CMSC 150
INTRODUCTION TO COMPUTING

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH INTRODUCTION TO PROGRAMMING IN JAVA: AN INTERDISCIPLINARY APPROACH, SEDGEWICK AND WAYNE (PEARSON ADDISON-WESLEY 2007)

LECTURE 9

- CREATING DATA TYPES

REVIEW
DATA TYPES AND OBJECT-ORIENTED PROGRAMMING

- **Data type. Object.** Set of values and operations on those values.
- **Object-oriented Programming** – design principle for large programs
  - **Composition/Abstraction** – Modeling objects (HAS-A relationship)
  - **Encapsulation** – combining data and operations (methods); data hiding from misuse (private vs public)
  - **Inheritance** – Types and sub-types (IS-A relationship)
  - **Polymorphism** – Abstract types that can act as other types (for algorithm design)
- **Instance** – A specific value/name for an object, i.e., a variable

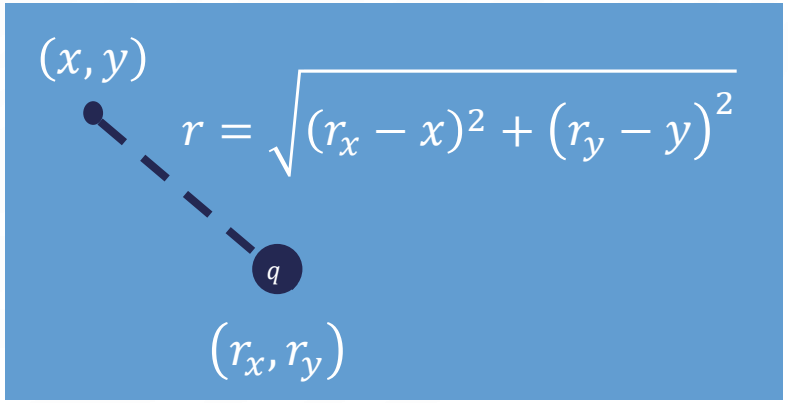
DEFINING DATA TYPES IN JAVA

- Java **class**. Defines a data type by specifying:
  - **Instance variables** – set of values
  - **Methods** – operations defined on those values
  - **Constructors** – create and initialize new objects

```
public class MyObject {
  //inside here is considered the class's scope
  //Only one public class per .java file is allowed
}
private class MyOtherObject {
  //Zero or more private classes are allowed in a file.
  //"Private" means only accessable/usable inside this
  //specific .java file
}
```

POINT CHARGE DATA TYPE

- Goal. Create a data type to manipulate point charges.
- Set of values. Three real numbers. Position (r_x, r_y) and electrical charge q
- Operations.
  - Create a new point charge at (r_x, r_y) with electric charge q
  - Determine electric potential $V = \frac{kq}{r}$ at (x, y) due to point charge
    - r – distance between (x, y) and (r_x, r_y)
    - k – electrostatic constant $= 8.99x10^9\ Nm^2/C^2$
  - Convert to string.



POINT CHARGE DATA TYPE

- API.

public class Charge		
	Charge(double x0, double y0, double q0)	
double	potentialAt(double x, double y)	*electric potential at (x, y) due to charge*
String	toString()	*string representation*

CHARGE DATA TYPE: A SIMPLE CLIENT

- Client program. Uses data type operations to calculate something.

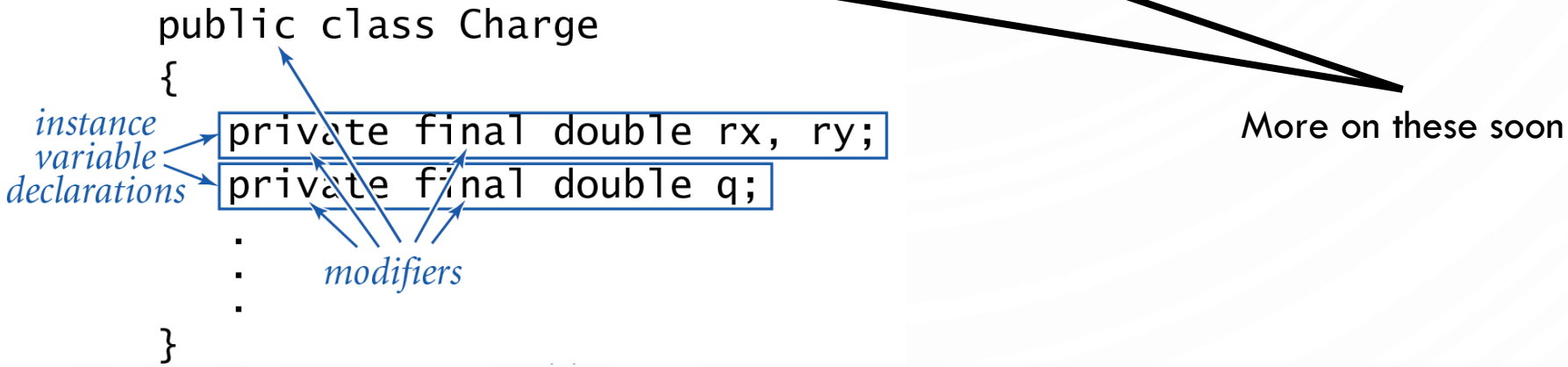
```
1. public static void main(String[] args) {
2.    double x = Double.parseDouble(args[0]);
3.    double y = Double.parseDouble(args[1]);
4.    Charge c1 = new Charge(.51, .63, 21.3);
5.    Charge c2 = new Charge(.13, .94, 81.9);
6.    double v1 = c1.potentialAt(x, y);
7.    double v2 = c2.potentialAt(x, y);
8.    StdOut.println(c1);
9.    StdOut.println(c2);
10.   StdOut.println(v1 + v2);
11.}
```

```
% java Charge .50 .50
21.3 at (0.51, 0.63)
81.9 at (0.13, 0.94)
2.74936907085912e12
```

automagically invokes the **toString()** method

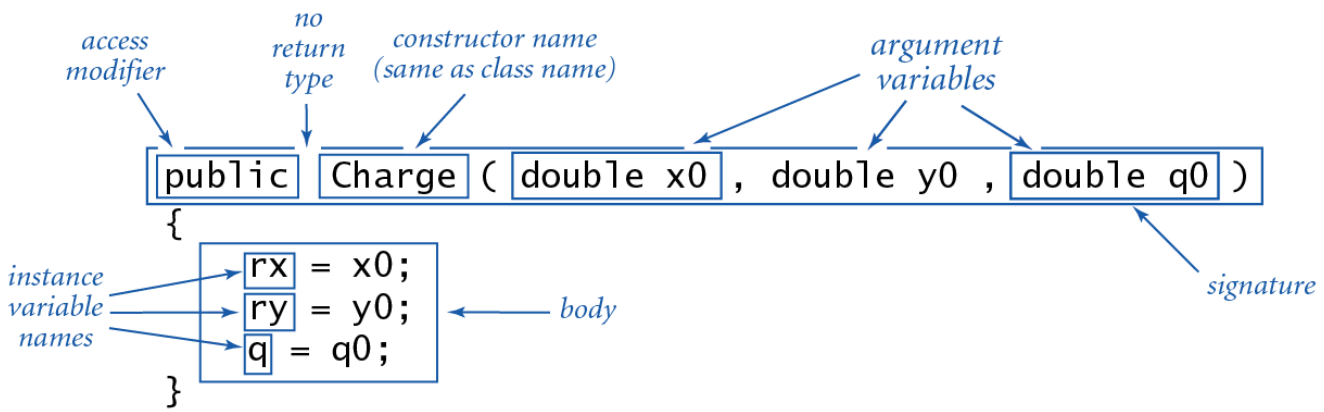
ANATOMY OF INSTANCE VARIABLES

- **Instance variables.** Specifies the set of values.
  - Declare outside any method inside the class scope
  - Always use access modifier **`private`** (unless you know better, e.g., `Math.PI`)
  - Use modifier **`final`** with instance variables that never change.

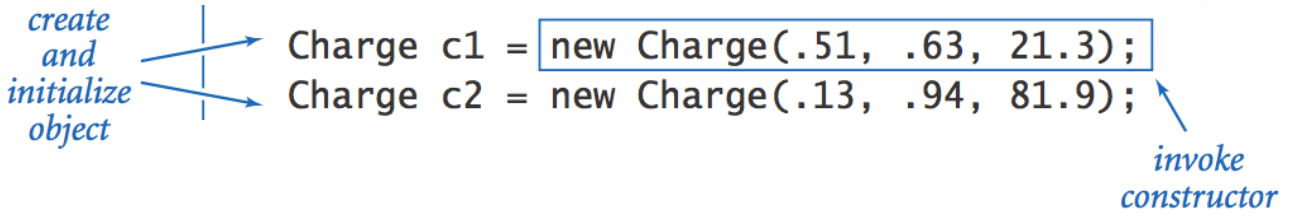


ANATOMY OF A CONSTRUCTOR

- Constructor. Specifies what happens when you create a new object

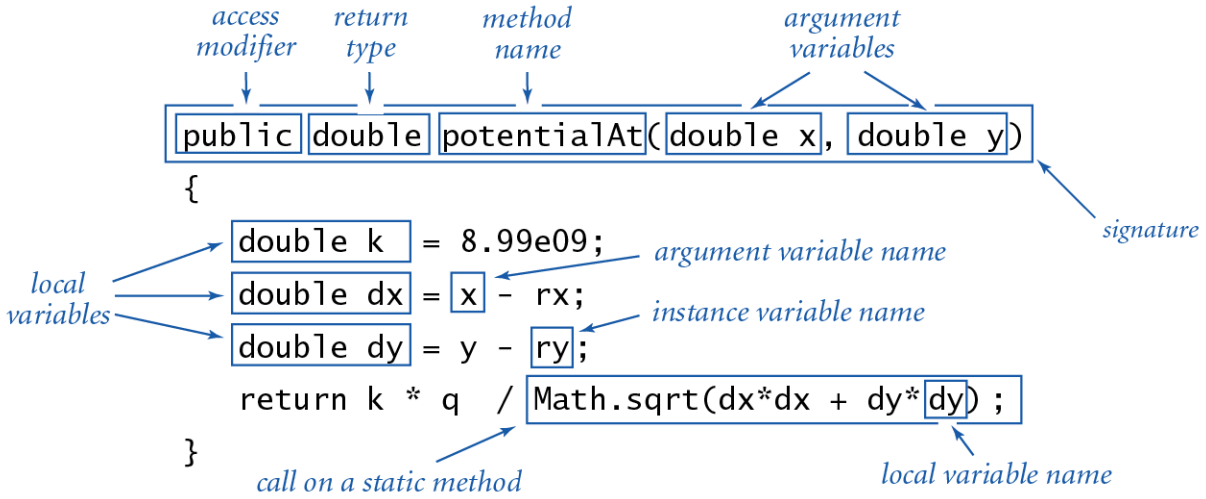


- Calling a constructor. Use new operator to create a new object.

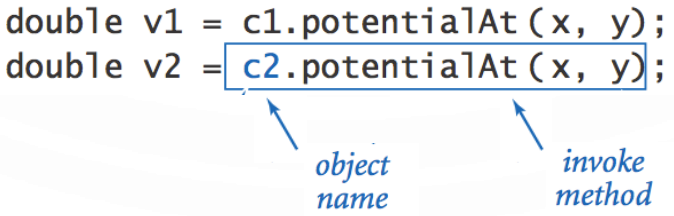


ANATOMY OF AN INSTANCE METHOD

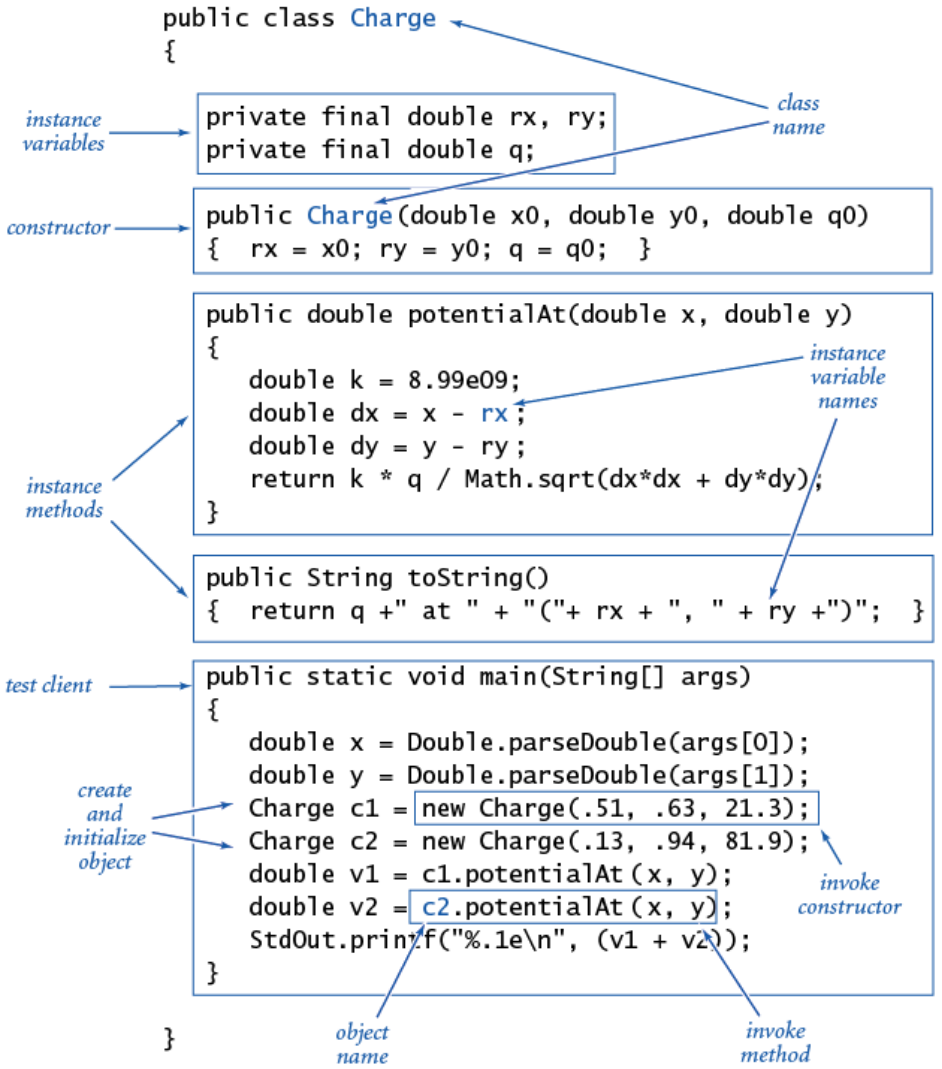
- Instance method. Define operations on instance variables.



- Invoking an instance method. Use dot operator to invoke a method.



ANATOMY OF A CLASS

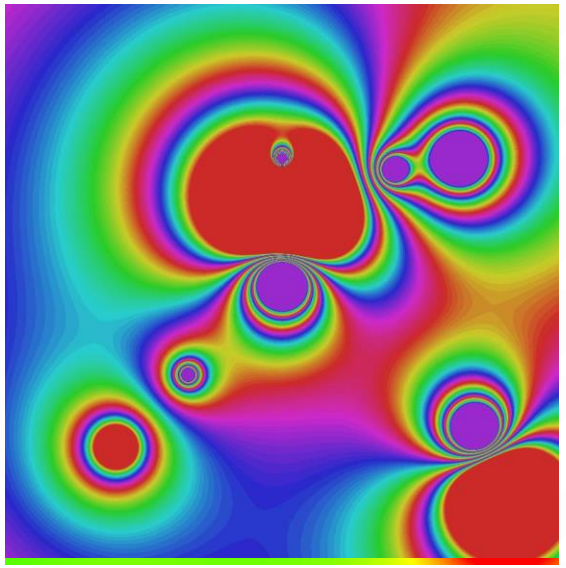


POTENTIAL VISUALIZATION

- Potential visualization. Read in N point charges from standard input; compute total potential at each point in unit square.

```
% more charges.txt
9
.51  .63 -100
.50  .50   40
.50  .72   10
.33  .33    5
.20  .20  -10
.70  .70   10
.82  .72   20
.85  .23   30
.90  .12  -50
```

```
% java Potential < charges.txt
```



POTENTIAL VISUALIZATION

- Arrays of objects. Allocate memory for the array with new; then allocate memory for each individual object with new.

```
1. // read in the data
2. int N = StdIn.readInt();
3. Charge[] a = new Charge[N];
4. for (int i = 0; i < N; i++) {
5.    double x0 = StdIn.readDouble();
6.    double y0 = StdIn.readDouble();
7.    double q0 = StdIn.readDouble();
8.    a[i] = new Charge(x0, y0, q0);
9. }
```

POTENTIAL VISUALIZATION

```
1. // plot the data
2. int SIZE = 512;
3. Picture pic = new Picture(SIZE, SIZE);
4. for (int i = 0; i < SIZE; i++) {
5.    for (int j = 0; j < SIZE; j++) {
6.       double V = 0.0;
7.       for (int k = 0; k < N; k++) {
8.          double x = 1.0 * i / SIZE;
9.          double y = 1.0 * j / SIZE;
10.         V += a[k].potentialAt(x, y);
11.      }
12.      Color color = getColor(V);   //Color is function of potential
13.      pic.set(i, SIZE-1-j, color); //Because (0,0) is upper left
14.   }
15.}
16.pic.show();
```

EXAMPLE
BOUNCING BALL

PROGRAM ALONG

EXAMPLE: BOUNCING BALL IN UNIT SQUARE

- Bouncing ball. Model a bouncing ball moving in the unit square with constant velocity.
  - Position x, y
  - Velocity x, y
  - Radius

  - Simple movement model
    - position' = position + velocity

OBJECT POINTERS – "REFERENCES"

- Object reference.
  - Allow client to manipulate an object as a single entity.
  - Essentially a machine address (pointer).

```
1. Ball b1 = new Ball();
2. b1.move();
3. b1.move();
4.
5. Ball b2 = new Ball();
6. b2.move();
7.
8. b2 = b1;
9. b2.move();
```

Address	Value
0x0	0
0x1	0
0x2	0
0x3	0
0x4	0
0x5	0
0x6	0
0x7	0
0x8	0
0x9	0
0xA	0
0xB	0
0xC	0

OBJECT POINTERS – "REFERENCES"

- Object reference.
  - Allow client to manipulate an object as a single entity.
  - Essentially a machine address (pointer).

```
1. Ball b1 = new Ball();
2. b1.move();
3. b1.move();
4.
5. Ball b2 = new Ball();
6. b2.move();
7.
8. b2 = b1;
9. b2.move();
```

Address	Value
0x0	**0.5**
0x1	**0.5**
0x2	**0.05**
0x3	**0.01**
0x4	**0.03**
0x5	0
0x6	0
0x7	0
0x8	0
0x9	0
0xA	0
0xB	0
0xC	0

OBJECT POINTERS – "REFERENCES"

- Object reference.
  - Allow client to manipulate an object as a single entity.
  - Essentially a machine address (pointer).

```
1. Ball b1 = new Ball();
2. b1.move();
3. b1.move();
4.
5. Ball b2 = new Ball();
6. b2.move();
7.
8. b2 = b1;
9. b2.move();
```

Address	Value
0x0	**0.55**
0x1	**0.51**
0x2	0.05
0x3	0.01
0x4	0.03
0x5	0
0x6	0
0x7	0
0x8	0
0x9	0
0xA	0
0xB	0
0xC	0

OBJECT POINTERS – "REFERENCES"

- Object reference.
  - Allow client to manipulate an object as a single entity.
  - Essentially a machine address (pointer).

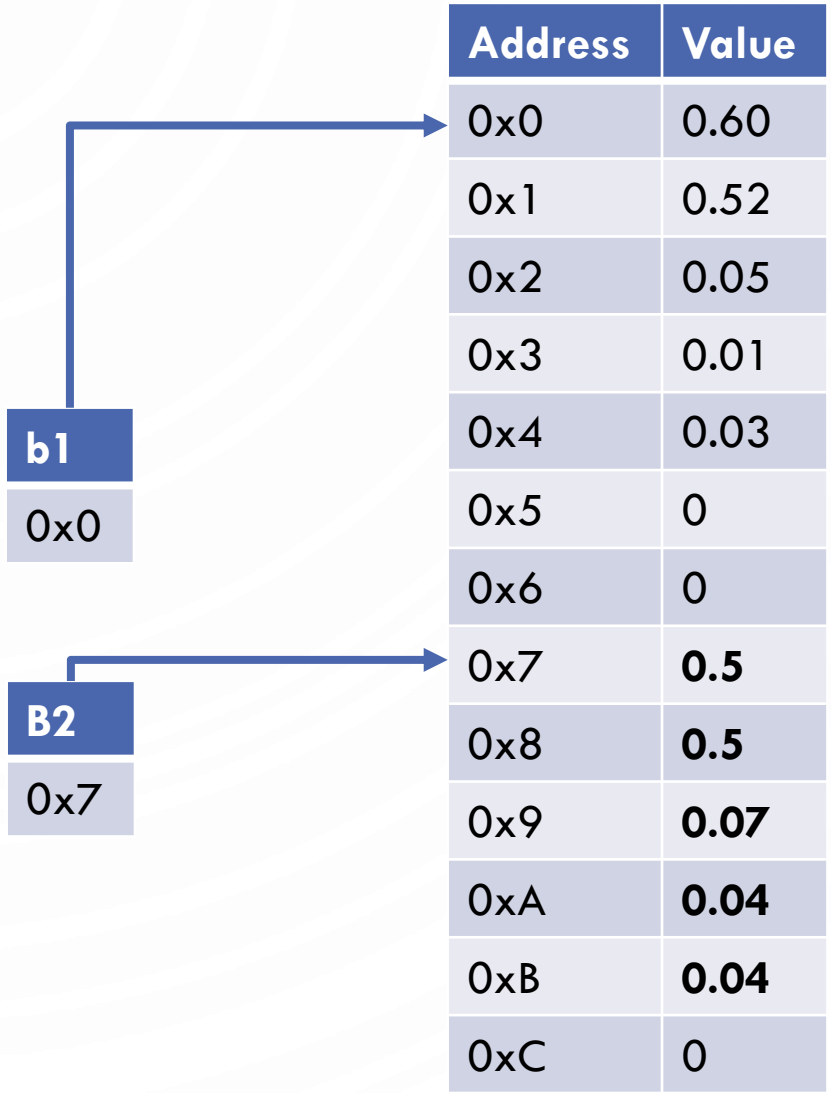
```
1. Ball b1 = new Ball();
2. b1.move();
3. b1.move();
4.
5. Ball b2 = new Ball();
6. b2.move();
7.
8. b2 = b1;
9. b2.move();
```

Address	Value
0x0	**0.60**
0x1	**0.52**
0x2	0.05
0x3	0.01
0x4	0.03
0x5	0
0x6	0
0x7	0
0x8	0
0x9	0
0xA	0
0xB	0
0xC	0

OBJECT POINTERS – "REFERENCES"

- Object reference.
  - Allow client to manipulate an object as a single entity.
  - Essentially a machine address (pointer).

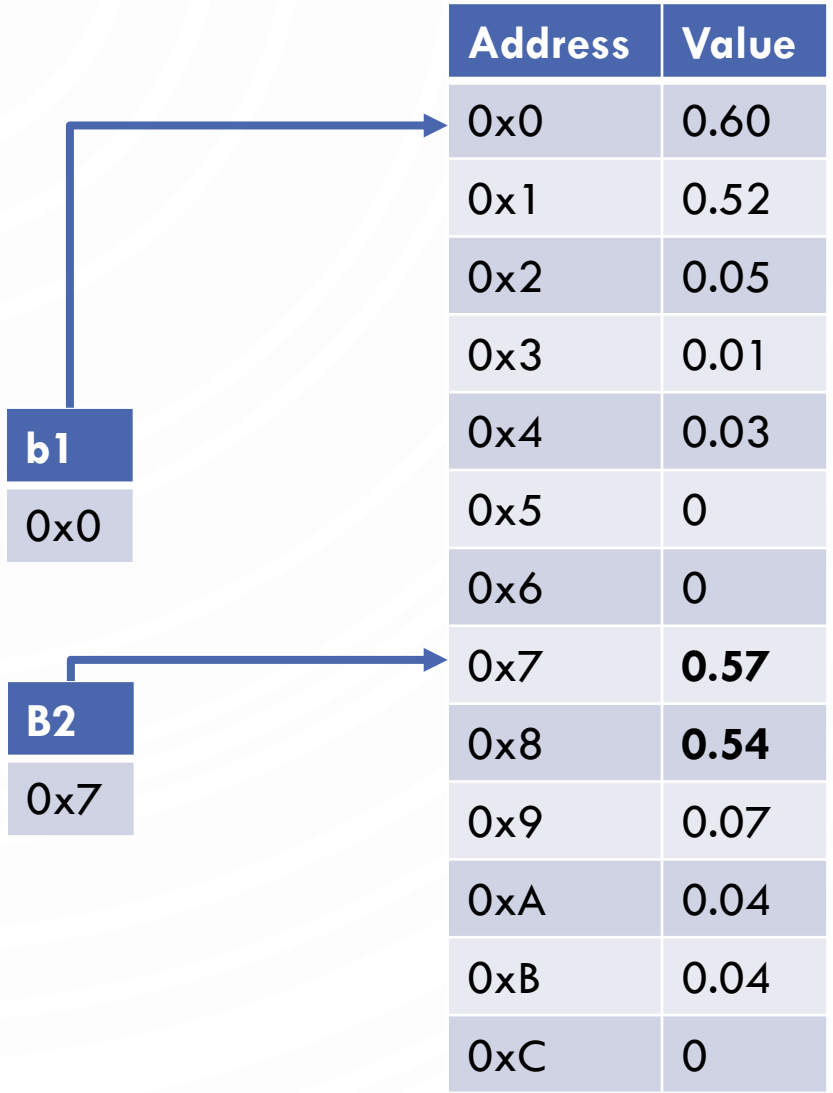
```
1. Ball b1 = new Ball();
2. b1.move();
3. b1.move();
4.
5. Ball b2 = new Ball();
6. b2.move();
7.
8. b2 = b1;
9. b2.move();
```



OBJECT POINTERS – "REFERENCES"

- Object reference.
  - Allow client to manipulate an object as a single entity.
  - Essentially a machine address (pointer).

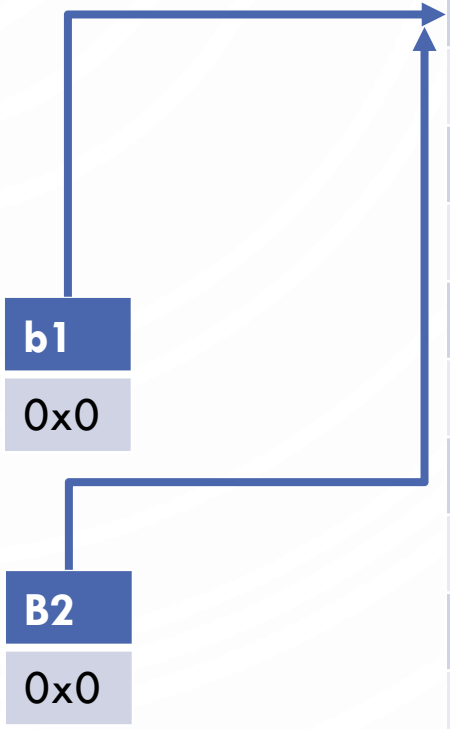
```
1. Ball b1 = new Ball();
2. b1.move();
3. b1.move();
4.
5. Ball b2 = new Ball();
6. b2.move();
7.
8. b2 = b1;
9. b2.move();
```



OBJECT POINTERS – "REFERENCES"

- Object reference.
  - Allow client to manipulate an object as a single entity.
  - Essentially a machine address (pointer).

```
1. Ball b1 = new Ball();
2. b1.move();
3. b1.move();
4.
5. Ball b2 = new Ball();
6. b2.move();
7.
8. b2 = b1;
9. b2.move();
```

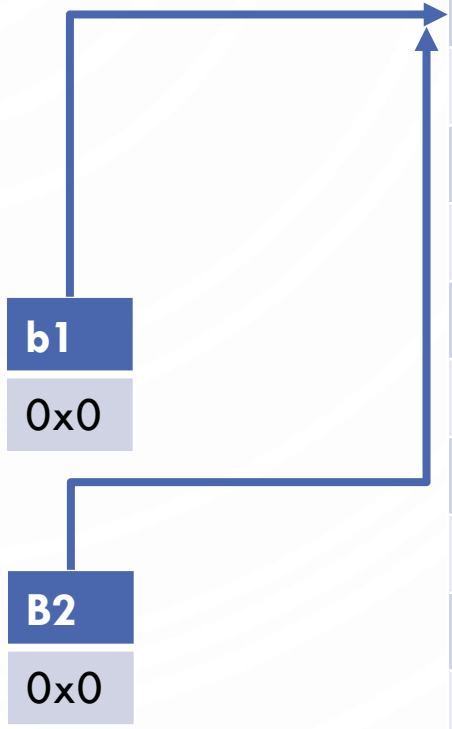


Address	Value
0x0	0.60
0x1	0.52
0x2	0.05
0x3	0.01
0x4	0.03
0x5	0
0x6	0
0x7	0.57
0x8	0.54
0x9	0.07
0xA	0.04
0xB	0.04
0xC	0

OBJECT POINTERS – "REFERENCES"

- Object reference.
  - Allow client to manipulate an object as a single entity.
  - Essentially a machine address (pointer).

```
1. Ball b1 = new Ball();
2. b1.move();
3. b1.move();
4.
5. Ball b2 = new Ball();
6. b2.move();
7.
8. b2 = b1;
9. b2.move();
```



Address	Value
0x0	**0.65**
0x1	**0.53**
0x2	0.05
0x3	0.01
0x4	0.03
0x5	0
0x6	0
0x7	0.57
0x8	0.54
0x9	0.07
0xA	0.04
0xB	0.04
0xC	0

OBJECT POINTERS – "REFERENCES"

- Some consequences.
  - Assignment statements copy references (not objects).
  - The == operator tests if two references refer to same object.
  - Pass copies of references (not objects) to functions.
    - efficient since no copying of data
    - function can change the object

EXAMPLE: BOUNCING BALL IN UNIT SQUARE

```
1.  public class Ball {
2.     private double rx, ry;
3.     private double vx, vy;
4.     private double radius;
5.
6.     public Ball() {
7.        rx = ry = 0.5;
8.        vx      = 0.015 - Math.random() * 0.03;
9.        vy      = 0.015 - Math.random() * 0.03;
10.       radius  = 0.01  + Math.random() * 0.01;
11.    }
12.
13.    public void move() {
14.       if ((rx + vx > 1.0) || (rx + vx < 0.0)) vx = -vx;
15.       if ((ry + vy > 1.0) || (ry + vy < 0.0)) vy = -vy;
16.       rx = rx + vx; ry = ry + vy;
17.    }
18.
19.    public void draw() {
20.       StdDraw.filledCircle(rx, ry, radius);
21.    }
22. }
```

CREATING MANY OBJECTS

- Each object is a data type value.
  - Use new to invoke constructor and create each one.
  - Ex: create N bouncing balls and animate them.

```
1.  public class BouncingBalls {
2.     public static void main(String[] args) {
3.        int N = Integer.parseInt(args[0]);
4.        Ball balls[] = new Ball[N];
5.        for (int i = 0; i < N; i++)
6.           balls[i] = new Ball();
7.
8.        while(true) {
9.           StdDraw.clear();
10.          for (int i = 0; i < N; i++) {
11.             balls[i].move();
12.             balls[i].draw();
13.          }
14.          StdDraw.show(20);
15.       }
16.    }
17. }
```

NEW ABSTRACTION VECTOR

- We can modify our code to create an abstraction for vector

```
public class Vector {
    private double x, y;
    public Vector(double a, double b) {
        x = a; y = b;
    }
    public Vector(Vector other) { //Note. This is a copy constructor
        x = other.x; y = other.y;
    }
    public double x() {return x;}
    public double y() {return y;}
    public Vector add(Vector other) {
        return new Vector(x + other.x, y + other.y);
    }
    public void addeq(Vector other) {
        x += other.x; y += other.y;
    }
}
```

UPDATED BALL

```
1.  public class Ball {
2.     private Vector pos; //points are vectors from the origin
3.     private Vector vel;
4.     private double radius;
5.
6.     public Ball() {
7.        pos = new Vector(0.5, 0.5);
8.        vel = new Vector(0.015 - Math.random() * 0.03, 0.015 - Math.random() * 0.03);
9.        radius  = 0.01  + Math.random() * 0.01;
10.    }
11.
12.    public void move() {
13.       if (pos.x()+vel.x() > 1.0 || pos.x()+vel.x() < 0.0) vel = new Vector(-vel.x(), vel.y());
14.       if (pos.y()+vel.y() > 1.0 || pos.y()+vel.y() < 0.0) vel = new Vector(vel.x(), -vel.y());
15.       pos.addeq(vel);
16.    }
17.
18.    public void draw() {
19.       StdDraw.filledCircle(pos.x(), pos.y(), radius);
20.    }
21. }
```

ADD GRAVITY!

- Alter velocity by an acceleration due to gravity before the position:

```
1.  public void move() {
2.     if (pos.x()+vel.x() > 1.0 || pos.x()+vel.x() < 0.0)
3.        vel = new Vector(-vel.x(), vel.y());
4.     if (pos.y()+vel.y() > 1.0 || pos.y()+vel.y() < 0.0)
5.        vel = new Vector(vel.x(), -vel.y());
6.     vel.addeq(new Vector(0.0, -0.05));
7.     pos.addeq(vel);
8. }
```

EXAMPLE
TURTLE GRAPHICS

TURTLE GRAPHICS

- Goal. Create a data type to manipulate a turtle moving in the plane.
- Set of values. Location and orientation of turtle.
- API.

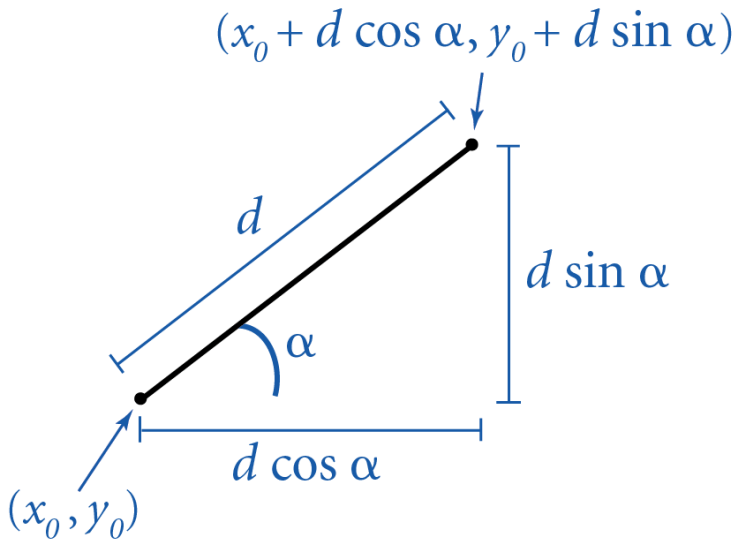
```
1. // draw a square
2. Turtle turtle =
        new Turtle(0.0, 0.0, 0.0);
3. turtle.goForward(1.0);
4. turtle.turnLeft(90.0);
5. turtle.goForward(1.0);
6. turtle.turnLeft(90.0);
7. turtle.goForward(1.0);
8. turtle.turnLeft(90.0);
9. turtle.goForward(1.0);
10.turtle.turnLeft(90.0);
```

public class Turtle

`Turtle(double x0, double y0, double a0)`	*create a new turtle at* (x_0, y_0) *facing* a_0 *degrees counterclockwise from the x-axis*
`void turnLeft(double delta)`	*rotate* `delta` *degrees counterclockwise*
`void goForward(double step)`	*move distance* `step`*, drawing a line*

TURTLE GRAPHICS

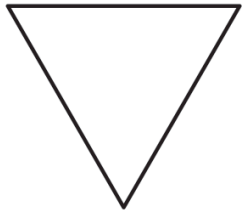
```
1.  public class Turtle {
2.     private double x, y;   // turtle is at (x, y)
3.     private double angle;  // facing this direction
4.
5.     public Turtle(double x0, double y0, double a0) {
6.        x = x0; y = y0; angle = a0;
7.     }
8.
9.     public void turnLeft(double delta) {
10.       angle += delta;
11.    }
12.
13.    public void goForward(double d) {
14.       double oldx = x, oldy = y;
15.       x += d * Math.cos(Math.toRadians(angle));
16.       y += d * Math.sin(Math.toRadians(angle));
17.       StdDraw.line(oldx, oldy, x, y);
18.    }
19. }
```



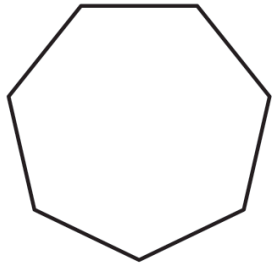
Turtle trigonometry

N-GON

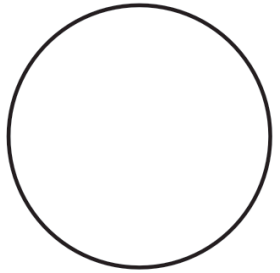
```
1. public class Ngon {
2.    public static void main(String[] args) {
3.       int N        = Integer.parseInt(args[0]);
4.       double angle = 360.0 / N;
5.       double step  = Math.sin(Math.toRadians(angle/2.0));
6.       Turtle turtle = new Turtle(0.5, 0, angle/2.0);
7.       for (int i = 0; i < N; i++) {
8.          turtle.goForward(step);
9.          turtle.turnLeft(angle);
10.       }
11.   }
12.}
```



3



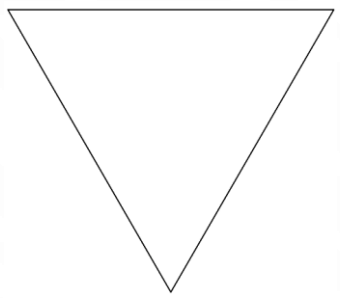
7



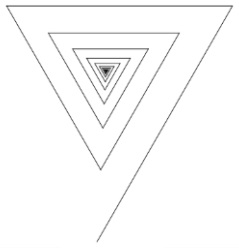
1440

SPIRAL

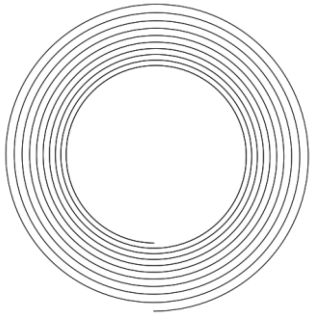
```
1.  public class Spiral {
2.     public static void main(String[] args) {
3.        int N        = Integer.parseInt(args[0]);
4.        double decay = Double.parseDouble(args[1]);
5.        double angle = 360.0 / N;
6.        double step  = Math.sin(Math.toRadians(angle/2.0));
7.        Turtle turtle = new Turtle(0.5, 0, angle/2.0);
8.        for (int i = 0; i < 10 * N; i++) {
9.           step /= decay;
10.          turtle.goForward(step);
11.          turtle.turnLeft(angle);
12.       }
13.    }
14. }
```



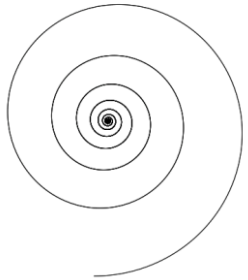
3 1.0



3 1.2



1440 1.00004



1440 1.0004

EXAMPLE
POINTS AND CIRCLES

POINTS IN THE PLANE

- Data type. Points in the plane.

```
1.  public class Point {
2.     private double x;
3.     private double y;
4.
5.     public Point() {
6.        x = Math.random();
7.        y = Math.random();
8.     }
9.
10.    public String toString() {
11.       return "(" + x + ", " + y + ")";
12.    }
13.
14.    public double distanceTo(Point p) {
15.       double dx = x - p.x;
16.       double dy = y - p.y;
17.       return Math.sqrt(dx*dx + dy*dy);
18.    }
19. }
```

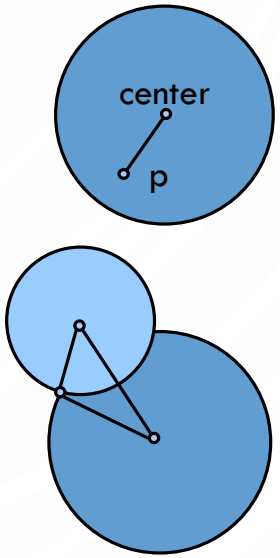
- Client. A sample client program that uses the Point data type.

```
1.  public class PointTest {
2.     public static void main(String[] args) {
3.        Point a = new Point();
4.        Point b = new Point();
5.        double distance = a.distanceTo(b);
6.        StdOut.println("a =  " + a);
7.        StdOut.println("b =  " + b);
8.        StdOut.println("distance =  " + distance);
9.     }
10. }
```

A COMPOUND DATA TYPE
CIRCLES

- Goal. Data type for circles in the plane.

```
1.  public class Circle {
2.     private Point center;
3.     private double radius;
4.
5.     public Circle(Point center, double radius) {
6.        this.center = center;  //"this" referes to "this" instance
7.        this.radius = radius;
8.     }
9.
10.    public boolean contains(Point p) {
11.       return p.dist(center) <= radius;
12.    }
13.    public double area() {
14.       return Math.PI * radius * radius;
15.    }
16.    public boolean intersects(Circle c) {
17.       return center.dist(c.center) <= radius + c.radius;
18.    }
19. }
```



IMMUTABILITY

- Immutable data type. Object's value cannot change once constructed.

mutable	*immutable*
`Picture`	`Charge`
`Histogram`	`Color`
`Turtle`	`Stopwatch`
`StockAccount`	`Complex`
`Counter`	`String`
Java arrays	primitive types

IMMUTABILITY
ADVANTAGES AND DISADVANTAGES

- Immutable data type. Object's value cannot change once constructed.

- Advantages.
  - Avoid aliasing bugs.
  - Makes program easier to debug.
  - Limits scope of code that can change values.
  - Pass objects around without worrying about modification.

- Disadvantage. New object must be created for every value.

FINAL ACCESS MODIFIER

- Final. Declaring an instance variable to be **final** means that you can assign it a value only once, in initializer or constructor.

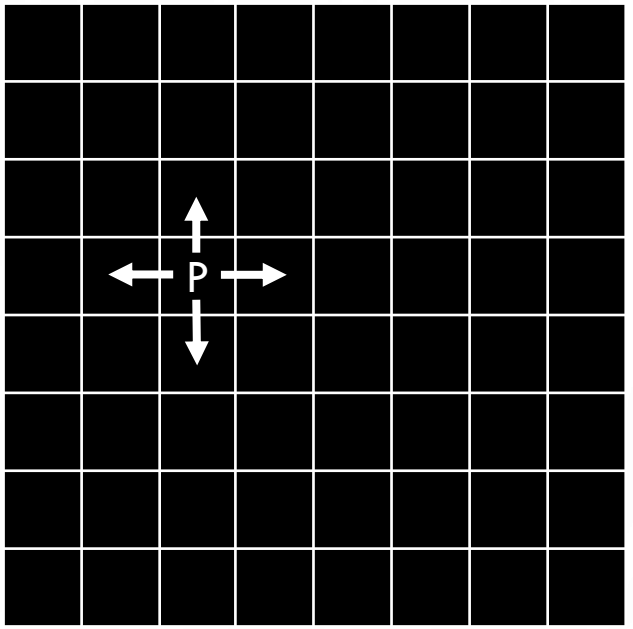
```
public class Counter {
   private final String name;
   private int count;
   ...
}
```

this value doesn't change once the object is constructed

this value changes by invoking instance method

- Advantages.
  - Helps enforce immutability.
  - Prevents accidental changes.
  - Makes program easier to debug.
  - Documents that the value cannot not change.

PRACTICE



- Grid world!
  - Create an object for a player which has an image and a position
    - Can only move in the cardinal directions
  - Create an object for a Grid world
    - Manages the players movements
    - Allow the player to enter a key (a,s,d,w) to walk within the grid
- If you finish, show me and then work on the next homework assignment.
  - You have ~1hour for this.