CHAPTER 9
OBJECTS AND CLASSES
CHAPTER 10
OBJECT-ORIENTED THINKING

MOTIVATIONS

- Suppose you want to develop a graphical user interface as shown below. How do you program it?
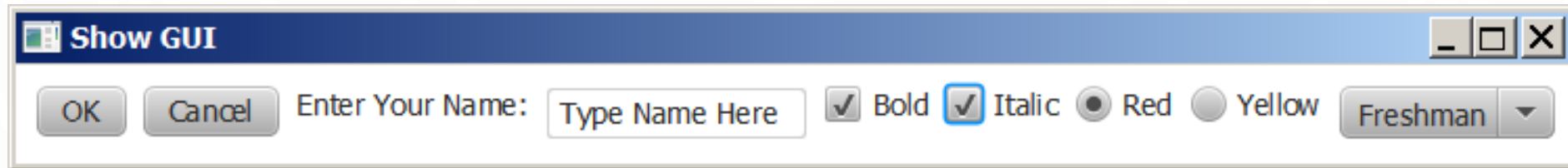
- Facebook?

- Simulation/animation for Pixar movies?

A FOUNDATION FOR PROGRAMMING

Any program you want!

Allows scaling to large programs!

Create your own data types

Objects

Functions and Modules

Arrays

Input and Output

Conditionals and Loops

Primitive data, Expressions, Math, String

WHAT ISN'T "NEW" IN "OBJECTS"?

- Some things we have seen and are familiar with, but do not fully understand the details:

  - `public class MyProgram` `//Seen this every program`
  - `String s;` `//not an integer, character, boolean, or floating-point number`
  - `Scanner in = new Scanner(System.in);` `//Making variables of complex types`
  - `s.charAt(5);` `//using methods tied to a variable's value`

DATA TYPES

- **Data type.** Set of values and operations on those values.

- **Primitive types.** Values directly map to machine representation; operations directly map to machine instructions.

Data Type	Set of Values	Operations
`boolean`	`true`, `false`	not, and, or, xor
`int`	$[-2^{31}, 2^{31})$	add, subtract, multiply
`double`	any of 2^64 real numbers	add, subtract, multiply

- We want to write programs that process other types of data.

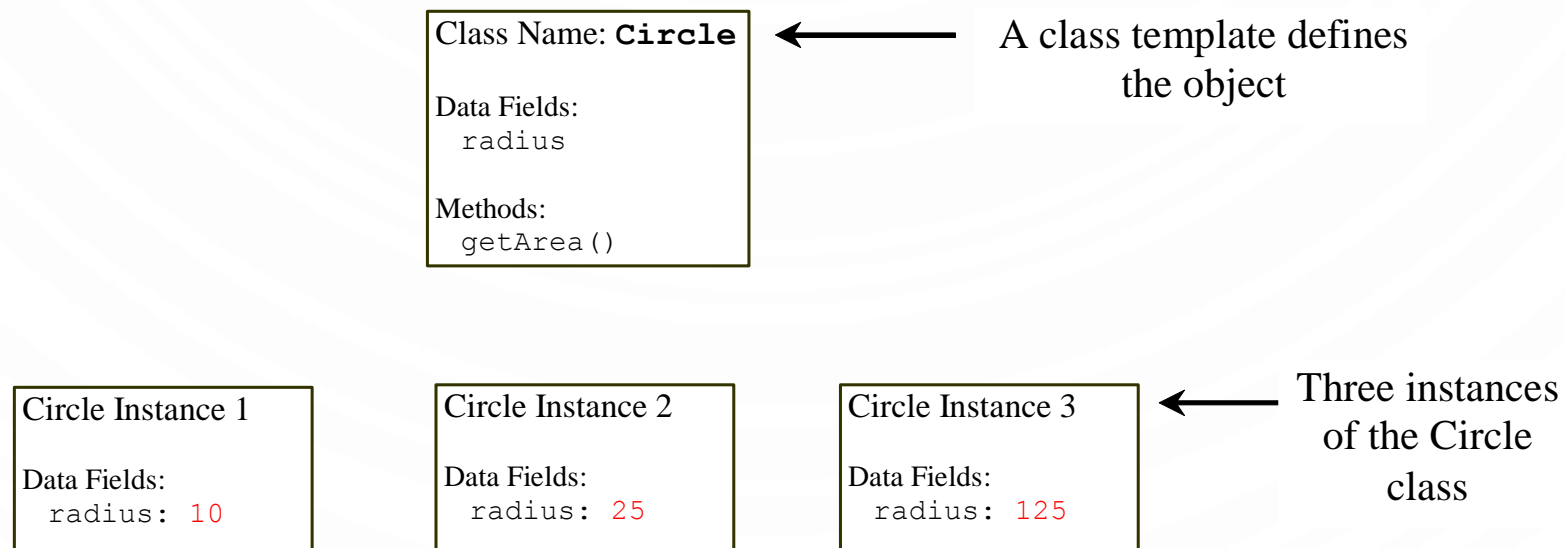 - Colors, pictures, strings, vectors, polygons, input streams, …

OBJECT-ORIENTED PROGRAMMING CONCEPTS

- **Object-oriented programming (OOP)** involves programming using objects

- An **object** represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behaviors.

    - The **state** of an object consists of a set of **data fields** (also known as **properties**) with their current values.

    - The **behavior** of an object is defined by a set of methods.

Data Type	Set of Values	Operations
`Color`	24 bits	`getRed(), brighten()`
`Picture`	2D array of **Color**s	`getPixel(i, j), setPixel(i, j)`
`String`	Sequence of characters	`length(), substring(), compare()`

OBJECTS

- An **object** has both a state and behavior. The state defines the object, and the behavior defines what the object does.
  - An object **class** defines its possible states and its behaviors
  - An object **instance** is a variable of the object type, i.e., it is a specific "value" or state

Class Name: **Circle** ← A class template defines the object

Data Fields:
 radius

Methods:
 getArea()

Circle Instance 1

Data Fields:
 radius: 10

Circle Instance 2

Data Fields:
 radius: 25

Circle Instance 3 ← Three instances of the Circle class

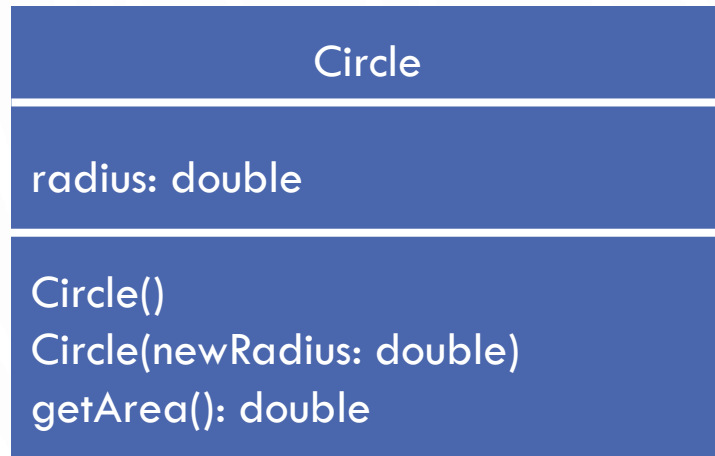Data Fields:
 radius: 125

CLASSES

- **Classes** are constructs that define objects of the same type

- A Java class uses

  - **Variables** to define data fields

  - **Methods** to define behaviors

  - A special type of methods, known as **constructors**, which are invoked to construct **instances** (objects) from the class
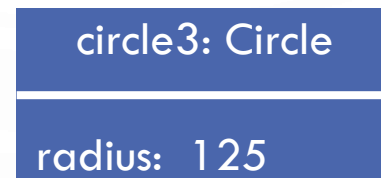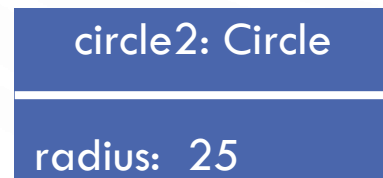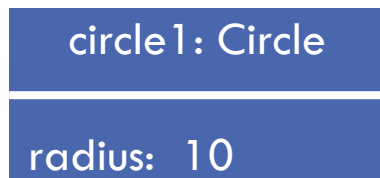
UML CLASS DIAGRAM

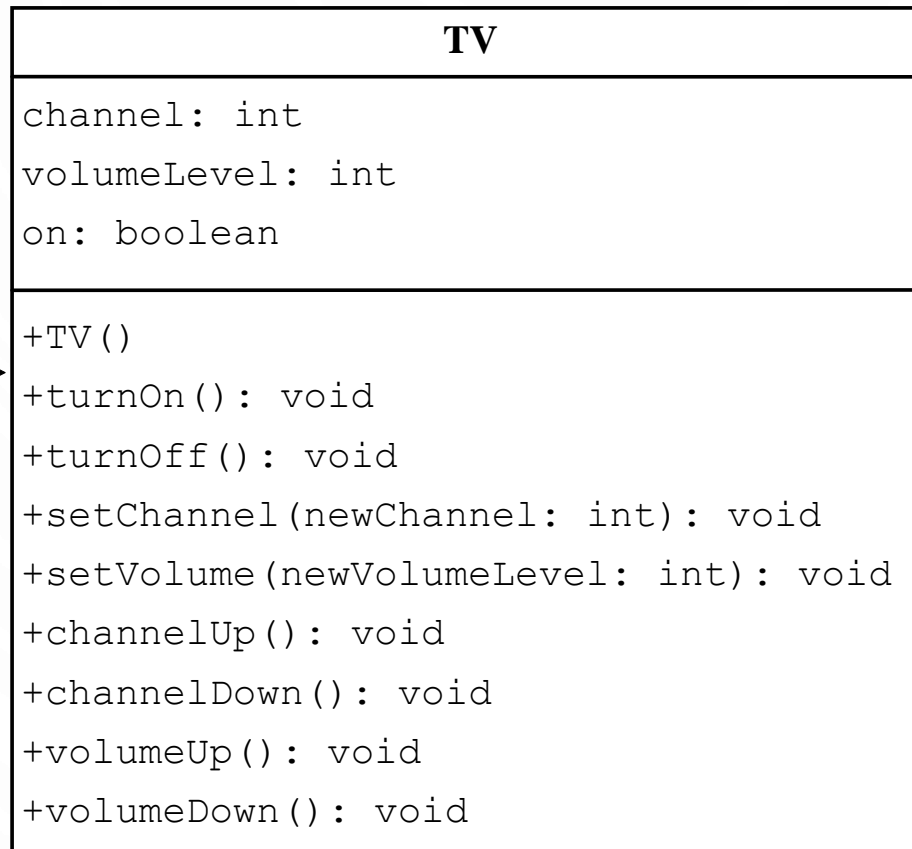| Circle |
| --- |
| radius: double |
| Circle()
Circle(newRadius: double)
getArea(): double |

← Class name

← Data fields

← Constructors and methods

circle1: Circle
radius: 10

circle2: Circle
radius: 25

circle3: Circle
radius: 125

← UML notation for instances (objects)

EXAMPLE UML DIAGRAM
DEFINING A TV OBJECT

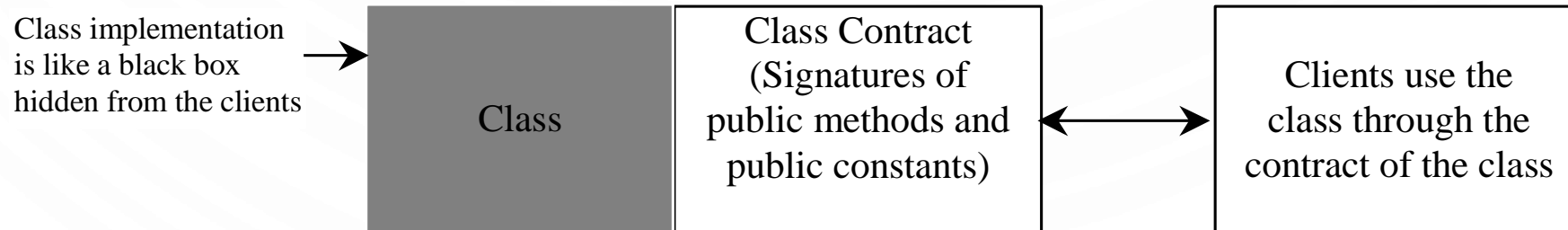| TV |
| --- |
| channel: int |
| volumeLevel: int |
| on: boolean |

The current channel (1 to 120) of this TV.
The current volume level (1 to 7) of this TV.
Indicates whether this TV is on/off.

The + sign indicates a public modifier. ⟶

```
+TV()
+turnOn(): void
+turnOff(): void
+setChannel(newChannel: int): void
+setVolume(newVolumeLevel: int): void
+channelUp(): void
+channelDown(): void
+volumeUp(): void
+volumeDown(): void
```

Constructs a default TV object.
Turns on this TV.
Turns off this TV.
Sets a new channel for this TV.
Sets a new volume level for this TV.
Increases the channel number by 1.
Decreases the channel number by 1.
Increases the volume level by 1.
Decreases the volume level by 1.

OBJECT-ORIENTED PROGRAMMING

- **Object-oriented Programming** – design principle for large programs
  - **Abstraction** – Modeling objects
  - **Composition** – Modeling object associations (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)

ABSTRACTION AND ENCAPSULATION

- **Abstraction** means to separate class implementation from the use of the class.
  - A description of the class lets the user know how the class can be used (class **contract**)
  - Thus, the user of the class does not need to know how the class is implemented
  - The detail of implementation is **encapsulated** and hidden from the user.

Class implementation is like a black box hidden from the clients → | Class | Class Contract (Signatures of public methods and public constants) | ↔ | Clients use the class through the contract of the class |

OBJECT COMPOSITION

- **Composition/Aggregation** models **has-a relationships** and represents an ownership relationship between two objects
  - The owner object is called an aggregating object and its class an aggregating class. The subject object is called an aggregated object and its class an aggregated class.
 - Typically represented as a data field in the aggregating object
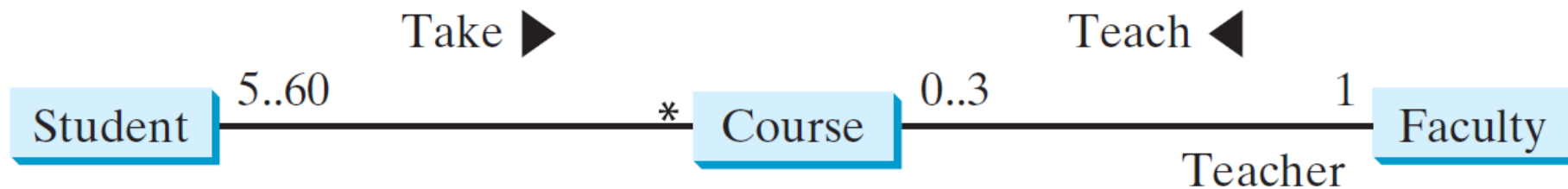
AGGREGATION OR COMPOSITION

- Many texts don't differentiate between the two, calling them both compositions – the idea of an object owning another object

- However, the technical difference is:
  - **Composition** – a relationship where the owned object cannot exist independent of the owner
  - **Aggregation** – a relationship where the owned object can exist independent of the owner

Association

Class A ——————————> Class B

Aggregation

Class A ◇——————— Class B

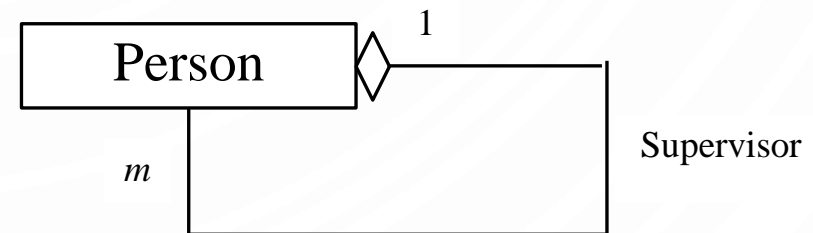Composition

Class A ◆——————— Class B
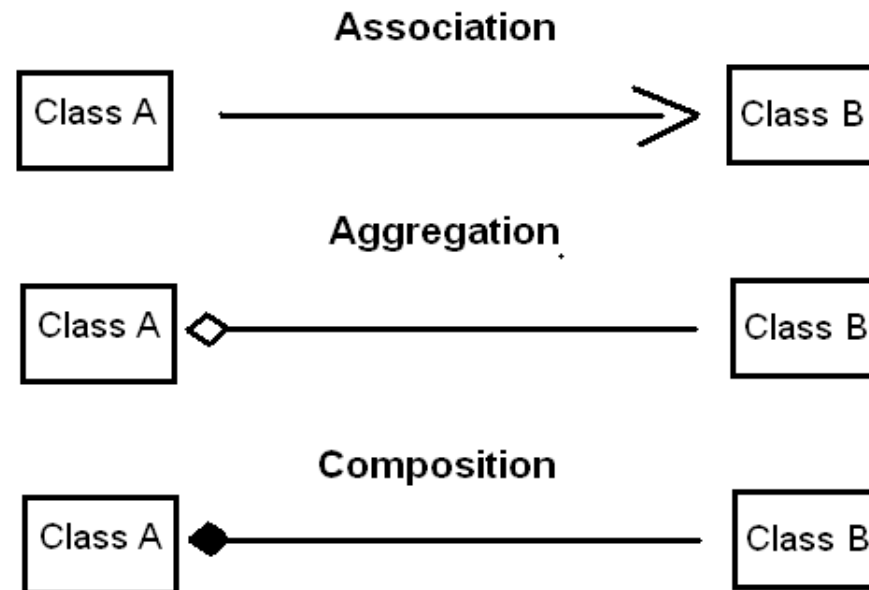
AGGREGATION BETWEEN SAME CLASS

- Aggregation may exist between objects of the same class. For example, a person may have a supervisor.
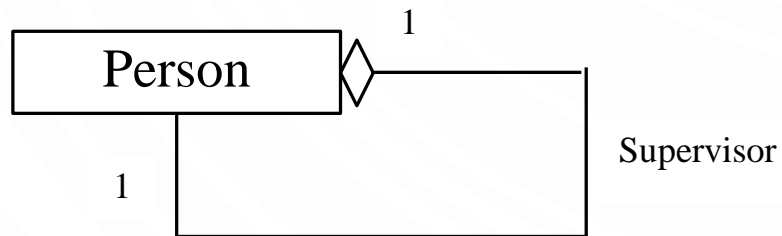
Aggregation of a single
Person owning a person

Aggregation of a single
Person owning multiple
persons

EXAMPLE
THE COURSE CLASS

Course
-courseName: String
-students: Student[]
-numberOfStudents: int
+Course(courseName: String)
+getCourseName(): String
+addStudent(student: Student): void
+dropStudent(student: Student): void
+getStudents(): Student[]
+getNumberOfStudents(): int

The name of the course.

An array to store the students for the course.

The number of students (default: 0).

Creates a course with the specified name.

Returns the course name.

Adds a new student to the course.
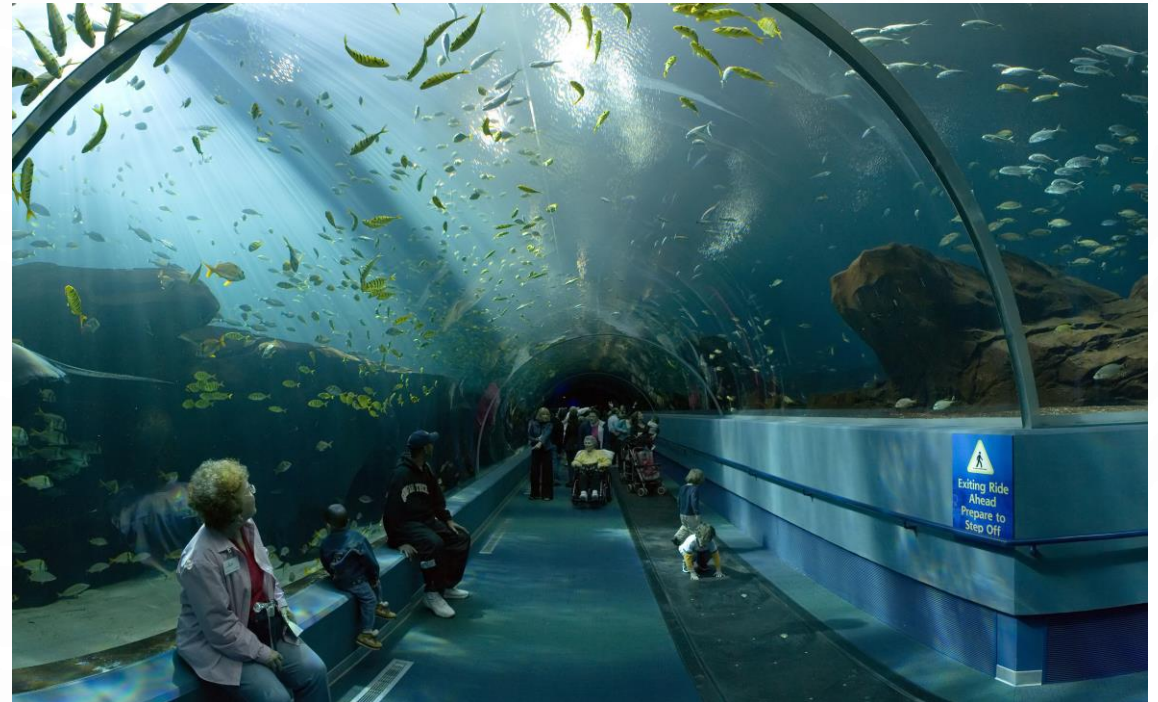
Drops a student from the course.

Returns the students in the course.

Returns the number of students in the course.

PRACTICE

- Describe objects (data and functions) for an Aquarium
  - Be descriptive
  - Objects can contain other objects!
  - Objects interact with other objects!

EXERCISE

- Describe objects (data and functions) for the world of Harry Potter
    - Be descriptive
    - Objects can contain other objects!
    - Objects interact with other objects!

CLASSES

```java
1.   public class Circle {
2.       /** Radius of the circle */
3.       private double radius = 1.0;
4.       /** Default construct a circle of radius 1 */
5.       public Circle() {
6.       }
7.       /** Contruct a circle of desired radius */
8.       public Circle(double r) {
9.         radius = r;
10.      }
11.      /** Compute the area of the circle */
12.      public double getArea() {
13.        return radius*radius*Math.PI;
14.      }
15. }
```

Data fields

Constructors

Methods

USING CLASSES

1. `Circle c1 = new Circle();` //declare and instantiate a circle
   //with the default constructor

2. `Circle c2 = new Circle(5);` //declare and instantiate a circle
   //with radius 5

3. `System.out.println(c2.getArea());` //Use the circle

4. `//System.out.println(c1.radius);` //Compiler error!
   //Cannot access radius.

DEFINING CLASSES

- Fields, methods, and constructors can appear in any order. However, to make life easy, use the following template:

```java
public class ClassName {
   /** First, place all public data fields
       (usually static, this is rare) */
   /** Next, place all private fields here */
   /** Next, define a default constructor
       followed by non-default constructors */
   /** Next, define all public methods */
   /** Next, define all private methods
       (helpers */
   /** Last, define all static methods */
}
```

- Example:

```java
public class Color {
   private int red   = 0,
               green = 0,
               blue  = 0;

   public Color() {}
   public Color(int r, int g, int b) {
      red = r; green = g; blue = b;
   }

   public int getRed()   {return red;}
   public int getGreen() {return green;}
   public int getBlue()  {return blue;}
}
```

CONSTRUCTORS

- **Constructors** are a special kind of methods that are invoked to construct objects.
- This is where you describe how memory for an object is **initialized**

```
Circle() {
}

Circle(double newRadius) {
    radius = newRadius;
}
```

CONSTRUCTORS

- A constructor with no parameters is referred to as a **no-arg constructor**

- Constructors must have the same name as the class itself

- Constructors do not have a return type, not even void

- Constructors are invoked using the **new** operator when an object is created. Constructors play the role of initializing objects.

  - `new ClassName();`

  - Example: `new Circle(2.3);`

DEFAULT CONSTRUCTOR

- A class may be defined without constructors

- In this case, a no-arg constructor with an empty body is implicitly defined in the class.

- This constructor, called a **default constructor**, is provided automatically only if no constructors are explicitly defined in the class.

DECLARING OBJECT REFERENCE VARIABLES

- To reference an object, assign the object to a **reference variable** (we saw the same when discussing arrays)

- To declare a reference variable, use the syntax:
  ClassName `objectRefVar;`

- Example:
  Circle `myCircle;`

- Like everything else, you may declare and initialize (create) in the same step
  ClassName `objectRefVar =` **new ClassName**`();`

- Example:

- **Circle** `myCircle =` **new Circle**`(5);`

ACCESSING OBJECT'S MEMBERS

- Referencing the object's data:
  - `objectRefVar.data`
  - e.g., `myCircle.radius`

- Invoking the object's method:
  - `objectRefVar.methodName(arguments)`
  - e.g., `myCircle.getArea()`

TRACING

1. `Circle myCircle = new Circle(5.0);`
2. `Circle yourCircle = new Circle();`
3. `yourCircle.radius = 100;`

Memory

TRACING

```
1. Circle myCircle = new Circle(5.0);
2. Circle yourCircle = new Circle();
3. yourCircle.radius = 100;
```

Declare myCircle

Memory

```
myCircle
null
```

TRACING

1. **`Circle`** `myCircle =` **`new Circle`**`(5.0);`
2. **`Circle`** `yourCircle =` **`new Circle`**`();`
3. `yourCircle.radius =` `100;`

Create a circle

Memory

```
myCircle
null

0xA
```

Circle	
radius	5

TRACING

1. `Circle` `myCircle =` `new` `Circle`(`5.0`);
2. `Circle` `yourCircle =` `new` `Circle`();
3. `yourCircle.radius =` `100`;

Assign memory location to reference variable

Memory

```
myCircle
0xA (reference)

0xA
```

Circle	
radius	5

TRACING

1. `Circle myCircle = new Circle(5.0);`
2. `Circle yourCircle = new Circle();`
3. `yourCircle.radius = 100;`

Declare yourCircle

Memory

```
myCircle                yourCircle
0xA (reference)         null


0xA
```

Circle	
radius	5

TRACING

1. **Circle** myCircle = **new Circle**(5.0);
2. **Circle** yourCircle = **new Circle**();
3. yourCircle.radius = 100;

Create a circle

Memory

myCircle   yourCircle
0xA (reference)   null

0xA   0xB

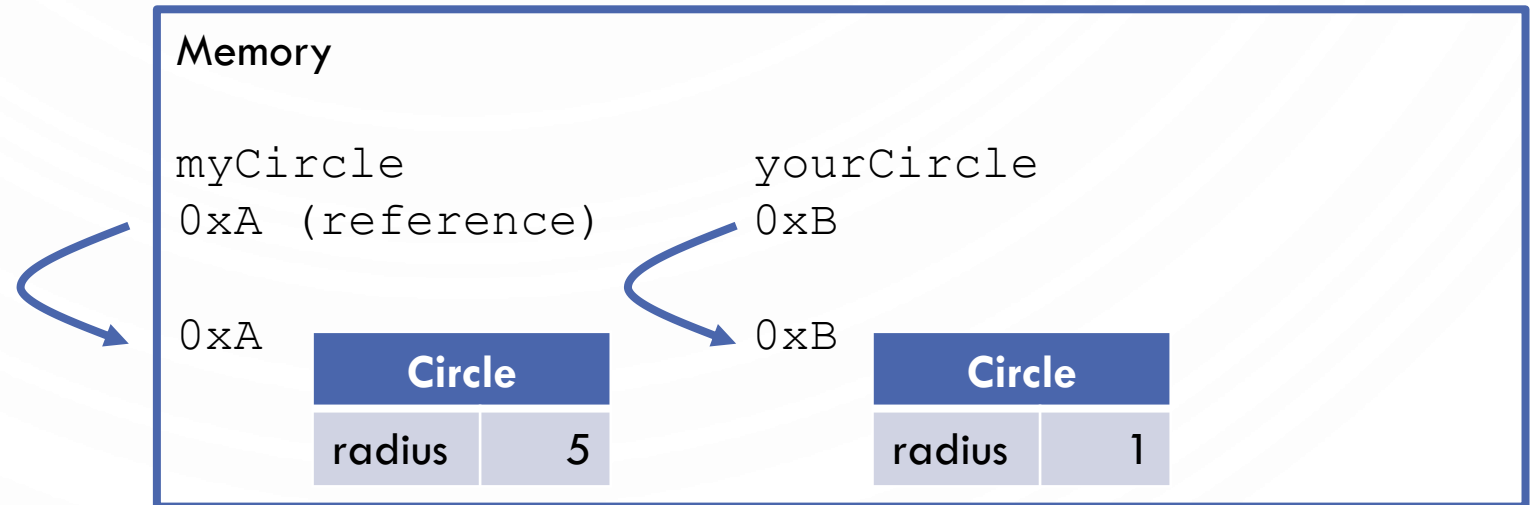| Circle	
radius	5

Circle	
radius	1

TRACING

1. **Circle** myCircle = **new Circle**(5.0);
2. **Circle** yourCircle = **new Circle**();
3. yourCircle.radius = 100;

Assign memory location to reference variable

Memory

myCircle
0xA (reference)

yourCircle
0xB

0xA

Circle	
radius	5

0xB

Circle	
radius	1

TRACING

1. `Circle myCircle = new Circle(5.0);`
2. `Circle yourCircle = new Circle();`
3. `yourCircle.radius = 100;`

Change radius in your circle (code assumes radius is public)

Memory

myCircle 0xA (reference)	yourCircle 0xB

0xA

Circle	
radius	5

0xB

Circle	
radius	100

CAUTION

- Recall that you use the following to invoke a method in the Math class
  - `Math.methodName(arguments)` (e.g., `Math.pow(3, 2.5)`)

- Can you invoke `getArea()` using `Circle.getArea()`?
  - The answer is no. All the methods used before this chapter are **static** methods, which are defined using the **static** keyword. However, `getArea()` is non-static. It must be invoked from an object using
  - `objectRefVar.methodName(arguments)` (e.g., `myCircle.getArea()`).

- More on this later

DATA FIELDS

- Unlike local variables, data fields are initialized with default values (if nothing else is specified)

- The data fields can be of reference types. For example, the following Student class contains a data field name of the String type.

- What would this look like in memory?

```
1.  public class Student {
2.      String name; //default value null
3.      int age;        //default value 0
4.      boolean isScienceMajor;
                        //default value false
5.      char gender; //default value '\u0000'
6.  }
```

THE NULL VALUE

- When a reference variable does not reference any object, the data field holds a special literal value, `null`.

- Null is typically the literal memory address 0x0 or integer value 0

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

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. }
```

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
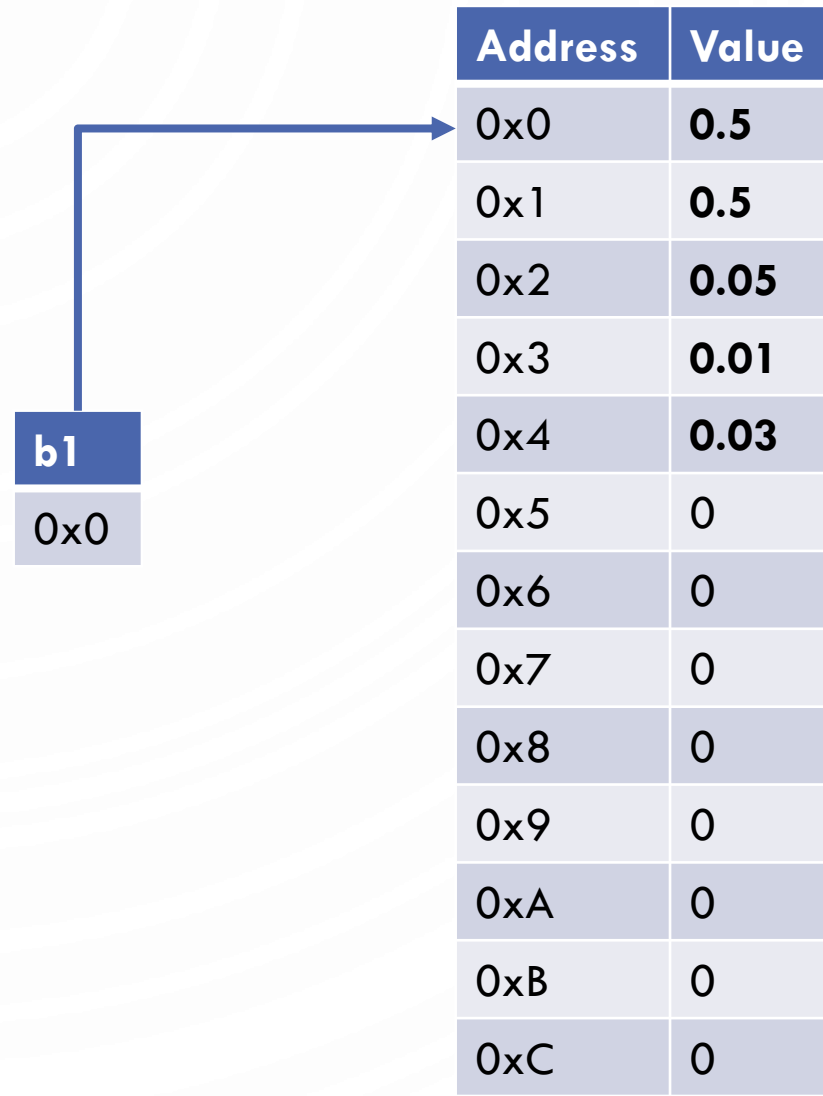
TRACING

```
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();
```

b1

0x0

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

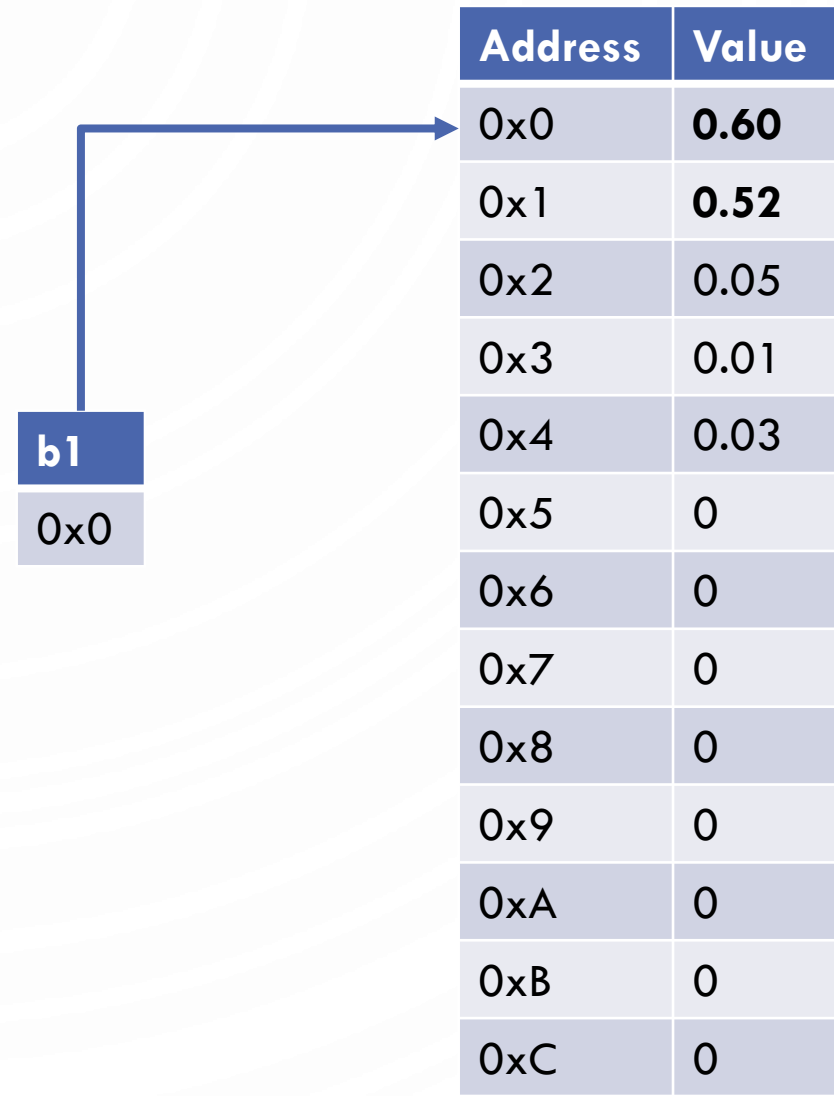# TRACING

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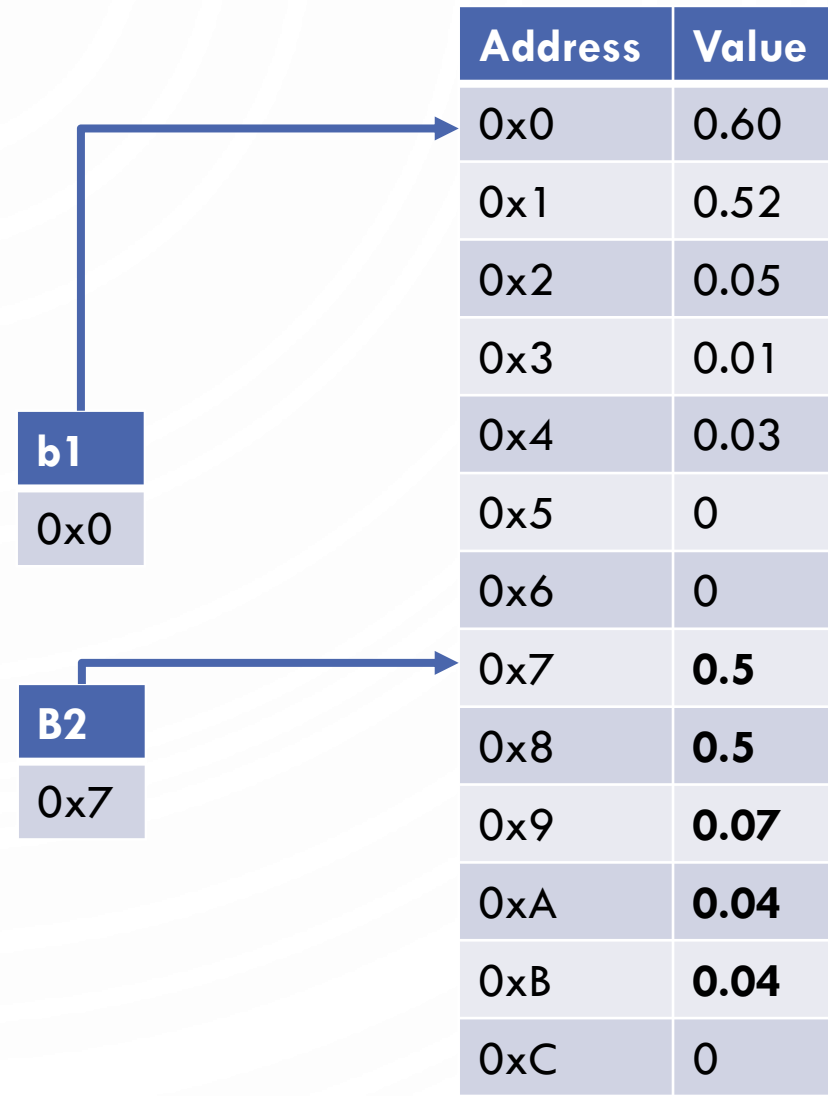
Address	Value
0x0	**0.55**
0x1	**0.51**
0x2	0.05
0x3	0.01
0x4	0.03
0x5	0
0x6	0
0x7	0
0x8	0
0x9	0
0xA	0
0xB	0
0xC	0

b1

0x0

TRACING

```
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();
```
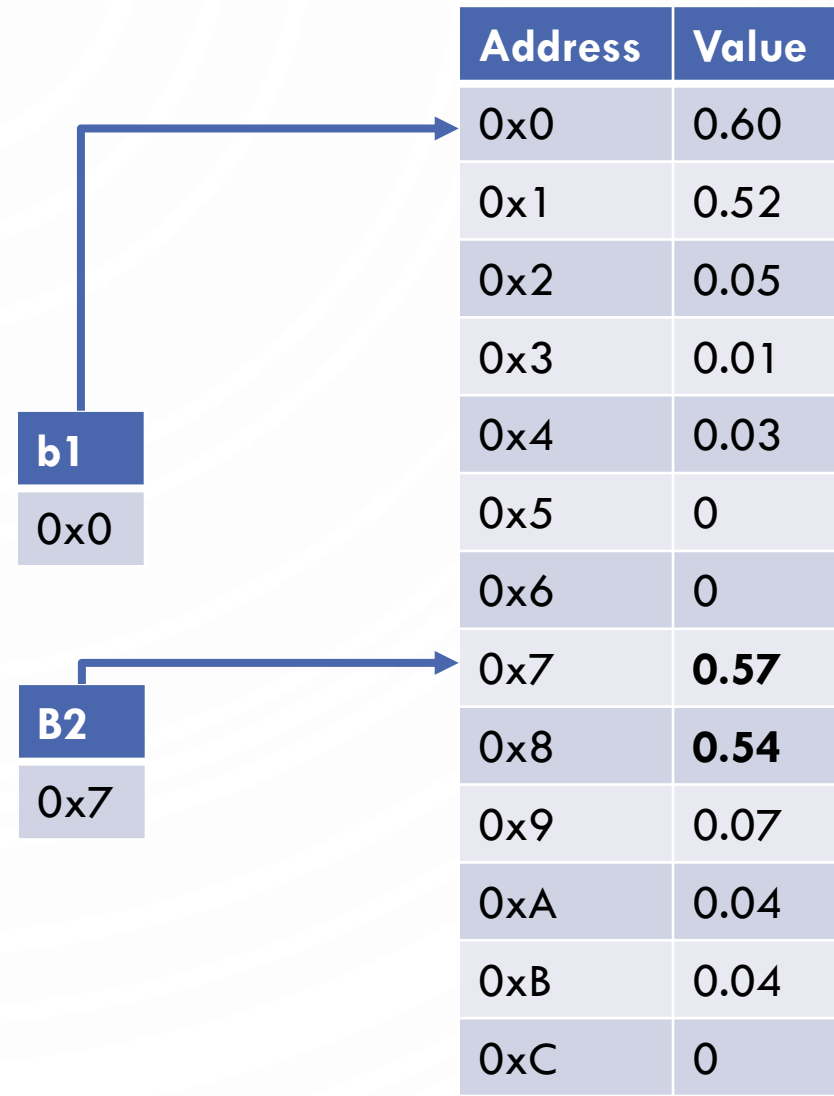
b1
0x0

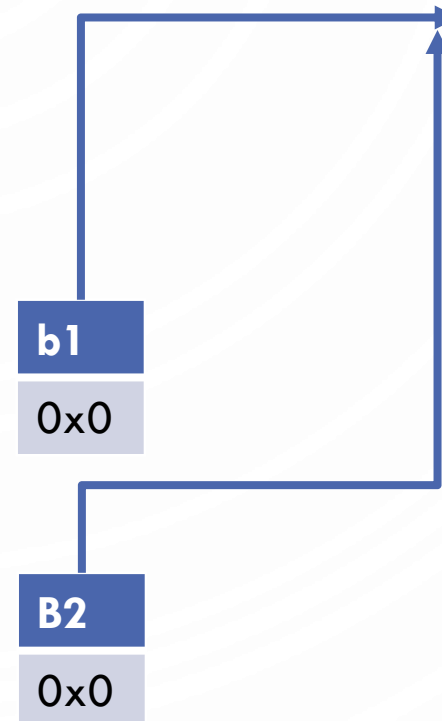| Address	Value
0x0	**0.60**
0x1	**0.52**
0x2	0.05
0x3	0.01
0x4	0.03
0x5	0
0x6	0
0x7	0
0x8	0
0x9	0
0xA	0
0xB	0
0xC	0

TRACING

```
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();
```
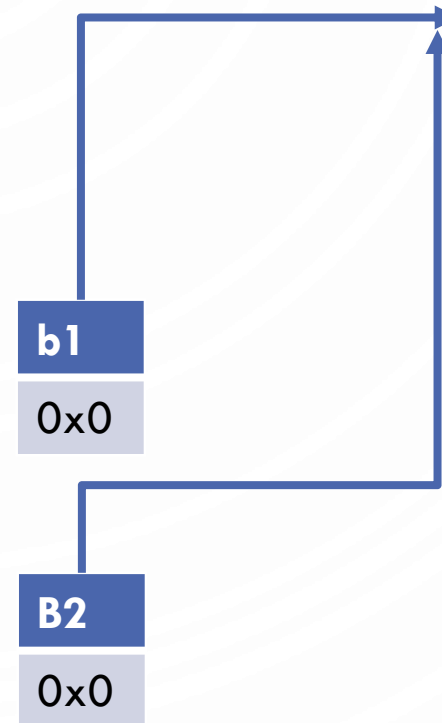
b1
0x0

B2
0x7

Address	Value
0x0	0.60
0x1	0.52
0x2	0.05
0x3	0.01
0x4	0.03
0x5	0
0x6	0
0x7	**0.5**
0x8	**0.5**
0x9	**0.07**
0xA	**0.04**
0xB	**0.04**
0xC	0

TRACING

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();

b1

0x0

B2

0x7

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

TRACING

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

b1
0x0

B2
0x0

TRACING

```
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

b1
0x0

B2
0x0

NEW ABSTRACTION VECTOR

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

```java
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

```java
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. }
```

ASSIGNMENT
PRIMITIVE DATA TYPES VS REFERENCE VARIABLE TYPES
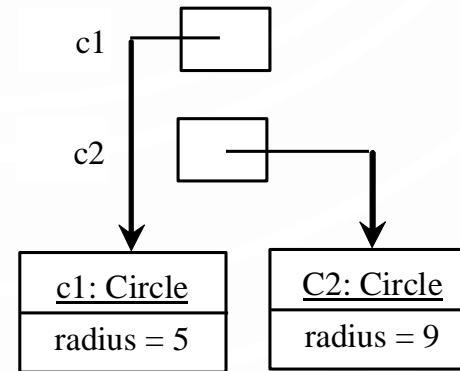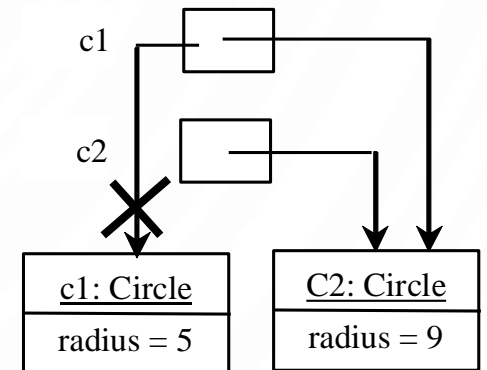
Primitive type assignment $i = j$

Object type assignment $c1 = c2$

Before:

After:

i | 1 |

j | 2 |

i | 2 |

j | 2 |

Before:

c1

c2

| c1: Circle |
| radius = 5 |

| C2: Circle |
| radius = 9 |

After:

c1

c2

| c1: Circle |
| radius = 5 |

| C2: Circle |
| radius = 9 |

GARBAGE COLLECTION

- As shown in the previous figure, after the assignment statement c1 = c2, c1 points to the same object referenced by c2. The object previously referenced by c1 is no longer referenced. This object is known as garbage. Garbage is automatically collected by JVM.

- TIP: If you know that an object is no longer needed, you can explicitly assign null to a reference variable for the object. The JVM will automatically collect the space if the object is not referenced by any variable .

INSTANCE VS STATIC

- Instance – a, or relating to a, specific object's value
  - Instance variables belong to a specific instance.
  - Instance methods are invoked by an instance of the class.

- Static – not a, or relating to a, specific object's value (related to the type). Uses the `static` keyword
  - Static variables are shared by all the instances of the class.
  - Static methods are not tied to a specific object.
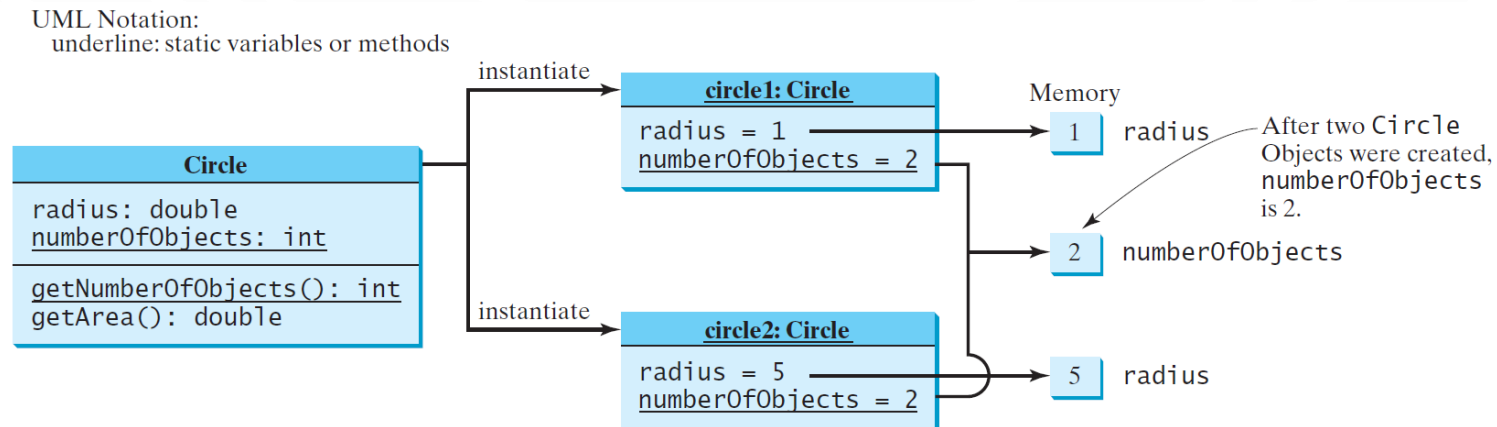
VISIBILITY MODIFIERS AND ACCESSOR/MUTATOR METHODS

- A **visibility modifier** defines the scope of a variable/method and enforces **encapsulation** (data hiding) in objects

- `public` – the class, data, or method is visible to any class in any package.

- `private` – the data or methods can be accessed only by the declaring class.

- By default (no modifier), the class, variable, or method can be accessed by any class in the same package (in between public and private)

- Typically, get and set methods are provided to read and modify private properties.
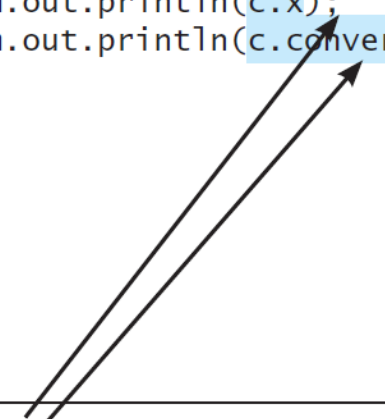
EXAMPLE OF PRIVATE VISIBILITY

- An object cannot access its private members, as shown in (b). It is OK, however,

  if the object is declared in its own class, as shown in (a).

```java
public class C {
  private boolean x;

  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }

  private int convert() {
    return x ? 1 : -1;
  }
}
```
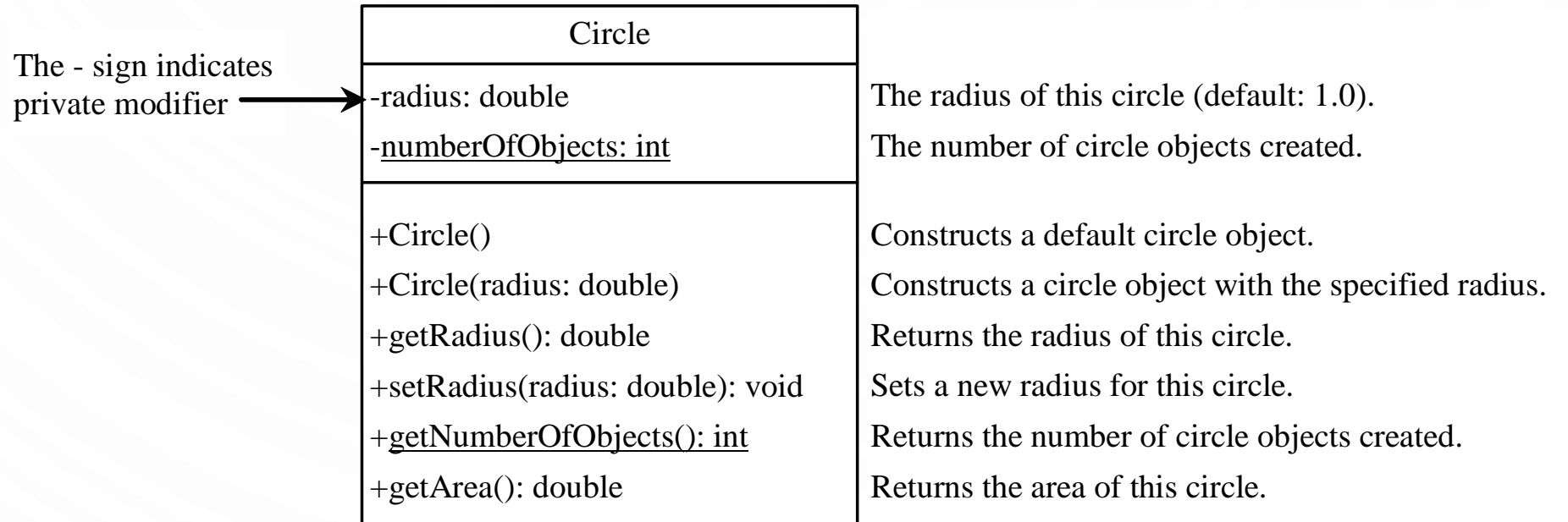
(a) This is okay because object **c** is used inside the class **C**.

```java
public class Test {
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }
}
```

(b) This is wrong because **x** and **convert** are private in class **C**.

WHY DATA FIELDS SHOULD BE PRIVATE?

- Promotes encapsulation
 - To protect data.
  - To make code easy to maintain.

The - sign indicates
private modifier →

Circle	
-radius: double	The radius of this circle (default: 1.0).
-numberOfObjects: int	The number of circle objects created.
+Circle()	Constructs a default circle object.
+Circle(radius: double)	Constructs a circle object with the specified radius.
+getRadius(): double	Returns the radius of this circle.
+setRadius(radius: double): void	Sets a new radius for this circle.
+getNumberOfObjects(): int	Returns the number of circle objects created.
+getArea(): double	Returns the area of this circle.

PASSING OBJECTS TO METHODS

- Recall – all parameters to functions are passed-by-value in Java
  - Passing-by-value for primitive type value (the value is copied to the parameter)
  - Passing-by-value for reference type value (the value is the reference to the object)

```java
public static void printArea(
    Circle c, int times) {
  for(int i = 0; i < times; ++i)
    System.out.println(
      c.getArea());
}

public static void main(
    String[] args) {
  Circle myCircle = new Circle();
  printArea(5, myCircle);
}
```
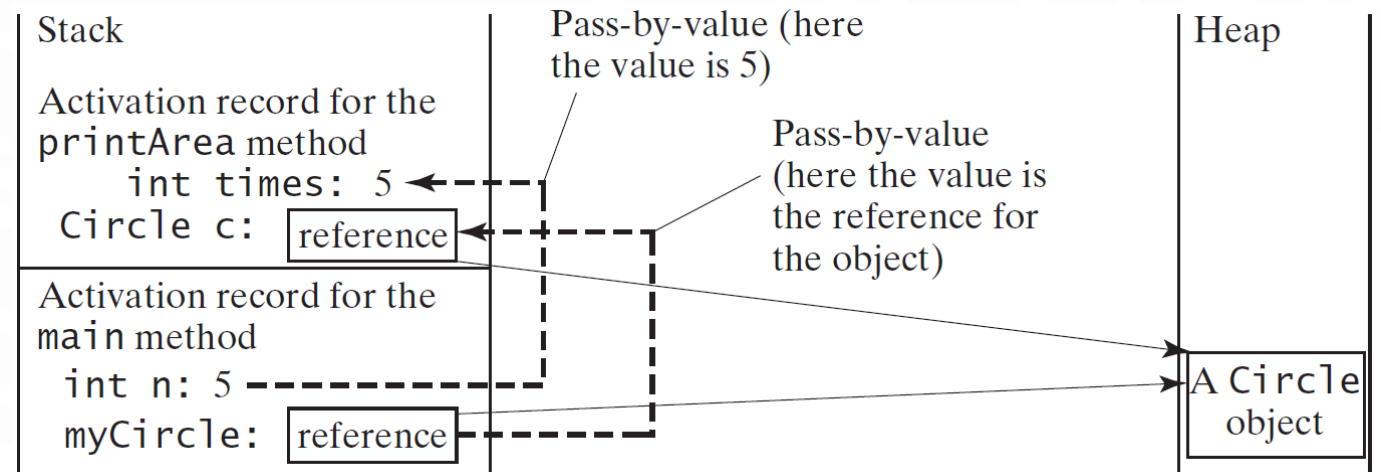
ARRAY OF OBJECTS

- **Circle**[] circleArray =
    new Circle[10];

- An array of objects is actually an array of reference variables

- Invoking
circleArray[1].getArea()
involves two levels of referencing.
circleArray references to the entire
array. circleArray[1] references to
a Circle object.
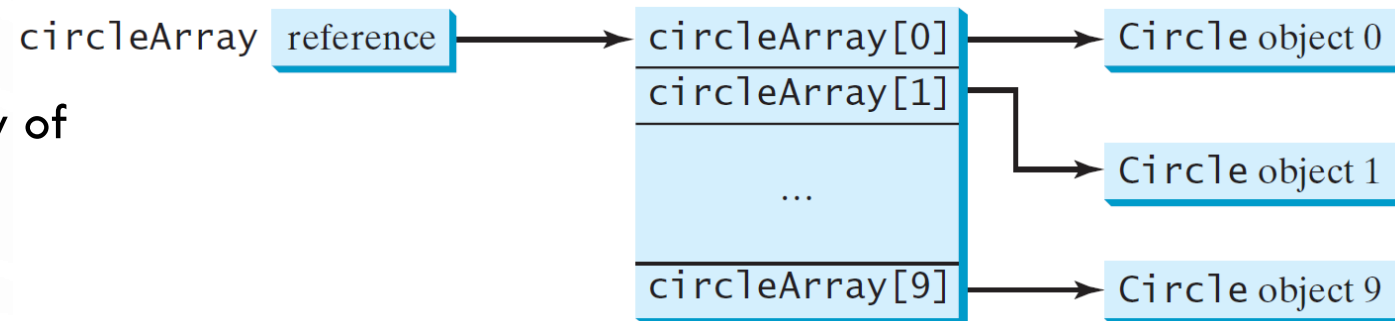
EXAMPLE BOUNCING BALL

PROGRAM ALONG

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. }
```

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.  }
```

SCOPE OF VARIABLES

- Recall – **scope** is the lifetime of a variable. It dictates where you as the programmer may refer to the identifier (name) in code
  - Rule – The scope of instance and static variables is the entire class (including inside of any method). They can be declared anywhere inside a class.
  - Rule – The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be initialized explicitly before it can be used.

THE THIS KEYWORD

- The `this` keyword is the name of a reference that refers to an object itself. One common use of the this keyword is reference a class's hidden data fields.

- Another common use of the this keyword to enable a constructor to invoke another constructor of the same class.

`this`

Circle
radius: 10
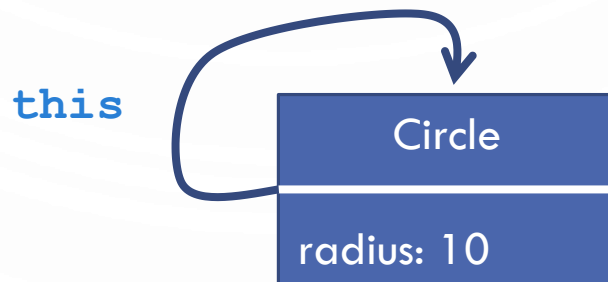
EXAMPLE
REFERENCE THE HIDDEN DATA FIELDS

```java
public class F {
   private int i = 5;
   private static double k = 0;

   void setI(int i) {
      this.i = i;
   }

   static void setK(double k) {
      F.k = k;
   }
}
```

```
Suppose that f1 and f2 are two objects of F.
F f1 = new F(); F f2 = new F();

Invoking f1.setI(10) is to execute
    this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
    this.i = 45, where this refers f2
```

CALLING OVERLOADED CONSTRUCTOR

```java
public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public Circle() {
        this(1.0);
    }

    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }
}
```

Or rename the parameter!

this must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted

IMMUTABLE OBJECTS AND CLASSES

- If the contents of an object cannot be changed once the object is created, the object is **immutable** and its class is called an **immutable class**.
  - If you delete the set method in the Circle class in Listing 8.10, the class would be immutable because radius is private and cannot be changed without a set method.
- A class with all private data fields and without mutators is not necessarily immutable. For example, the following class Student has all private data fields and no mutators, but it is mutable.

EXAMPLE
STUDENT IS STILL MUTABLE, HOW?

```
1.  public class Student {
2.     private int id;
3.     private BirthDate birthdate;
4.     public Student(int ssn, int year,
5.         int month, int day) {
6.       id = ssn;
7.       birthDate = new BirthDate(
          year, month, day);
8.     }
9.     public int getId() {
10.      return id;
11.    }
12.    public BirthDate getBirthDate() {
13.      return birthDate;
14.    }
15.}
```

```
1.  public class BirthDate {
2.     private int year;
3.     private int month;
4.     private int day;
5.     public BirthDate(int newYear,
6.       int newMonth, int newDay) {
7.       year = newYear;
8.       month = newMonth;
9.       day = newDay;
10.    }
11.    public void setYear(int newYear) {
12.      year = newYear;
13.    }
14.}
```

WHAT CLASS IS IMMUTABLE?

- For a class to be immutable, it must mark all data fields private and provide no mutator methods and no accessor methods that would return a reference to a mutable data field object.

IMMUTABLE WRAPPER CLASSES

- **Boolean**
- **Character**
- **Short**
- **Byte**
- **Integer**
- **Long**
- **Float**
- **Double**
- **String**

- All are immutable without a no-arg constructor
- All provide limits of their data types (e.g., **Integer**.MAX_VALUE and **Double**.POSITIVE_INFINITY)
- All provide functions to convert between each other (e.g., **Integer**.parseInt() and **String**.valueOf())
- Since Java 5, primitive types can be automatically be converted to their immutable class counterpart (called **boxing**)

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

```java
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. }
```

$(x_0 + d \cos \alpha, y_0 + d \sin \alpha)$

d

$d \sin \alpha$

α

(x_0, y_0)

$d \cos \alpha$
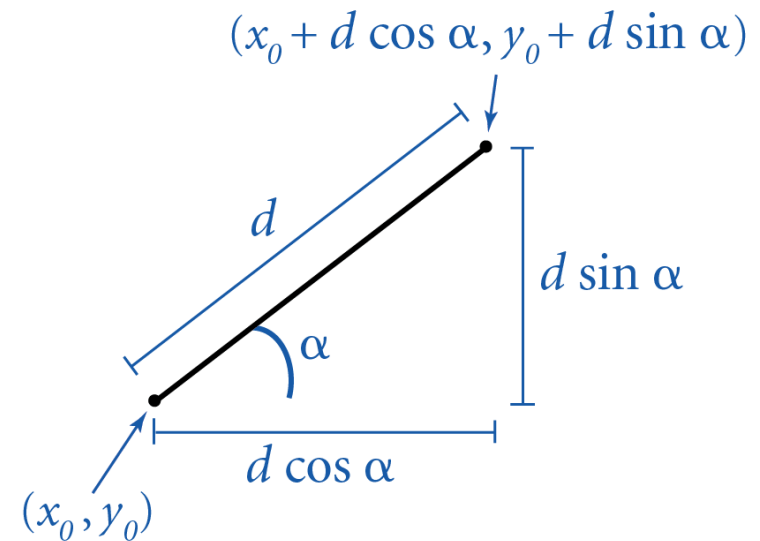
Turtle trigonometry
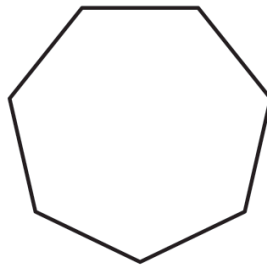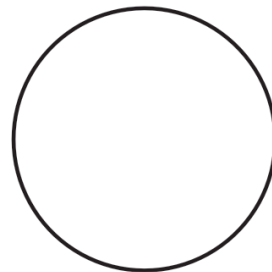
N-GON

```java
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.}
```
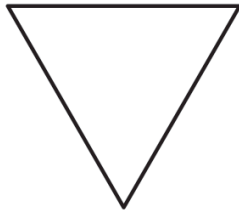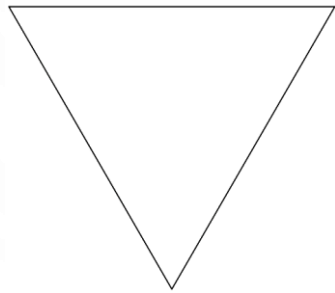

3   **7**   **1440**
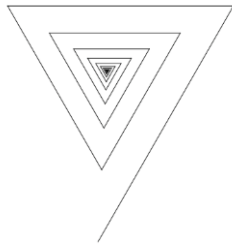
SPIRAL

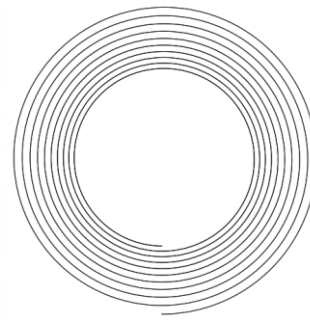```
1. public class Spiral {
2.   public static void main(String[] args) {
3.   int N   = Integer.parseInt(args[0]);
4.   double decay = Double.parseDouble(args[1]);
5.   double angle = 360.0 / N;
6.   double step = Math.sin(Math.toRadians(angle/2.0));
7.   Turtle turtle = new Turtle(0.5, 0, angle/2.0);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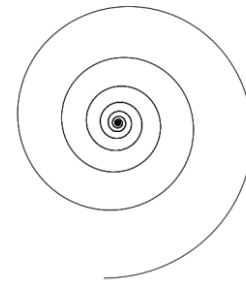


3 1.0          3 1.2          1440 1.00004          1440 1.0004

# 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.