CSE 143
Lecture 3

More ArrayList;
object-oriented programming

reading: 10.1; 8.1 - 8.7

slides created by Marty Stepp
http://www.cs.washington.edu/143/
Out-of-bounds

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

```java
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
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Marty</td>
<td>Kevin</td>
<td>Vicki</td>
<td>Larry</td>
</tr>
</tbody>
</table>
ArrayList "mystery"

```java
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?

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

• Answer:

```
[20, 40, 60, 80, 100]
```
ArrayList<String> list = new ArrayList<String>();
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]
public static void name(ArrayList<Type> name) {

• Example:
  // Removes all plural words from the given list.
  public static void removePlural(ArrayList<String> list) {
      for (int i = 0; i < list.size(); i++) {
          String str = list.get(i);
          if (str.endsWith("s")) {
              list.remove(i);
              i--;
          }
      }
  }

• You can also return a list:
  public static ArrayList<Type> methodName(params) {
• Write a method `addStars` that accepts an array 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, *]`

• Write a method `removeStars` that accepts an array list of strings, assuming that every other element is a `*`, and removes the stars (undoing what was done by `addStars` above).
public static void addStars(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i += 2) {
        list.add(i, "*");
    }
}

public static void removeStars(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i++) {
        list.remove(i);
    }
}
• 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:
    
    - `list1`: [1, 4, 8, 9, 11, 15, 17, 28, 41, 59]
    - `list2`: [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]
Other Exercises

- Write a method `reverse` that reverses the order of the elements in an `ArrayList` of strings.

- Write a method `capitalizePlurals` that accepts an `ArrayList` of strings and replaces every word ending with an "s" with its uppercased version.

- Write a method `removePlurals` that accepts an `ArrayList` of strings and removes every word in the list ending with an "s", case-insensitively.
Object-Oriented Programming

reading: 8.1 - 8.7
Classes and objects

• **class**: A program entity that represents either:
  1. A program / module, or
  2. A template for a new type of objects.

• **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.
Blueprint analogy

iPod blueprint

**state:**
- current song
- volume
- battery life

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song

---

iPod #1

**state:**
- song = "1,000,000 Miles"
- volume = 17
- battery life = 2.5 hrs

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song

---

iPod #2

**state:**
- song = "Letting You"
- volume = 9
- battery life = 3.41 hrs

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song

---

iPod #3

**state:**
- song = "Discipline"
- volume = 24
- battery life = 1.8 hrs

**behavior:**
- power on/off
- change station/song
- change volume
- choose random song
**Clients of objects**

- **client program**: A program that uses objects.
  - Example: *Bomb* is a client of *DrawingPanel* and *Graphics*.

```java
Bomb.java (client program)
public class Bomb {
    public static void main(String[] args) {
        new DrawingPanel(...)
        new DrawingPanel(...)
        ...
    }
}
```

```java
DrawingPanel.java (class)
public class DrawingPanel {
    ...
}
```
• **field**: A variable inside an object that is part of its state.
  – Each object has *its own copy* of each field.

• Declaration syntax:

  ```java
  private type name;
  ```

  – Example:

  ```java
  public class Point {
      private int x;
      private int y;
      ...
  }
  ```
Instance methods

- **instance method** (or **object method**): A method inside each object of a class that gives behavior to each object.

  ```java
  public type name(parameters) {
    statements;
  }
  ```

  – same syntax as static methods, but without `static` keyword

  Example:

  ```java
  public void translate(int dx, int dy) {
    x = x + dx;
    y = y + dy;
  }
  ```
The implicit parameter

- **implicit parameter:** The object on which an instance method is being called.
  - If we have a `Point` object `p1` and call `p1.translate(5, 3);` the object referred to by `p1` is the implicit parameter.
  - If we have a `Point` object `p2` and call `p2.translate(4, 1);` the object referred to by `p2` is the implicit parameter.
  - The instance method can refer to that object's fields.
    - We say that it executes in the *context* of a particular object.
    - `translate` can refer to the `x` and `y` of the object it was called on.
Constructors

• **constructor**: Initializes the state of new objects.

```java
public type(parameters) {
    statements;
}
```

– runs when the client uses the `new` keyword
– no return type is specified; implicitly "returns" the new object

```java
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }
}
```
BankAccount exercise

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

  ```java
  public BankAccount(int id)
  public void deposit(double amount)
  public void withdraw(double amount)
  public double getBalance()
  public int getID()
  ```

• How would we make each account object keep a log of all deposit/withdrawal transactions?
  
  - Desired: a `printLog` method that shows all transactions so far.

    Deposit of $7.84
    Withdrawal of $2.53
    Deposit of $6.19
Objects storing collections

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

```java
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.