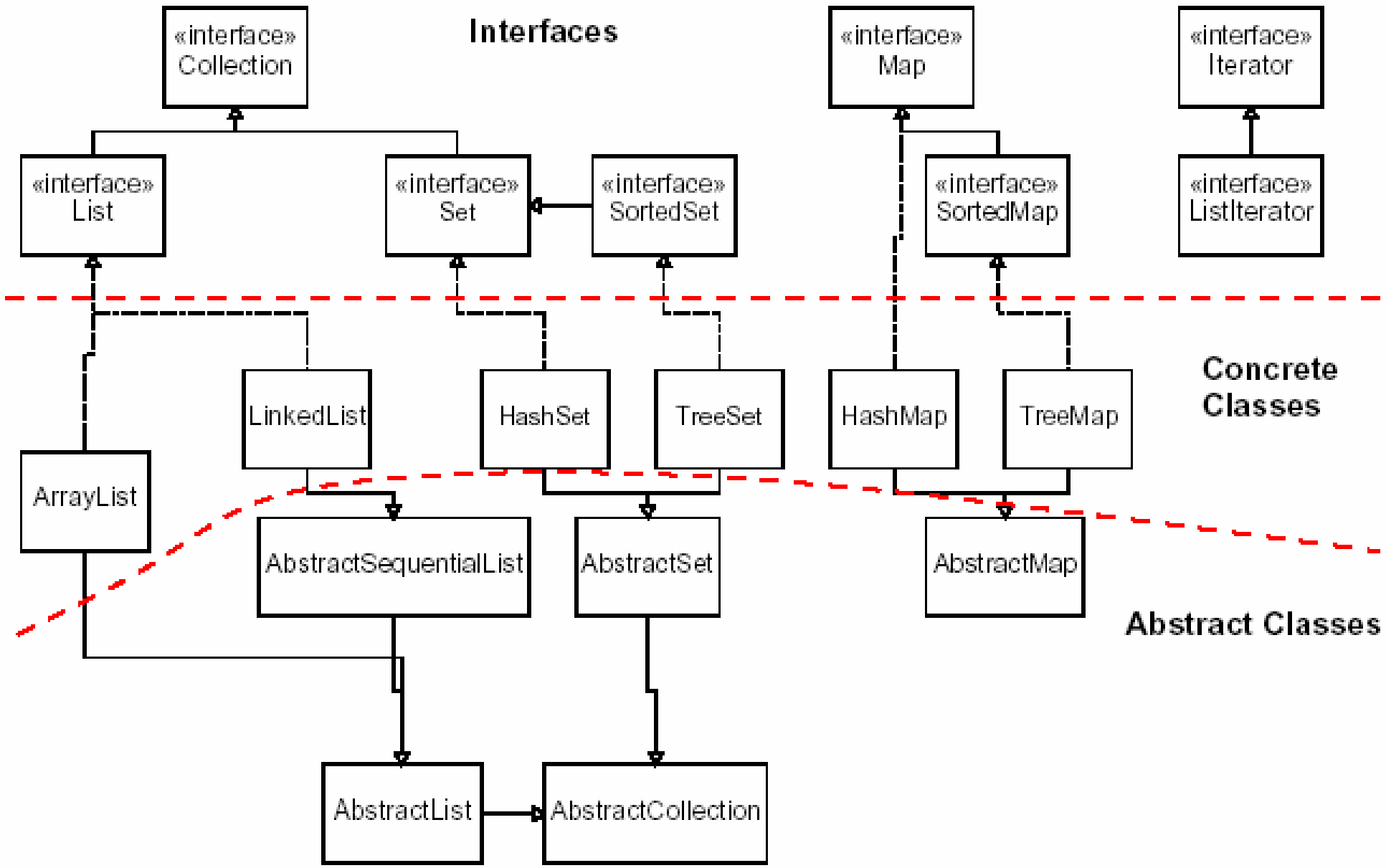
- **collection**: an object that stores data; a.k.a. "data structure"
  - the objects stored are called **elements**
  - some collections maintain an ordering; some allow duplicates
  - typical operations: *add*, *remove*, *clear*, *contains* (search), *size*

  - examples found in the Java class libraries:
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`

  - all collections are in the `java.util` package

```
import java.util.*;
```

Java collection framework



`ArrayList` methods (10.1)*

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

* *(a partial list; see 10.1 for other methods)*

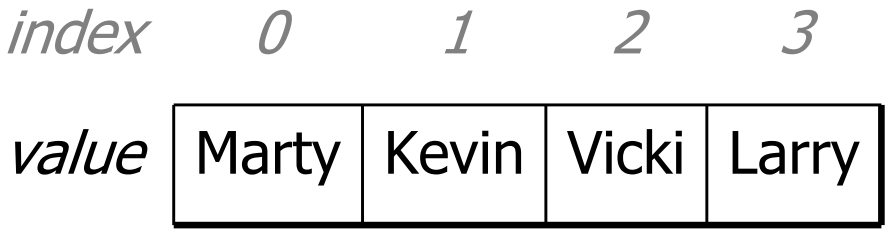
`ArrayList` methods 2

`addAll(`**list**`)` `addAll(`**index**`, `**list**`)`	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
`contains(`**value**`)`	returns true if given value is found somewhere in this list
`containsAll(`**list**`)`	returns true if this list contains every element from given list
`equals(`**list**`)`	returns true if given other list contains the same elements
`iterator()` `listIterator()`	returns an object used to examine the contents of the list (seen later)
`lastIndexOf(`**value**`)`	returns last index value is found in list (-1 if not found)
`remove(`**value**`)`	finds and removes the given value from this list
`removeAll(`**list**`)`	removes any elements found in the given list from this list
`retainAll(`**list**`)`	removes any elements *not* found in given list from this list
`subList(`**from**`, `**to**`)`	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
`toArray()`	returns the elements in this list as an array

Out-of-bounds

- Legal indexes are between **0** and the **list's size() - 1**.
  - Reading or writing any index outside this range will cause an `IndexOutOfBoundsException`.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty");    names.add("Kevin");
names.add("Vicki");    names.add("Larry");
System.out.println(names.get(0));          // okay
System.out.println(names.get(3));          // okay
System.out.println(names.get(-1));         // exception
names.add(9, "Aimee");                     // exception
```



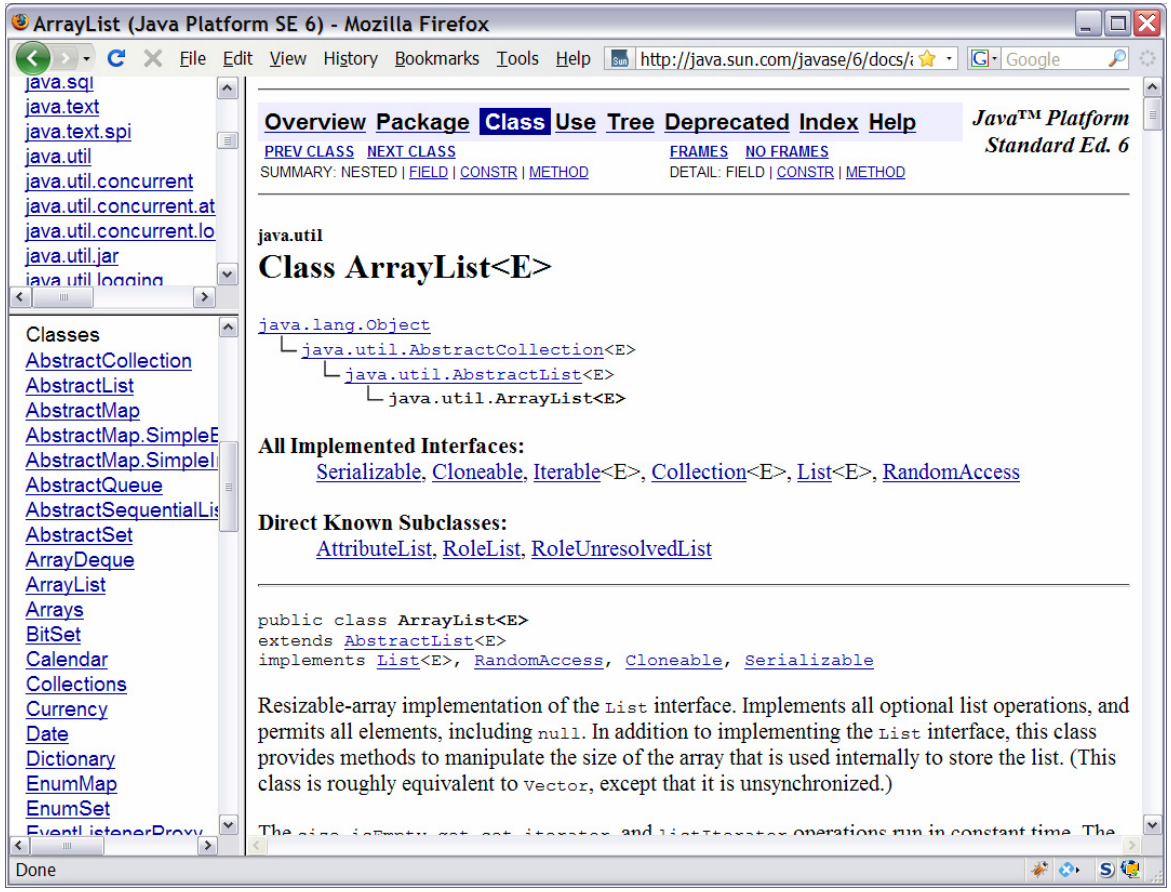
Collections class

Method name	Description
`binarySearch(`**list**`, `**value**`)`	returns the index of the given value in a sorted list (< 0 if not found)
`copy(`**listTo**`, `**listFrom**`)`	copies **listFrom**'s elements to **listTo**
`fill(`**list**`, `**value**`)`	sets every element in the list to have the given value
`max(`**list**`), min(`**list**`)`	returns largest/smallest element
`replaceAll(`**list**`, `**old**`, `**new**`)`	replaces an element value with another
`reverse(`**list**`)`	reverses the order of a list's elements
`shuffle(`**list**`)`	arranges elements into a random order
`sort(`**list**`)`	arranges elements into ascending order

```
ArrayList<String> names = new ArrayList<String>();
...
Collections.sort(names);
```

Learning about classes

- The Java API Specification is a huge web page containing documentation about every Java class and its methods.
  - The link to the API Specs is on the course web site.



`ArrayList` of primitives?

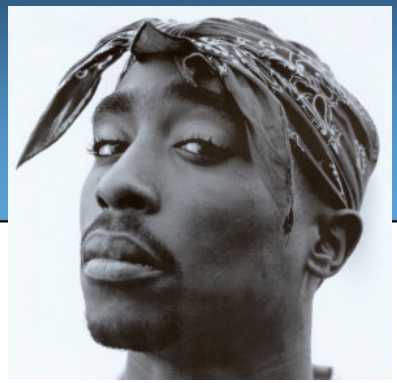
- The type you specify when creating an `ArrayList` must be an object/class type; it cannot be a primitive type.

```
// illegal; int cannot be a type parameter
ArrayList<int> list = new ArrayList<int>();
```

- But we can still use `ArrayList` with primitive types by using special classes called *wrapper* classes in their place.

```
// legal; creates a list of ints
ArrayList<Integer> list = new ArrayList<Integer>();
```

Wrapper classes



Primitive Type	Wrapper Type
int	Integer
double	Double
char	Character
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:

```
ArrayList<Double> grades = new ArrayList<Double>();
grades.add(3.2);
grades.add(2.7);
...
double myGrade = grades.get(0);
```

ArrayList "mystery"

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (int i = 1; i <= 10; i++) {
    list.add(10 * i);   // [10, 20, 30, 40, ..., 100]
}
```

- What is the output of the following code?

```
for (int i = 0; i < list.size(); i++) {
    list.remove(i);
}
System.out.println(list);
```

- Answer: `[20, 40, 60, 80, 100]`
  - *Observation:* If the list size or contents are being changed in a loop, that may lead to surprising or incorrect behavior.

ArrayList "mystery" 2

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) {
    list.add(2 * i);    // [2, 4, 6, 8, 10]
}
```

- What is the output of the following code?

```
int size = list.size();
for (int i = 0; i < size; i++) {
    list.add(i, 42);    // add 42 at index i
}
System.out.println(list);
```

- Answer: `[42, 42, 42, 42, 42, 2, 4, 6, 8, 10]`

Exercise

- Write a method `addStars` that accepts a list of strings as a parameter and places a `*` after each element.
  - Example: if an array list named `list` initially stores:

    `[the, quick, brown, fox]`

  - Then the call of `addStars(list);` makes it store:

    `[the, *, quick, *, brown, *, fox, *]`

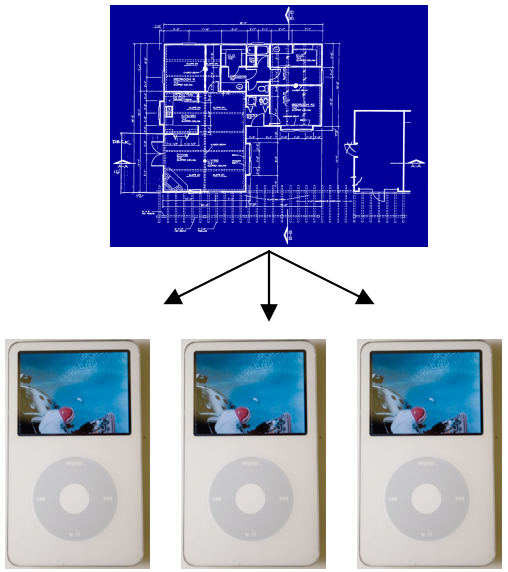
```
// solution
public static void addStars(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i += 2) {
        list.add(i, "*");
    }
}
```

Exercise

- Write a method `intersect` that accepts two sorted array lists of integers as parameters and returns a new list that contains only the elements that are found in both lists.
  - Example: if lists named `list1` and `list2` initially store:
  - `[1, 4, 8, 9, 11, 15, 17, 28, 41, 59]`
  - `[4, 7, 11, 17, 19, 20, 23, 28, 37, 59, 81]`
  - Then the call of `intersect(list1, list2)` returns the list:
  - `[4, 11, 17, 28, 59]`

Classes and objects

- **class**: A program entity that represents:
  - A complete program or module, or
  - A template for a type of objects.
  - (`ArrayList` is a class that defines a type.)



- **object**: An entity that combines **state** and **behavior**.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.
  - **abstraction**: Separation between concepts and details. Objects provide abstraction in programming.

Elements of a class

```
public class BankAccount {
    private String name;      // fields:
    private int id;           //  data encapsulated
    private double balance;   //  inside each object

    public BankAccount(String name, int id) {
        this.name = name;     // constructor:
        this.id = id;         //  initializes
        this.balance = 0.0;   //  new objects
    }

    public void deposit(double amount) {
        this.balance += amount; // instance method:
    }                           //  each object's
    ...                         //  behavior
}
```


"**implicit parameter**": object on which a method was called

`BankAccount` exercise

- Suppose we have a class `BankAccount` with the methods:

```
public BankAccount(String name, int id)
public void deposit(double amount)
public void withdraw(double amount)
public double getBalance()
public int getID()
```

- Make each account keep a log of all its transactions.
  - Desired: a `printLog` method that shows all transactions so far.

```
Deposit of $7.82
Withdrawal of $2.55
Deposit of $6.18
```

Objects storing collections

- An object can have an array, list, or other collection as a field.

```
public class Course {
    private double[] grades;
    private ArrayList<String> studentNames;

    public Course() {
        grades = new double[4];
        studentNames = new ArrayList<String>();
        ...
    }
```

- Now each object stores a collection of data inside it.