UNSW Business School
Information Systems and Technology Management

INFS1609/2609 Fundamentals of Business Programming

Lecture 8
Object-oriented Programming!

yenni.tim@unsw.edu.au

INFS1609/2609 Fundamentals of Business Programming

Topics for this week:

- To describe **objects** and **classes**, and use classes to model objects
- To create objects using **constructors**
- To access an object's data and methods using the **object member access operator** (.)
- To use the keyword **this** to refer to the calling object itself
- To distinguish between **instance** and **static** variables and methods
- Visibility modifiers
- To learn about encapsulation and the use of getter and setter

Main references
- *Textbook: Chapter 9*
- *Other online references posted on Ed*

Java is a Object-oriented Programming Language

- 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

Java is a Object-oriented Programming Language

- 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* (behavior defines what the object *does*)

Java is a Object-oriented Programming Language

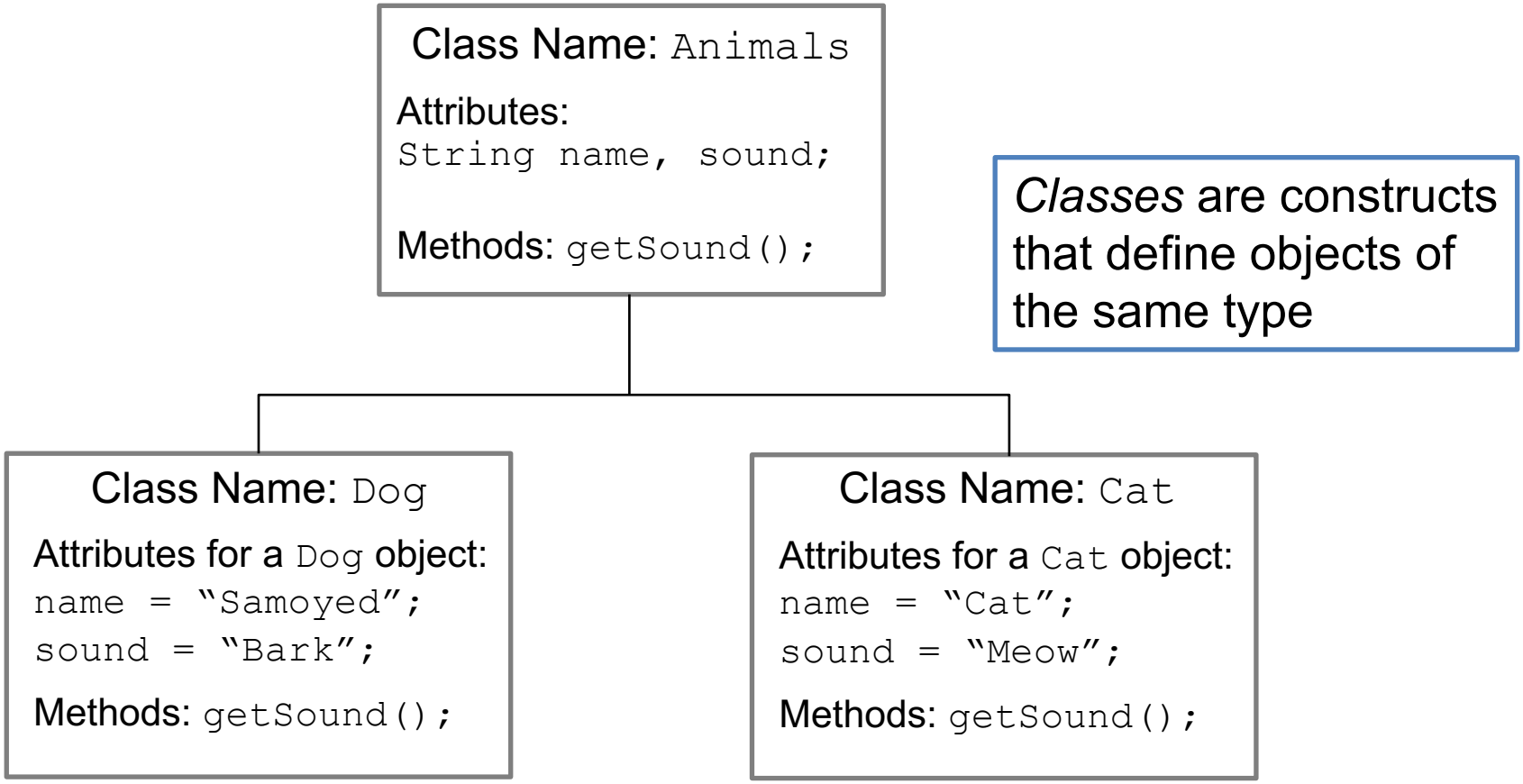
- A class is a blueprint, a construct that define objects of the same type

A Class: Blueprint (e.g., a recipe for a cake)

An instance of the class = **An Object**
= A cake that you baked based on the recipe

You can bake as many cakes as you want using the same recipe

Object-oriented Programming (OOP)

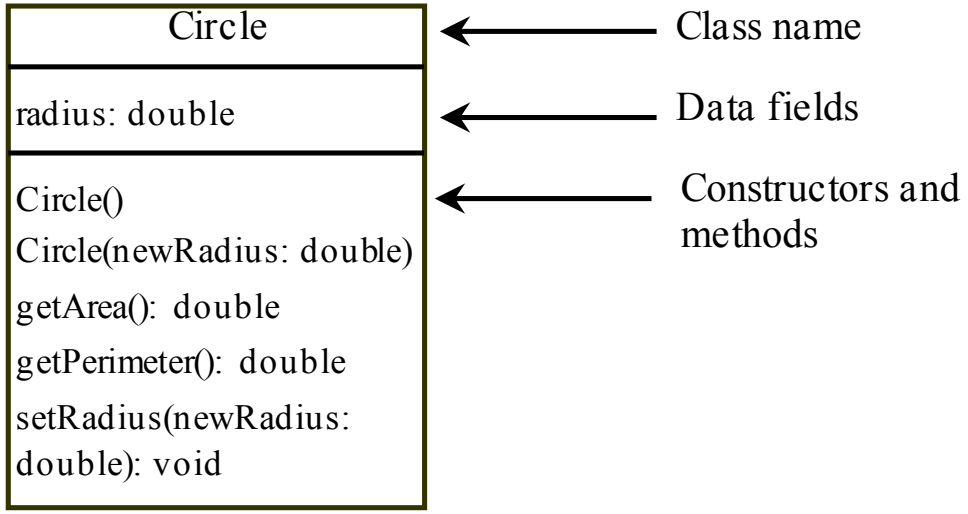


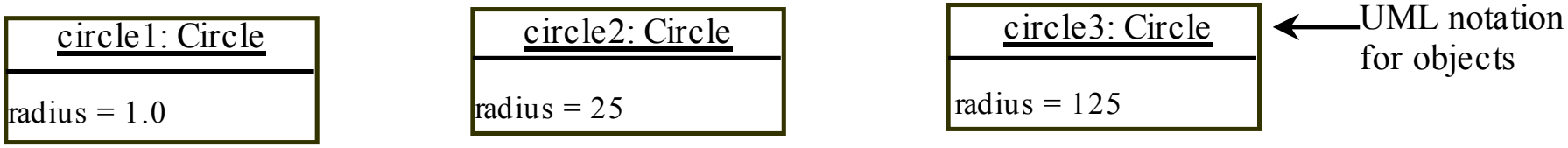
Classes

- A Java class uses **variables** to define **data fields** and **methods** to define **behaviors**
- A class provides a special type of methods, known as **constructors**, which are invoked to construct objects from the class

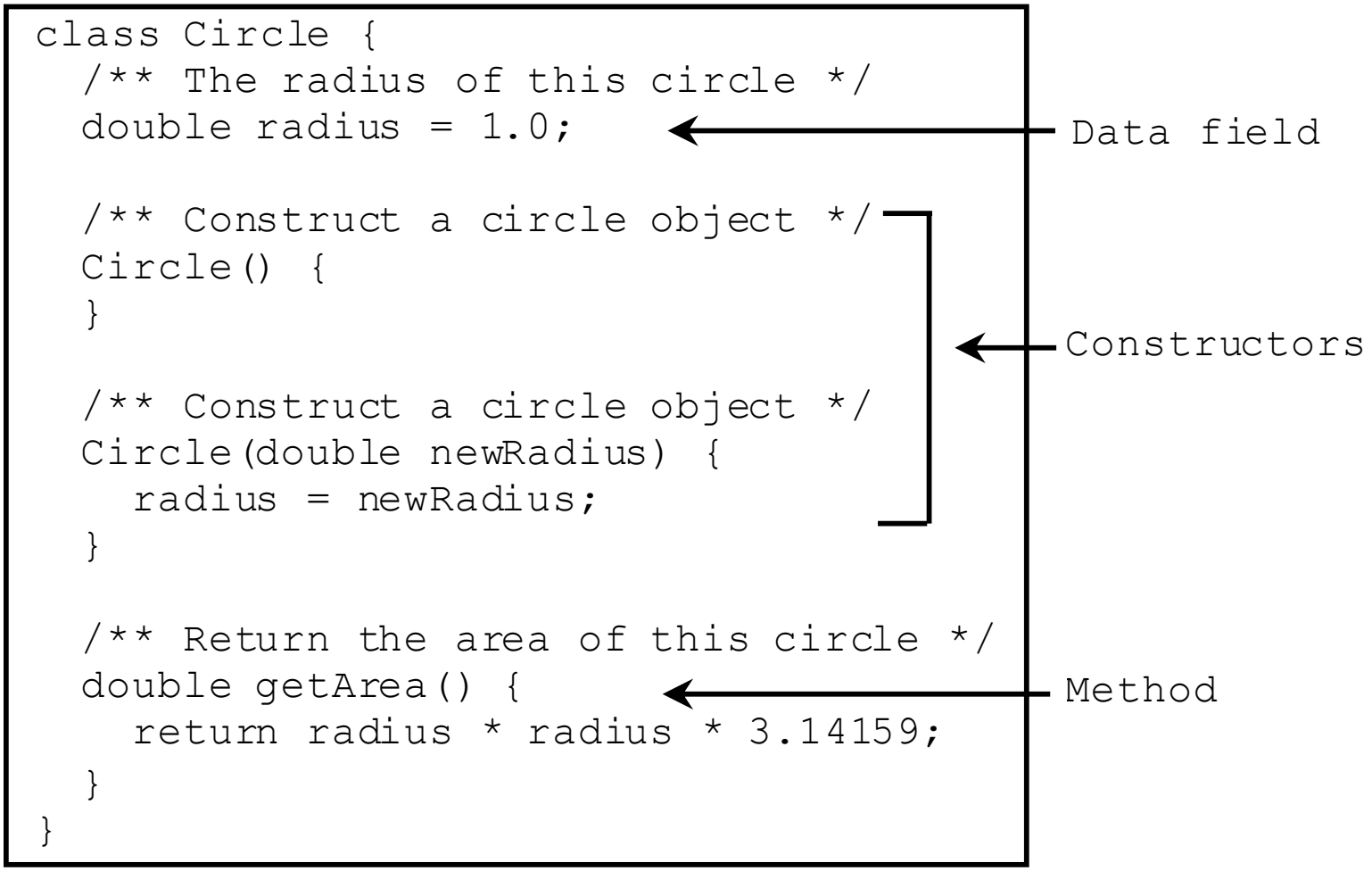
UML Class Diagram

UML Class Diagram





INFS1609/2609 Fundamentals of Business Programming



Constructors

- Constructors must have the **same name** as the class
- Constructors do not have a return type – **not even void**
- Constructors play the role of initializing objects: **invoked using the `new` operator**

```
Circle() {
}

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

Example:

```
new Circle();

new Circle(5.0);
```

Constructors

- A class may be defined without constructors – in this case a `no-arg` **default constructor** with an empty body is implicitly defined in the class

Declaring an object reference variables

- To declare a reference variable, use the syntax:

```
ClassName objectRefVar;
```

- Example:

```
Circle myCircle;
```

Declaring and creating objects in one step:

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

Example:

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

Accessing object's attributes and methods (".")

- Referencing the object's data:

```
objectRefVar.data
```

e.g., `myCircle.radius`

- Invoking the object's method:

```
objectRefVar.methodName(arguments)
```

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

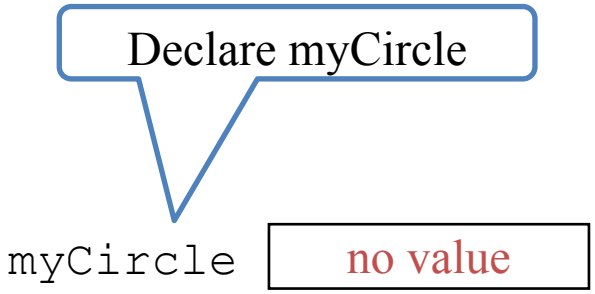
Accessing Object's attributes and methods (".")

(Trace code)

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



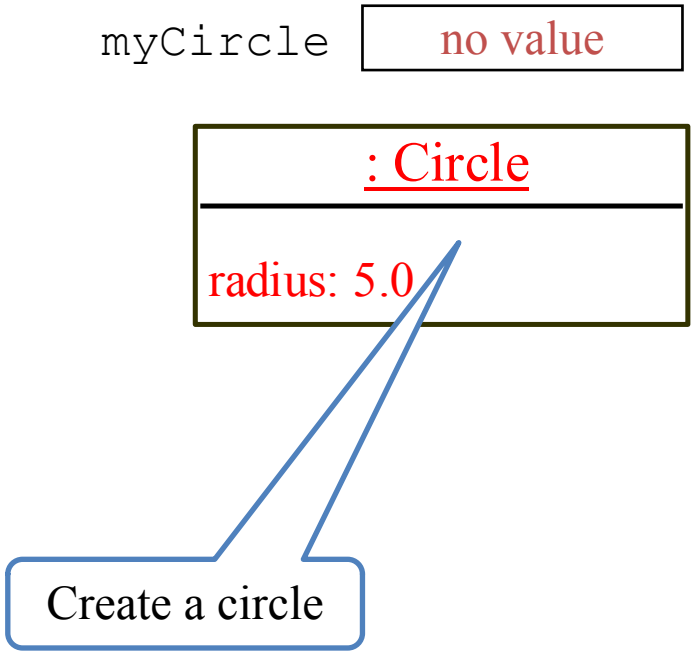
Accessing Object's attributes and methods (".")

(Trace code)

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



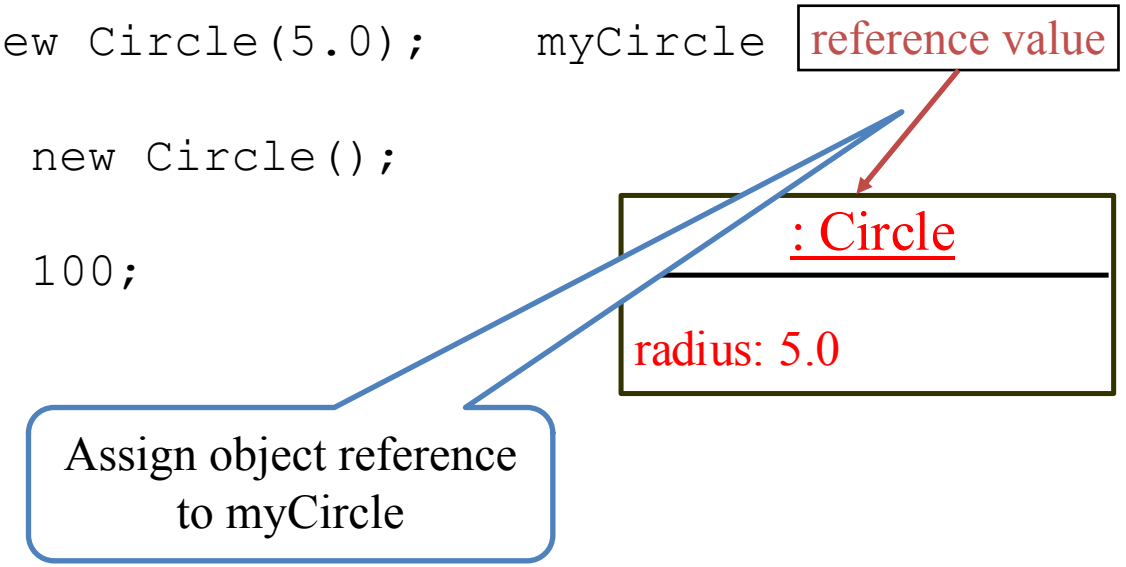
Accessing Object's attributes and methods (".")

(Trace code)

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



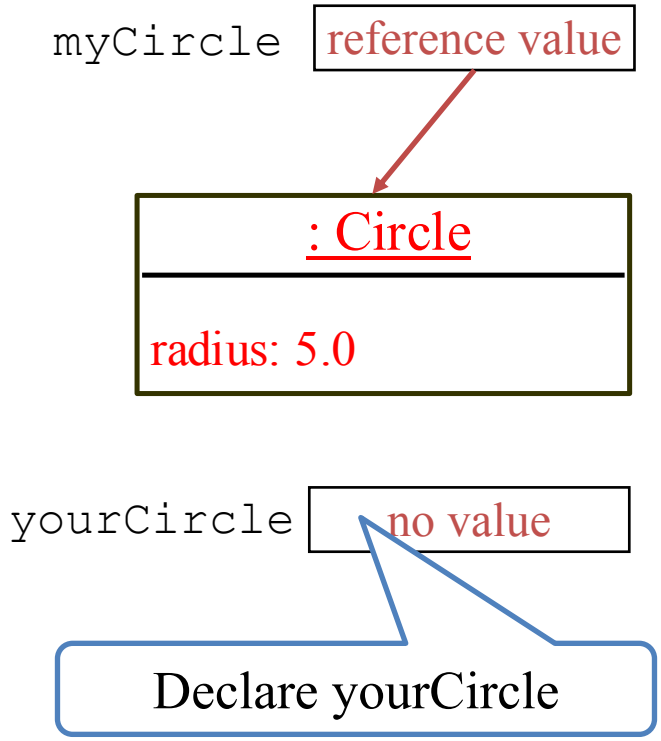
Accessing Object's attributes and methods (".")

(Trace code)

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



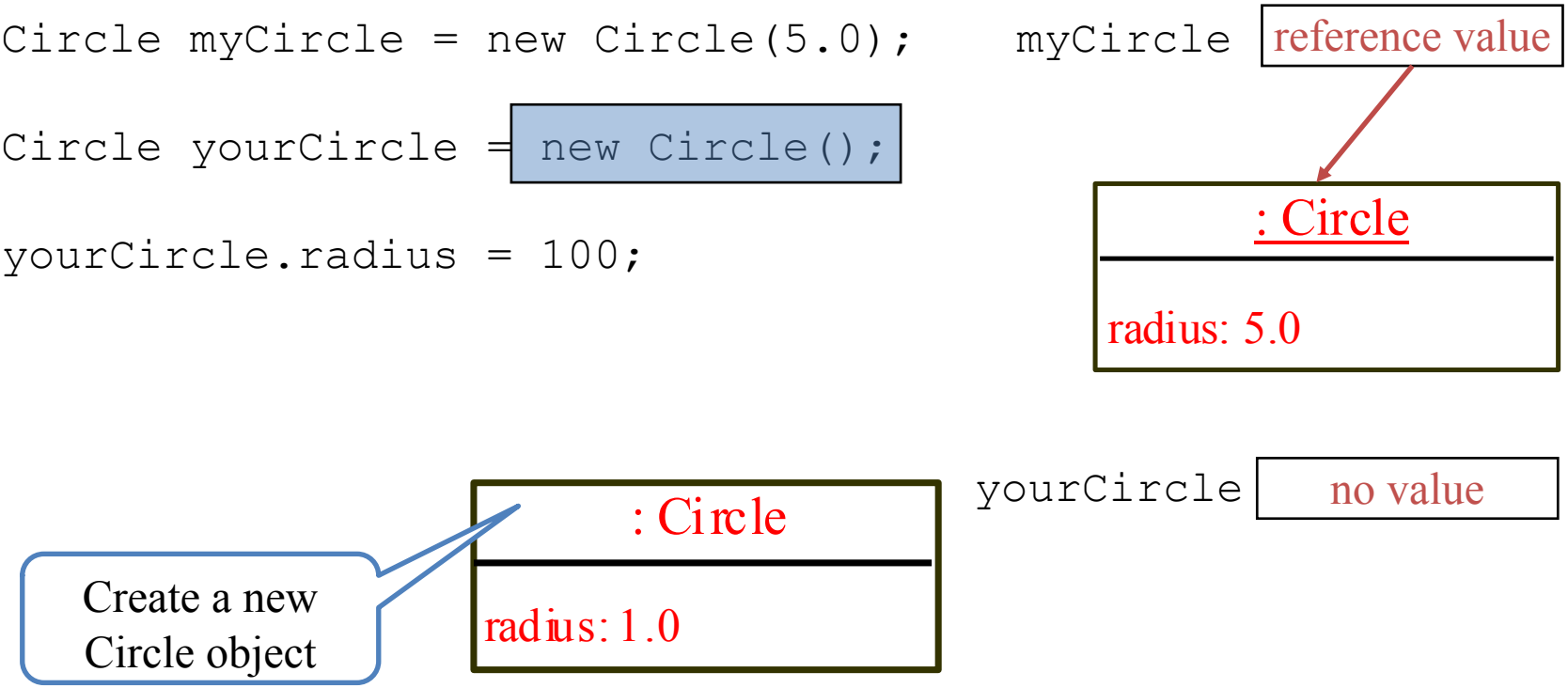
Accessing Object's attributes and methods (".")

(Trace code)

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



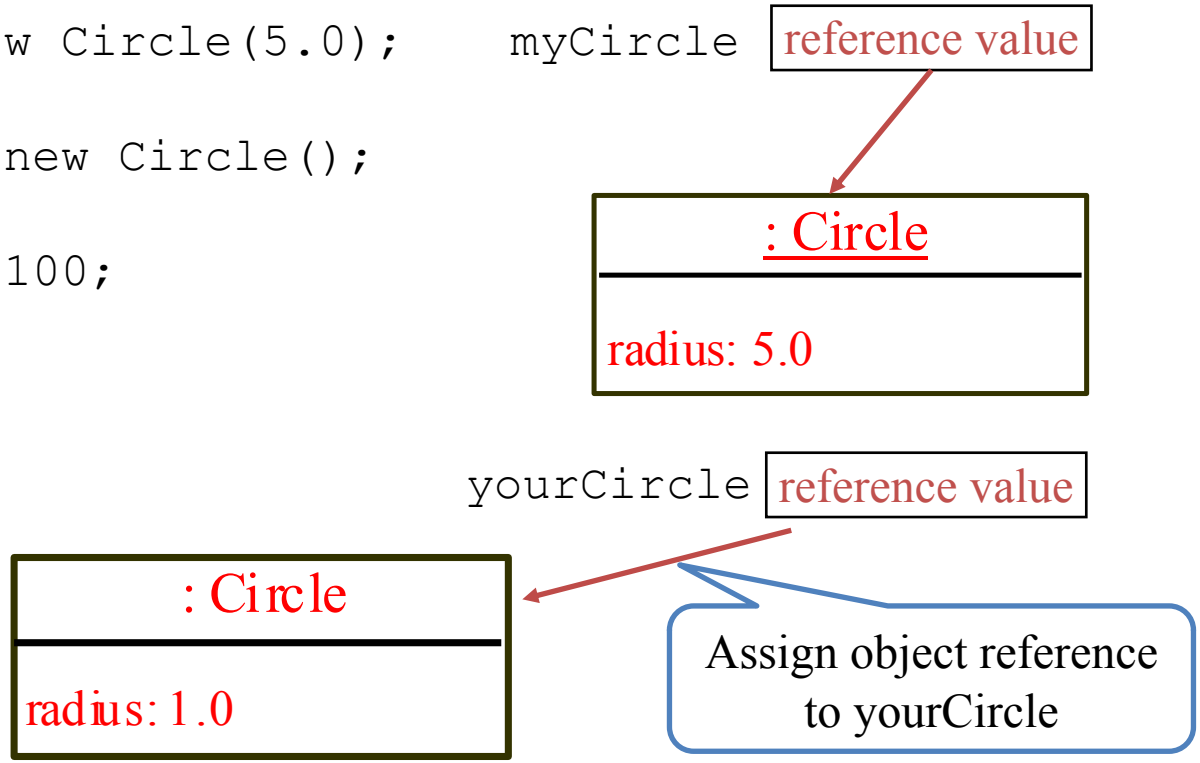
Accessing Object's attributes and methods (".")

(Trace code)

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



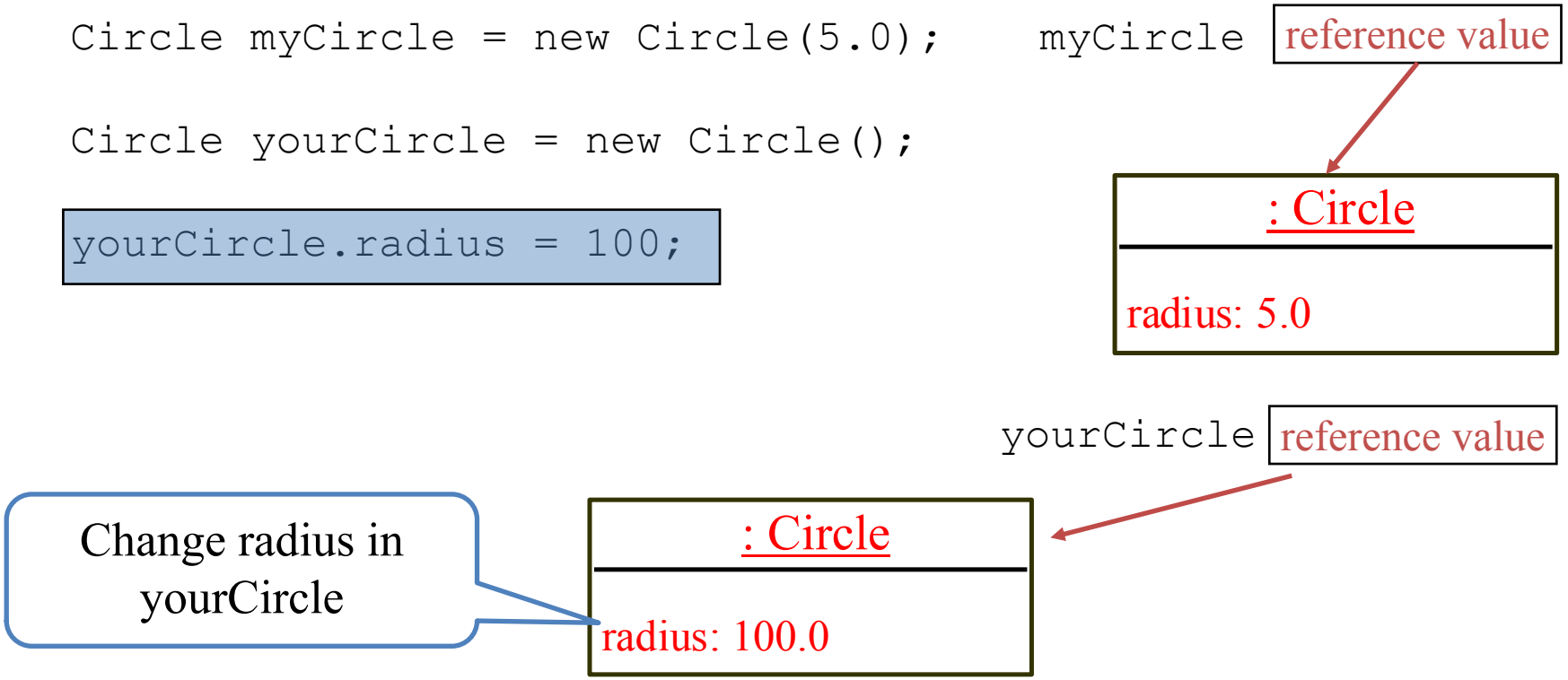
Accessing Object's attributes and methods (".")

(Trace code)

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Example:

Creating objects, accessing data and using methods

[Supplementary Videos on Ed]

https://liveexample.pearsoncmg.com/html/TestSimpleCircle.html

Data Fields

- Data fields (i.e. attributes of a class) can be of reference types
- For example, the following Student class contains a data field name of the String type:

```
public class Student {
  String name; // name has default value null
  int age; // age has default value 0
  boolean isScienceMajor; // isScienceMajor has default value
                          false
  char gender; // c has default value '\u0000'
}
```

Data Fields

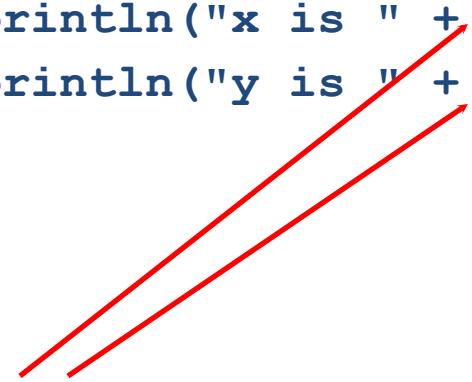
- If a data field of a reference type does not reference any object, the data field holds a **special literal value**, `null`
- Recall: the default value of a data field **is** `null` **for a reference type**, 0 for a numeric type, false for a boolean type, and '\u0000' for a char type. e.g.:

```
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);
  }
}
```

Data Fields

Note: however, Java assigns no default value to a **local variable** inside a method:

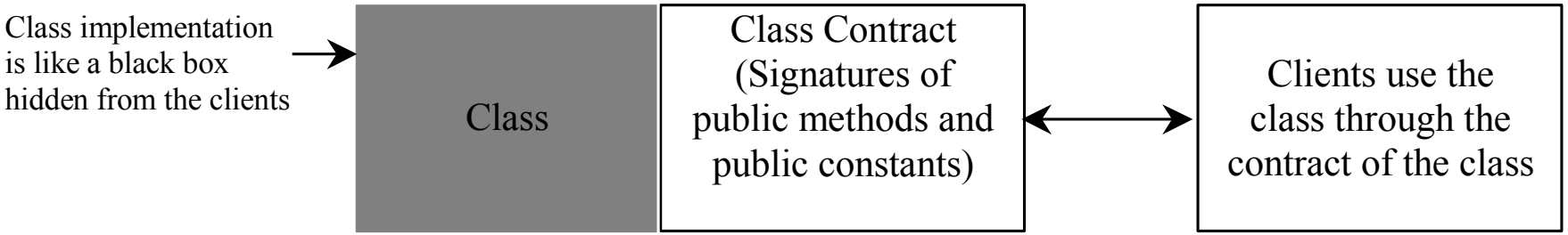
```
public class Test {
  public static void main(String[] args) {
    int x; // x has no default value
    String y; // y has no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```



Compile error: variable not initialized

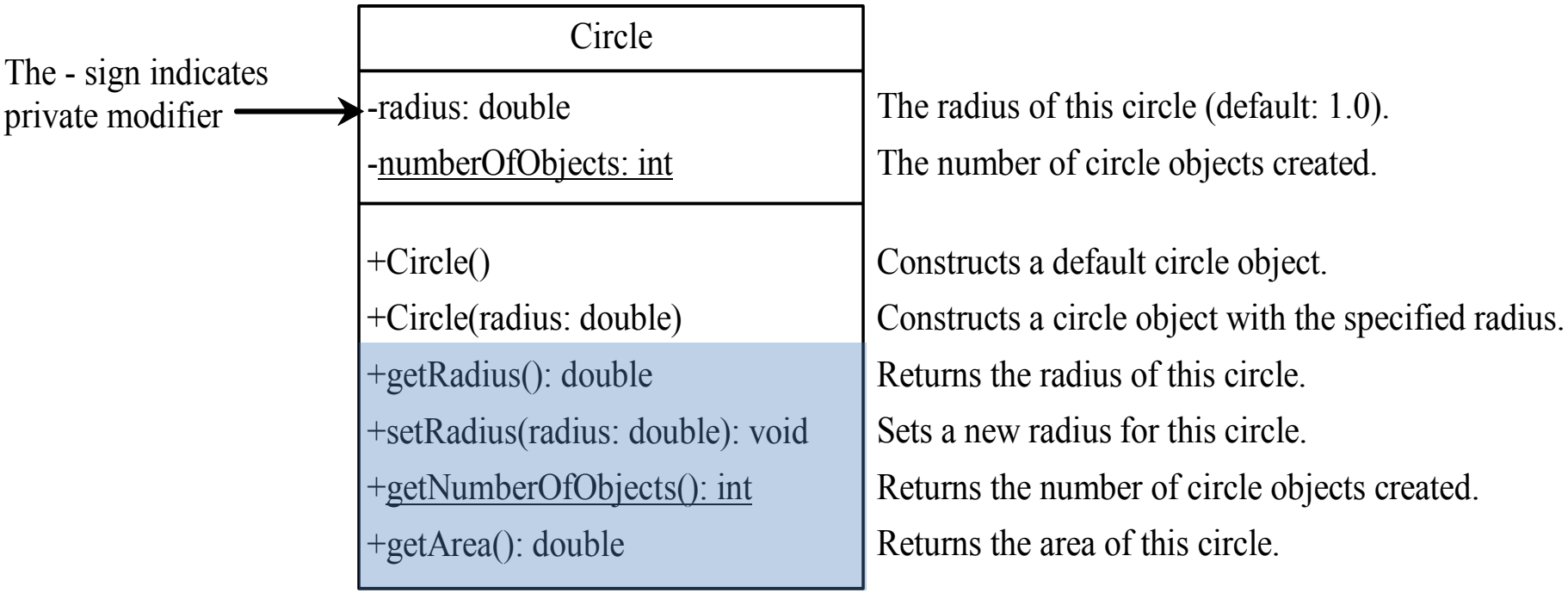
Class Abstraction and Encapsulation

- Class **abstraction** means to separate class **implementation** from the **use** of the class
- The creator of the class provides a description of the class and let the user know **how** the class can be used
- The user of the class does not need to know how the class is implemented
- Class **encapsulation**: the details of implementation are encapsulated and **hidden** from the user



Data Field Encapsulation

- To prevent direct modification of data fields, data fields should be declared as `private`



Visibility Modifiers

- `public`: accessible from any other classes
- `private`: accessible only from within its own class
- `protected`: a `protected` data or method in a `public` class can be accessed by any class in the same package or its subclasses
- If no visibility modifier is used: the default = `package-private` or `package-access`
  - Accessible by any class in the same package

INFS1609/2609 Fundamentals of Business Programming

Visibility Modifiers

`private, none (if no modifier is used), protected, public`

Visibility Modifiers

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Data Field Encapsulation

- By default, the class, variable, or method can be accessed by any class in the same package
- `private` data or methods can be accessed **only by the declaring class**
- The `get` and `set` methods are used to read and modify `private` properties

Data Field Encapsulation

```java
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```java
package p1;

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();
  }
}
```

```java
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();
  }
}
```

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

Data Field Encapsulation

- Note: an object cannot access its private members, as shown in (b). Its ok, however, if the object is declared in its own class (as shown in (a))

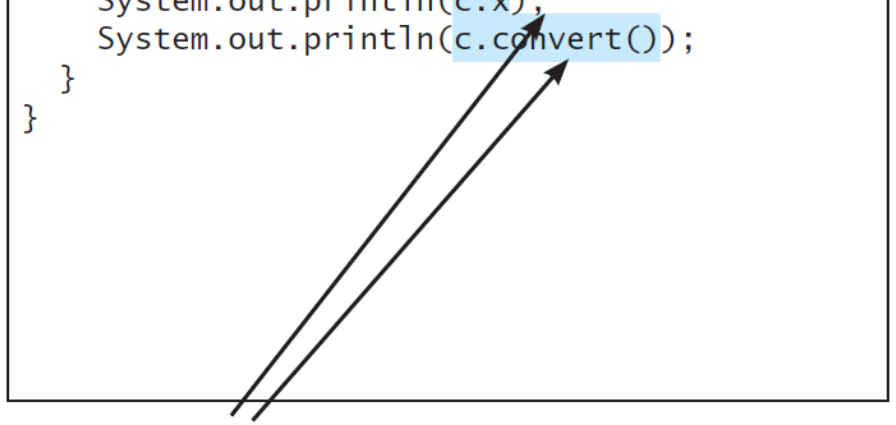
```
public class C {
  private boolean x;

  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`.

```
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`.

Data Field Encapsulation

- Why data fields should be private?
  - To protect data
  - To make code easy to maintain

Sample code:

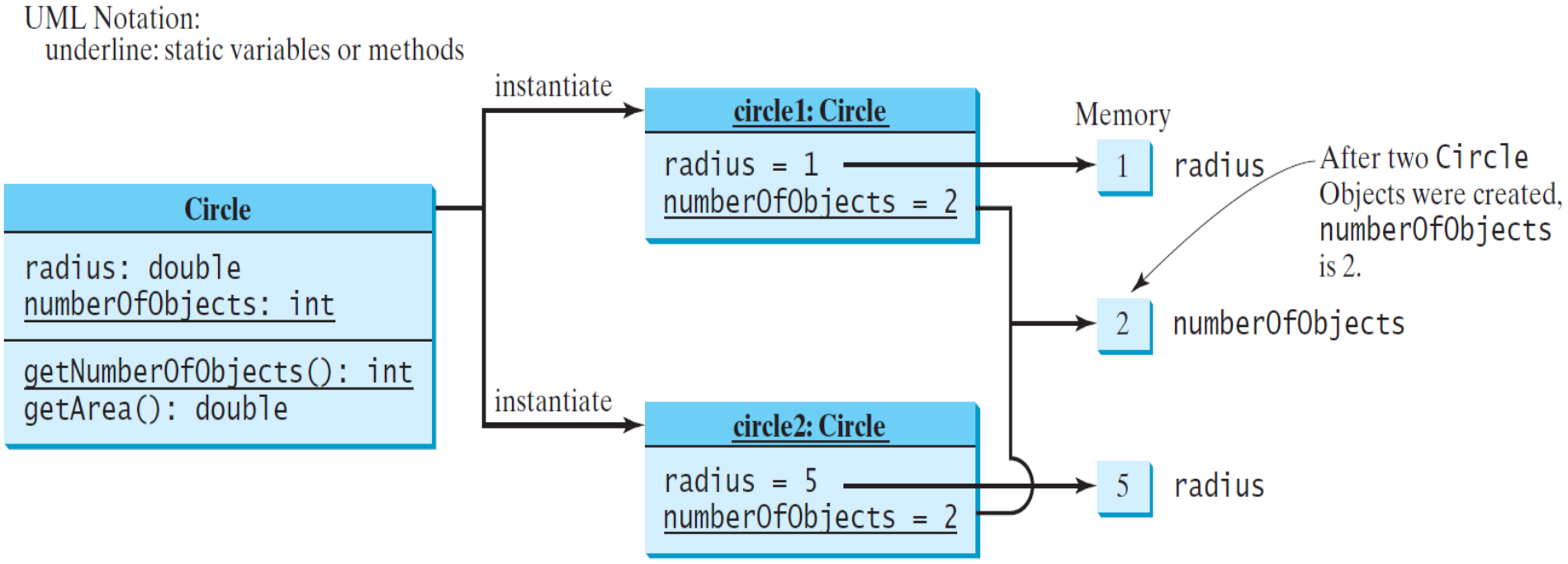
https://liveexample.pearsoncmg.com/html/CircleWithPrivateDataFields.html

Instance versus Static Variables and Methods:

- Instance variables belong to a **specific instance**
- Instance methods are invoked by an instance of the class

- Static variables are shared by **all the instances of the class**
- Static methods are **not** tied to a specific object

Instance versus Static Variables and Methods:



Instance versus Static Variables and Methods:

Example

Demonstrate the roles of instance and class variables and their uses

[Supplementary Videos on Ed]

https://liveexample.pearsoncmg.com/html/CircleWithStaticMembers.html

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.

Scope of Variables:

- If a local variable has the same name as a class's variable, the **local variable takes precedence** AND the class's variable with the same name is **hidden**

- The `this` keyword can be used to reference to the **hidden instance variable**: e.g., when a data field name is often used as the parameter name in a setter method
- The `this` keyword allows us to reference the **object** that invokes an instance method

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

The `this` keyword:

```
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
```

Reflection

What have you learned today?

Thank you!