OOP: Objects and Classes

Lecture 2

OO Programming Concepts

- Object-oriented programming (OOP) involves programming using **objects**.
  - An object represents an entity in the real world that can be distinctly identified, such as a desk, a button, a car, etc.

- An object has…
  - Unique **identity** (think memory address)
  - **State**, consisting of a set of data fields (also known as properties) with their current values
  - **Behavior**, defined by a set of methods

Classes

- A **class** is a template, blue-print, or *contract* that defines what an object's data fields and methods will be
  - Typically in its own file (name of the file = name of the class)
- Every object is an *instance* of some class
  - Think of the class as the data type, whereas an object is a variable of that type

Example: Circles

- Class: Circle
  - All circles **have**…
    - A radius
  - All circles **can**…
    - Tell you their area
    - Get/set radius

Class: Circle

Data Fields:
- radius

Methods:
- getArea()
- getRadius()
- setRadius(r)

- Some example circles…
  - c1: radius=5
  - c2: radius=10
  - c3: radius=5
    - Distinct from c1!
  - c4: radius=1

Object: c1	Object: c2	Object: c3	Object: c4
- Class: Circle	- Class: Circle	- Class: Circle	- Class: Circle
Data Fields:	Data Fields:	Data Fields:	Data Fields:
- radius=5	- radius=10	- radius=5	- radius=1

Code: Circles

Circle.java

```
public class Circle {

	private double radius = 1.0;

	public Circle() {
	}

	public Circle(double r) {
		setRadius(r);
	}

	public double getRadius() {
		return radius;
	}

	public void setRadius(double r) {
		if (r>0) {
			radius = r;
		}
	}

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

Anywhere

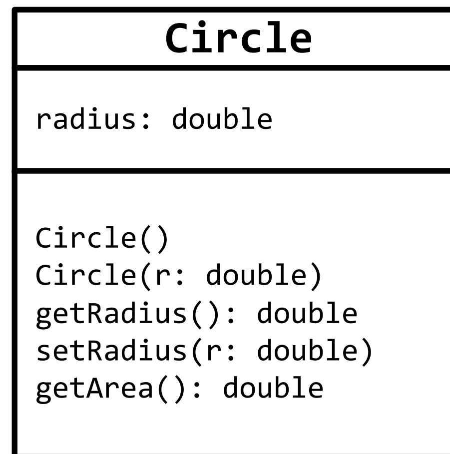
```
Circle c1 = new Circle(5);
Circle c2 = new Circle(10);
Circle c3 = new Circle(5);
Circle c4 = new Circle();
```

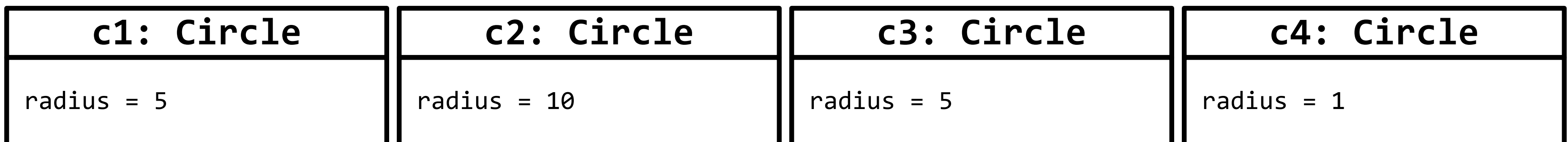
Output: Circles

```
Circle c1 = new Circle(5);
Circle c2 = new Circle(10);
Circle c3 = new Circle(5);
Circle c4 = new Circle();

System.out.printf("Circle 1 (%s): r=%.2f A=%.2f%n",
                    c1, c1.getRadius(), c1.getArea());
System.out.printf("Circle 2 (%s): r=%.2f A=%.2f%n",
                    c2, c2.getRadius(), c2.getArea());
System.out.printf("Circle 3 (%s): r=%.2f A=%.2f%n",
                    c3, c3.getRadius(), c3.getArea());
System.out.printf("Circle 4 (%s): r=%.2f A=%.2f%n",
                    c4, c4.getRadius(), c4.getArea());
```

UML: Circles





Constructors

- A **constructor** is a special type of method that is invoked to construct an object from its class
- All classes have at least one constructor
- All constructor(s) for a class…
  - Have the same name as the class
  - Have no return type (not even void)
  - A constructor with no parameters is referred to as a **no-arg constructor**
- A constructor is invoked exactly once for an object automatically via the **new** operator

Default Constructors

- A class may be defined without any constructors
- In this case, a no-arg constructor with an empty body is implicitly defined in the class
- This **default constructor** is provided automatically only if no constructors are explicitly defined in the class

Example (1)

Circle.java

```
public class Circle {

	private double radius = 1.0;

	public Circle(double r) {
		setRadius(r);
	}

	public double getRadius() {
		return radius;
	}

	public void setRadius(double r) {
		if (r>0) {
			radius = r;
		}
	}

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

Anywhere

```
Circle c1 = new Circle(5);
Circle c2 = new Circle(10);
Circle c3 = new Circle(5);
```

~~`Circle c4 = new Circle();`~~

Example (2)

Circle.java

```
public class Circle {

	private double radius = 1.0;

	public double getRadius() {
		return radius;
	}

	public void setRadius(double r) {
		if (r>0) {
			radius = r;
		}
	}

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

Anywhere

~~Circle c1 = new Circle(5);~~

~~Circle c2 = new Circle(10);~~

~~Circle c3 = new Circle(5);~~

```
Circle c4 = new Circle();
```

Reference Variables

- To reference an object, assign the object to a **reference variable**

- To declare a reference variable…

```
ClassName objectRefVar;
```

- Example:

```
Circle c;
```

Referencing a Newly Created Object

- The **new** operator creates a new object and returns a reference

```
new Circle();
```

- Thus the typical pattern…


```
Circle c = new Circle();
```

Accessing Object's Members

- Accessing class/object data/methods is achieved via the dot (`.`) operator

- Member variables

```
objRefVar.data
```

e.g. `c.radius`

- Member methods

```
objRefVar.methodName(args)
```

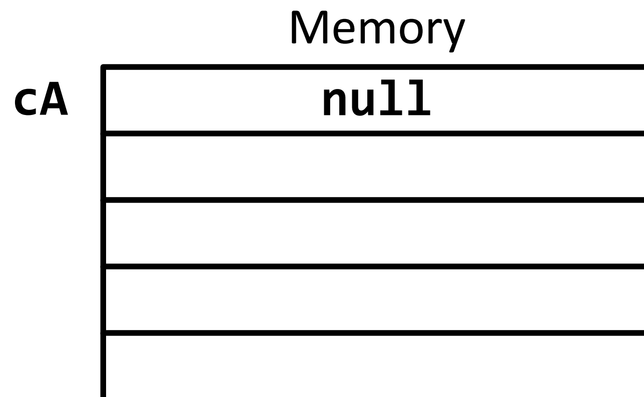
e.g. `c.getArea()`

Trace Code (1)

```
Circle cA = new Circle(5.0);
Circle cB = new Circle();
cB.setRadius(100);
```

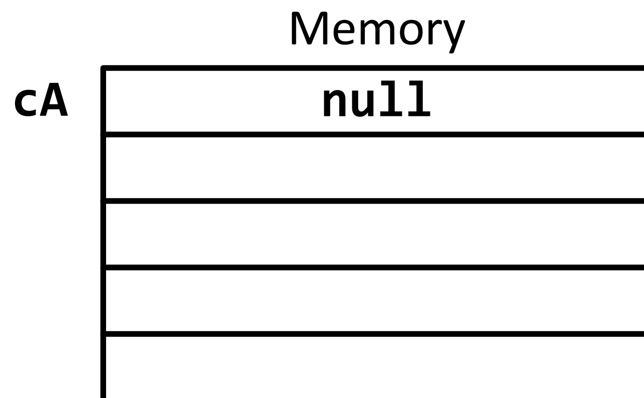
Trace Code (2)

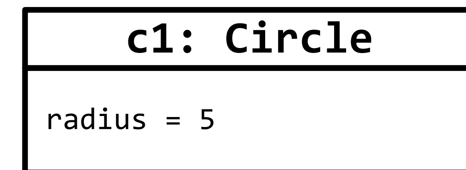
```
Circle cA = new Circle(5.0);
Circle cB = new Circle();
cB.setRadius(100);
```



Trace Code (3)

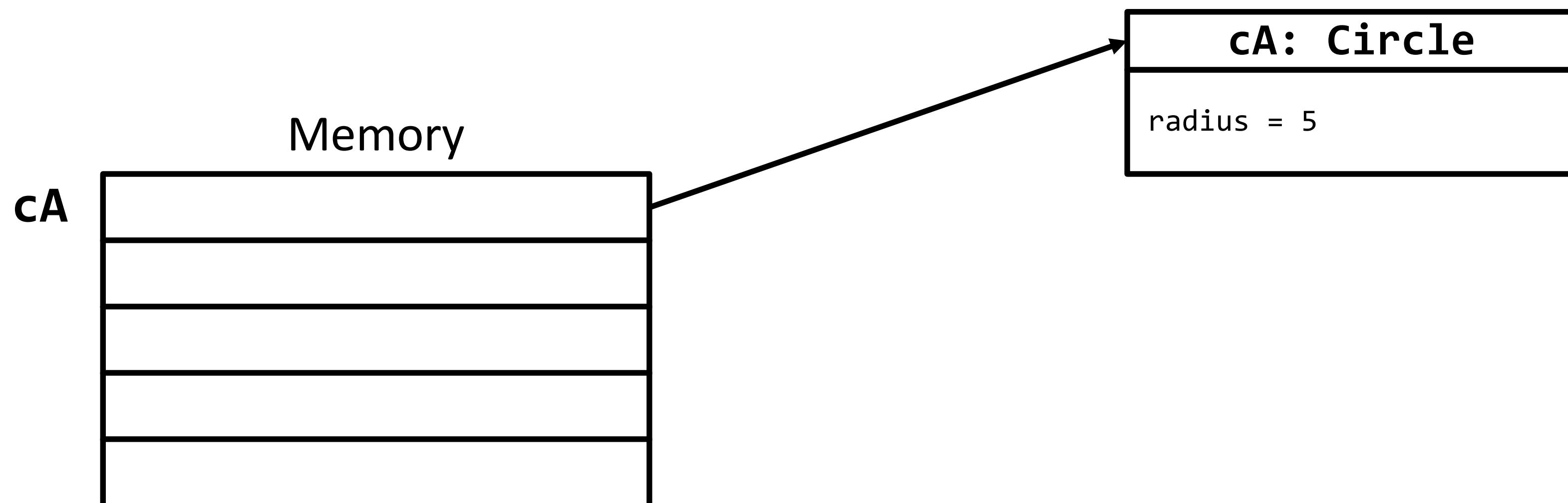
```
Circle cA = new Circle(5.0);
Circle cB = new Circle();
cB.setRadius(100);
```





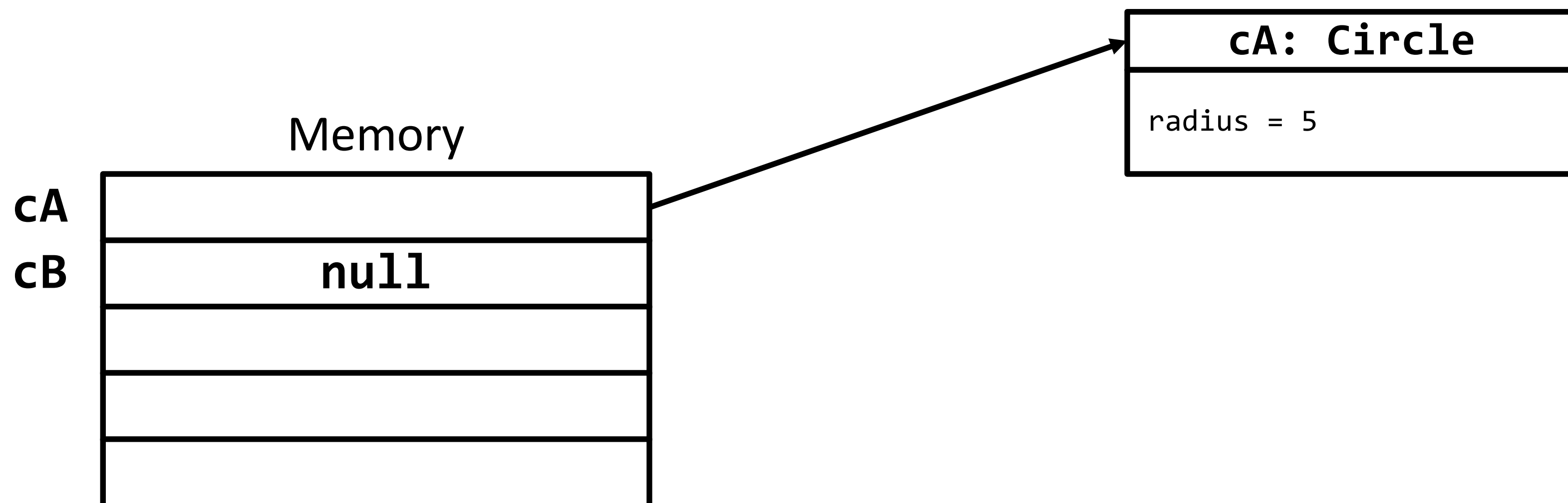
Trace Code (4)

```
Circle cA = new Circle(5.0);
Circle cB = new Circle();
cB.setRadius(100);
```



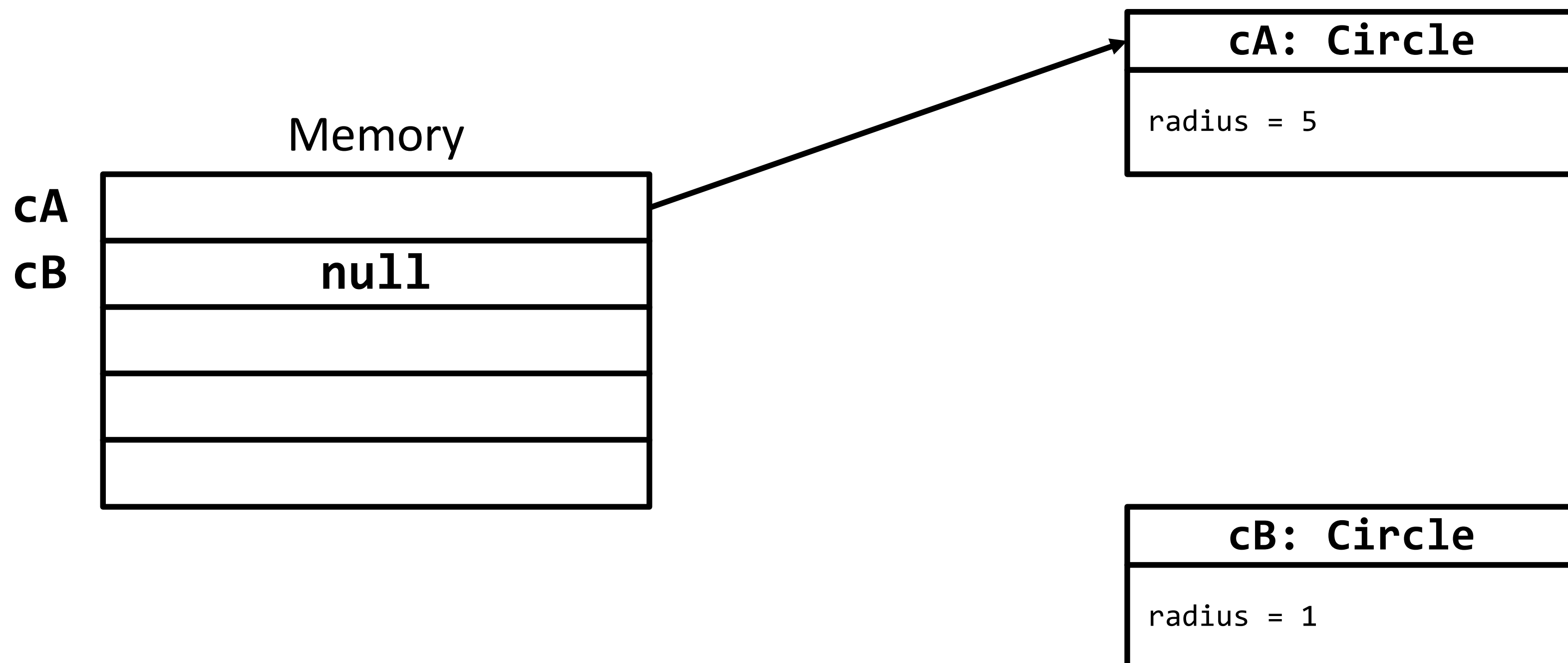
Trace Code (5)

```
Circle cA = new Circle(5.0);
Circle cB = new Circle();
cB.setRadius(100);
```



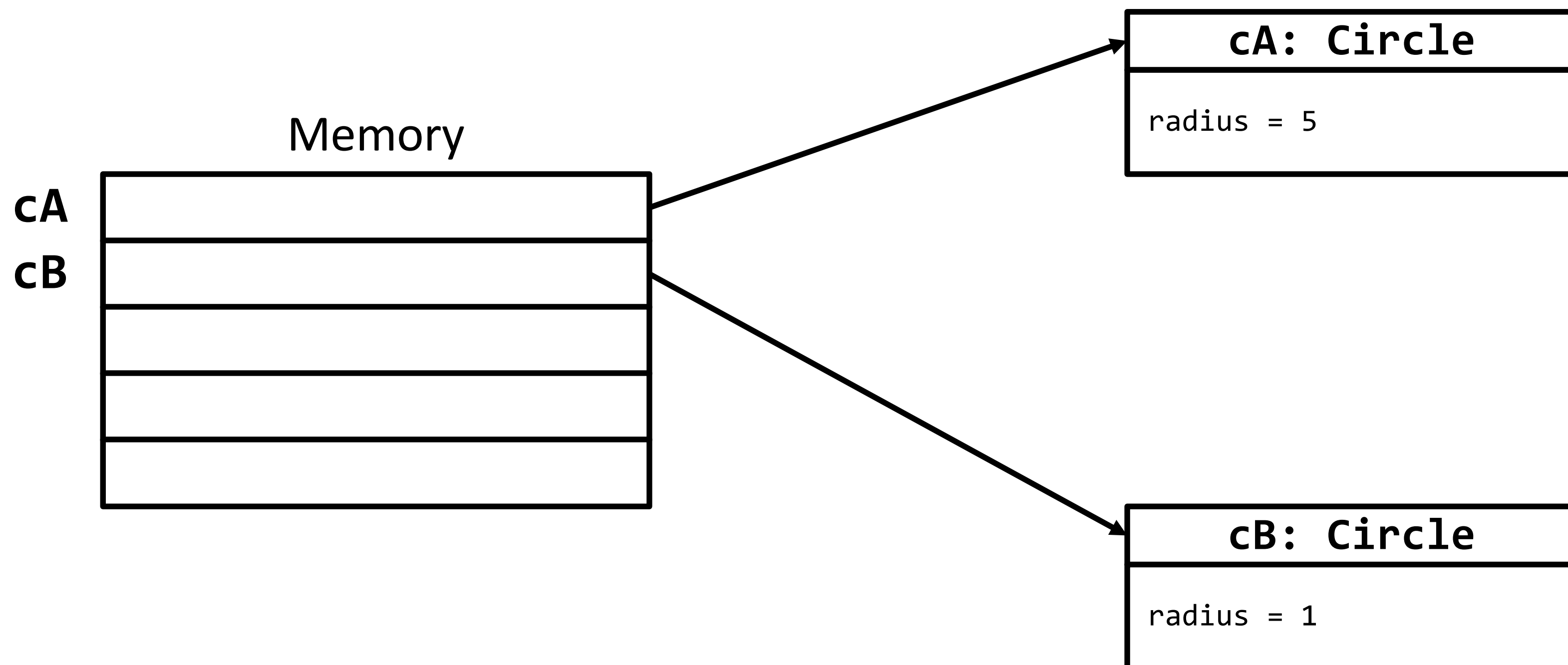
Trace Code (6)

```
Circle cA = new Circle(5.0);
Circle cB = new Circle();
cB.setRadius(100);
```



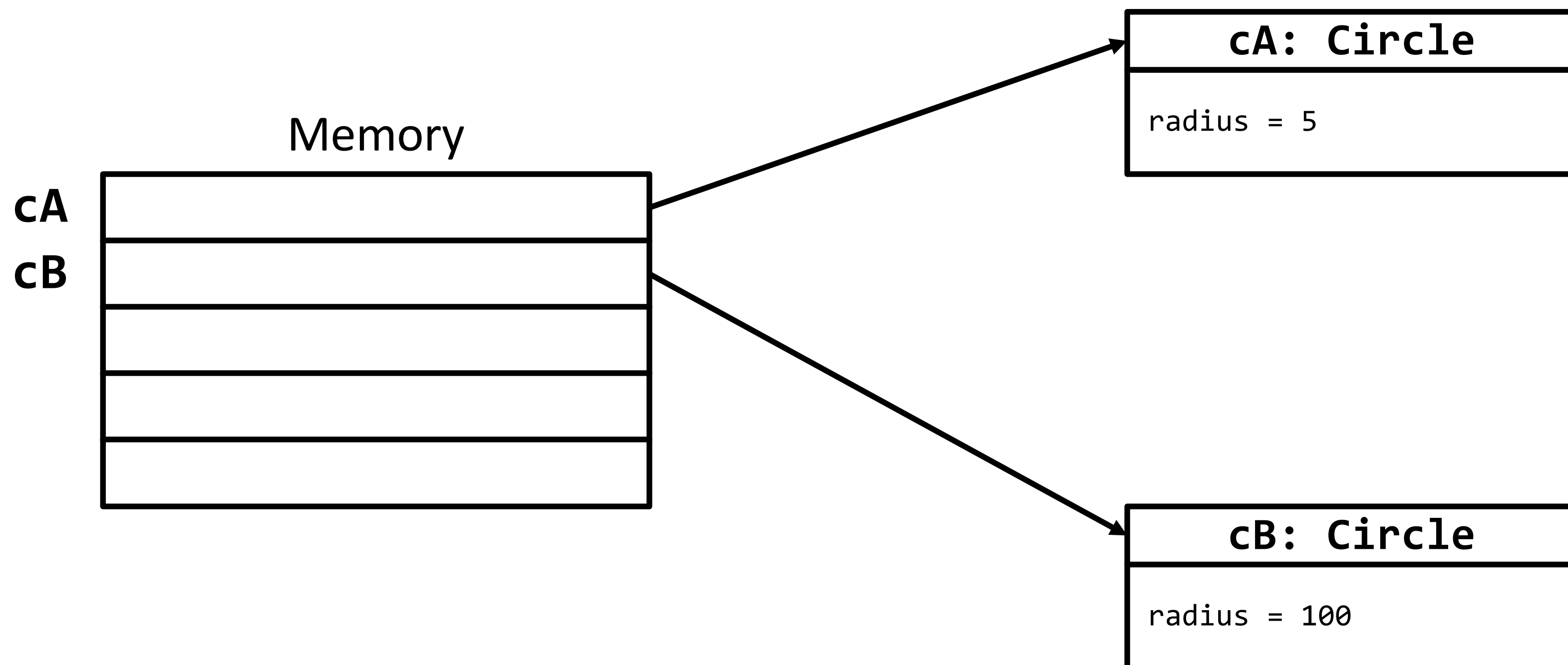
Trace Code (7)

```
Circle cA = new Circle(5.0);
Circle cB = new Circle();
cB.setRadius(100);
```



Trace Code (8)

```
Circle cA = new Circle(5.0);
Circle cB = new Circle();
cB.setRadius(100);
```



The Value `null`

- In Java, **`null`** is a special literal value that indicates an invalid reference (that is, the variable does not refer to an object)

- Trying to access member data/methods of **`null`** results in a **`NullPointerException`**

```
Circle c4 = null;
System.out.printf("%s%n", c4);
System.out.printf("%d%n", c4.getRadius());
```

Default Values for Member Data

- The default value of member data for an object depends on the data type
  - Reference: `null`
  - Numeric: `0`
  - `boolean`: `false`
  - `char`: `'\u0000'` (basically ASCII 0)

- However, Java assigns no default value to local variables inside methods

Example

Stuff.java

```
public class Stuff {
   public int value;
}
```

Anywhere

```
Stuff things = new Stuff();

int x;

System.out.printf("%d%n",
   things.value);
```

~~System.out.printf("%d%n",
   x);~~

Exercise

- Define a Student class to have the following data fields
  - lastName (String)
  - age (int)
  - isScienceMajor (boolean)
  - firstInitial (char)

- Create an instance of the Student class and print out the default value of all the data fields

Solution

Student.java

```
public class Student {
    public String lastName;
    public int age;
    public boolean isScienceMajor;
    public char firstInitial;
}
```

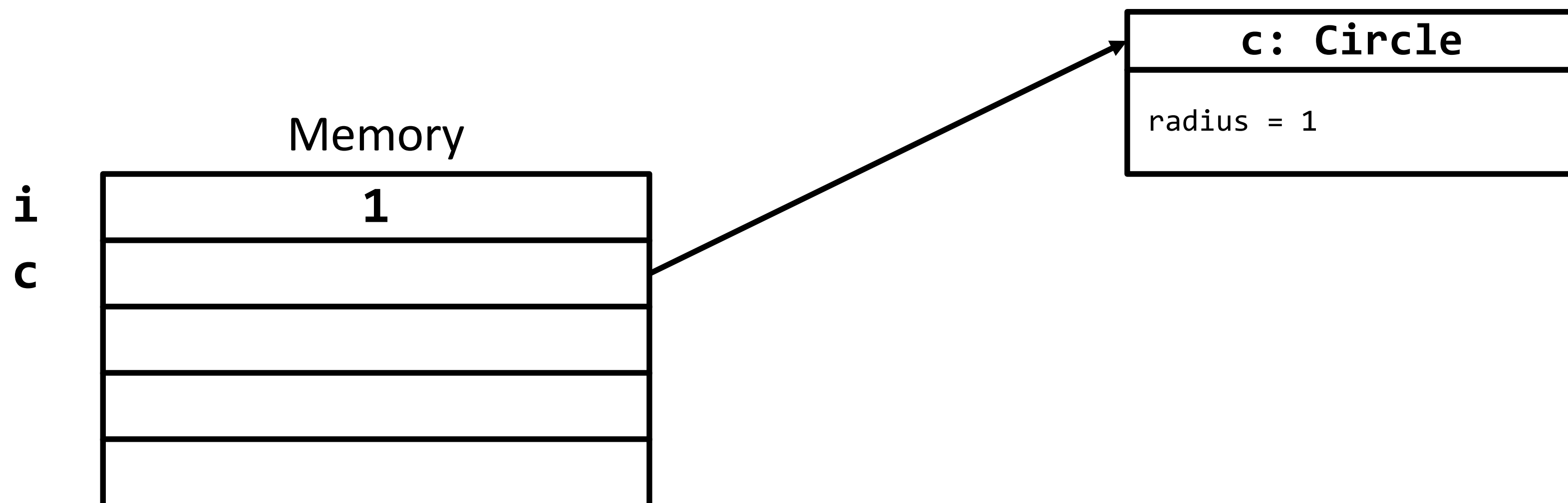
Anywhere

```
Student s1 = new Student();

System.out.printf("%s%n",
    s1.lastName); // null
System.out.printf("%d%n",
    s1.age); // 0
System.out.printf("%b%n",
    s1.isScienceMajor); // false
System.out.printf("%c%d%n",
    s1.firstInitial,
    (int) s1.firstInitial); // 0
```

Primitive Data Types vs. Objects

```
int i = 1;
Circle c = new Circle();
```



Primitive Assignment

Code

```
int i = 1, j = 2;
```

```
i = j;
```

Memory

- Before assignment

	Memory
i	1
j	2

- After assignment

	Memory
i	2
j	2

Object Assignment

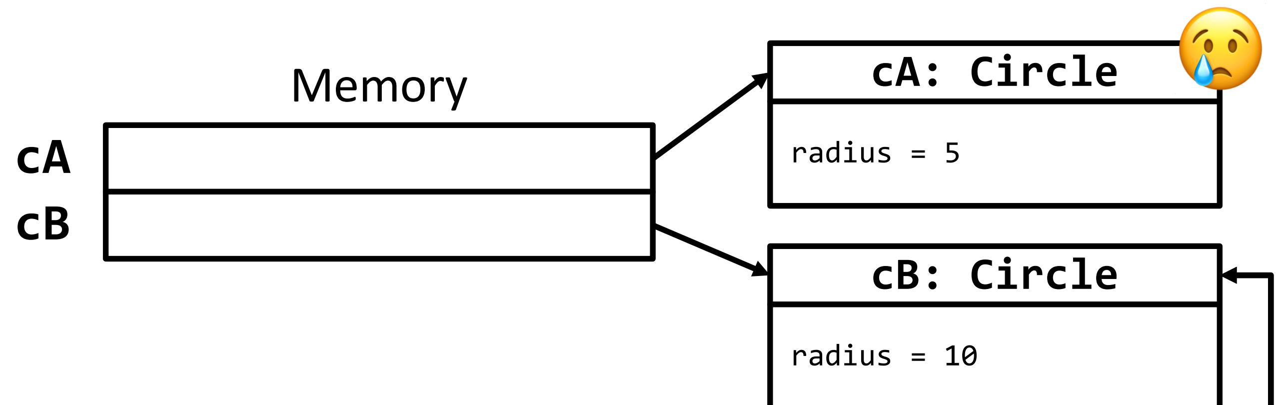
Code

```
Circle cA =
    new Circle(5);
Circle cB =
    new Circle(10);
```

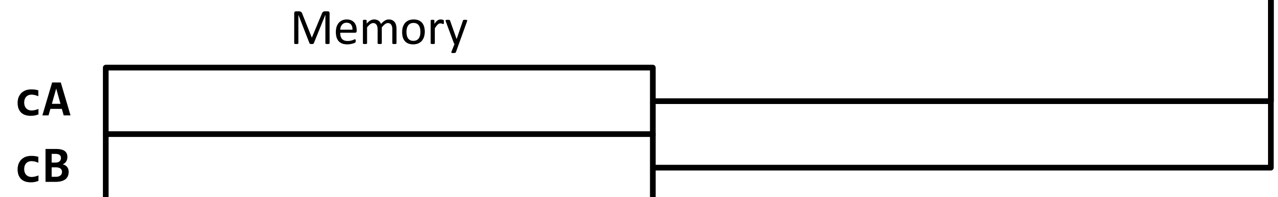
```
cA = cB;
```

Memory

- Before assignment



- After assignment



Garbage Collection

- 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 cleaned up by the JVM via **Garbage Collection (GC)**

GC Pro Tips

- 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 … eventually (i.e. you don't have direct control)

Static vs. Instance

Static

- Shared by ALL instances of a class
  - Method: only can use static variables
  - Variable: can be used by any instance
- Invoked via Class.var/method()
- Specified via `static` keyword

Instance

- Tied to a specific instance of a class
  - Method: can use static AND member variables
  - Variable: can only be used by its instance
- Invoked via objRef.var/method()
- Specified via the lack of `static` keyword

Example

Person.java

```
public class Person {
	private static int numPeople = 0;
	private final int myId;

	public Person() {
		numPeople++;
		myId = numPeople;
	}

	public int getId() {
		return myId;
	}

	public String getBorgId() {
		return String.format("%d of %d",
			myId, numPeople);
	}

	public static int getNumPeople() {
		return numPeople;
	}
}
```

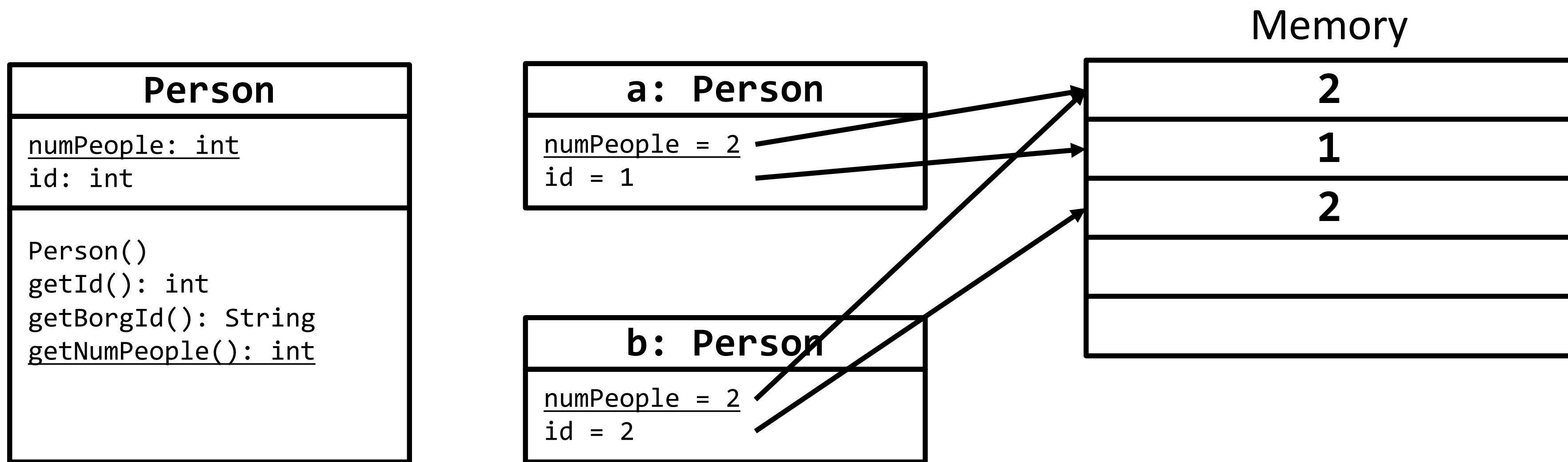
Anywhere

```
System.out.printf("Peeps: %d%n",
    Person.getNumPeople());

Person a = new Person();
System.out.printf("A: %s%n",
    a.getBorgId());
System.out.printf("Peeps: %d%n",
    Person.getNumPeople());

Person b = new Person();
System.out.printf("A: %s%n",
    a.getBorgId());
System.out.printf("B: %s%n",
    b.getBorgId());
System.out.printf("Peeps: %d%n",
    Person.getNumPeople());
```

UML & Memory



Packages

- Packages are a way to organize classes
  - Useful for managing large projects, allowing overlap of class names, and controlling access to sensitive data (more in Visibility)

- Specified via the package keyword

```
package packageName;
```

  - Must be first in the file, none = “default” package (discouraged)

Package Naming & Directory Structure

- It is common practice to preface the package name with a unique identifier owned by the author (by convention a URL in reverse)
  - Example: **`edu.wit.cs.comp1050`**

- Java requires that any dots (`.`) in the package name correspond to folders in the file system

Visibility Modifiers

- Way of specifying what code can “see” (i.e. directly access) a variable/method
  - By default, operate under a “need-to-know” basis (i.e. most constraining)
  - No modifier = *package-private*
  - More on protected/subclass soon!

	Class	Package	Subclass	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	
no modifier	✓	✓		
`private`	✓			

Example (1)

Foo.java

```
package p1;

public class Foo {
	private int x;
	int y;
	public int z;

	public Foo() {
		x = y = z = 1;
		f1();
		f2();
		f3();
	}

	private void f1() {}
	void f2() {}
	public void f3() {}
}
```

Bar.java

```
package p1;

public class Bar {
	public static void main(String[] args) {
		Foo f = new Foo();

		f.x*=2;
		f.y++;
		f.z--;

		f.f1();
		f.f2();
		f.f3();
	}
}
```

(The lines `f.x*=2;` and `f.f1();` are struck through in red.)

Example (2)

Foo.java

```
package p1;

public class Foo {
     private int x;
     int y;
     public int z;

     public Foo() {
          x = y = z = 1;
          f1();
          f2();
          f3();
     }

     private void f1() {}
     void f2() {}
     public void f3() {}
}
```

Baz.java

```
package p2;

import p1.Foo;

public class Baz {
     public static void main(String[] args) {
          Foo f = new Foo();

          f.x*=2;
          f.y++;
          f.z--;

          f.f1();
          f.f2();
          f.f3();
     }
}
```

Checkup

What is wrong with the following code…

```
public class Test {
   int x;

   public Test(String t) {
      System.out.printf("Hello");
   }

   public static void main(String[] args) {
      Test test = new Test();
      System.out.println(test.x);
   }
}
```

Solution

What constructor is being used for the first line of main()…?

Passing Objects to Methods

- For both primitives and objects, the value is passed to the method
- However, the “value” of an object is the reference (think: memory address), and so the object can actually be changed within the method

Example

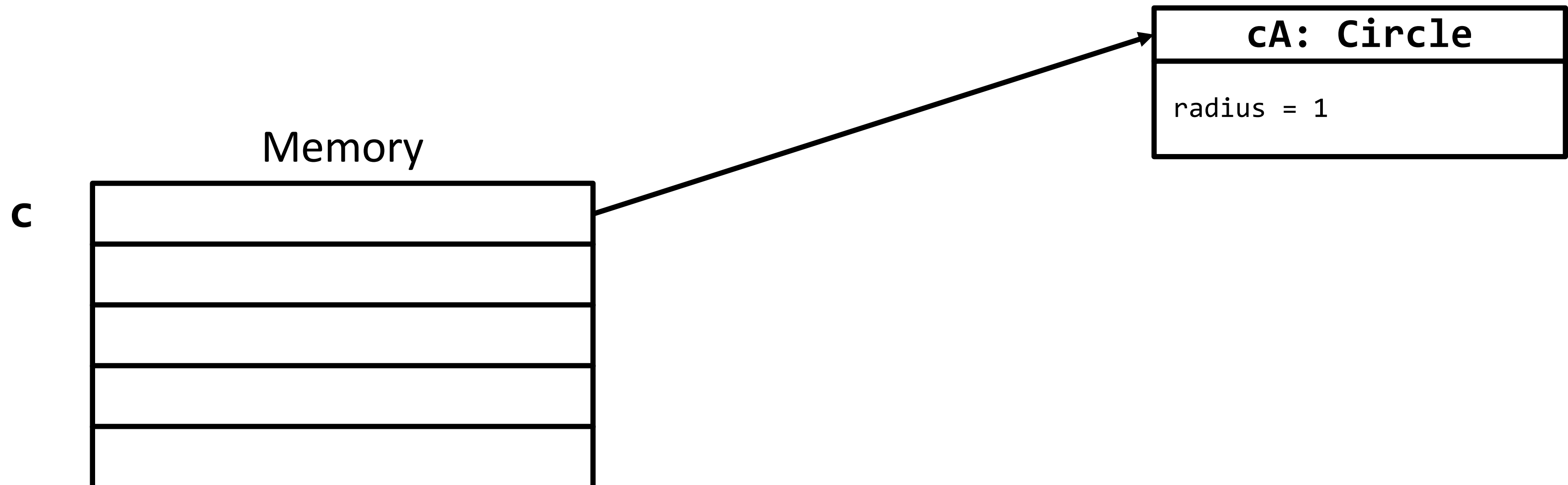
```
public static void inc(Circle c, int x) {
   c.setRadius(c.getRadius()+1);
   x++;
}

public static void main(String[] args) {
   Circle c = new Circle(1);
   int x = 1;

   System.out.printf("%.2f %d%n", c.getRadius(), x);
   inc(c, x);
   System.out.printf("%.2f %d%n", c.getRadius(), x);
}
```

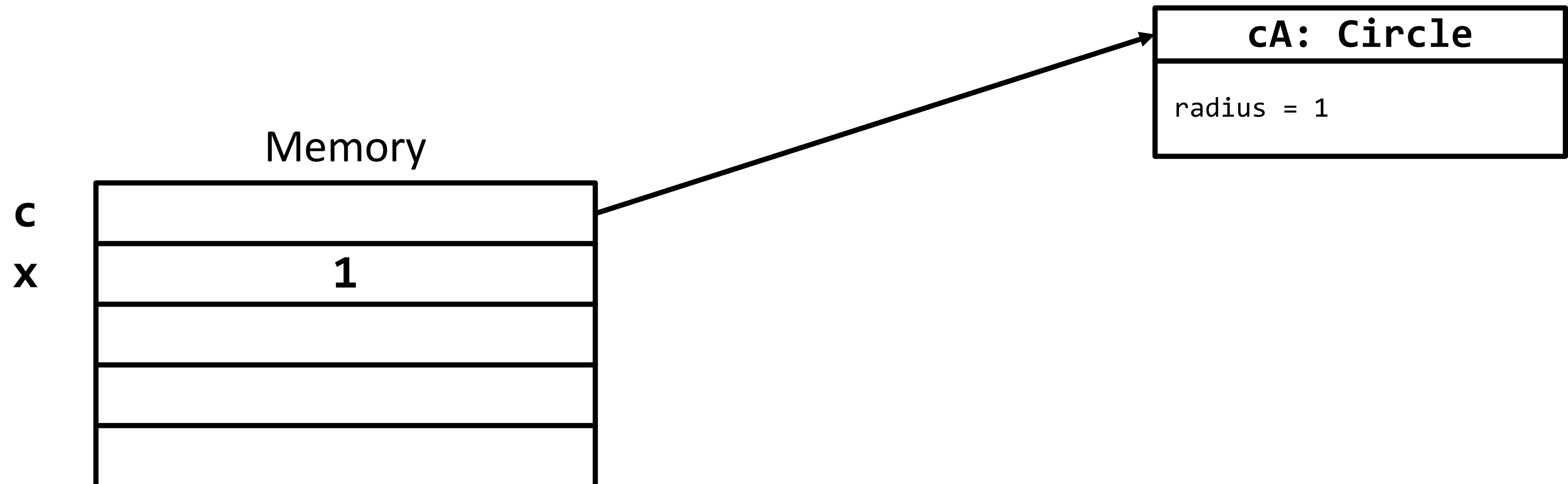
A Peek Into Memory (1)

```
Circle c = new Circle(1);
```



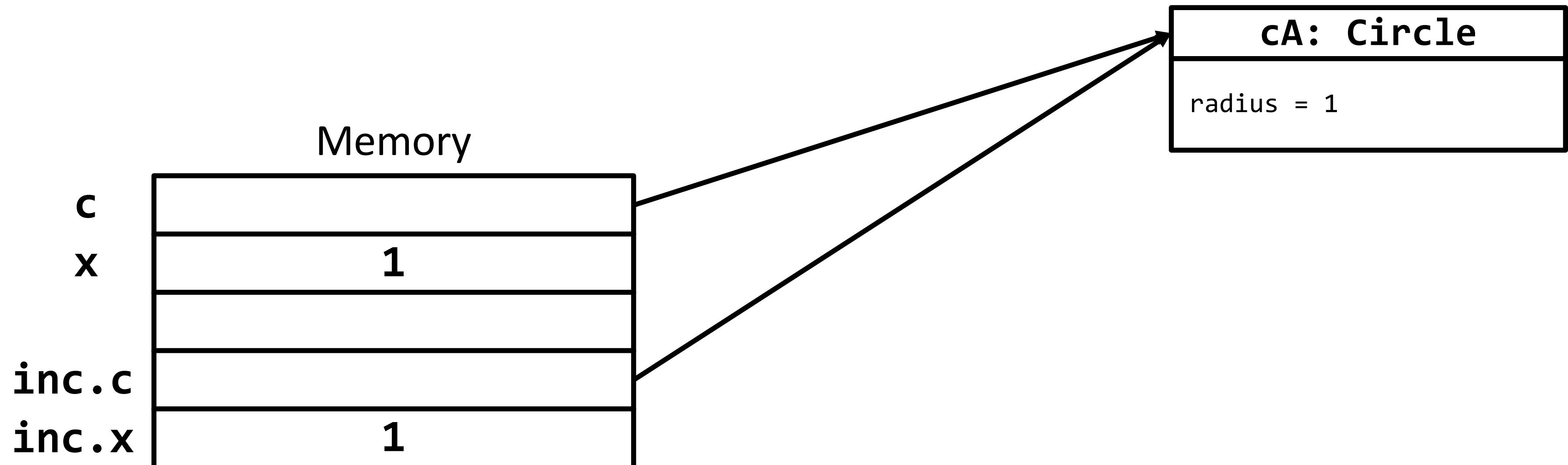
A Peek Into Memory (2)

```
int x = 1;
```



A Peek Into Memory (3)

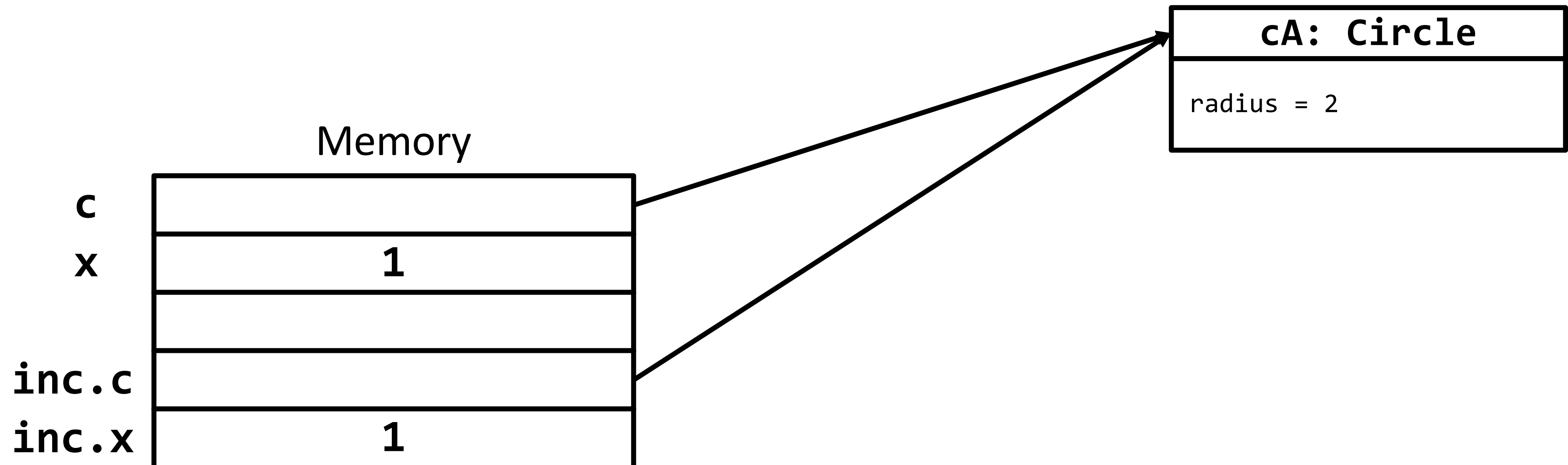
```
inc(c, x);
```



A Peek Into Memory (4)

in inc()

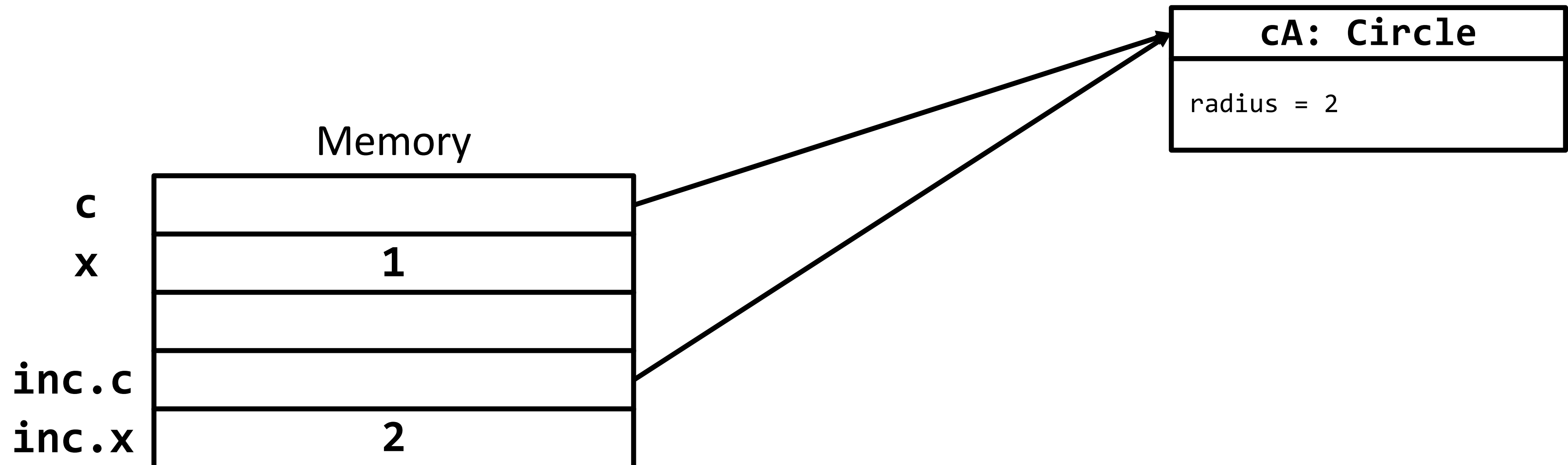
```
c.setRadius(c.getRadius()+1);
```



A Peek Into Memory (5)

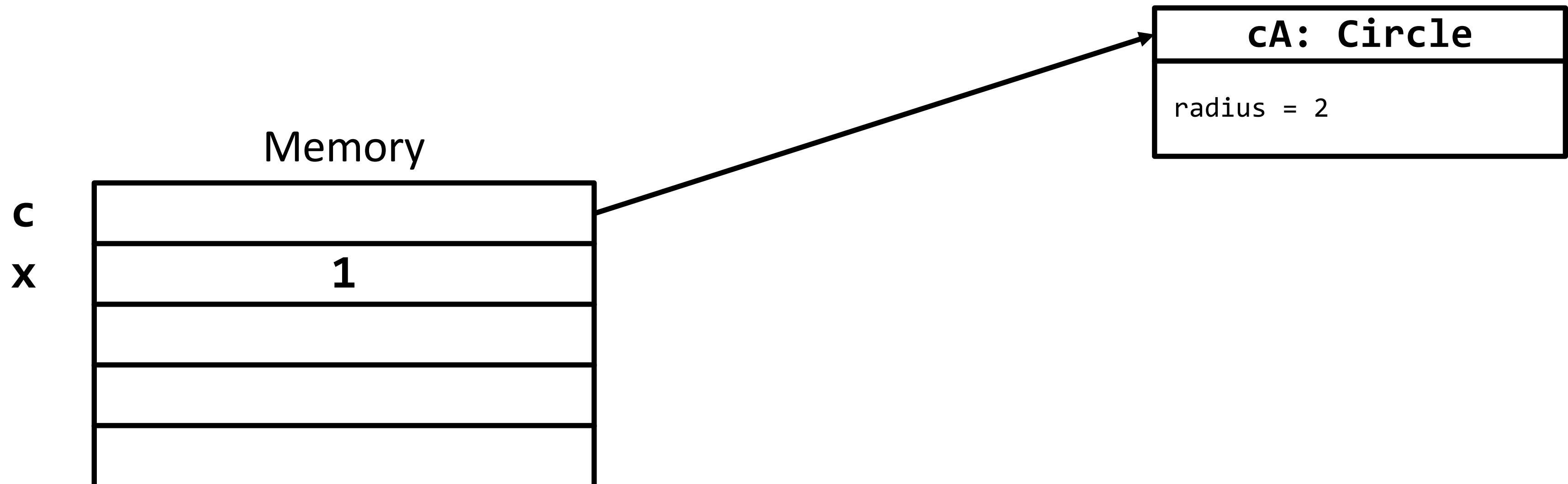
in inc()

x++;



A Peek Into Memory (5)

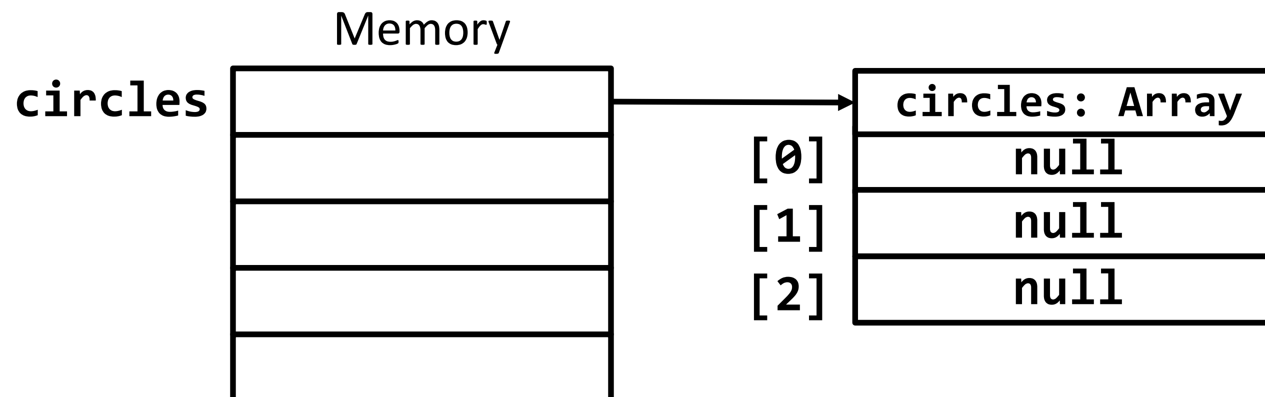
after inc()



Arrays of Objects

An array of objects is actually an array of reference variables

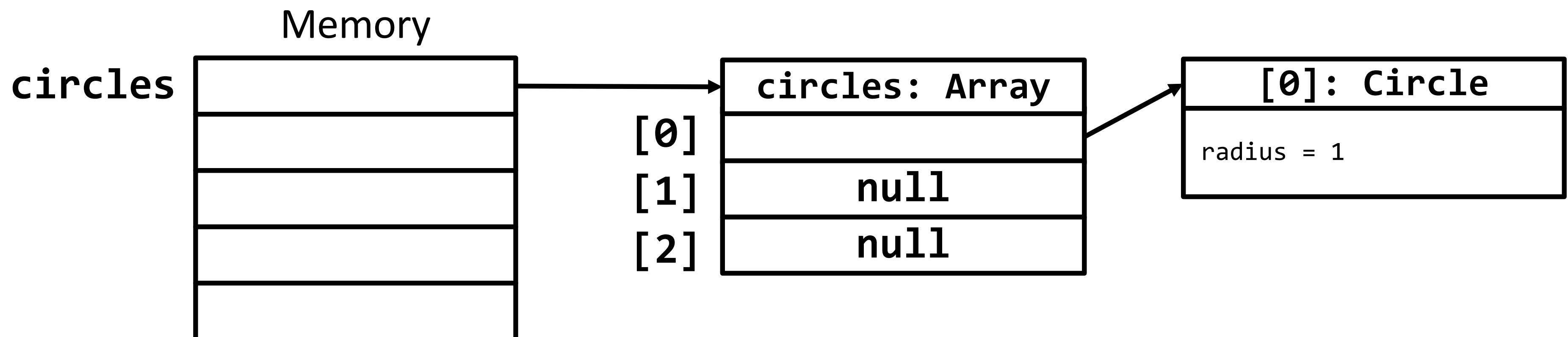
```
Circle[] circles = new Circle[3];
```



Arrays of Objects

An array of objects is actually an array of reference variables

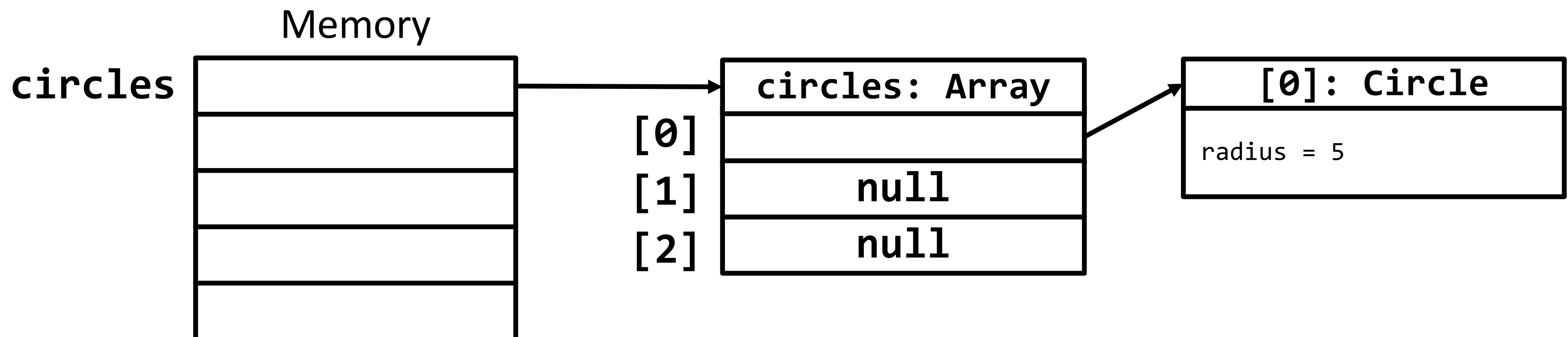
```
Circle[] circles = new Circle[3];
circles[0] = new Circle();
```



Arrays of Objects

An array of objects is actually an array of reference variables

```
Circle[] circles = new Circle[3];
circles[0] = new Circle();
circles[0].setRadius(5);
```



`this` Keyword

- Within an object method, `this` refers to the "current" object
- Common uses
  - Refer to private variables that have the same name as a parameter
  - Within a constructor, invoke another constructor of the same class

Example (1)

Circle.java

```
public class Circle {

	private double radius = 1.0;

	public Circle() {
	}

	public Circle(double radius) {
		setRadius(radius);
	}

	public double getRadius() {
		return radius;
	}

	public void setRadius(double radius) {
		if (radius>0) {
			this.radius = radius;
		}
	}

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

Notes

- If a method parameter has the same name as a member variable, the parameter name "hides" the member variable
- To access the member variable, use **`this.varName`**

Example (2)

Circle.java

```
public class Circle {

	private double radius;

	public Circle() {
		this(1.0);
	}

	public Circle(double radius) {
		setRadius(radius);
	}

	public double getRadius() {
		return radius;
	}

	public void setRadius(double radius) {
		if (radius>0) {
			this.radius = radius;
		}
	}

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

Notes

- The default constructor now calls the specialized constructor
- This makes sure all attempts to change the radius (via construction or user) pass through common validation, reducing the risk of error

The `toString` Method

- The **`toString`** method is used to return a string representation of an object
- This is useful when debugging and using terminal/file output on objects
- If the class does not have a method, a … less-than-useful string will be shown (more on how this works soon!)

Example (1)

Name.java

```
public class Name {
	final private String fName;
	final private String lName;

	public Name(String fName, String lName) {
		this.fName = fName;
		this.lName = lName;
	}
}
```

Anywhere

```
Name javaInventor = new Name("James", "Gosling");

System.out.printf("Java was invented by %s.%n",
	javaInventor);
```

Example (2)

Name.java

```
public class Name {
	final private String fName;
	final private String lName;

	public Name(String fName, String lName) {
		this.fName = fName;
		this.lName = lName;
	}

	public String toString() {
		return String.format("%s %s",
				fName, lName);
	}
}
```

Anywhere

```
Name javaInventor = new Name("James", "Gosling");

System.out.printf("Java was invented by %s.%n",
	javaInventor);
```

Take Home Points

- This lecture has covered many of the basic elements of classes and objects
- It is important to remember primitive vs. object memory organization, as it has effects on assignment, parameters, etc.