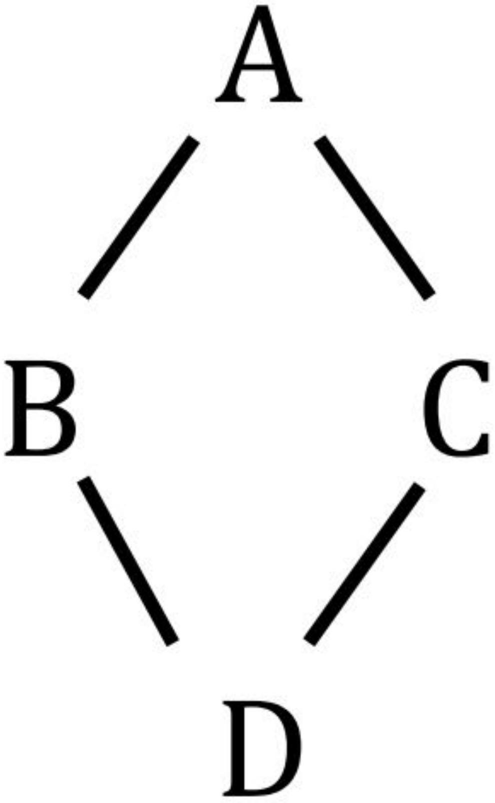
Interface

Moumita Asad
Lecturer
IIT, DU

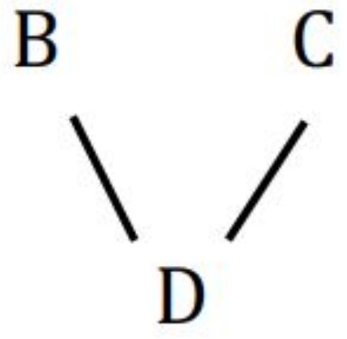
The Diamond Problem

- Suppose, both class B and C declare a method m() and class D calls m().
- Which method should be called, the one in B or the one in C?



The Diamond Problem

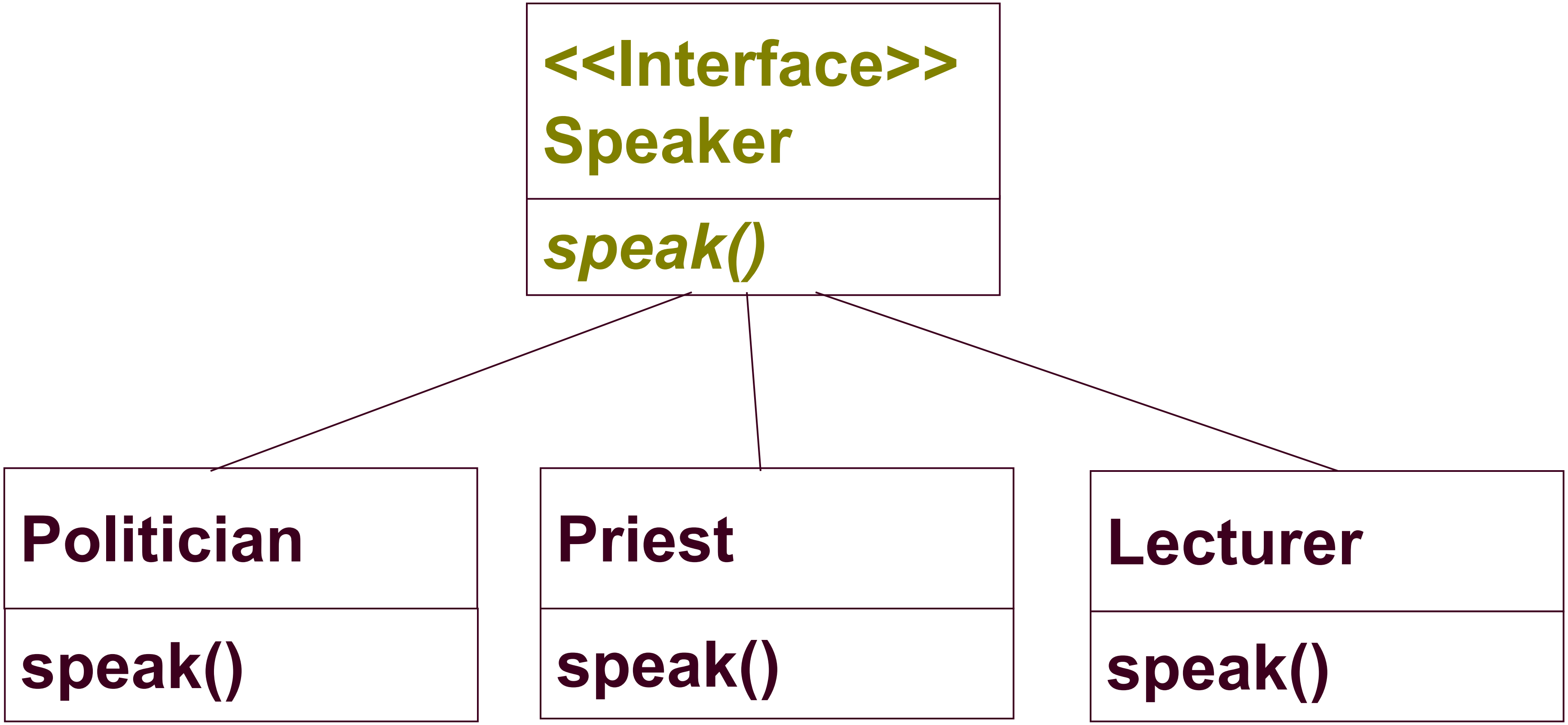
- For many Java cases, it’s not a diamond problem, it’s a Vee problem



Interface

- *Interface* is a conceptual entity similar to a Abstract class.
- Can contain only constants (final variables) and abstract method (no implementation) - Different from Abstract classes.
- Use when a number of classes share a common interface.
- Since interface only abstract methods and final fields/variables, it is the responsibility of the class that implements an interface to supply the code for methods.
- A class can implement any number of interfaces, but cannot extend more than one class at a time.
- Therefore, interfaces are considered as an informal way of realizing multiple inheritance in Java.

Interface - Example



<<Interface>>
Speaker

speak()

Politician

speak()

Priest

speak()

Lecturer

speak()

Interface - Example

- Syntax (appears like abstract class):

```
interface InterfaceName {
      // Constant/Final Variable Declaration
      // Methods Declaration
}
```

```
interface Speaker {
      public void speak( );
}
```

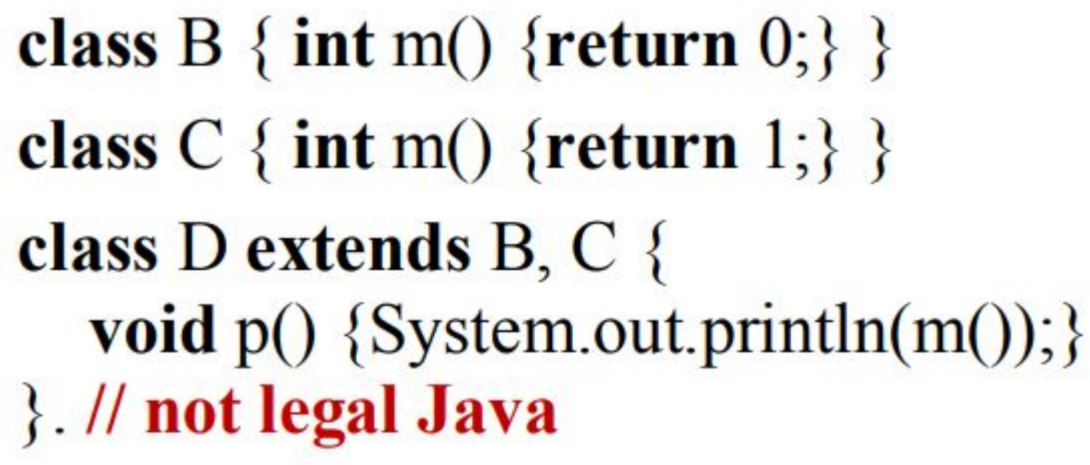
Interface - Example

```
class  Politician implements Speaker {
        public void speak(){
                System.out.println(“Talk politics”);
        }
}
```

```
class  Priest implements Speaker {
        public void speak(){
                System.out.println(“Religious Talks”);
        }
}
```

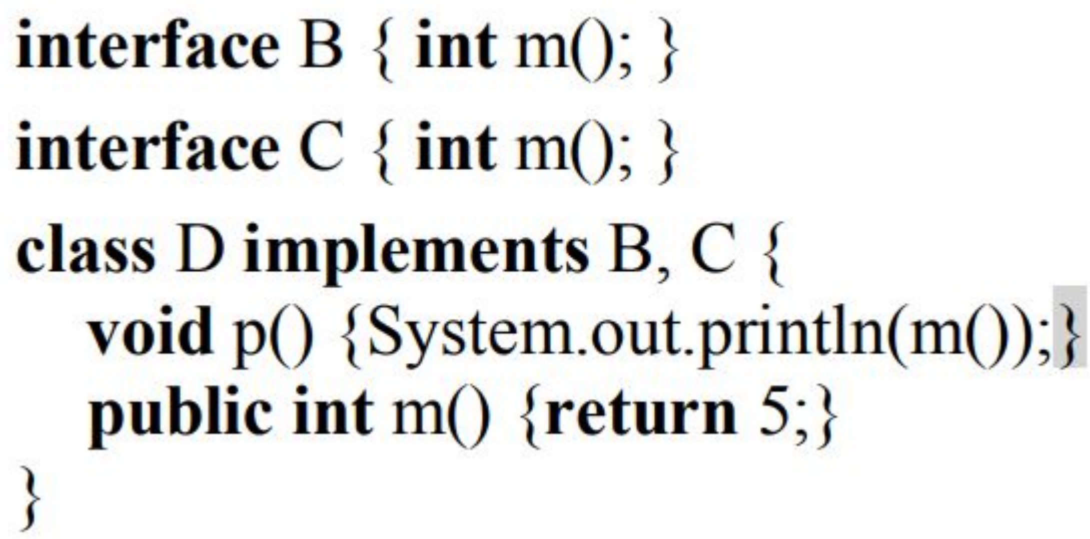
```
class  Lecturer implements Speaker {
        public void speak(){
                System.out.println(“Talks Object Oriented Programming!”);
        }
}
```

Multiple inheritance



```
class B { int m() {return 0;} }
class C { int m() {return 1;} }
class D extends B, C {
    void p() {System.out.println(m());}
}. // not legal Java
```

Multiple interface



```
interface B { int m(); }
interface C { int m(); }
class D implements B, C {
    void p() {System.out.println(m());}
    public int m() {return 5;}
}
```

Default Interface Methods

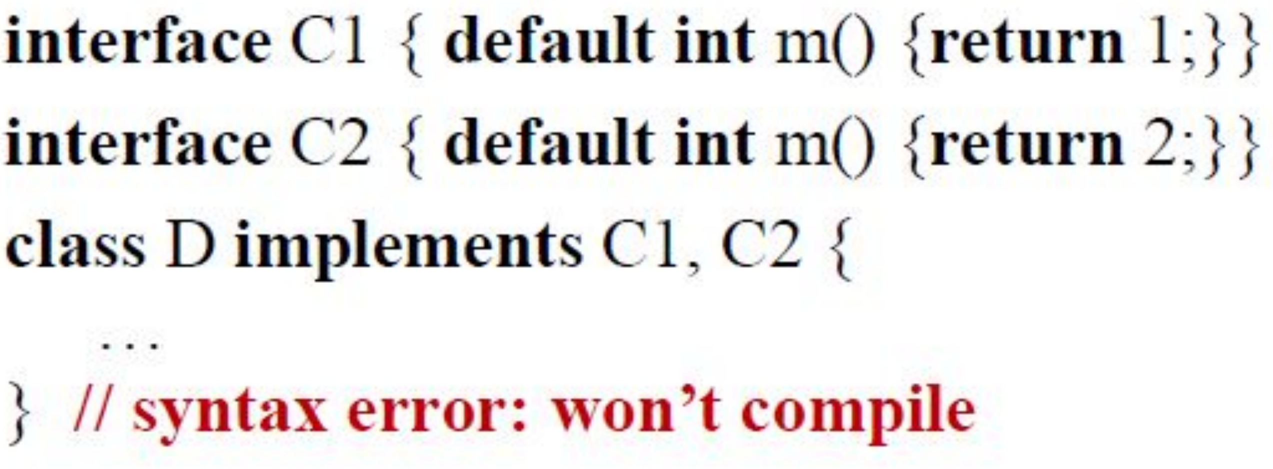
- Prior to JDK 8, an interface could not define any implementation
- By use of a default method, it is now possible for an interface method to provide a body, rather than being abstract
- Motivations:
  1. To provide a means by which interfaces could be expanded without breaking existing code.
     - In the past, if a new method were added to a popular, widely used interface, the addition of that method would break existing code because no implementation would be found for that new method
  2. To specify methods in an interface that are, essentially, optional, depending on how the interface is used.

Default Method Example

```
public interface MyIF {
   int getNumber();
   // This is a default method. Notice that it provides
   // a default implementation.
   default String getString() {
   return "Default String";
   }
}
class MyIFImp implements MyIF {
   public int getNumber() {
   return 100;
   }
}
```

Note

- □ In cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result.
- □ But if the class override the method, the class implementation takes priority over an interface default implementation.



```
interface C1 { default int m() {return 1;}}
interface C2 { default int m() {return 2;}}
class D implements C1, C2 {
    ...
} // syntax error: won't compile
```

Interface Variables

- Variables can be declared inside of interface declarations.
- They are implicitly final and static, meaning they cannot be changed by the implementing class.

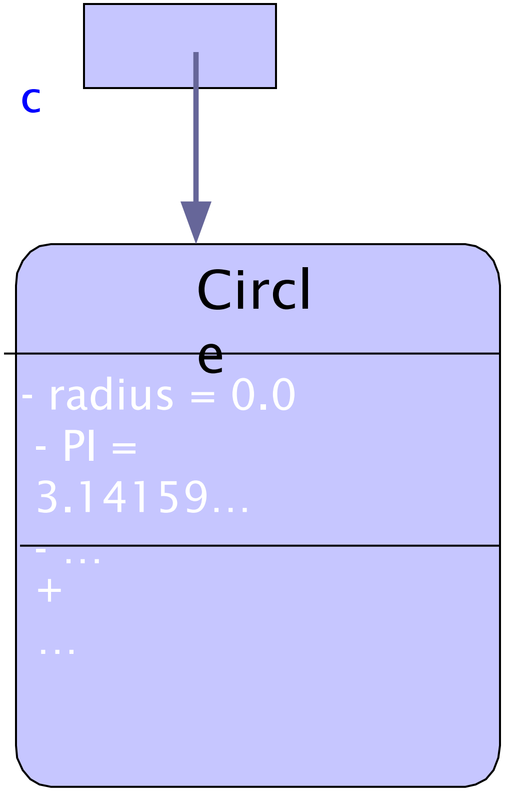
```
interface SampleInterface{
    int UPPER_LIMIT = 100;
}

public class InterfaceVariablesExample implements SampleInterface{
    public static void main(String[] args) {
        System.out.println("UPPER LIMIT = " + UPPER_LIMIT);
        // UPPER_LIMIT = 150; // Can not be modified
    }
}
```

UNDERSTANDING STATIC

Circle class

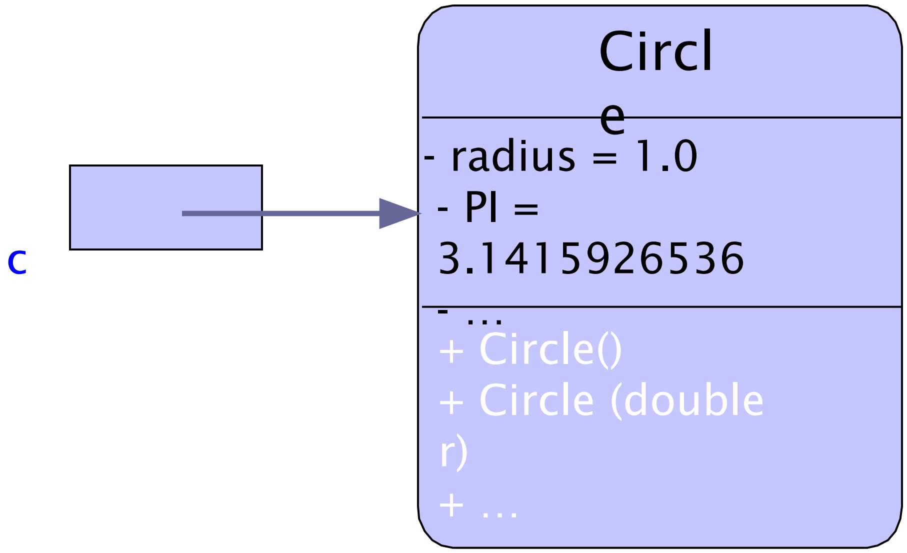
```
Circle c = new Circle();
```



```
public class Circle {
   double radius;
   double PI = 3.1415926536;
}
```

What happens in memory

- Consider: Circle c = new Circle();
- A double takes up 8 bytes in memory
- Thus, a Circle object takes up 16 bytes of memory
  - As it contains two doubles



c

Circle

- radius = 1.0

- PI = 3.1415926536

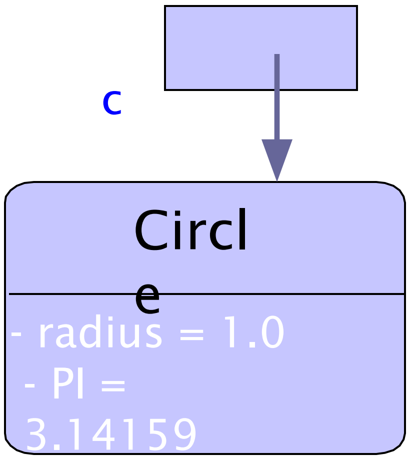
- …

+ Circle()

+ Circle (double r)

+ …

Shorthand representation



c

Circle

- radius = 1.0

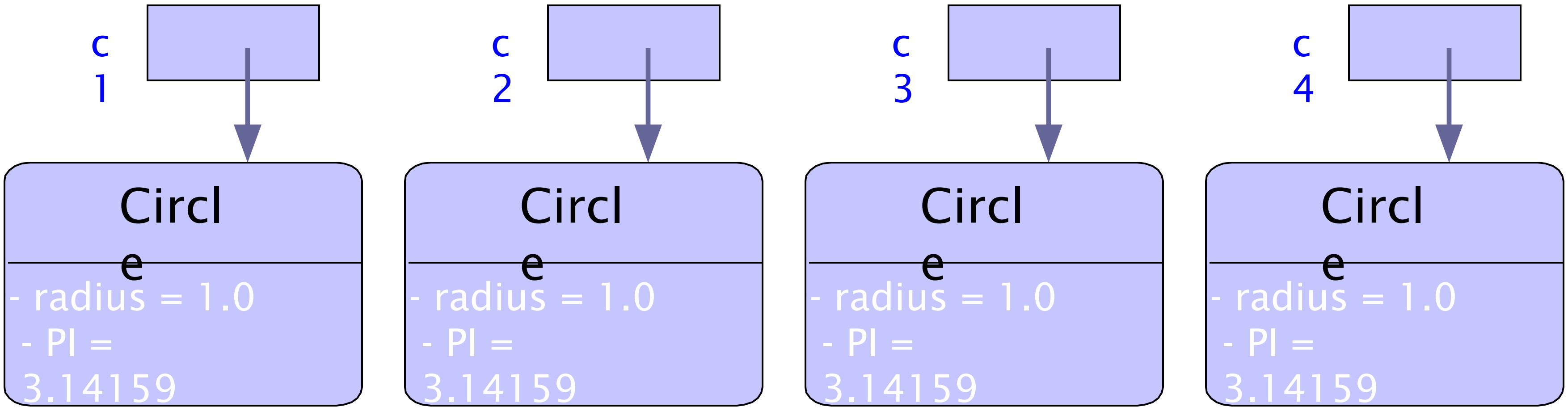
- PI = 3.14159

Consider the following code

```
public class CircleTest {
    public static void main (String[] args) {
        Circle c1 = new Circle();
        Circle c2 = new Circle();
        Circle c3 = new Circle();
        Circle c4 = new Circle();
    }
}
```

What happens in memory

- There are 4 Circle objects in memory
  - Taking up a total of 4*16 = 64 bytes of memory



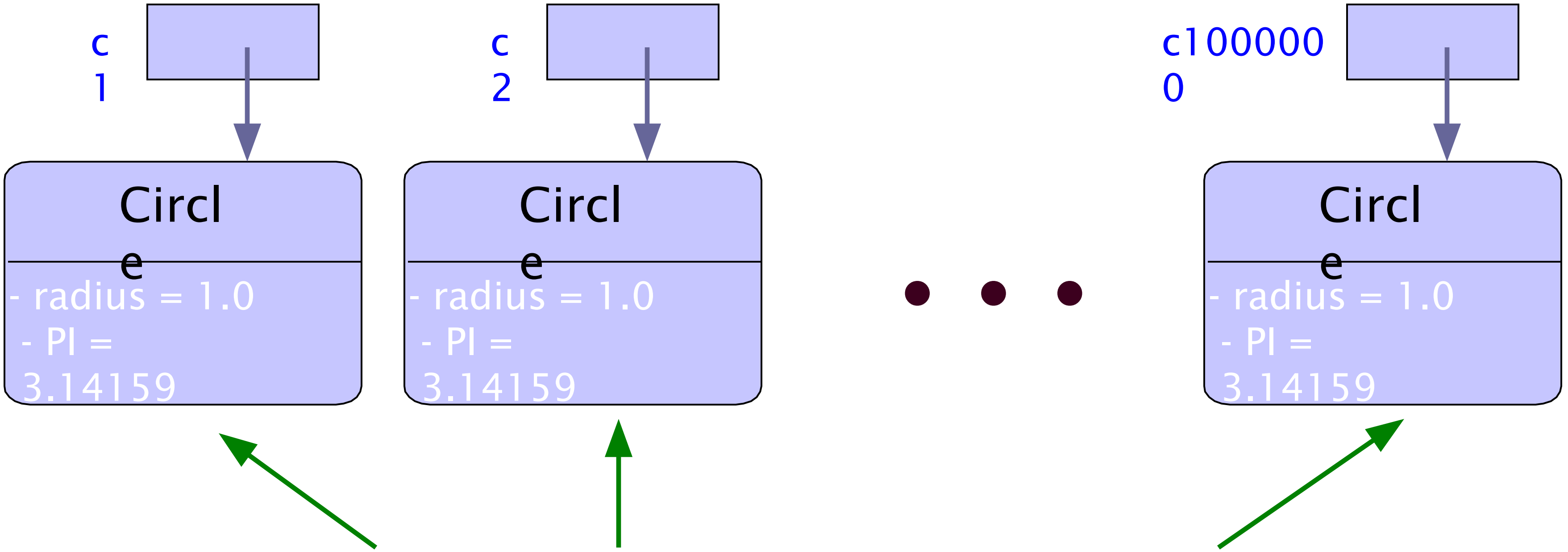
Consider the following code

```
public class CircleTest {
    public static void main (String[] args) {
        Circle c1 = new Circle();
        //...
        Circle c1000000 = new Circle();
    }
}
```

This program creates 1 million Circle objects!

What happens in memory

- There are 1 million Circle objects in memory
  - Taking up a total of 1,000,000*16 ≈ 16 Mb of memory



Note that the final PI field is repeated 1 million times

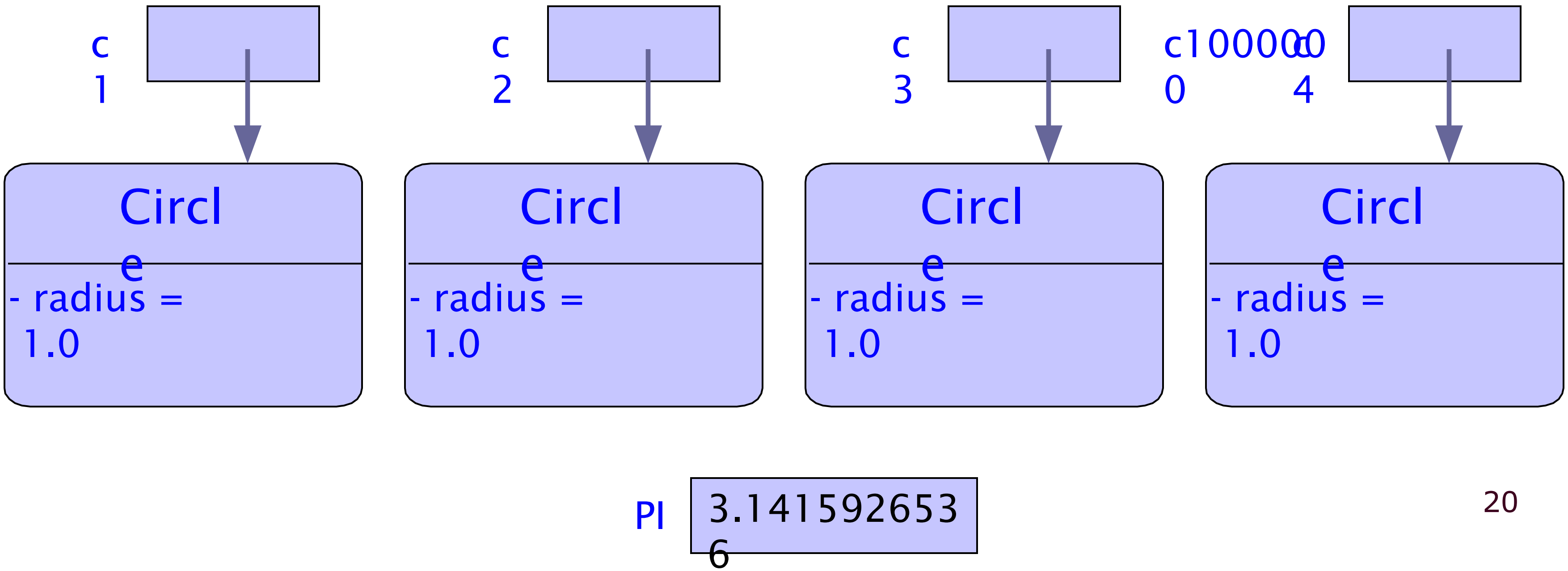
The use of `static` for fields

- If a variable is `static`, then there is only ONE of that variable for ALL the objects
  - That variable is shared by *all* the objects

Total memory usage: 8 Mb + 8 bytes (1,000,000+1 = 1,000,001 doubles)

Total memory usage: 40 bytes (4+1=5 doubles)

Total memory usage: 16 bytes (1+1=2 doubles)



Even more on `static` fields

- There is only one copy of a `static` field no matter how many objects are declared in memory
  - Even if there are zero objects declared!
  - The one field is "common" to all the objects

- Static variables are called class variables
  - As there is one such variable for all the objects of the class
  - Whereas non-static variables are called instance variables

- Thus, you can refer to a static field by using the class name:
  - Circle.PI

Even even more on `static` fields

- This program also prints 4.3:

```
Circle c1 = new Circle();
Circle c2 = new Circle();
Circle c3 = new Circle();
Circle c4 = new Circle();
Circle.PI = 4.3;
System.out.println (c2.PI);
```

Adding a method

```
public class Circle {
    double radius;
    final static double PI = 3.1415926536;

    // Constructors...

    double computeArea () {
       return PI*radius*radius;
    }

}
```

Note that a (non-static) method can use both instance and class variables

Using that method

```
public class CircleTest {
    public static void main (String[] args) {
        Circle c = new Circle();
        c.radius = 2.0;
        double area = c.computeArea();
        System.out.println (area);
  }
}
```

Prints 12.566370614356

Back to the `static` discussion

- Remember that there is one (and only one) static PI field, regardless of how many objects are declared

- Consider the following method:

```
double getPI() {
  return PI;
}
```

- It doesn’t read or modify the “state” of any object
  - In this example, it doesn’t read/write the radius

- In fact, that particular method doesn’t care anything about the objects declared
  - It’s only accessing a `static` field

Make getPI() static

- Consider the following:

```
static double getPI() {
    return PI;
  }
```

- As the method is static, it can ONLY access static fields

- A static method does not care about the "state" of an object
  - Examples: Math.sin(), Math.tan(), Math.cos()
    - They don't care about the state of any Math object
    - They only perform the computation

Invoking static methods

- As with `static` fields, they can be called using either an object or the class name:

```
Circle c = new Circle();
System.out.println (c.getPI());
System.out.println (Circle.getPI());
```

- Static methods are also called class methods

`static` methods and non-`static` fields

- Consider the following (illegal) Circle method:

```
static double getRadius() {
    return radius;
  }
```

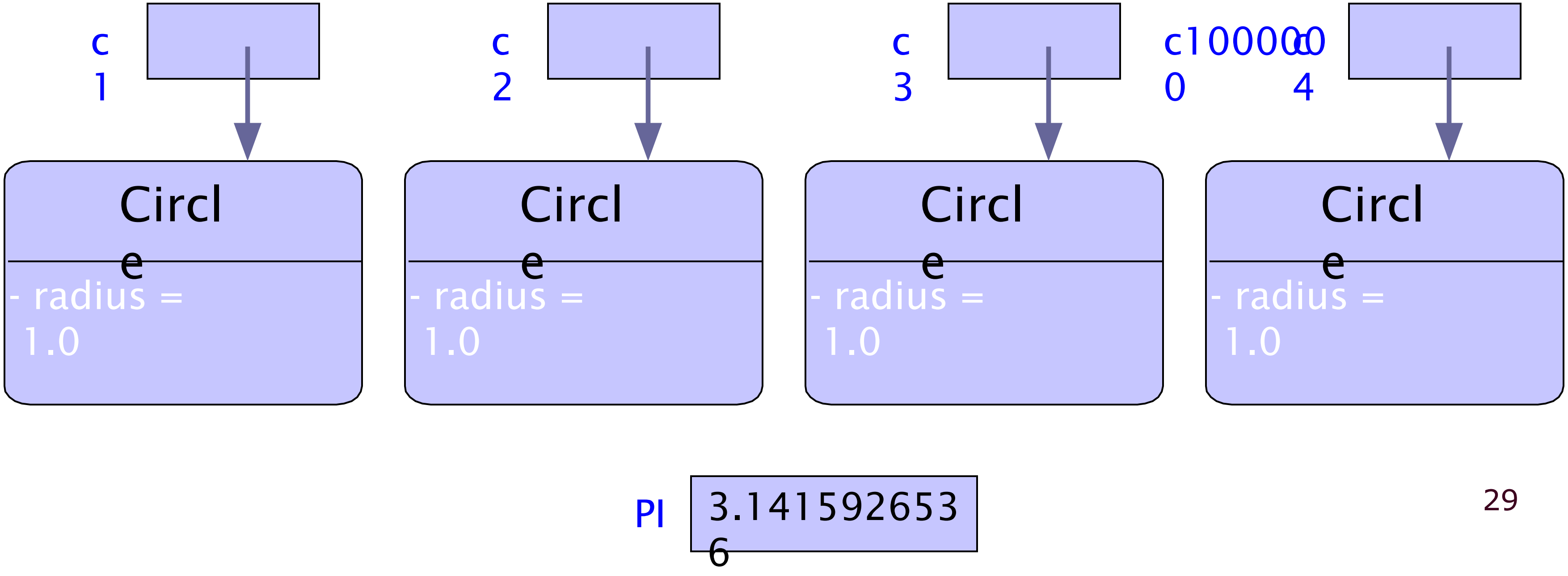
- And the code to invoke it:

```
public static void main (String[] args) {
  Circle c1 = new Circle();
  Circle c2 = new Circle();
  Circle c3 = new Circle();
  Circle c4 = new Circle();
  System.out.println (Circle.getRadius());
}
```

What happening in memory

- There are 4 Circle objects in memory
  1000000 Circle objects in memory

- Which radius field does Circle.getRadius() want?



static and non-static rules

- Non-static fields and methods can ONLY be accessed by the object name
- Static fields and methods can be accessed by EITHER the class name or the object name
- Non-static methods can refer to BOTH static and non-static fields
- Static methods can ONLY access static fields of the class they are part of