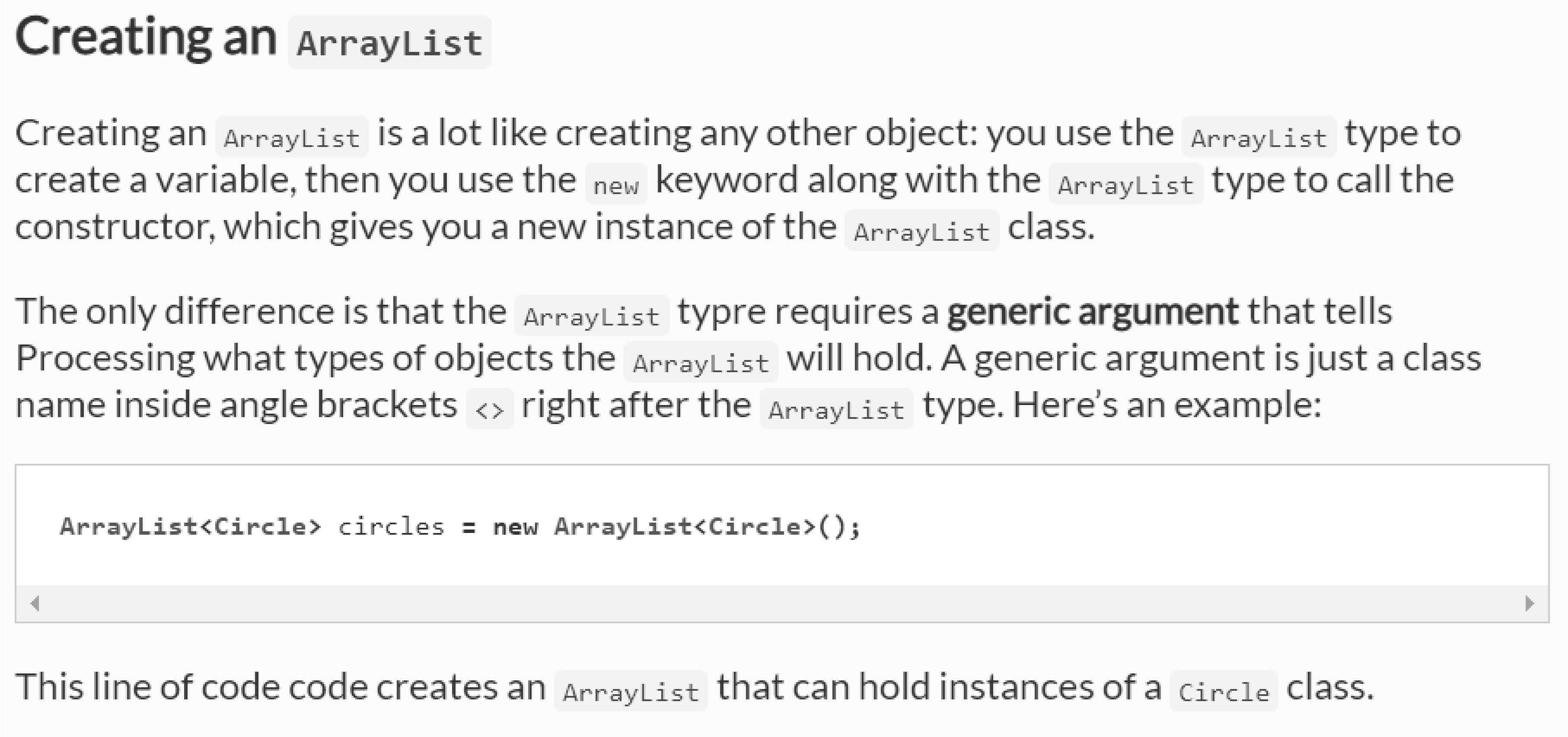
Week 7a : Array-List Programming

Arrays and Array-List

Overview

- Creating an ArrayList
- Adding to an ArrayList
- Getting from an ArrayList
- Removing from an ArrayList
- Example



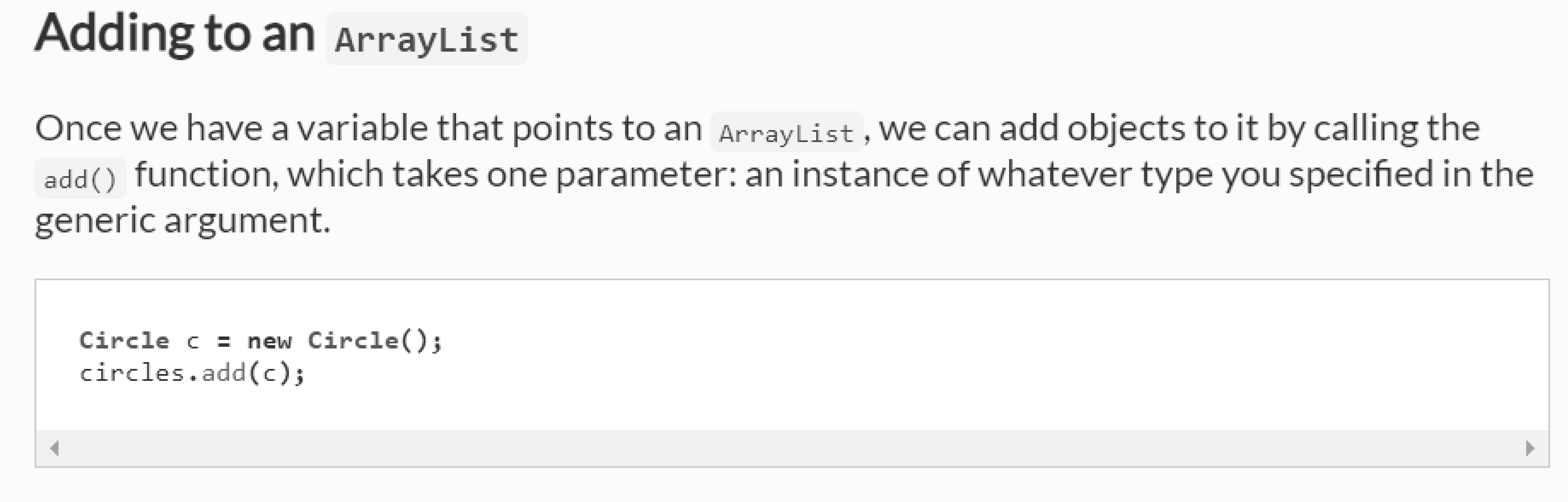
Creating an `ArrayList`

Creating an `ArrayList` is a lot like creating any other object: you use the `ArrayList` type to create a variable, then you use the `new` keyword along with the `ArrayList` type to call the constructor, which gives you a new instance of the `ArrayList` class.

The only difference is that the `ArrayList` typre requires a **generic argument** that tells Processing what types of objects the `ArrayList` will hold. A generic argument is just a class name inside angle brackets `<>` right after the `ArrayList` type. Here's an example:

```
ArrayList<Circle> circles = new ArrayList<Circle>();
```

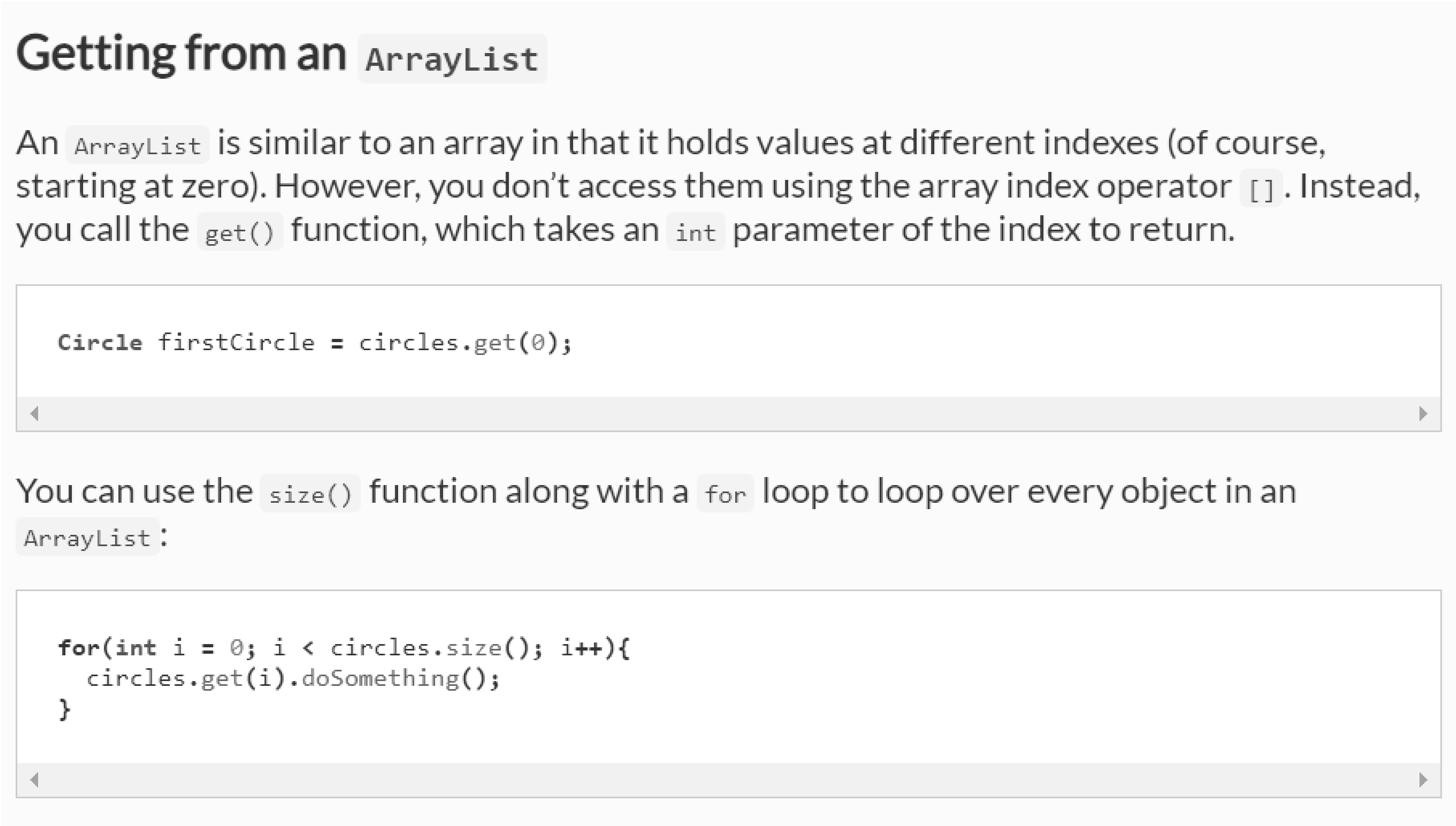
This line of code code creates an `ArrayList` that can hold instances of a `Circle` class.



Adding to an `ArrayList`

Once we have a variable that points to an `ArrayList`, we can add objects to it by calling the `add()` function, which takes one parameter: an instance of whatever type you specified in the generic argument.

```
Circle c = new Circle();
circles.add(c);
```



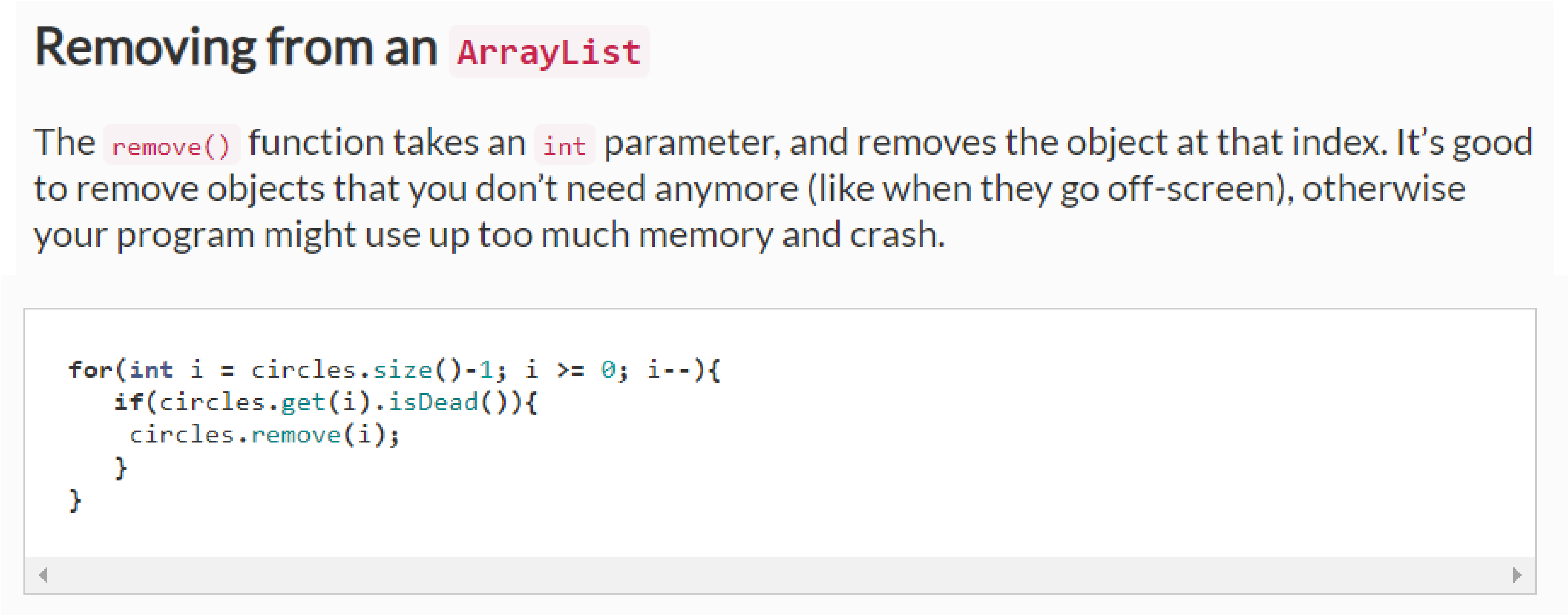
Getting from an `ArrayList`

An `ArrayList` is similar to an array in that it holds values at different indexes (of course, starting at zero). However, you don't access them using the array index operator `[]`. Instead, you call the `get()` function, which takes an `int` parameter of the index to return.

```
Circle firstCircle = circles.get(0);
```

You can use the `size()` function along with a `for` loop to loop over every object in an `ArrayList`:

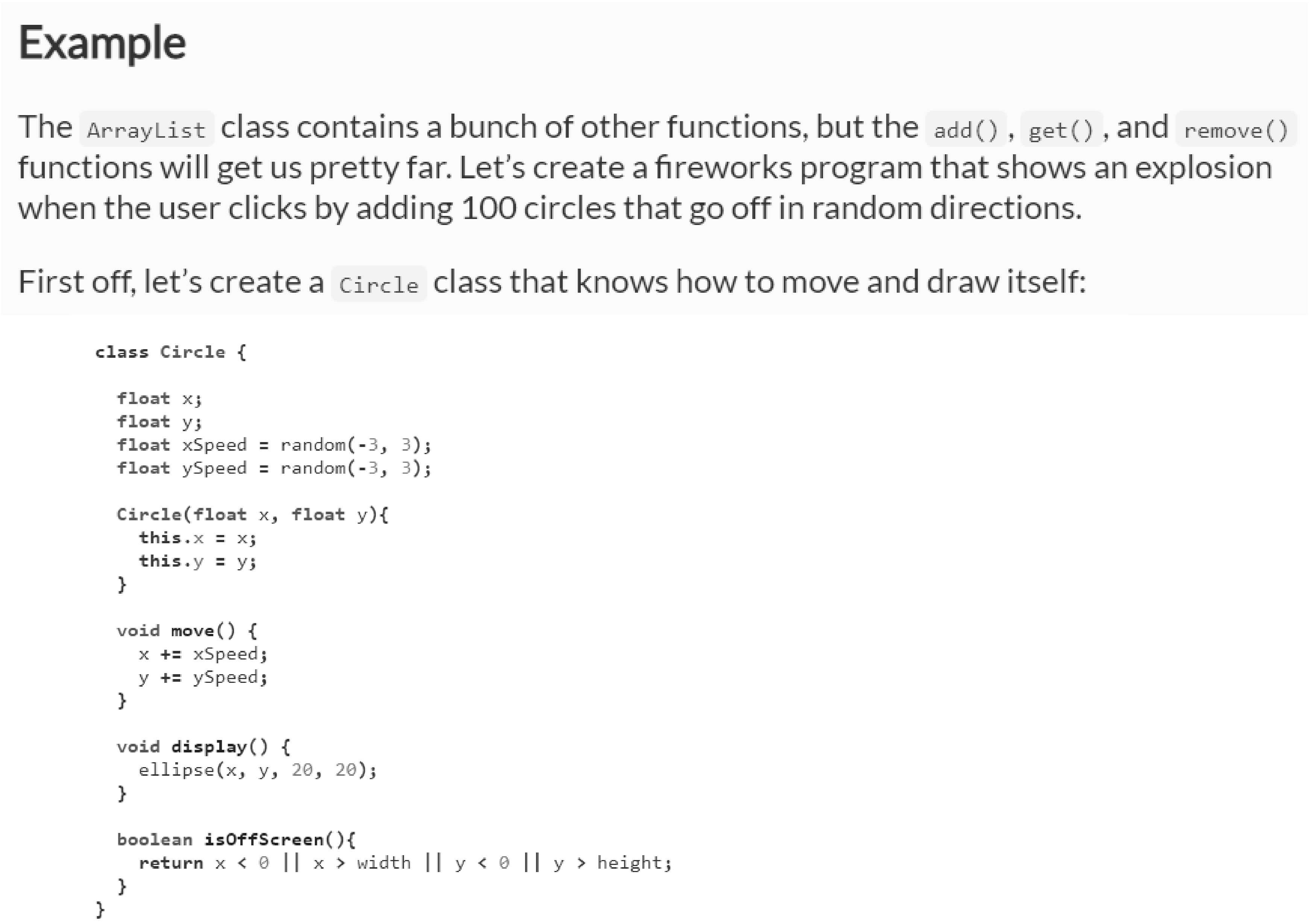
```
for(int i = 0; i < circles.size(); i++){
  circles.get(i).doSomething();
}
```



Removing from an `ArrayList`

The `remove()` function takes an `int` parameter, and removes the object at that index. It’s good to remove objects that you don’t need anymore (like when they go off-screen), otherwise your program might use up too much memory and crash.

```
for(int i = circles.size()-1; i >= 0; i--){
   if(circles.get(i).isDead()){
    circles.remove(i);
   }
}
```

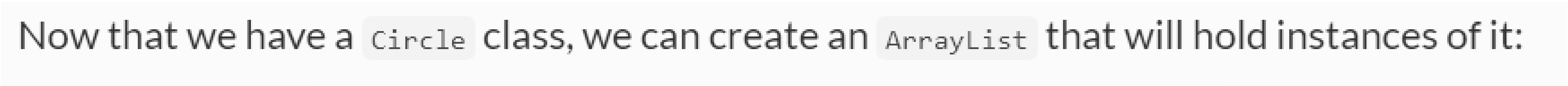


Example

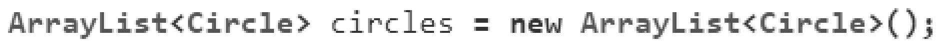
The `ArrayList` class contains a bunch of other functions, but the `add()`, `get()`, and `remove()` functions will get us pretty far. Let's create a fireworks program that shows an explosion when the user clicks by adding 100 circles that go off in random directions.

First off, let's create a `Circle` class that knows how to move and draw itself:

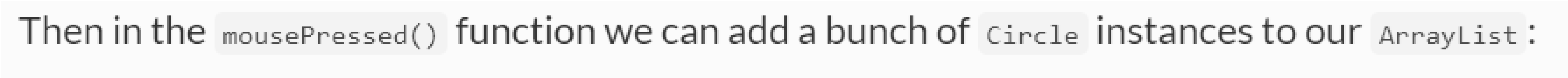
```
class Circle {

  float x;
  float y;
  float xSpeed = random(-3, 3);
  float ySpeed = random(-3, 3);

  Circle(float x, float y){
    this.x = x;
    this.y = y;
  }

  void move() {
    x += xSpeed;
    y += ySpeed;
  }

  void display() {
    ellipse(x, y, 20, 20);
  }

  boolean isOffScreen(){
    return x < 0 || x > width || y < 0 || y > height;
  }
}
```



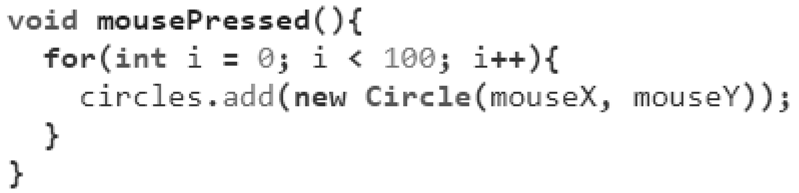
Now that we have a `Circle` class, we can create an `ArrayList` that will hold instances of it:



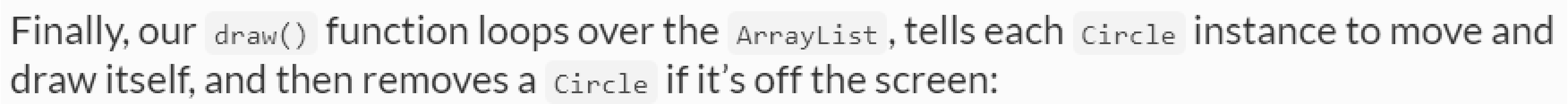
```
ArrayList<Circle> circles = new ArrayList<Circle>();
```



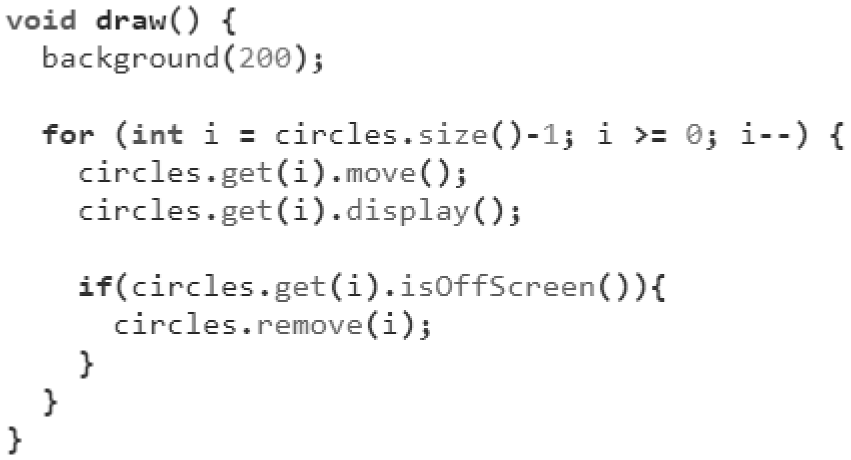
Then in the `mousePressed()` function we can add a bunch of `Circle` instances to our `ArrayList`:



```
void mousePressed(){
  for(int i = 0; i < 100; i++){
    circles.add(new Circle(mouseX, mouseY));
  }
}
```



Finally, our `draw()` function loops over the `ArrayList`, tells each `Circle` instance to move and draw itself, and then removes a `Circle` if it’s off the screen:



```
void draw() {
  background(200);

  for (int i = circles.size()-1; i >= 0; i--) {
    circles.get(i).move();
    circles.get(i).display();

    if(circles.get(i).isOffScreen()){
      circles.remove(i);
    }
  }
}
```

Class Activity:

- Combine the Codes!
- Create Array-list
  - Class Circle()

Solution:

- In Class Discussion/Lab

Home work

- Modify the fireworks program to use random colors and sizes. Make the circles fade over time.
- Assignment#2 : idea
- For example if I drew a garden scene, You might use an ArrayList of Flower instances to add flowers whenever the user clicks.

Week 7b : Next Stage in Programming

Algorithms

Look for more complex fun!

Wk4b : Putting It All Together

- Algorithms
  - Where have we been? Where are we going?
  - Dance to the Beat
  - From Idea to Parts
  - Part 1: The Catcher (game)
  - Part 2: Intersection
  - Part 3: The Timer
  - Part 4: Raindrops
  - Integration
- : Debugging
- : Libraries

Where have we been?

- Stick Bugg was our friend.
  - We learned to draw shapes from Stick bugg
  - We learned how to use variables from Stick bugg.
  - We learned interaction (mouse movement) from Stick bugg.
  - We used ifs, loops and functions, then objects (with functions), and arrays of objects and made them all move and change colors.
- But this has all been playing with the basics.

Where are we going?

- Now let’s put the things we’ve learned together into a ‘real’ program!
- As programs become more complex, we call them ‘projects’.
  - Projects require planning and often involve multiple objects and tasks
  - Start with an idea
  - Break it into Parts: Plan, code and test each one
    - Algorithm Pseudocode
    - Algorithm Code
    - Objects
  - Integrate the parts

Algorithms: Steps to code

- In computer programming, an algorithm is the sequence of steps required to perform a task
- Every single example we have created so far in the weeks involved an algorithm
- Similar to a Recipe:
  - Preheat oven to 400 degrees (F)
  - Place four boneless chicken breasts in a baking dish
  - Spread mustard evenly over chicken
  - Bake for 30 minutes

Pseudocode is used for computer programs instead of recipe steps

Problem Statement to Pseudocode

- Programs begin from a problem statement such as:
  - Sum the sequence of numbers between 1 and n
- The mathematical version of this would be:
  - SUM(n) = 1 + 2 + 3 +… + n where n is a whole number greater than 0
- Pseudocode Steps:

```
1. Set SUM to 0 and a counter to1
2. Get the value of n
3. Repeat the following steps while counter is less than or equal to n:
      a. Calculate SUM + counter and save the result in SUM
      b. Increase the value of counter by 1
4. Output the number saved in SUM
```

Pseudocode often uses words like:
'Get', 'Set', 'Repeat', 'Calculate', and 'Output'

Pseudocode: Find the variables

- Examine the pseudocode to find likely variables
- Pseudocode Steps:

Any named value that changes during the process

```
1. Set SUM to 0 and a counter to1
2. Get the value of n
3. Repeat the following steps while counter is less than or equal to n:
        a. Calculate SUM + counter and save the result in SUM
        b.  Increase the value of counter by 1
4. Output the number saved in SUM
```

Pseudocode to Code

- Pseudocode Steps:

```
1. Set SUM to 0 and a counter to1
2. Get the value of n
3. Repeat the following steps while counter is less than or equal to n:
      a. Calculate SUM + counter and save the result in SUM
      b.  Increase the value of counter by 1
4. Output the number saved in SUM
```

```
int sum = 0;
int i = 1;
int n = 10;
while (i < = n) {
  sum = sum + i;
  i++;
}
println(sum);
```

Programmers often use a single letter name for a ‘counter’ variable such as ‘i’.

From Idea to Parts

- Simple Ideas can be developed with a few steps:

  (1) developing an idea

  (2) working out an algorithm to implement that idea

  (3) writing out the code to implement that algorithm

- Some ideas are too complex to solve all at once, so we add a few more steps:

  (1) developing an idea

  (2) **breaking that idea into smaller manageable parts**

  (3) working out the algorithm **for each part**

  (4) writing the code **for each part**

  (5) **working out the algorithm for all the parts together**

  (6) **integrating the code for all of the parts together**

Rain Game Idea:

- Programs begin from a problem statement such as:
  - The object of this game is to catch raindrops before they hit the ground.
  - Every so often (depending on the level of difficulty), a new drop falls from the top of the screen at a random horizontal location with a random vertical speed.
  - The player must catch the raindrops with the mouse with the goal of not letting any raindrops reach the bottom of the screen.

Rain Game Parts:

- What are the logical parts of this idea?
  - **Part 1.** Make a circle controlled by the mouse. This circle will be the user controlled "rain catcher"
  - **Part 2.** Test if two circles intersect. This will be used to determine if the rain catcher has caught a raindrop.
  - **Part 3.** A timer that executes a function every *N* seconds. This will be used to animate raindrops 'falling' down the screen and make new raindrops.
  - **Part 4.** Make circles fall from the top of the screen to the bottom. These will be the raindrops. We'll make them look pretty at the very end.

Rain Game Parts: Still too complex?

- We want to develop an object-oriented solution. Think Objects. What are the objects and mechanisms on those objects that we will need?
- Are these parts easy or hard to do?
  - **Part 1.** “rain catcher” -- easy. Follow mouse, draw circle. See Previous week.
  - **Part 2.** “intersecting circles” – bouncing ball (refer), calc distance (refer), maybe use something similar to ‘rollover’ in wk3a… -- doable
  - **Part 3.** ‘timer’ – there must be a ‘time’ tool. Research required, but probably do-able.
  - **Part 4.** ‘falling circles’ – Array of circle objects like Array of Car objects from wk3b. -- doable

Part 1: 'The Catcher'

- Pseudocode for draw() method:
  - Erase background
  - Draw an ellipse at the mouse location
- Translating it into code is easy:

```
void setup() {
  size(400,400);
  smooth();
}

void draw() {
  background(255);
  stroke(0);
  fill(175);
  ellipse(mouseX,mouseY,64,64);
}
```

Part 1: Catcher Object

```
class Catcher {
  float r;   // radius
  float x,y; // location

  Catcher(float tempR) {
    r = tempR;
    x = 0;
    y = 0;
  }
  void setLocation(float tempX,   float tempY) {
    x = tempX;
    y = tempY;
  }
  void display() {
    stroke(0);
    fill(175);
    ellipse(x,y,r*2,r*2);
  }
}
```

- Instance vars for size, loc
- Constructor with size
- How to move it
- From draw() method

Part 2: ‘Intersection’

- Start from ‘bouncing ball’ class (Wk3a-e.g.)
- Determine if two bouncing circles intersect
  - One will be the ‘raindrop’ and one the ‘catcher’
- Plan the ‘intersect’ method
  - Will need to be part of one of the classes
  - Will need a reference to the other object to compare locations

Part 2: 'Intersection' Test Plan

- Algorithm Steps
  - Setup:
    - Create two ball objects.
  - Draw:
    - Move balls.
    - If ball #1 intersects ball #2, change color of both balls to white. Otherwise, leave color gray.
    - Display balls.
- Certainly the hard work here is the intersection test, which we will get to in a moment.

Part 2: 'Intersection' – Bouncing Ball Class

- Plan a simple bouncing "Ball" class without an intersection test (yet).
- Translating to code won't be too hard..

Data:

• *X* and *Y* location.

• Radius.

• Speed in *X* and *Y* directions.

Functions:

• Constructor

– Set radius based on argument

– Pick random location.

– Pick random speed.

• Move

– Increment *X* by speed in *X* direction.

– Increment *Y* by speed in *Y* direction.

– If Ball hits any edge, reverse direction.

• Display

– Draw a circle at *X* and *Y* location.

Part 2: ‘Intersection’ – Bounding Ball Class

Data:

- *X* and *Y* location.
- Radius.
- Speed in *X* and *Y* directions.

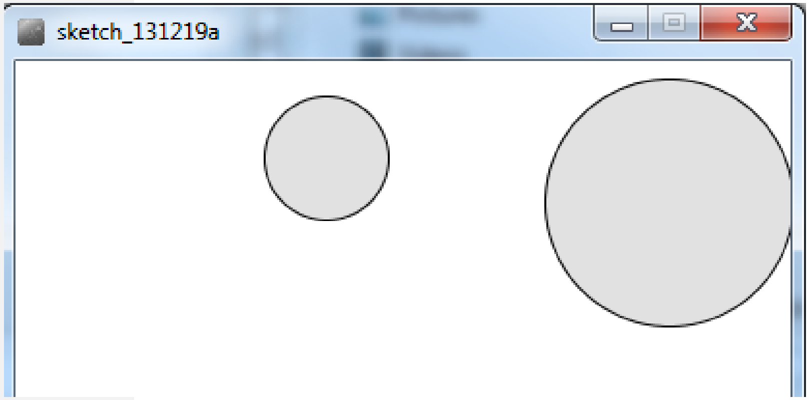
Functions:

- Constructor
  - Set radius based on argument
  - Pick random location.
  - Pick random speed.
- Move
  - Increment *X* by speed in *X* direction.
  - Increment *Y* by speed in *Y* direction.
  - If Ball hits any edge, reverse direction.
- Display
  - Draw a circle at *X* and *Y* location.

```
class Ball {
  float x,y; // location
  float r; // radius
  float xspeed,yspeed; // spds
  Ball(float tempR) {
    r = tempR;
    // set loc and speed
  }
  void move() {
    // move x and y per speeds
    // Check horizontal edges
    // Check vertical edges
  }
  void display() {
    // Same as catcher
  }
}
```

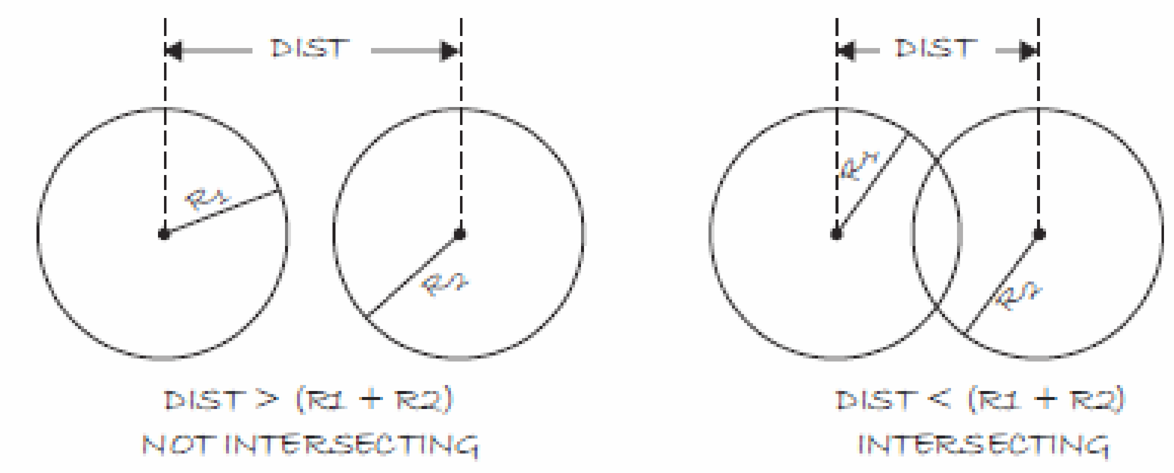
Part 2: 'Intersection' – Two Balls Objects

```
// Two ball variables
Ball ball1;
Ball ball2;
void setup() {
  size(400,400);
  smooth();
  // Instantiate balls
  ball1 = new Ball(64);
  ball2 = new Ball(32);
}
void draw() {
  background(0);
  // Move and display balls
  ball1.move();
  ball2.move();
  ball1.display();
  ball2.display();
}
```



Part 2: ‘Intersection’ Logic

- How to tell if two balls intersect
  - In Processing, we know we can calculate the distance between two points using the `dist()` function (today). We also need the radius of each circle.
  - If they are on the same horizontal plane:



  - Processing’s **`dist()`** function calculates the distance between any two points and returns a float

Part 2: ‘Intersection’ of Two Balls Objects

- The `intersect()`function (first try)
  - Needs six parameters: two x,y points, two radii
  - Returns true or false based on whether the two circles intersect
  - If distance is less than the sum of radii the circles touch

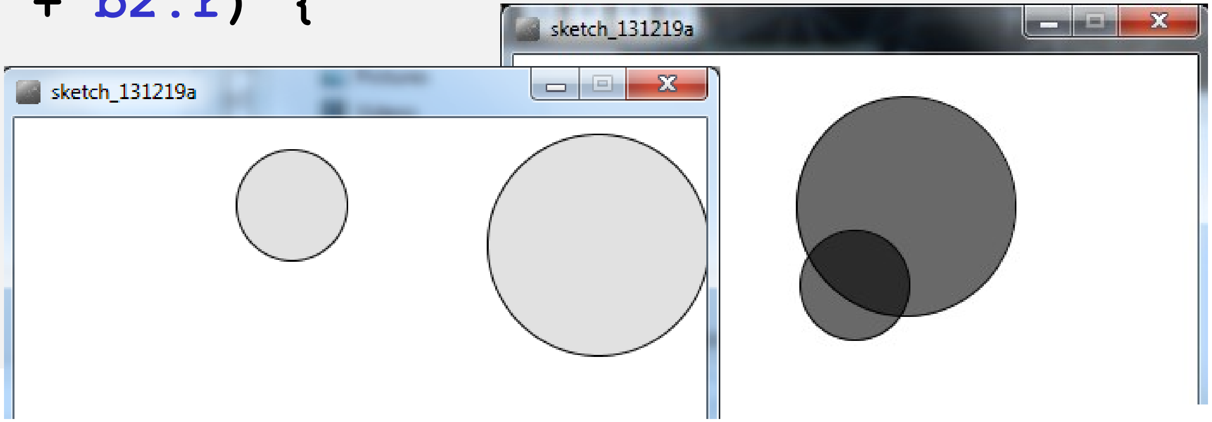
```
boolean intersect(float x1, float y1, float x2, float
y2, float r1, float r2) {
  float distance = dist(x1,y2,x2,y2); // Get distance
  if (distance < r1 + r2) { // Compare dist to r1 + r2
    return true;
  } else {
    return false;
  }
}
```

Part 2: 'Intersection': Do you intersect me?

- But wait… Do we need all of those parameters?
  - If the intersect() function is part of an object ball1, it has access to ball1 instance variables x, y and r.
  - With a reference to ball2, it can ask ball2 its x, y and r

```
if (ball1.intersect(ball2) …
```

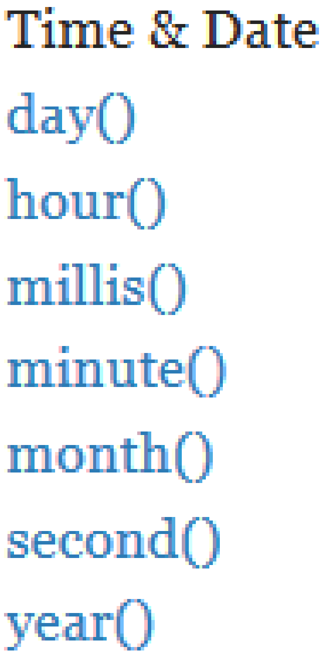
```
boolean intersect(Ball b2) {
  float distance = dist(x,y,b2.x,b2.y); // Calc Dist
  if (distance < r + b2.r) {
    return true;
  } else {
    return false;
  }
}
```



Part 3: Timer Research

- Our next task is to develop a timer that executes a function every N seconds
- We will work on this in two steps:
  1. Experiment with the tools in the main body (draw)
  2. Create a Timer class to do the work
- Research Processing Time and Date:
  - Help, Reference Time & Date shows:
  - Hmm. Let’s see how **millis()** works

```
void draw() {
  int m = millis();
  noStroke();
  fill(m % 255);
  rect(25, 25, 50, 50);
}
```



Time & Date

day()

hour()

millis()

minute()

month()

second()

year()

Part 3: Timer Planning

- Experiment:
  - Change the background color every 5 secs (5000 millis)

  Setup:
  - Save the time at startup (note this should always be zero, but it is useful to save it in a variable anyway). Call this “ savedTime ” .

  Draw:
  - Calculate the time passed as the current time
    - (i.e., millis( )) minus savedTime. Save this as “ passedTime ” .
  - If passedTime is greater than 5,000
    - Fill a new random background
    - Reset the timer by setting savedTime to the current time

Part 3: Timer Test Code

```
int savedTime;
int totalTime = 5000;
void setup() {
  size(200,200);
  background(0);
  savedTime = millis();
}
void draw() {
  // Calculate how much time has passed
  int passedTime = millis() - savedTime;
  // Has five seconds passed?
  if (passedTime > totalTime) {
    println( " 5 seconds have passed! " );
    background(random(255)); // Color a new background
    savedTime = millis();    // Reset timer
  }
}
```

Part 3: Timer Class Planning

- Let's think about what data is involved in the timer.
  - A timer must know the time when it started (savedTime) and how long it needs to run (totalTime).
- Data:
  - savedTime
  - totalTime
- The timer must be able to start as well as check and see if it is finished.
- Functions:
  - Constructor(): Pass the amount of time to run
  - start( )
  - isFinished( ): returns true or false

Part 3: Timer Class

```
class Timer {
  int savedTime; // When Timer started
  int totalTime; // How long Timer should last

  Timer(int tempTotalTime) {  // Constructor
    totalTime = tempTotalTime;
  }
  void start() {           // Start the timer
    savedTime = millis();  // Store the current time
  }
  boolean isFinished() {   // Have totalTime millis passed?
    // Check how much time has passed
    int passedTime = millis()- savedTime;
    if (passedTime > totalTime) {
      return true;
    } else {
      return false;
    }
  }
}
```

```
void draw() {
  if (timer.isFinished()) {
    background(random(255));
    timer.start();
  }
}
```

Part 4: Raindrops

- We want an array of Raindrop objects falling from the top of the window to the bottom
- We will be creating an array of moving objects
  - It is useful to approach this part as a series of even smaller steps: subparts of Part 4
  - Think of the individual elements and behaviors we will need.
- Part 4 Subparts:
  - 4.1. A single moving raindrop
  - 4.2. An array of raindrop objects
  - 4.3. Flexible number of raindrops (appearing one at a time)
  - 4.4. Fancier raindrop appearance

Part 4.1: A single moving raindrop

- What does a drop do?
  - Make a shape move downward…
    - Easy. Add 1 to the Y coordinate each time through draw
  - And make it disappear when it hits the bottom
    - Easy. Test if the y is greater than the height of the screen
- But we plan to make the drop an object
  - What data does a raindrop need?
    - Location, speed, color, size

```
class Drop {
  float x,y;   // location
  float speed; // Speed of raindrop
  color c;
  float r;     // Radius of raindrop
```

Part 4.1: Drop Class Methods

- What methods will the Drop object need?
  - Constructor
  - move()
  - display()
  - reachedBottom()

```
void move() {
  y += speed; // Increment by speed
}
void display() {
  fill(50,100,150);
  noStroke();
  ellipse(x,y,r*2,r*2);
}
boolean reachedBottom() {
  if (y > height + r*4) {
    return true;
  } else {
    return false;
  }
}
```

```
Drop() {
  r = 8; // All drops ar
  x = random(width); //
  y = -r*4; // above the
  speed = random(1,5);
  c = color(50,100,150);
}
```

Part 4.1: The Drop Test

- In setup
  - Create the drop (call the Constructor)
- In draw
  - Move the drop down one pixel
  - Display if we have not reached the bottom

```
Drop drop;
void setup() {
  size(200,200);
  ______________________________;
}
void draw() {
  background(255);
  drop.________________;
  ____________________________;
  ____________________________;
}
```

Part 4.1: The Drop Test (Answer)

- In setup
  - Create the drop (call the Constructor)
- In draw
  - Move the drop down one pixel
  - Display if we have not reached the bottom

```
Drop drop;
void setup() {
  size(200,200);
  drop = new Drop();
}
void draw() {
  background(255);
  drop.move();
  if (!drop.reachedBottom())
    drop.display();
}
```

Part 4.2: Array of Drops

- How many drops are we going to need?
  - Let’s test with 50 for now

```
Drop[] drops = new Drop[50];

void setup() {
  size(400,400);
  smooth();
  for (int i = 0; i < drops.length; i++) {
    drops[i] = new Drop();
  }
}
void draw() {
  background(255);
  for (int i = 0; i < drops.length; i++) {
    drops[i].move();
    drops[i].display();
  }
}
```

- Set them all up in a loop in setup
- Move and Display them all in a loop in draw

They all appear at the same time for now.
Spec says ‘appearing one at a time’…

Part 4.3: Flexible number of Drops Plan

- Goals:
  - Setup an array for up to 1000 drops
  - Keep track of the total drops that we have

Setup:

• Create an array of drops with 1,000 spaces in it.

• Set totalDrops = 0.

Draw:

• Create a new drop in the array (at the location totalDrops). Since totalDrops starts at 0, we will first create a new raindrop in the first spot of the array.

• Increment totalDrops (so that the next time we arrive here, we will create a drop in the next spot in the array).

• If totalDrops exceeds the array size, reset it to zero and start over.

• Move and display all available drops (i.e., totalDrops).

Part 4.3: Increasing Drops

```
Drop[] drops = new Drop[1000];
int totalDrops = 0;
void setup() {
  size(400,400);
  smooth();
  background(0);
}
void draw() {
  background(255);
  drops[totalDrops] = new Drop();
  totalDrops++;
  if (totalDrops >= drops.length) {
    totalDrops = 0;
  }

  for (int i = 0; i < totalDrops; i++) {
    drops[i].move();
    drops[i].display();
  }
}
```

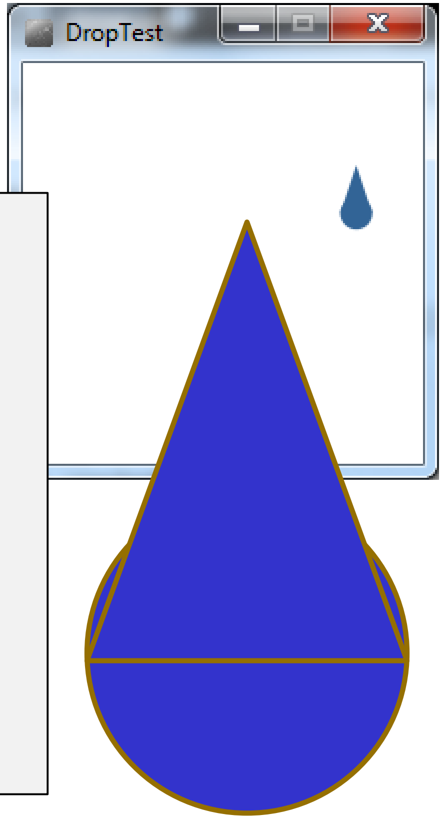
- New variable to keep track of drops
- Make a new drop each time
- Add to totalDrops
- If we hit the end, start over
- Move and display all drops

The for loop stops at totalDrops

Part 4.4: A Prettier Drop

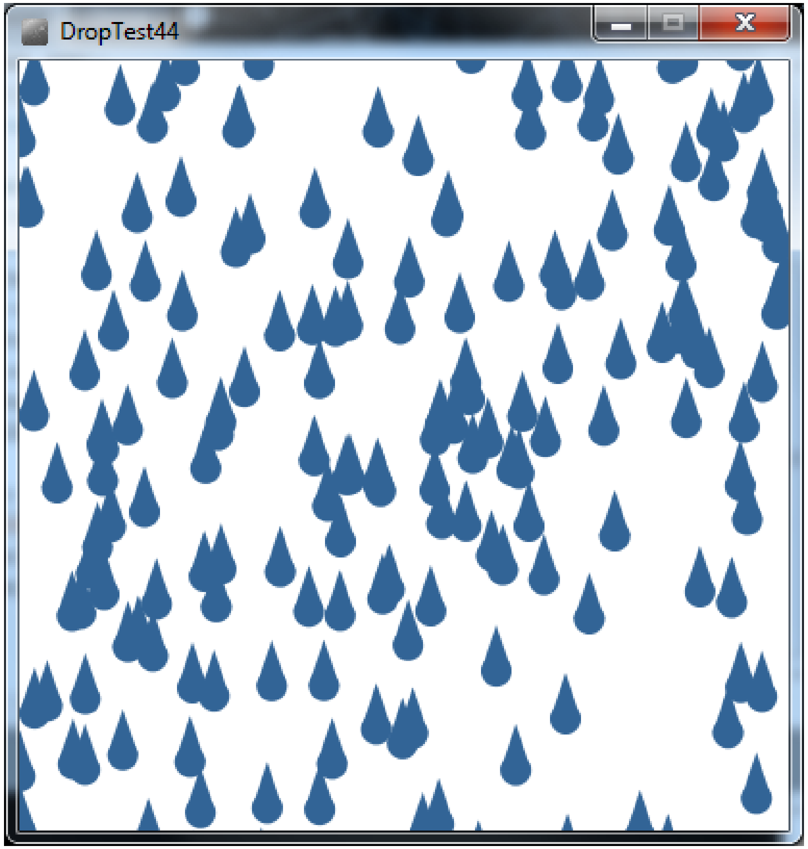
- All we will need to do is plan the beautiful drop and then change the Drop class display() function.
  - A drop knows it's 'center point' (x and y) and radius (r).
  - Draw a triangle above the ellipse
  - Then draw the ellipse as usual

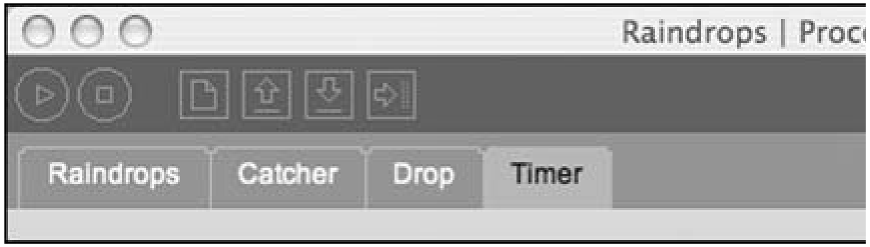
```
  void display() {
    fill(c);
    noStroke();
    // Try a triangle then an ellipse
    // (x, y is ellipse center)
    //     left edge,right edge,top
    triangle(x-r, y, x+r, y, x, y-(r*3));
    ellipse(x, y, r*2, r*2);
  }
```



Part 4.4: Big Drop Test

- Lots of pretty drops!
- Fall at different speeds!
- What's left?
  - Integrate Timer
  - Integrate Catcher
  - Make a new 'main'
  - Keep score! Win!





Summary

- Programmers need to plan projects

- Lab- Work

  (Complete today and open for discussion for next Week)

Task:

- Improve the drop-catcher Game with Levels!
- Programmers Plan Stuff!
- Part 5….
- Part 6….