Lists

Recall: Public Interface

The **public interface** of a class comprises the public methods and instance variables defined in the class.

- These public methods and instance variables are the only ways that objects of the class can be used once created.

- Private instance variables and helper methods make the implementation of the public interface possible, but are not exposed to the outside.

Rules & "Public Interface" of Arrays

- Once an array object is declared, its size can't change.

- An array can only contain elements of one declared type.

- Array elements are accessed/modified using `[]`, and we get array length with `.length`
  - This is the extent of the "public interface" of arrays

Imagining a Better Future

What if we had a data structure that was a bit easier to use?

- **Something that can change size to accomodate more elements.**

- Something that has the same methods for getting & setting values in an ordered sequence **in addition to other useful operations.**
  - deleting elements? searching for elements? subsequences?

The ArrayList

- In Java (and other languages...), the `List` data type describes an ordered, resizeable collection of elements.

- `List.java` defines a public interface for what any `List` can do
  - Java includes many different classes that implement this public interface: `Vector`, `LinkedList`, `ArrayList`, and many more.
  - The following examples will all use `ArrayList`, but any of the others would work, too.

Importing

To use an `ArrayList` in your program, make sure to put the import statement at the top of the file.

```java
import java.util.ArrayList;
```

Declaring ArrayLists

In order to declare a variable for an ArrayList and initialize an empty one:

```
ArrayList<DataType> values = new ArrayList<DataType>();
```

- In the angle brackets (`<>`) goes the data type that the ArrayList will store
  - For example, we say that `ArrayList<String>` is an ArrayList of `Strings`.
  - The contents of the angle brackets on the left and right should match.
    - (you can also safely omit the contents of the right angle brackets)

ArrayList Types

Lists cannot support primitive types like `int`, `double`, `boolean`, or `char`.

Java has a bunch of *wrapper* classes that can be used instead.

Wrong!	Right.
ArrayList<int>	ArrayList<Integer>
ArrayList<double>	ArrayList<Double>
ArrayList<char>	ArrayList<Character>
ArrayList<boolean>	ArrayList<Boolean>

Adding to an ArrayList

`add(T element)` adds the value `element` to the ArrayList.

```
ArrayList<String> staffNames = new ArrayList<String>();
staffNames.add("Hannah");
staffNames.add("Jared");
staffNames.add("Elie");
staffNames.add("Mia");
staffNames.add("Ruth");
System.out.println(staffNames);
```

🖨️ ➡️ `["Hannah", "Jared", "Elie", "Mia", "Ruth"]`

...did you catch that?

🚨 🚨 🚨 🚨 🚨

You can print out the contents of an ArrayList with a single call to `System.out.println()`*!*

🚨 🚨 🚨 🚨 🚨

Anyway...

`add(T element)` adds the value `element` to the List

- Elements are added to the end

- the size of the ArrayList will grow by one

- The List can grow arbitrarily large, although there are occasionally performance penalties when growing the List.
  - *don't worry about this for now* 🕐

Getting from a List

`get(int index)` returns the element at the specified location. Valid indices range from `0` to `list.size() - 1` (just like in arrays.)

```java
ArrayList<String> staffNames = new ArrayList<String>();
staffNames.add("Hannah");
staffNames.add("Jared");
staffNames.add("Elie");
staffNames.add("Mia");
staffNames.add("Ruth");
System.out.println(staffNames.get(2));
```

🖨️ ➡️ `"Elie"`

Getting from a List

get(int index) returns the element at the specified location. Valid indices range from 0 to list.size() - 1 (just like in arrays.)

```java
ArrayList<String> staffNames = new ArrayList<String>();
staffNames.add("Hannah");
staffNames.add("Jared");
staffNames.add("Elie");
staffNames.add("Mia");
staffNames.add("Ruth");
System.out.println(staffNames.get(0));
```

🖨️ ➡️ "Hannah"

Getting from a List

`get(int index)` returns the element at the specified location. Valid indices range from `0` to `list.size() - 1` (just like in arrays.)

```java
ArrayList<String> staffNames = new ArrayList<String>();
staffNames.add("Hannah");
staffNames.add("Jared");
staffNames.add("Elie");
staffNames.add("Mia");
staffNames.add("Ruth");
System.out.println(staffNames.get(-1));
```

Exception in thread "main" java.lang.IndexOutOfBoundsException: Index -1 out of bounds for length 5 🚨🚨🚨🚨

Getting from a List

get(int index) returns the element at the specified location. Valid indices range from 0 to list.size() − 1 (just like in arrays.)

```java
ArrayList<String> staffNames = new ArrayList<String>();
staffNames.add("Hannah");
staffNames.add("Jared");
staffNames.add("Elie");
staffNames.add("Mia");
staffNames.add("Ruth");
System.out.println(staffNames.get(10));
```

Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 10 out of bounds for length 5 🚨🚨🚨🚨

Updating Values in a List

`set(int index, T element)` sets the value at the specified location to be `element`.

Valid indices range from `0` to `list.size() - 1` still.

```java
ArrayList<String> staffNames = new ArrayList<String>();
staffNames.add("Hannah");
staffNames.add("Jared");
staffNames.add("Elie");
staffNames.add("Mia");
staffNames.add("Ruth");
staffNames.set(2, "Evil Elie");
System.out.println(staffNames);
```

🖨️➡️ `["Hannah", "Jared", "Evil Elie", "Mia", "Ruth"]`

Updating Values in a List

`set(int index, T element)` actually returns the value that's being replaced, if you want it.

```java
ArrayList<String> staffNames = new ArrayList<String>();
staffNames.add("Hannah");
staffNames.add("Jared");
staffNames.add("Elie");
staffNames.add("Mia");
staffNames.add("Ruth");
String oldName = staffNames.set(2, "Evil Elie");
System.out.println(oldName);
```

🖨️➡️ `"Elie"`

The Length of a List

`size()` returns the number of elements inside of the specified ArrayList.

```java
ArrayList<String> staffNames = new ArrayList<String>();
System.out.print(staffNames.size() + " ");
staffNames.add("Hannah");
System.out.print(staffNames.size() + " ");
staffNames.add("Jared");
System.out.print(staffNames.size() + " ");
```

🖨️➡️ `0 1 2`

There's more where that came from…

Check out the documentation if you want to use other methods of an ArrayList. Here are a few handy ones.

Method	Return Type	Purpose
add(int index, T element)	void	Put an element at a specific place in the ArrayList
clear()	void	Remove all elements from the list
remove(int index)	element type	Delete and return whatever lives at the specified index.

Refresher: Iterating over Arrays

```java
Rectangle[] shapes = new Rectangle[10];
// assume that we add some Rectangles to the array here!

for (int i = 0; i < shapes.length; i++) {
    Rectangle currentElement = shapes[i];
    System.out.println("shapes has " + currentElement + " at index " + i);
}
```

Iterating over Lists

It's pretty similar to array iteration, replacing `shapes[idx]` with `shapes.get(idx)` and `shapes.length` with `shapes.size()`:

```java
ArrayList<Rectangle> shapes = new ArrayList<Rectangle>();
// assume that we add some Rectangles to the ArrayList here!

for (int i = 0; i < shapes.size(); i++) {
    Rectangle currentElement = shapes.get(i);
    System.out.println("shapes has " + currentElement + " at index " + i);
}
```

Enhanced Iteration over ArrayLists

You can also use the **Enhanced For Loop** to iterate over ArrayLists

```
ArrayList<Rectangle> shapes = new ArrayList<Rectangle>();
// assume that we add some stuff to the ArrayList here!

for (Rectangle currentElement : shapes) {
    System.out.println("currentElement is now " + currentElement);
}
```

The iteration order is the same: it'll proceed from index `0` to `shapes.size() - 1`.

Within the enhanced for loop, the value of `currentElement` is set directly to be a `Rectangle`—no indexing necessary or even possible.

Technically, you can do this over plain old arrays, too.

Problem: Acceptable Names

Given an ArrayList of names, make sure that they start with an uppercase letter. If they don't, print that you're fixing the name and then modify the list to have the correctly formatted name.

You'll find that `Character.isUpperCase(char c)` and `Character.toUpperCase(char c)` will come in handy.

Make sure to change the ArrayList *in-place*, which is to say that you shouldn't create a new ArrayList to complete this task.

Solution:

```java
public void fixFormatting(ArrayList<String> names) {
    for (int i = 0; i < names.size(); i++) {
        String currentName = names.get(i);
        char firstLetter = currentName.charAt(0);
        if (!Character.isUpperCase(firstLetter)) {
            char firstLetterUpper = Character.toUpperCase(firstLetter);
            String rest = currentName.substring(1);
            String fixedName = firstLetterUpper + rest;
            names.set(i, fixedName);
            System.out.print("Fixing name " + currentName);
            System.out.println(" at position " + i);
        }
    }
}
```

Problem: Concatenate

Given two ArrayLists, create a new ArrayList containing first the values from the first ArrayList followed by the values from the second ArrayList

Make sure to create and return a new ArrayList

```
public ArrayList<Integer> concatenate(ArrayList<Integer> first, ArrayList<Integer> second) {

}
```

Solution:

```java
public ArrayList<Integer> concatenate(ArrayList<Integer> first, ArrayList<Integer> second) {
    ArrayList<Integer> newList = new ArrayList<Integer>();

    for (int i : first) {
        newList.add(i);
    }
    for (int i : second) {
        newList.add(i);
    }
    return newList;
}
```

Problem: Weave

Given two ArrayLists, create a new ArrayList with the values from the input lists woven together like so:

```
a = [1, 3, 5, 7];
b = [2, 4, 6, 8, 10, 12];
weave(a, b) --> [1, 2, 3, 4, 5, 6, 7, 8, 10, 12]
```

Make sure to create and return a new ArrayList.

```java
public ArrayList<Integer> weave(ArrayList<Integer> first, ArrayList<Integer> second) {

}
```

Solution:

```java
public ArrayList<Integer> weave(ArrayList<Integer> first, ArrayList<Integer> second) {
    ArrayList<Integer> newList = new ArrayList<Integer>();
    int firstIdx = 0;
    int secondIdx = 0;
    while (firstIdx < first.size() || secondIdx < second.size()) {
        if (firstIdx < first.size()) {
            newList.add(first.get(firstIdx));
            firstIdx++;
        }
        if (secondIdx < second.size()) {
            newList.add(second.get(secondIdx));
            secondIdx++;
        }
    }
    return newList;
}
```

Using Lists in Creative Ways

Bouncing Ball Simulation

Ingredients:

- `Ball.java`, a class that defines how a 2D ball moves & bounces on a screen
- `BouncingBalls.java`, a class that:
    - contains a main method so that the simulation is runnable
    - creates an `ArrayList<Ball>` in which to store the objects to be simulated
    - defines a "physics" (animation) loop

A Bit of Physics

To simulate an object's motion in 2D space over time, we need to keep track of its:

- position (p_x, p_y)
  - where the object is **right now**

- velocity/speed (v_x, v_y)
  - how much the object should move from where it is right now to where it will be next time we look

- acceleration (a_x, a_y)
  - how much the object's velocity should change from what it is right now to what it will be next time we look
  - we'll hold acceleration constant

A Bit of Physics

Since our simulation is run using a loop, we do our calculations in *discrete steps*.

- We denote the step number using superscripts, so p_x^t means "x position at step t"

- We'll assume a constant unit timestep, meaning that we don't have to account for the length of the timestep in our equations
  - *(ignore this point if the details of physical simulations are not interesting to you)*

A Bit of Physics

Equation	Meaning	Code
$p_x^{t+1} = p_x^t + v_x^t$	x position in the next iteration is equal to the x position now plus the x speed now	px = px + vx
$p_y^{t+1} = p_y^t + v_x^t$	y position in the next iteration is equal to the y position now plus the y speed now	py = py + vy
$v_x^{t+1} = v_x^t + a_x^t$	x speed in the next iteration is equal to the x speed now plus the x acceleration	vx = vx + ax
$v_y^{t+1} = v_y^t + a_y^t$	y speed in the next iteration is equal to the y speed now plus the y acceleration	vy = vy + ay

Implementing `Ball.java`

What behaviors does a `Ball` object need to exhibit as part of a simple physics simulation?

- Needs to be drawable so that we can see the simulation
- Needs to move & bounce pursuant to the previous equations

Methods:
`public void draw()`, `public void update()`

Implementing `Ball.java`

What properties does a `Ball` object need to store in order to perform these operations?

- position, x and y

- velocity, x and y

- acceleration, x and y
  - we'll ignore x acceleration, and y acceleration is just gravity

- radius
  - used for drawing
  - used for deciding when to bounce

The Simulator

The simulator will be responsible for initializing and keeping track of all of the balls in the simulation.

- How will we store all of the objects being simulated?
    - Create an `ArrayList<Ball>`
- How will we draw each of the objects being simulated?
    - Iterate through the array list and call the `draw()` method on each of the `Ball` objects.
- How will we get each of the objects to move and bounce?
    - Iterate through the array list and call the `update()` method on each of the `Ball` objects

The Simulator

```java
import java.util.ArrayList;

public class BouncingBalls {
    public static void main(String[] args) {
        int N = 40;
        ArrayList<Ball> allBalls = new ArrayList<Ball>();
        for (int i = 0; i < N; i++) {
            allBalls.add(new Ball());
        }

        PennDraw.setCanvasSize(600, 600);
        PennDraw.enableAnimation(30);
        while (true) {
            PennDraw.clear();
            for (int i = 0; i < N; i++) {
                Ball current = allBalls.get(i);
                current.draw();
                current.update();
            }
            PennDraw.advance();
        }
    }
}
```

A First Pass at the "Bouncing" Ball

```java
public class Ball {
    private double px, py, vx, vy, gravity, radius;

    public Ball() {
        px = Math.random();
        py = Math.random();
        vx = -0.005 + (Math.random() * 0.01); // [-0.005, 0.005]
        vy = -0.005 + (Math.random() * 0.01);
        gravity = -0.0001;
        radius = 0.02 + Math.random() * 0.04; // [0.02, 0.06]
    }

    public void draw() {
        PennDraw.filledCircle(px, py, radius);
    }

    public void update() {
        px = px + vx;
        py = py + vy;
        vy = vy + gravity;
    }
}
```
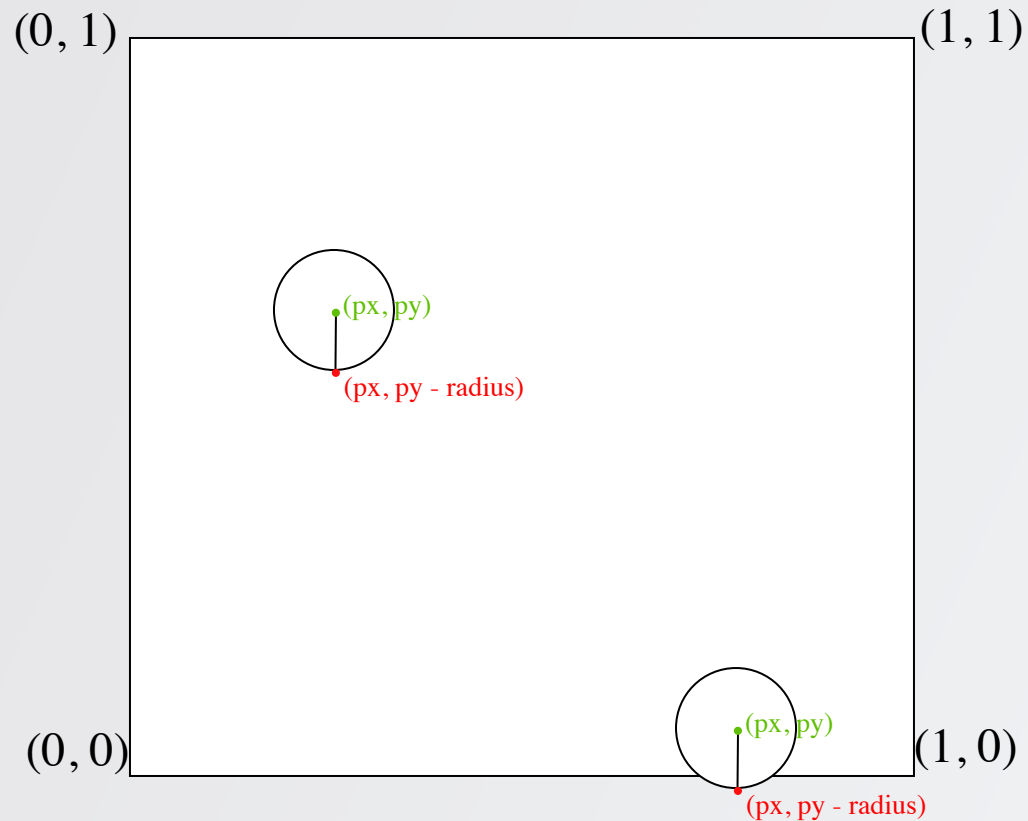
39

Problem: No Bouncing!

Currently, the balls just drop off the sides or bottom of the screen. How can we get them to bounce?

- Check if the ball has gone past the left, right, or bottom of the screen

- Simulate a bounce by inverting the velocity for the next update step

A Bounce

On the left, we have a sketch of the canvas with two balls.

- Which one should "bounce"?

- How can you formalize what it means for a ball to bounce off of the bottom of the screen?

Checking a Bounce

A ball should bounce off the bottom of the screen when, at time step t:

- The ball is traveling downwards ($v_y^t < 0$)

- The bottom of the ball is at or below the bottom of the screen ($p_y^t - \text{radius} \leq 0$)

Modeling a Bounce

What happens when an object bounces off of a surface?

- The object should change direction

- The object should lose a bit of momentum

The Bounce:

$$v_y = -0.9 * v_y$$

A Better **update()**

```java
public void update() {
    px = px + vx;
    py = py + vy;
    vy = vy + gravity;

    if (vy < 0 && py - radius <= 0) {
        vy = -0.9 * vy;
    }
}
```

A Best `update()`

```java
public void update() {
    px = px + vx;
    py = py + vy;
    vy = vy + gravity;

    if (vy < 0 && py - radius <= 0) {
        vy = -0.9 * vy;
    }


    if ((vx < 0 && px - radius <= 0) ||
        (vx > 0 && px + radius >= 1)) {
        vx = -0.9 * vx;
    }
}
```