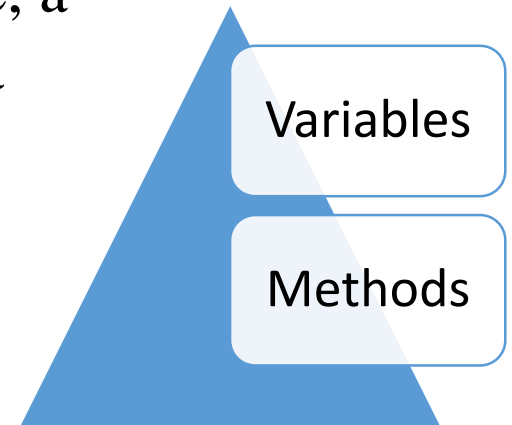


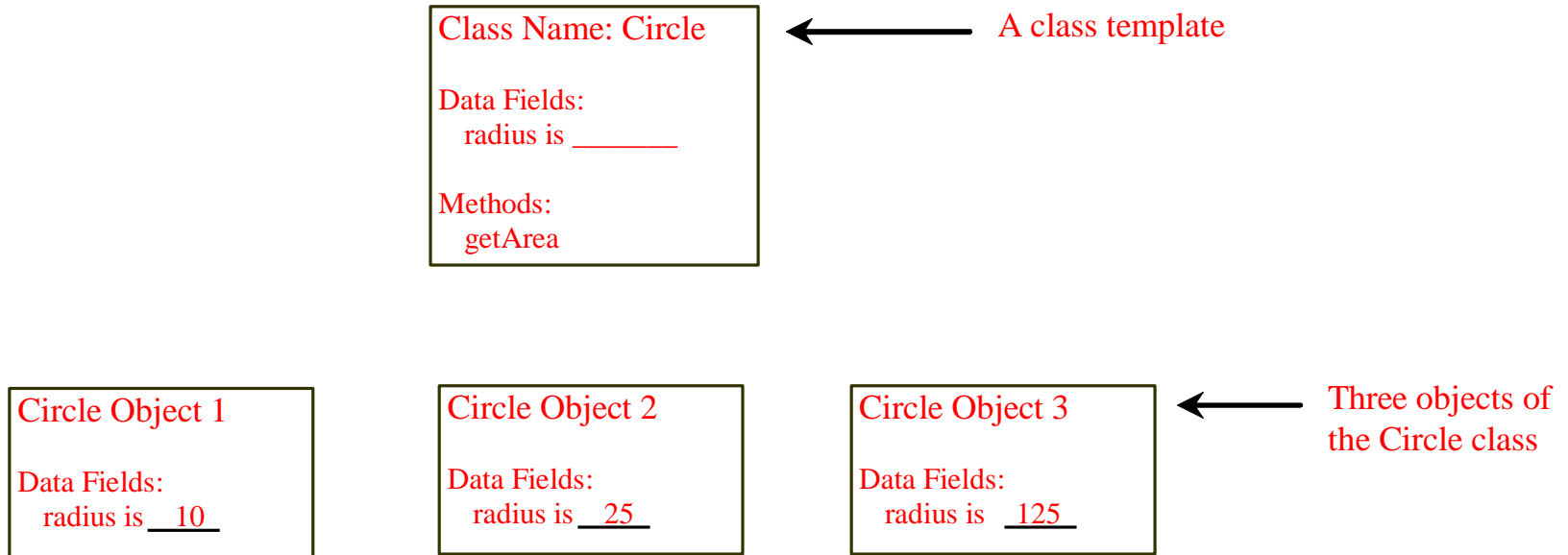
Objects and Classes

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.



Objects



An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

Classes

Classes are constructs that define objects of the same type. A Java class uses variables to define data fields and methods to define behaviors. Additionally, a class provides a special type of methods, known as constructors, which are invoked to construct objects from the class.

Classes

```
class Circle {  
    /** The radius of this circle */  
    double radius = 1.0;
```

← Data field

```
    /** Construct a circle object */  
    Circle() {  
    }
```

```
    /** Construct a circle object */  
    Circle(double newRadius) {  
        radius = newRadius;  
    }
```

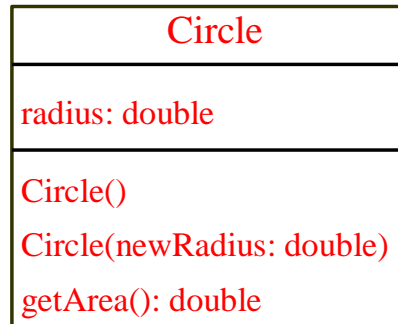
← Constructors

```
    /** Return the area of this circle */  
    double getArea() {  
        return radius * radius * 3.14159;  
    }  
}
```

← Method

UML Class Diagram

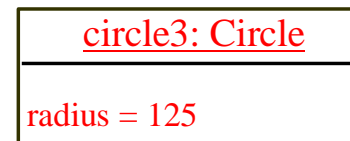
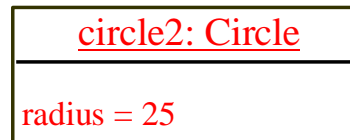
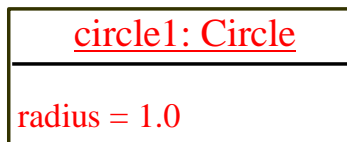
UML Class Diagram



← Class name

← Data fields

← Constructors and methods



← UML notation for objects

Circle, TestCircle, TV, TestTV

Constructors

```
Circle() {  
}
```

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

Constructors are a special kind of methods
that are invoked to construct objects.

Constructors, cont.

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.

Creating Objects Using Constructors

```
new ClassName ( ) ;
```

Example:

```
new Circle ( ) ;
```

```
new Circle (5.0) ;
```

Default Constructor

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

Declaring Object Reference Variables

To reference an object, assign the object to a reference variable.

To declare a reference variable, use the syntax:

```
ClassName objectRefVar;
```

Example:

```
Circle myCircle;
```

Declaring/Creating Objects in a Single Step

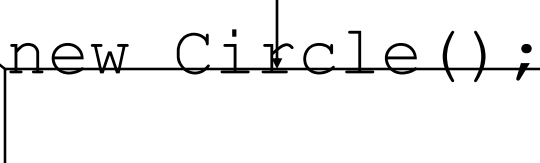
```
ClassName objectRefVar = new ClassName();
```

Example:

Assign object reference

Create an object

```
Circle myCircle = new Circle();
```



The diagram consists of a line starting from the text 'Assign object reference', which branches into two paths. One path leads to the variable 'myCircle' in the code 'Circle myCircle = new Circle();'. The other path leads to the 'new' keyword. A second line starts from the text 'Create an object', which leads to the 'new' keyword. A rectangular box is drawn around the 'new Circle()' part of the code.

Accessing Objects

- Referencing the object's data:

`objectRefVar.data`

e.g., `myCircle.radius`

- Invoking the object's method:

`objectRefVar.methodName (arguments)`

e.g., `myCircle.getArea()`

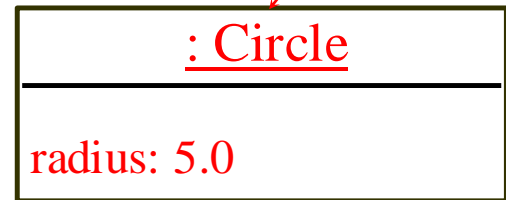
Trace Code, cont.

```
Circle myCircle = new Circle(5.0);
```

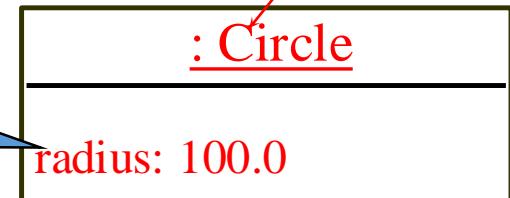
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle **reference value**



yourCircle **reference value**



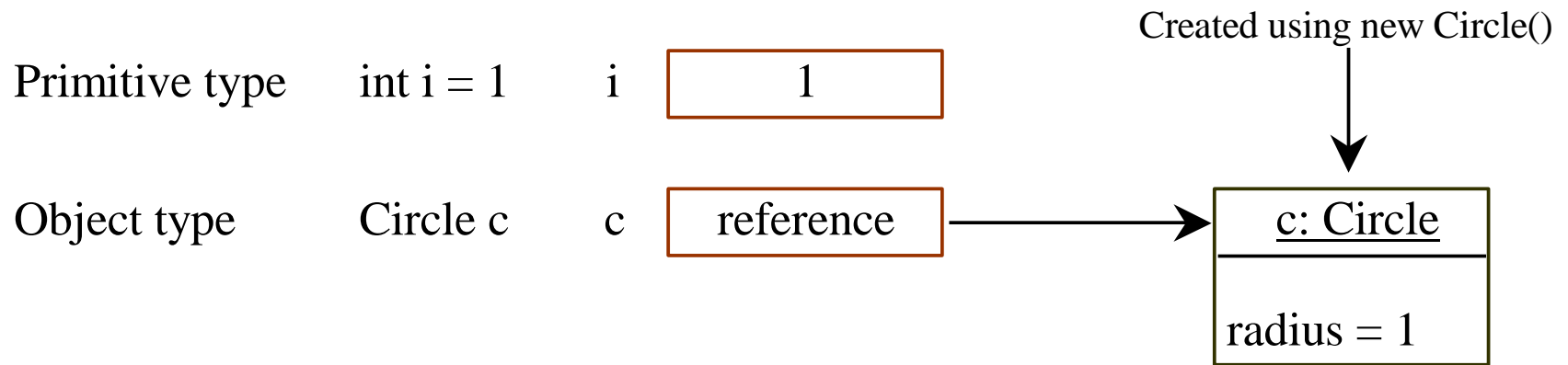
Change radius in
yourCircle

Default Value for a Data Field

The default value of a data field is null for a reference type, 0 for a numeric type, false for a boolean type, and '\u0000' for a char type. However, Java assigns no default value to a local variable inside a method.

```
public class Test {  
    public static void main(String[] args) {  
        Student student = new Student();  
        System.out.println("name? " + student.name);  
        System.out.println("age? " + student.age);  
        System.out.println("isScienceMajor? " + student.isScienceMajor);  
        System.out.println("gender? " + student.gender);  
    }  
}
```

Differences between Variables of Primitive Data Types and Object Types



Copying Variables of Primitive Data Types and Object Types

Primitive type assignment $i = j$

Before:

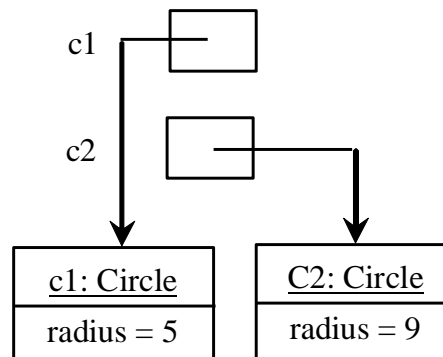


After:

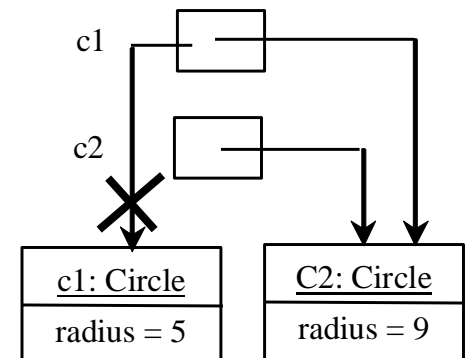


Object type assignment $c1 = c2$

Before:



After:



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 .

The Random Class

You have used Math.random() to obtain a random double value between 0.0 and 1.0 (excluding 1.0). A more useful random number generator is provided in the java.util.Random class.

java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (exclusive).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (exclusive).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (exclusive).
+nextBoolean(): boolean	Returns a random boolean value.

Instance Variables, and Methods

Instance variables belong to a specific instance.

Instance methods are invoked by an instance of the class.

Static Variables, Constants, and Methods

Static variables are shared by all the instances of the class.

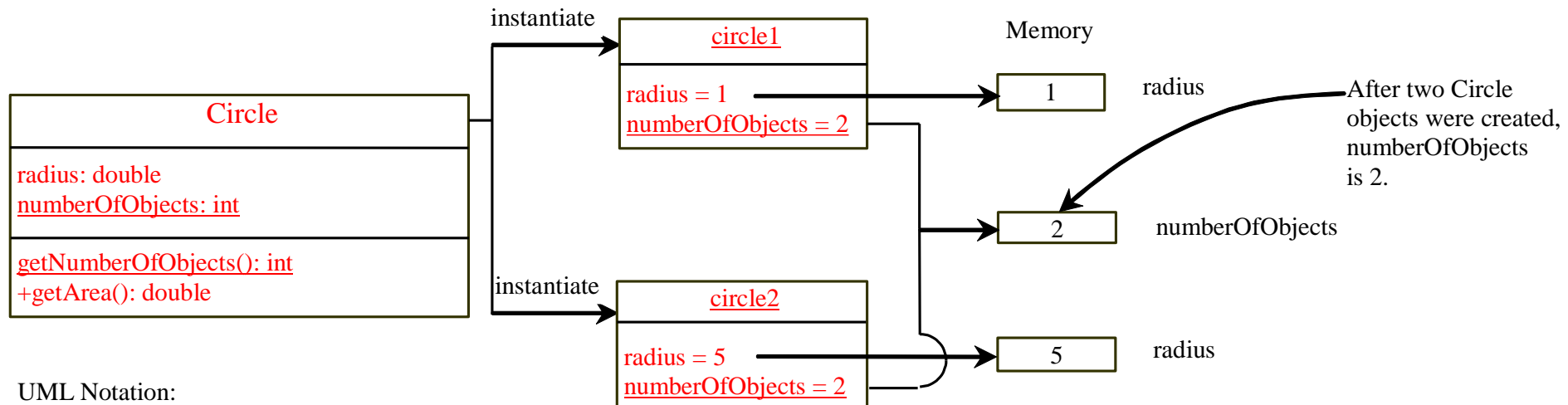
Static methods are not tied to a specific object.

Static constants are final variables shared by all the instances of the class.

Static Variables, Constants, and Methods, cont.

To declare static variables, constants, and methods,
use the static modifier.

Static Variables, Constants, and Methods, cont.



UML Notation:

+: public variables or methods
underline: static variables or methods

Scope of Variables

- The scope of instance and static variables is the entire class. They can be declared anywhere inside a class.
- 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.

Access/Visibility Modifiers and Accessor/Mutator Methods Getter/Setter methods

By default, the class, variable, or method can be accessed by any class in the same package.

☞ `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.

The get and set methods are used to read and modify private properties.

package p1;

```
class C1 {  
    public int x;  
    int y;  
    private int z;  
  
    public void m1() {  
    }  
    void m2() {  
    }  
    private void m3() {  
    }  
}
```

```
public class C2 {  
    void aMethod() {  
        C1 o = new C1();  
        can access o.x;  
        can access o.y;  
        cannot access o.z;  
  
        can invoke o.m1();  
        can invoke o.m2();  
        cannot invoke o.m3();  
    }  
}
```

package p2;

```
public class C3 {  
    void aMethod() {  
        C1 o = new C1();  
        can access o.x;  
        cannot access o.y;  
        cannot access o.z;  
  
        can invoke o.m1();  
        cannot invoke o.m2();  
        cannot invoke o.m3();  
    }  
}
```

package p1;

```
class C1 {  
    ...  
}
```

```
public class C2 {  
    can access C1  
}
```

package p2;

```
public class C3 {  
    cannot access C1;  
    can access C2;  
}
```

The private modifier restricts access to within a class, the default modifier restricts access to within a package, and the public modifier enables unrestricted access.

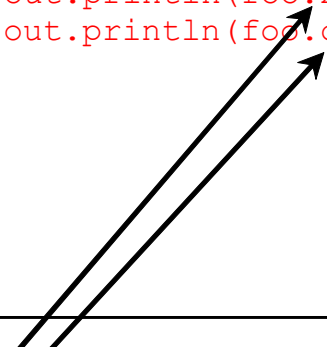
NOTE

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).

```
public class Foo {  
    private boolean x;  
  
    public static void main(String[] args) {  
        Foo foo = new Foo();  
        System.out.println(foo.x);  
        System.out.println(foo.convert());  
    }  
  
    private int convert(boolean b) {  
        return x ? 1 : -1;  
    }  
}
```

(a) This is OK because object foo is used inside the Foo class

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



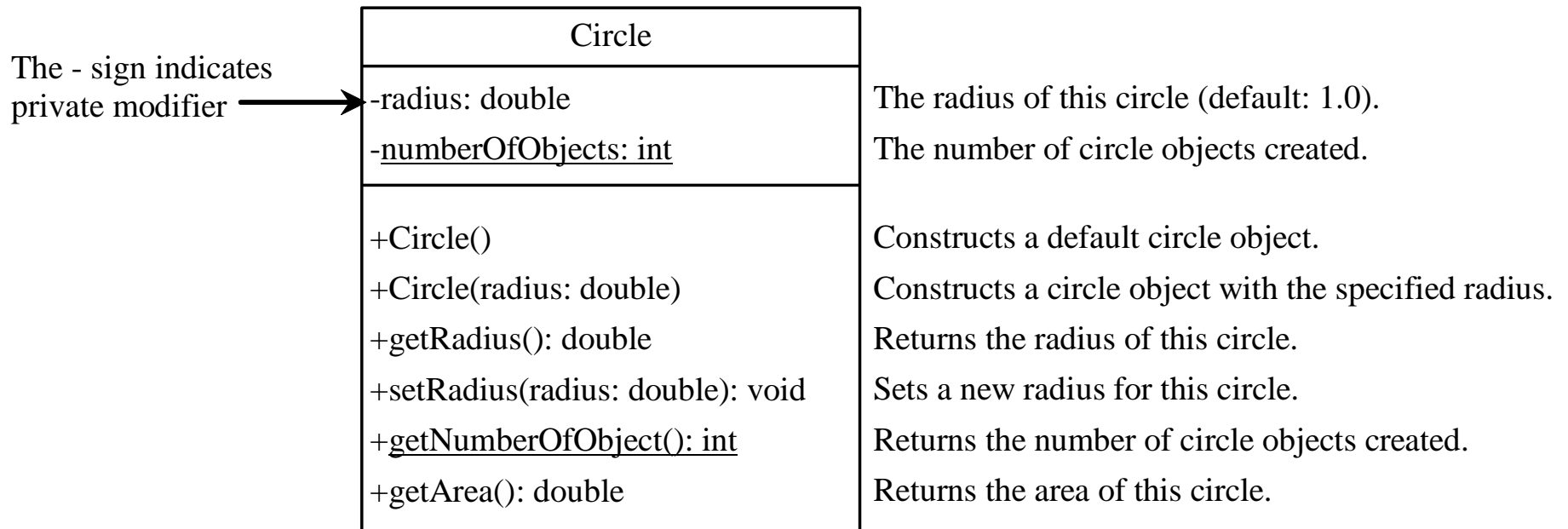
(b) This is wrong because x and convert are private in Foo.

Why Data Fields Should Be private?

To protect data.

To make class easy to maintain.

Example of Data Field Encapsulation



Circle3

TestCircle3

The *this* Keyword

Reference the Hidden Data Fields

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*.

```
public class Foo {  
    private int i = 5;  
    private static double k = 0;  
  
    void setI(int i) {  
        this.i = i;  
    }  
  
    static void setK(double k) {  
        Foo.k = k;  
    }  
}
```

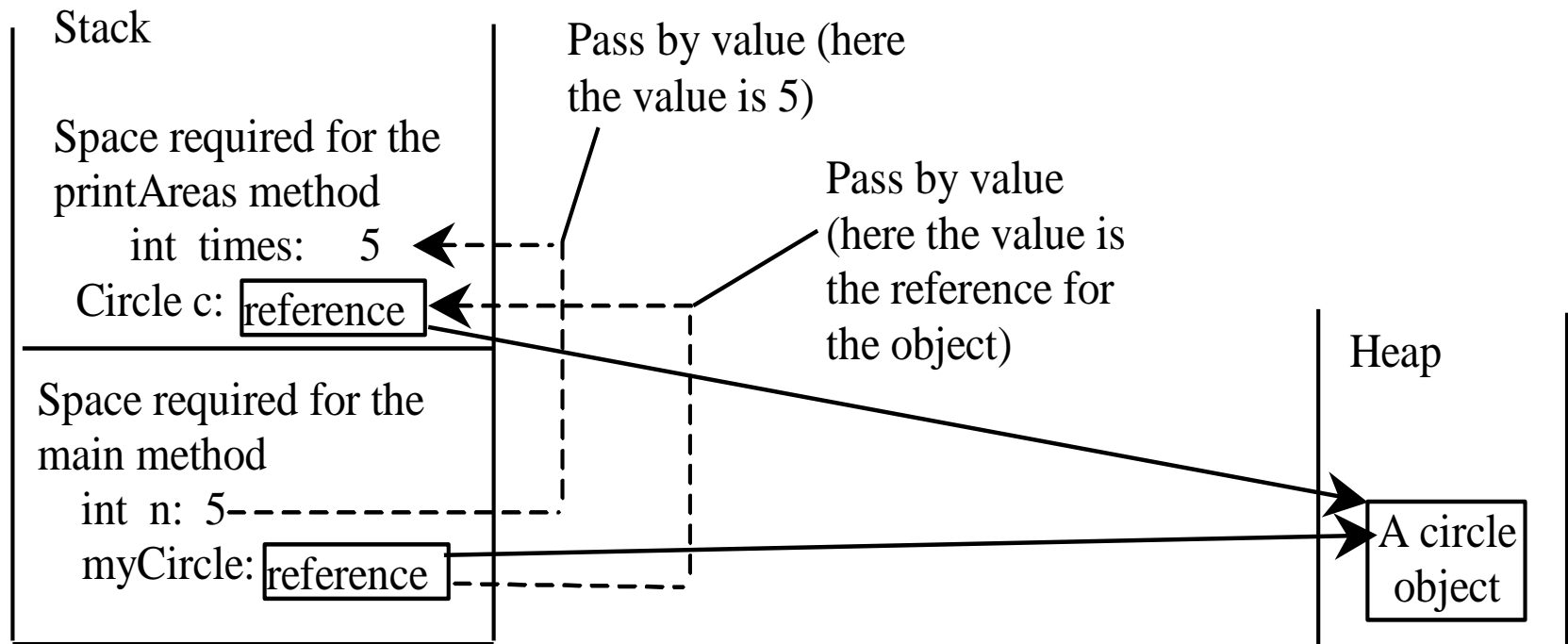
Suppose that f1 and f2 are two objects of Foo.

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

Passing Objects to Methods

- ➡ Passing by value for primitive type value (the value is passed to the parameter)
- ➡ Passing by value for reference type value (the value is the reference to the object)



Array of Objects

```
Circle[] circleArray = new Circle[10];
```

An array of objects is actually an *array of reference variables*. So invoking `circleArray[1].getArea()` involves two levels of referencing as shown in the next figure. `circleArray` references to the entire array. `circleArray[1]` references to a `Circle` object.

Array of Objects, cont.

```
Circle[] circleArray = new Circle[10];
```

