Chapter 9 Objects and Classes

OO Programming Concepts

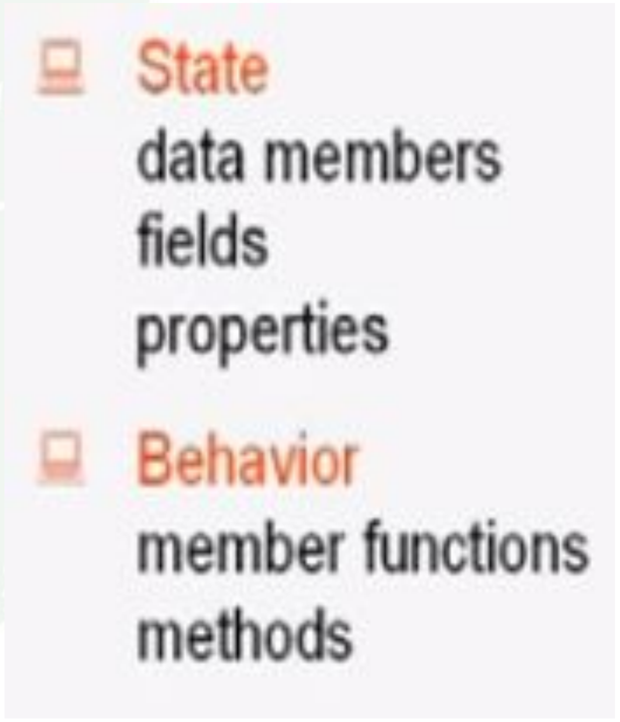
Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behaviors.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

Problems with Procedural Languages

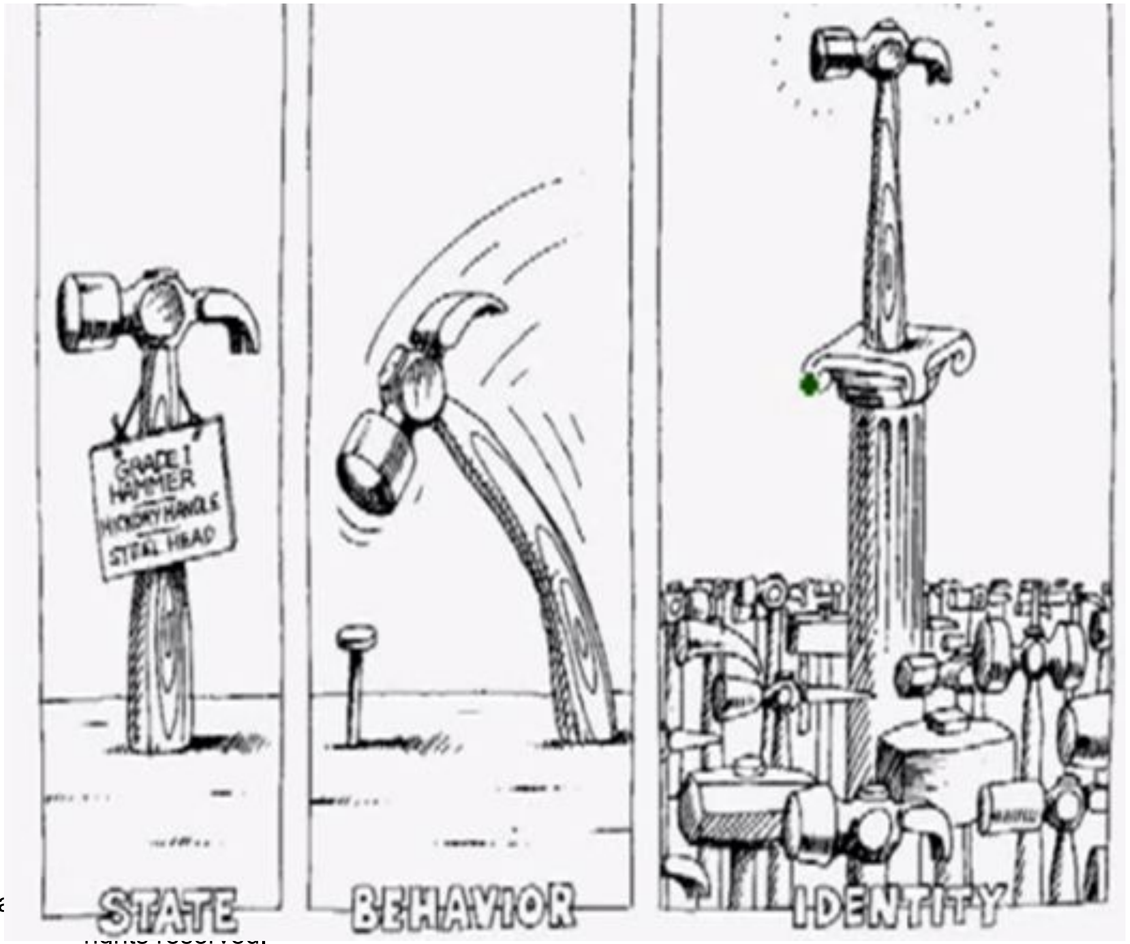
- Data does not have an owner.
- Difficult to maintain data integrity.
- Functions are building blocks.
- Many functions can modify a given block of data.
- Difficult to trace bug sources when data is corrupted.

What is Object?

- An object has **state**, exhibits some well defined **behaviour**, and has a unique **identity**.

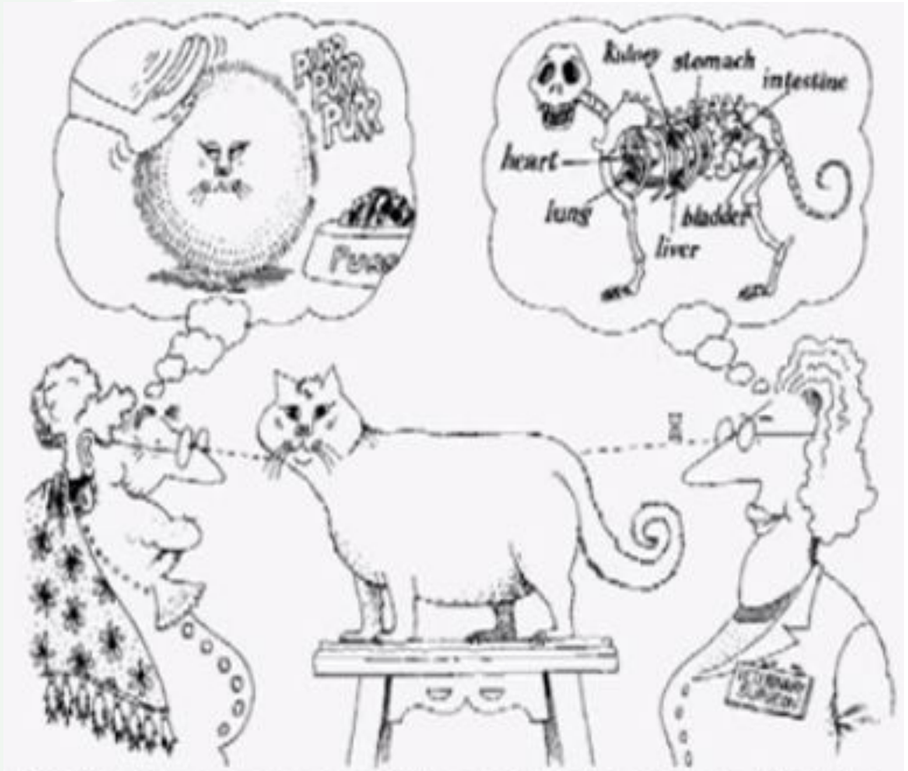


- State
  - data members
  - fields
  - properties
- Behavior
  - member functions
  - methods

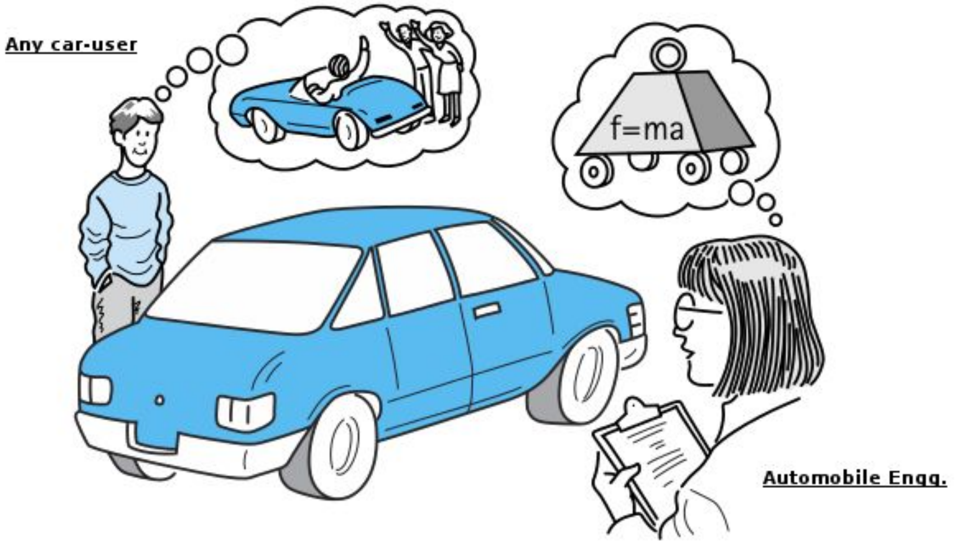


Abstraction - Modeling

- **Abstraction** focuses upon the **essential** characteristics of some object, relative to the perspective of the viewer.



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

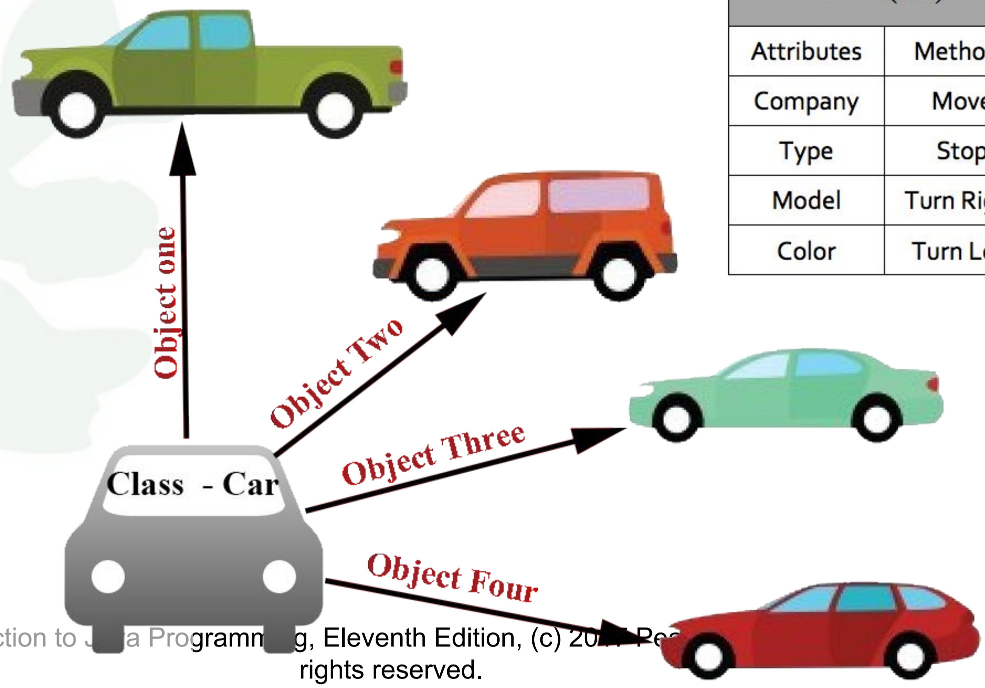


An abstraction includes the essential details relative to the perspective of the viewer

What is Class?

- A **class** represents a set of objects that share common structure and a common behavior.
- A **class** is a **blueprint** or **prototype** that defines the variables and methods common to all objects of a certain kind.

Class (Car)	
Attributes	Methods
Company	Move
Type	Stop
Model	Turn Right
Color	Turn Left

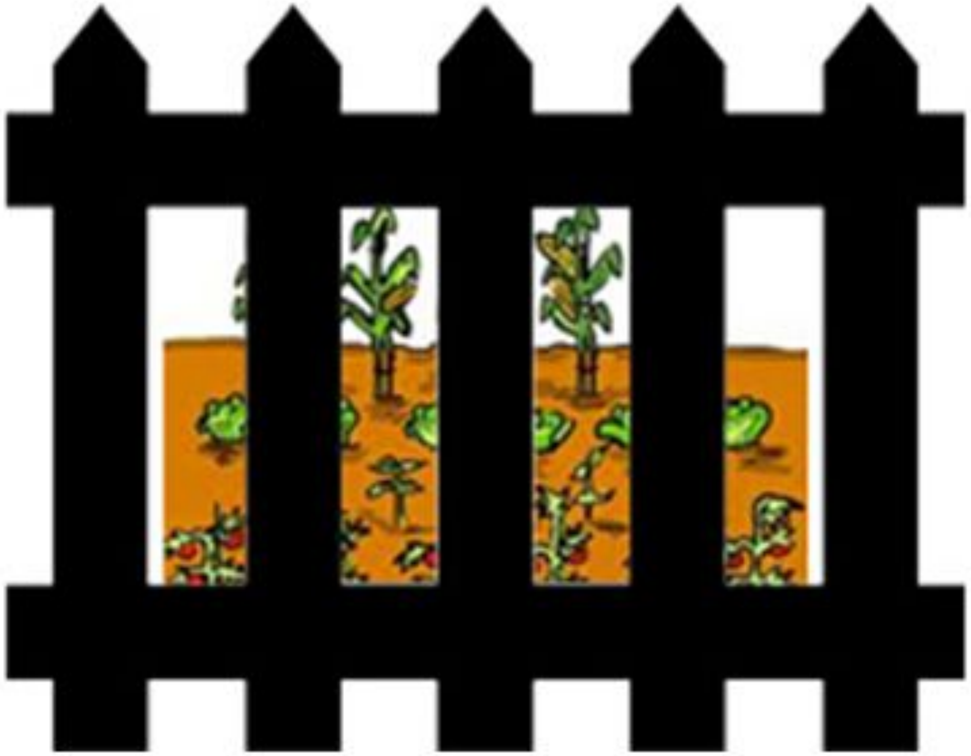


Class Access

PROBLEM: You have a garden and it is **public**. Anyone can take the properties of the garden when they want.

Class Access cont.

SOLUTION? Put a high fence around my garden, now it is safe! But waite, I can no longer access my own garden.



Class Access cont.

SOLUTION: Hire a **private** guard and give him **rules** on who is able to access the garden. Anyone wanting to use the garden must get permission from guard. garden is now **safe** and **accessible**.

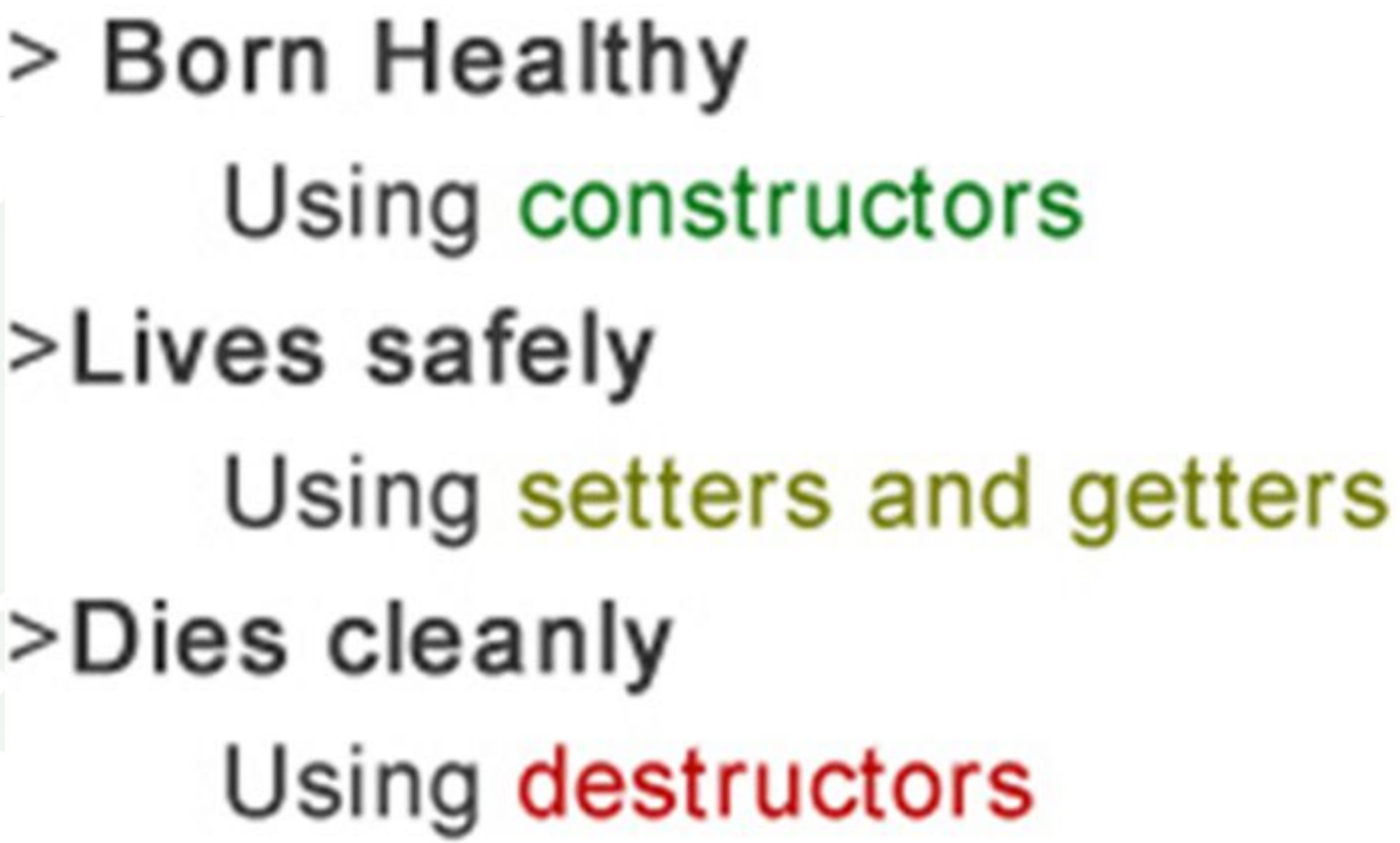
Class Access cont.

Setters and Getters to Safeguard Data
Data
Set Property
Get Property
Outside
Requester
Private
Property
Setter&Getter
Requester

Initialization of Objects

- **Constructors** ensure correct initialization of all data. They are automatically called at the time of object creation.
- **Destructors** on the other hand ensure the de allocation of resources before an object dies or goes out of scope.

Lifecycle of an Object



> Born Healthy

Using constructors

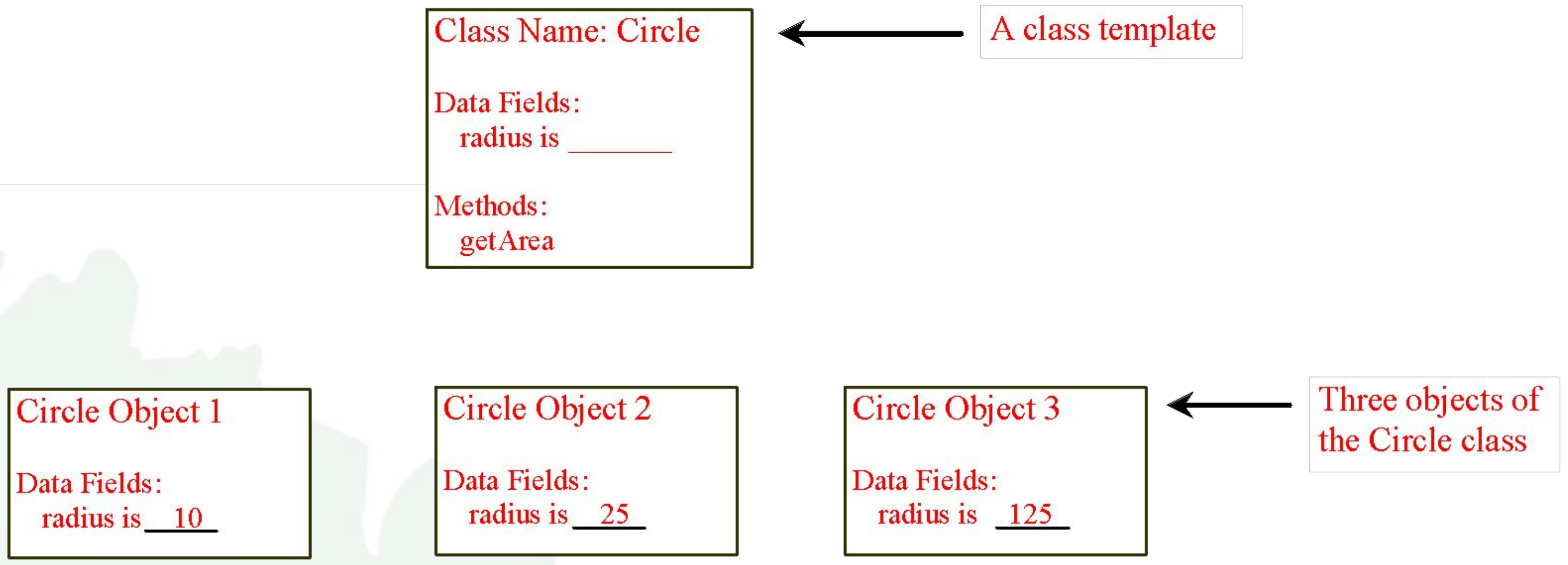
>Lives safely

Using setters and getters

>Dies cleanly

Using destructors

Objects

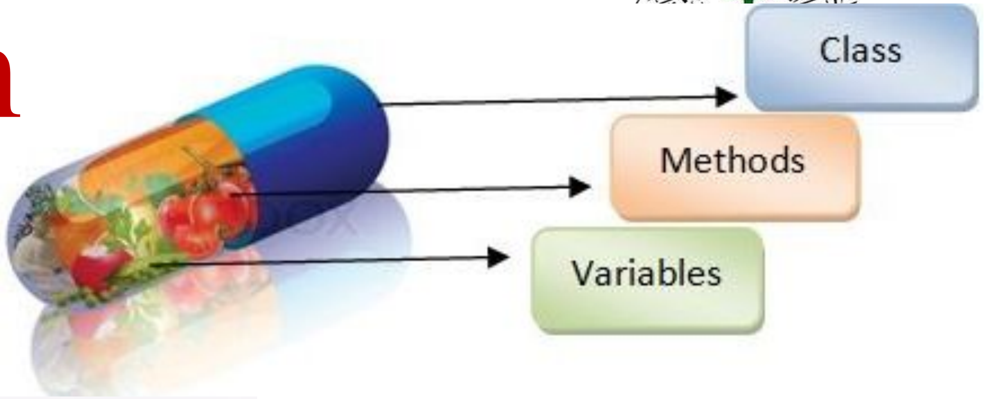


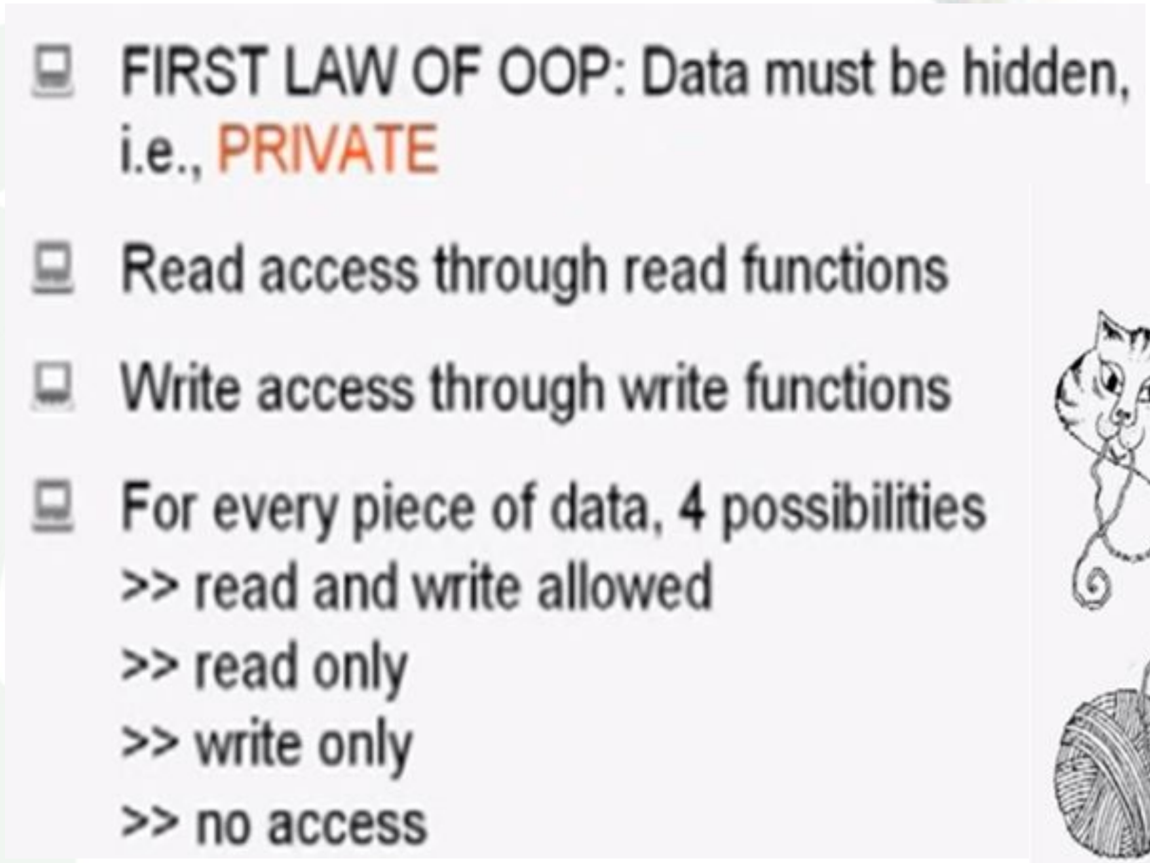
An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

Classes

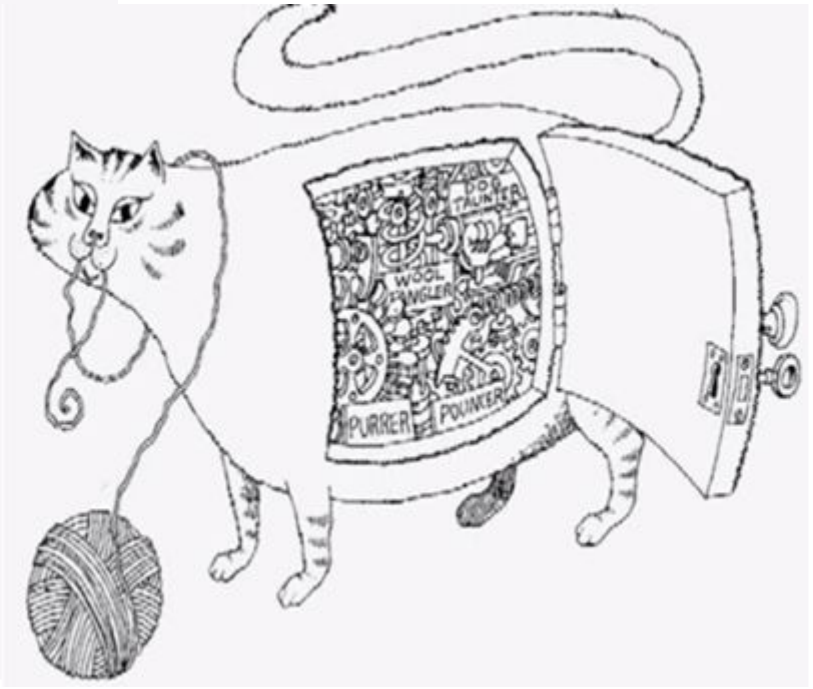
Classes are constructs that define objects of the same type. A Java class uses variables to define data fields and methods to define behaviors. Additionally, a class provides a special type of methods, known as constructors, which are invoked to construct objects from the class.

Encapsulation





- FIRST LAW OF OOP: Data must be hidden, i.e., PRIVATE
- Read access through read functions
- Write access through write functions
- For every piece of data, 4 possibilities
  >> read and write allowed
  >> read only
  >> write only
  >> no access



Encapsulation

- Encapsulation is used to hide unimportant implementation details from other objects.
- In real world
  - When you want to change gears on your car:
    - You don't need to know how the gear mechanism works.
    - You just need to know which lever to move.

Encapsulation cont.

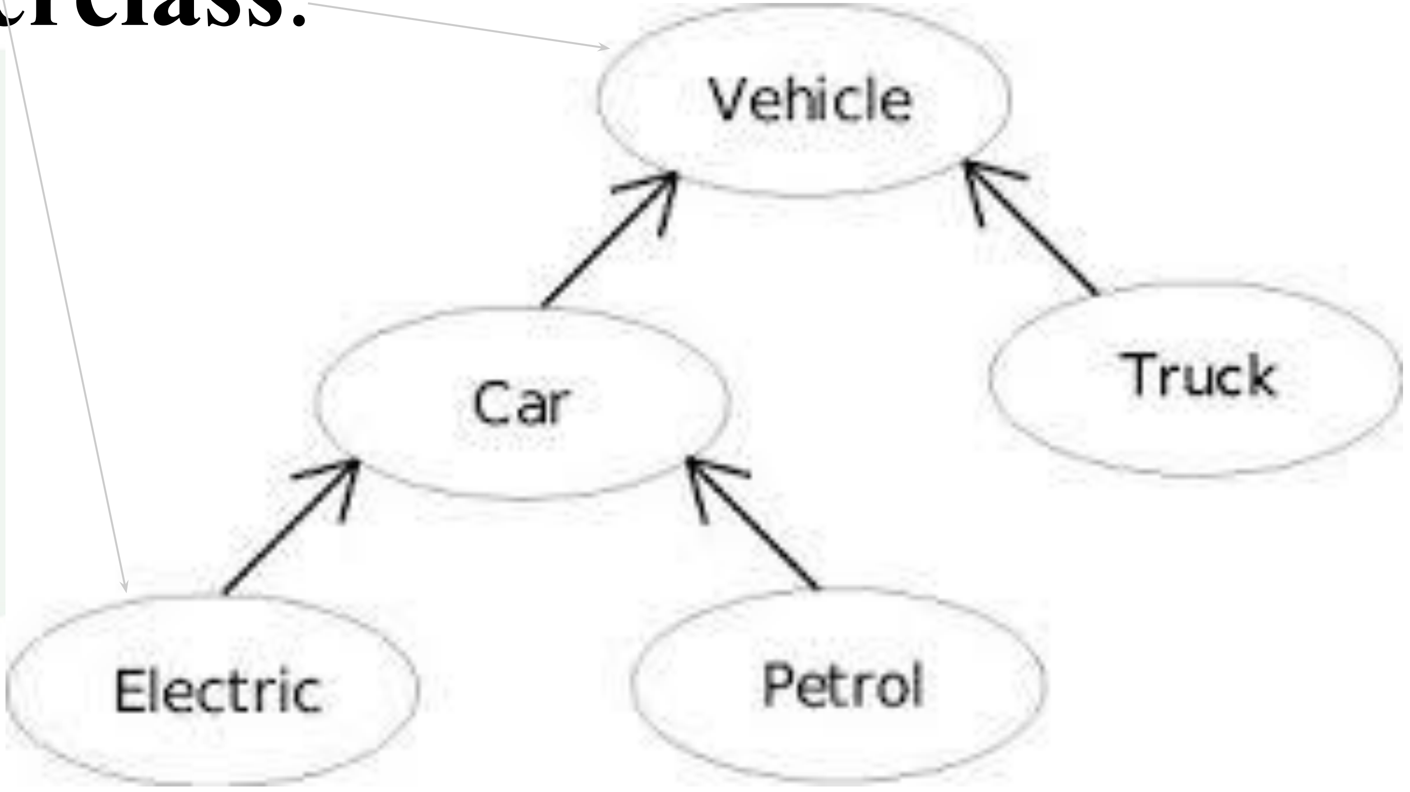
- In software programs:
  - You don’t need to know how a class is implemented.
  - You just need to know which methods to invoke.
  - Thus, the implementation details can change at any time without affecting other parts of the program.

Inheritance

- **Extending** the functionality of a class or
- **Specializing** the functionality of the class.

Inheritance cont.

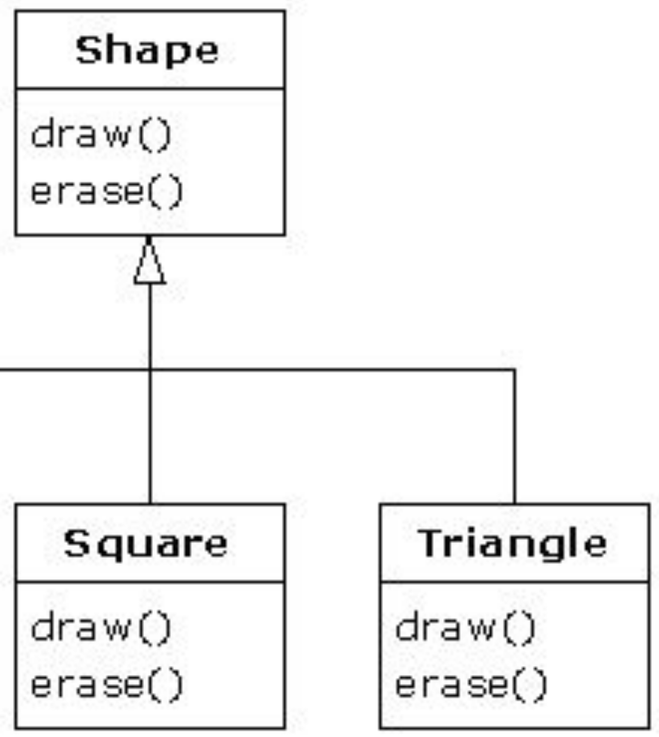
- **Subclasses**: a subclass may inherit the structure and behaviour of it's **superclass**.



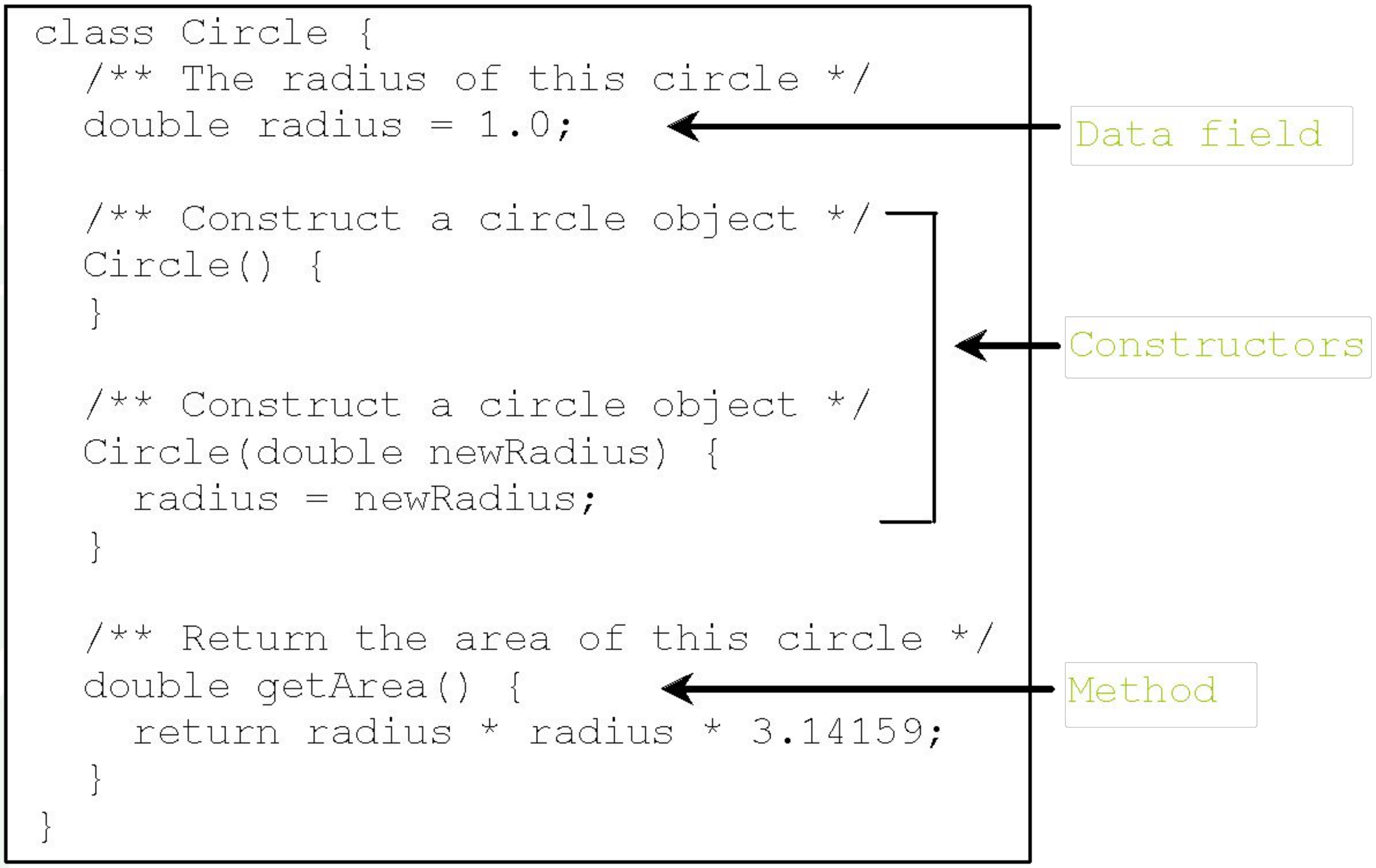
Polymorphism

- **Polymorphism** refers to the ability of an object to provide different behaviours (use different implementations) depending on its own nature. Specifically, depending on its position in the class hierarchy.

drawShape (class Shape)

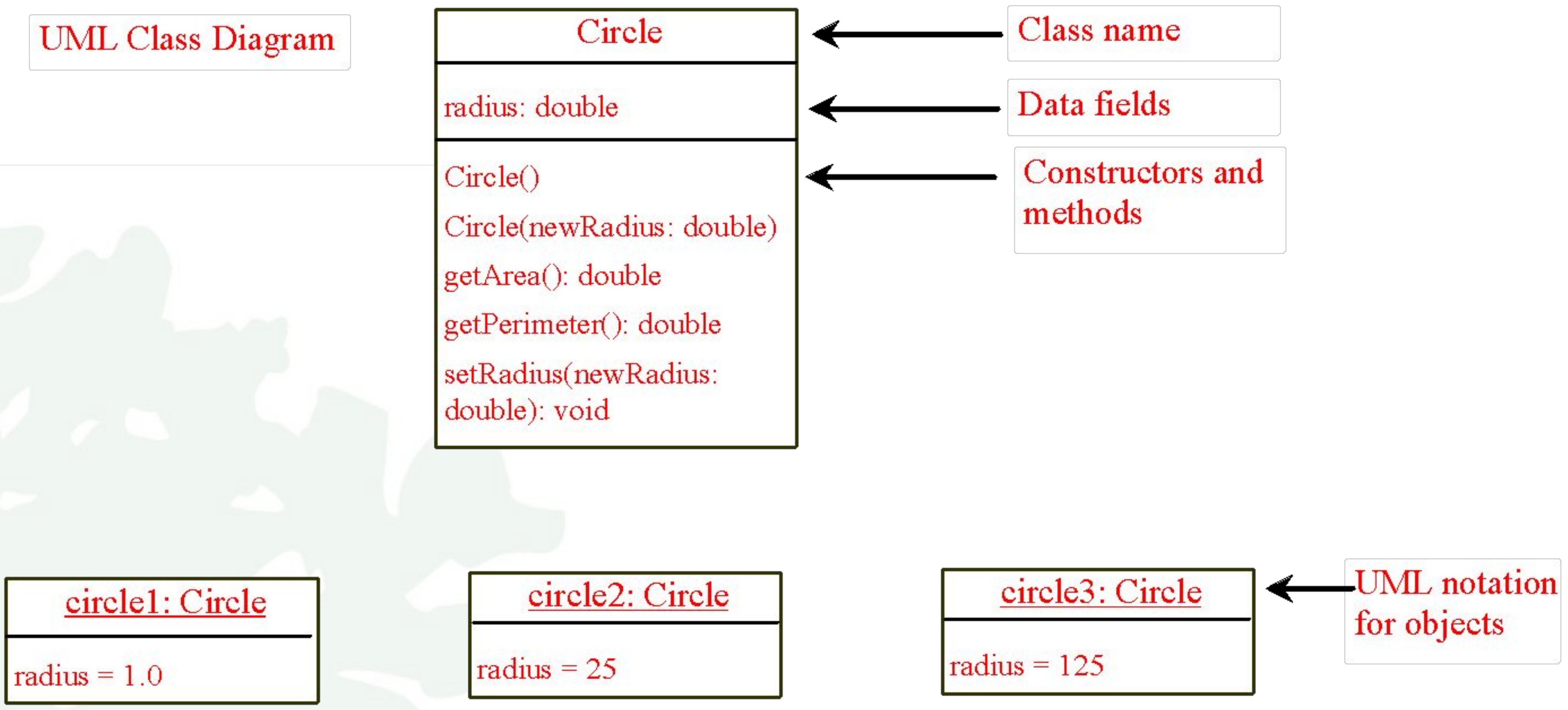


Classes



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

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

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

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

UML Class Diagram

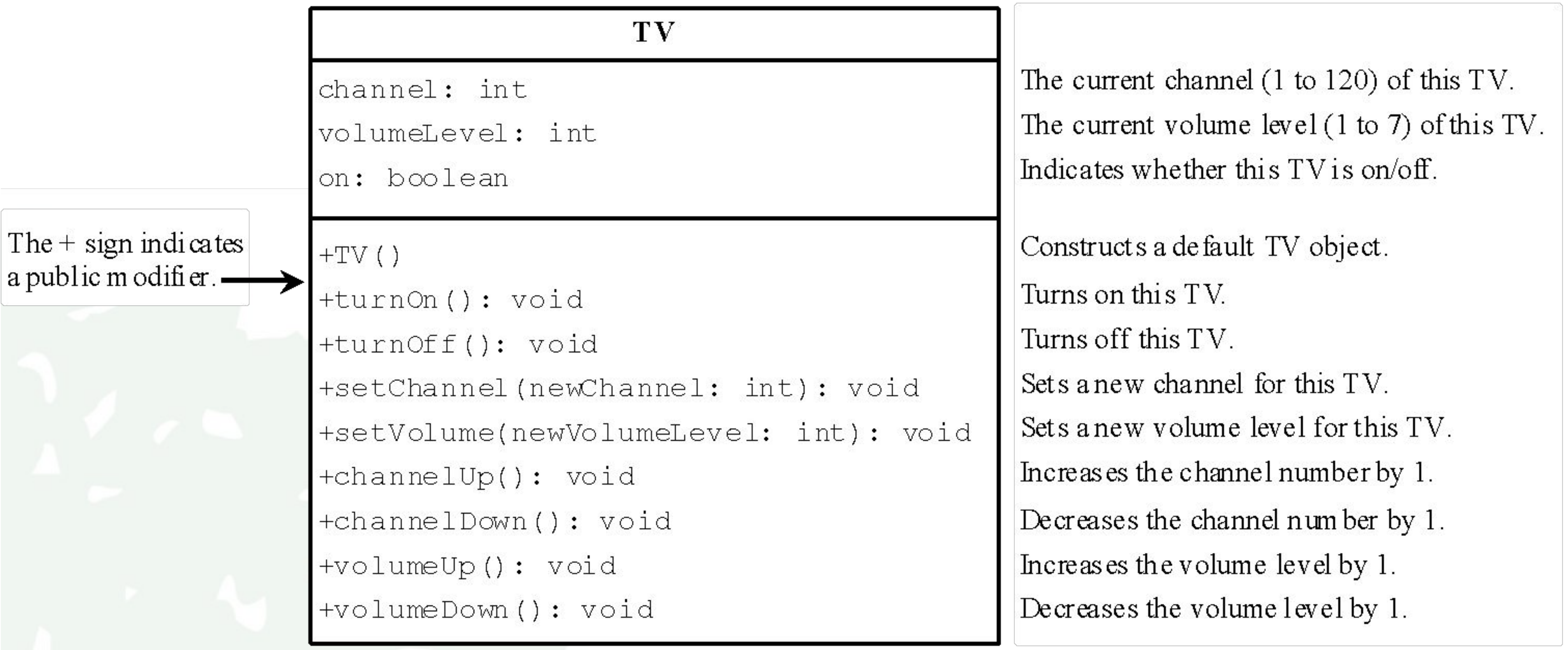


Example: Defining Classes and Creating Objects

Objective: Demonstrate creating objects, accessing data, and using methods.

TestSimpleCircle Run

Example: Defining Classes and Creating Objects



TV	
channel: int	The current channel (1 to 120) of this TV.
volumeLevel: int	The current volume level (1 to 7) of this TV.
on: boolean	Indicates whether this TV is on/off.
+TV()	Constructs a default TV object.
+turnOn(): void	Turns on this TV.
+turnOff(): void	Turns off this TV.
+setChannel(newChannel: int): void	Sets a new channel for this TV.
+setVolume(newVolumeLevel: int): void	Sets a new volume level for this TV.
+channelUp(): void	Increases the channel number by 1.
+channelDown(): void	Decreases the channel number by 1.
+volumeUp(): void	Increases the volume level by 1.
+volumeDown(): void	Decreases the volume level by 1.

The + sign indicates a public modifier.

TV

TestTV

Run

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 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.

To declare a reference variable, use the syntax:

```
ClassName objectRefVar;
```

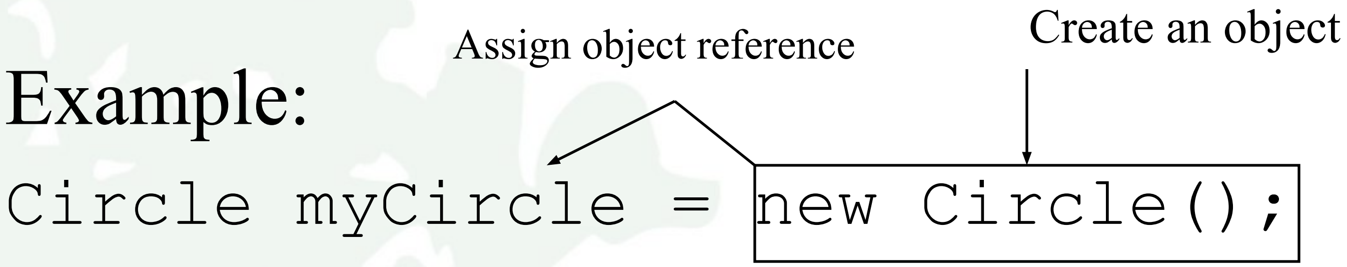
Example:

```
Circle myCircle;
```

Declaring/Creating Objects in a Single Step

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

Example:



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

Accessing Object's Members

- Referencing the object's data:

  `objectRefVar.data`

  e.g., `myCircle.radius`

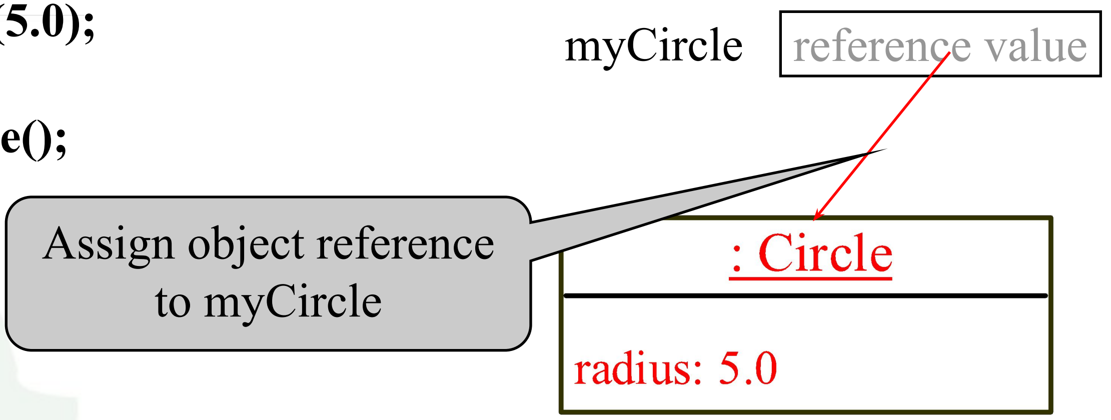
- Invoking the object's method:

  `objectRefVar.methodName(arguments)`

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

Trace Code, cont.

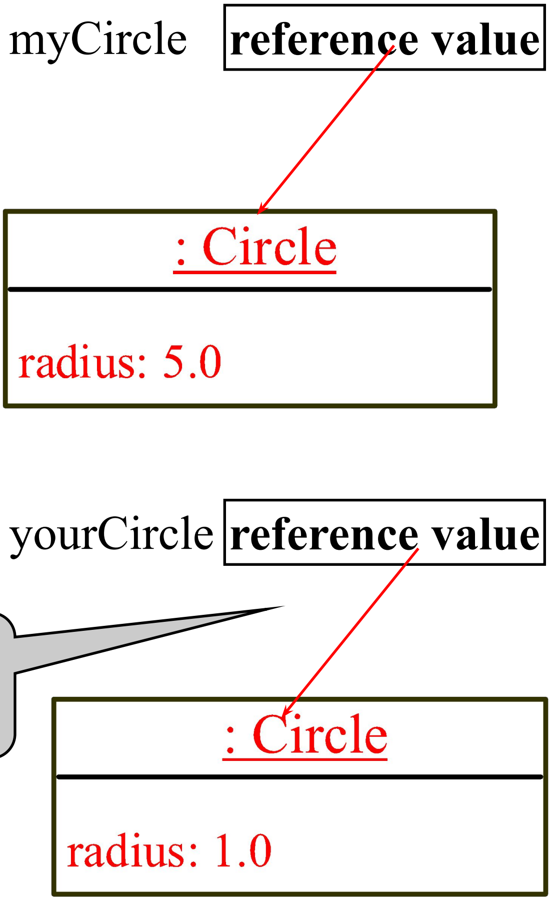
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



animation

Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

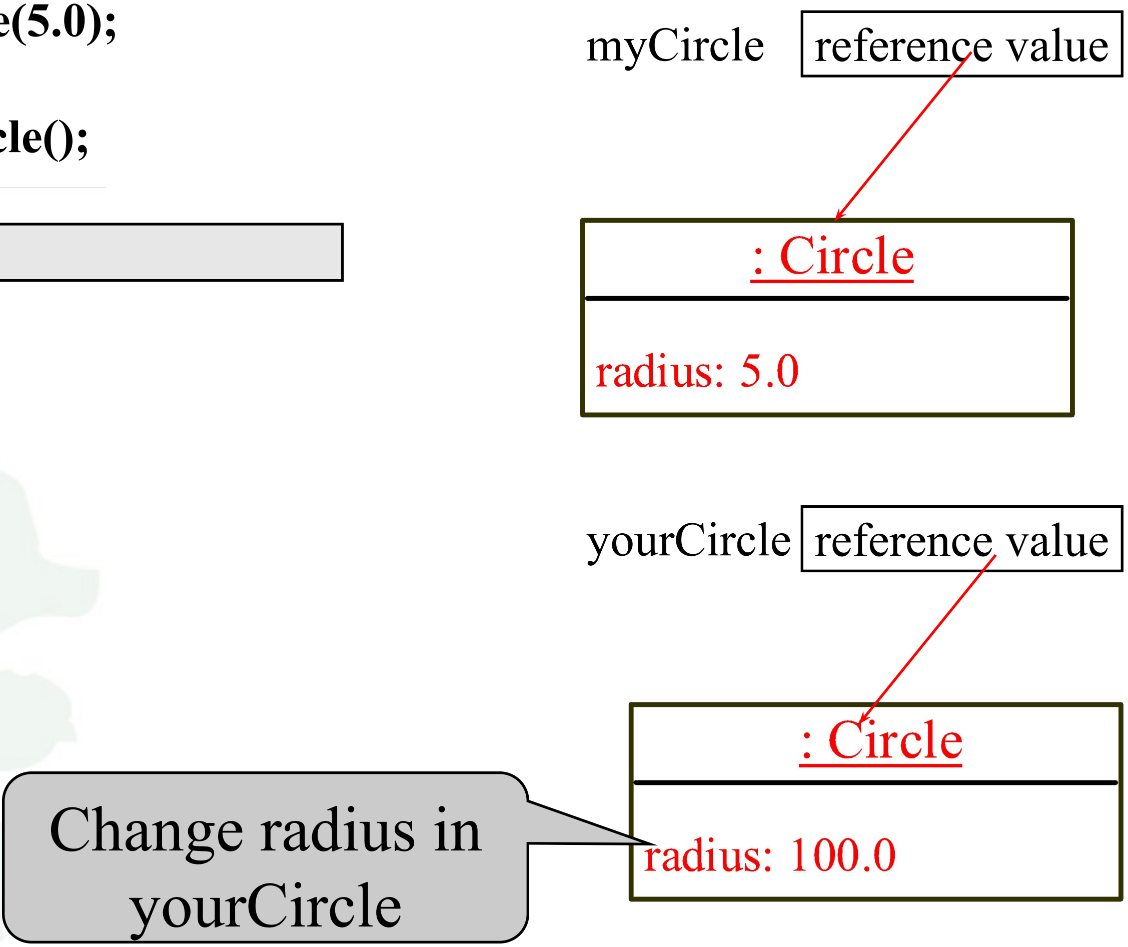


Assign object reference to yourCircle

animation

Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Caution

Recall that you use

Math.methodName(arguments) (e.g., Math.pow(3, 2.5))

to invoke a method in the Math class. Can you invoke getArea() using SimpleCircle.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 explanations will be given in the section on "Static Variables, Constants, and Methods."

Reference Data Fields

The data fields 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'
}
```

The null Value

If a data field of a reference type does not reference any object, the data field holds a special literal value, null.

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);
  }
}
```

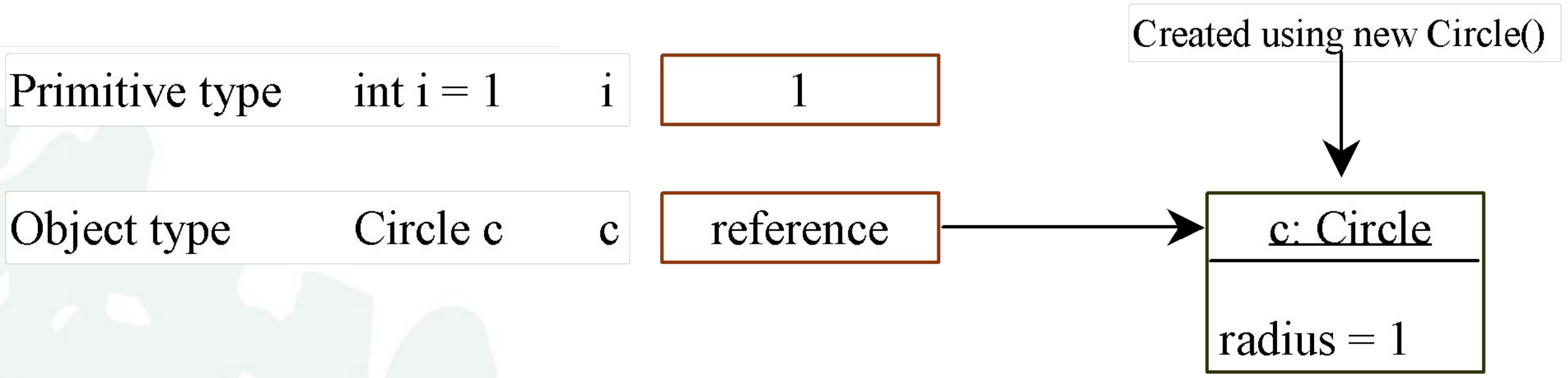
Example

Java assigns no default value to a local variable inside a method.

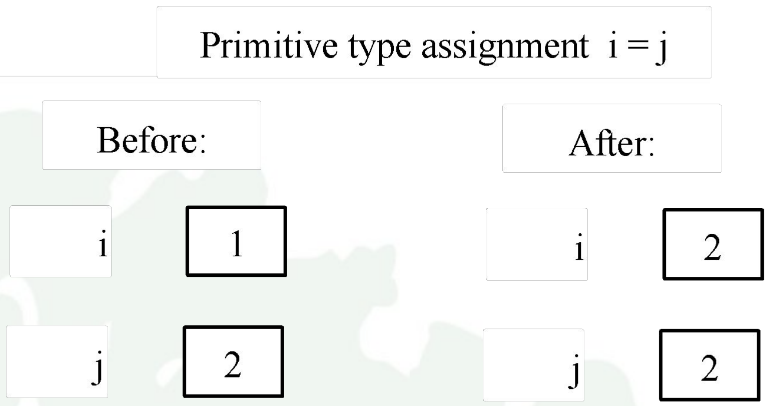
```
public class Test {
  public static void main(String[] args) {
    int x; // x has no default value
    String y; // y has no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```

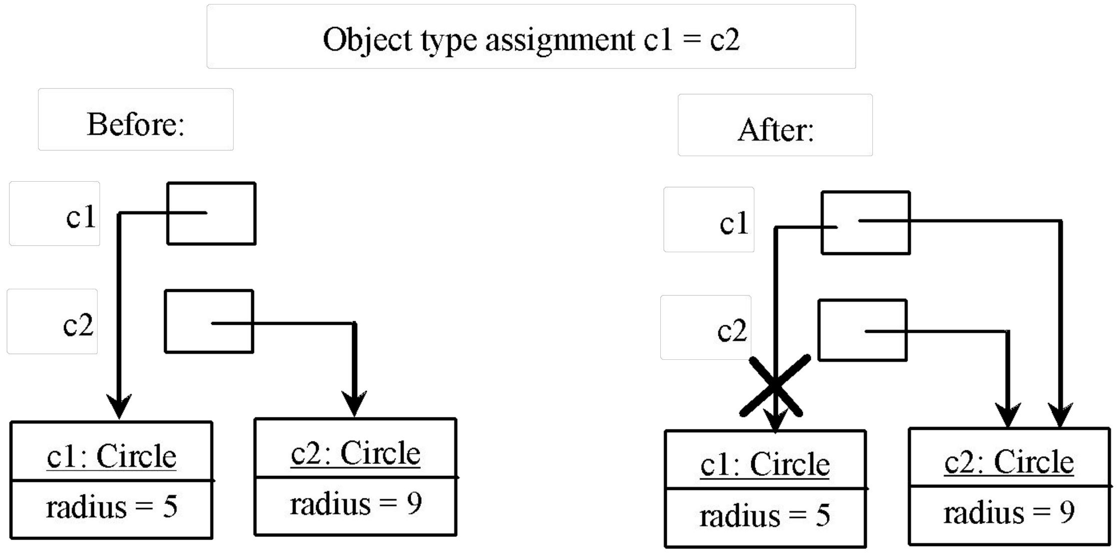
Compile error: variable not initialized

Differences between Variables of Primitive Data Types and Object Types



Copying Variables of Primitive Data Types and Object Types





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.

Garbage Collection, cont

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 .

!?What's wrong

```java
public class ShowErrors {
  public static void main(String[] args) {
    ShowErrors t = new ShowErrors(5);
  }
}
```

(a)

```java
public class ShowErrors {
  public static void main(String[] args) {
    ShowErrors t = new ShowErrors();
    t.x();
  }
}
```

(b)

```java
public class ShowErrors {
  public void method1() {
    Circle c;
    System.out.println("What is radius "
      + c.getRadius());
    c = new Circle();
  }
}
```

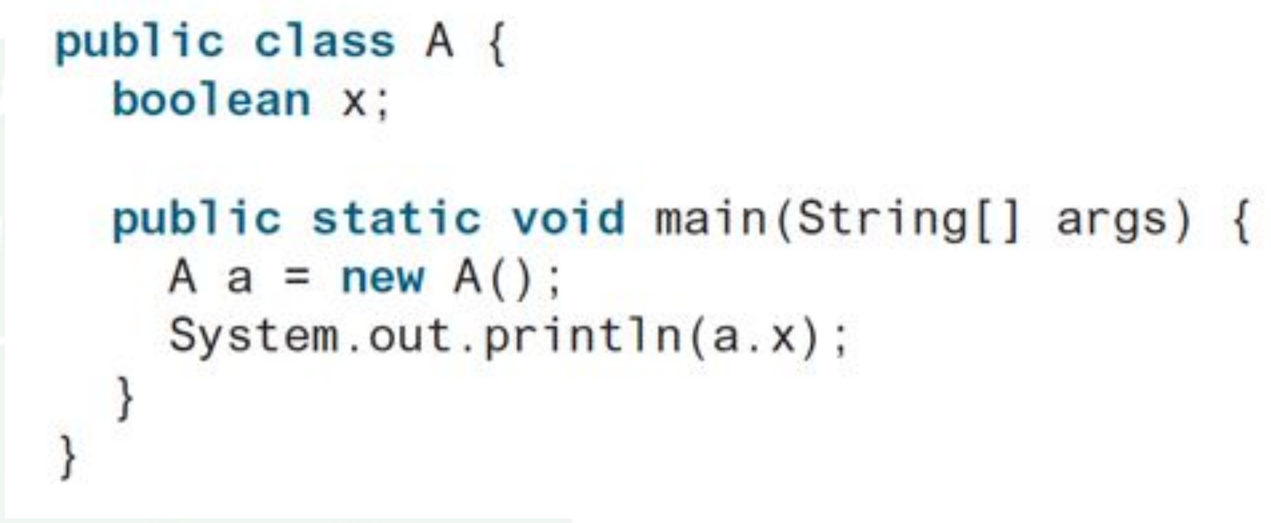
```java
public class ShowErrors {
  public static void main(String[] args) {
    C c = new C(5.0);
    System.out.println(c.value);
  }
}

class C {
  int value = 2;
}
```

What's Wrong?!

```java
class Test {
  public static void main(String[] args) {
    A a = new A();
    a.print();
  }
}

class A {
  String s;

  A(String newS) {
    s = newS;
  }

  public void print() {
    System.out.print(s);
  }
}
```

BIRZEIT UNIVERSITY

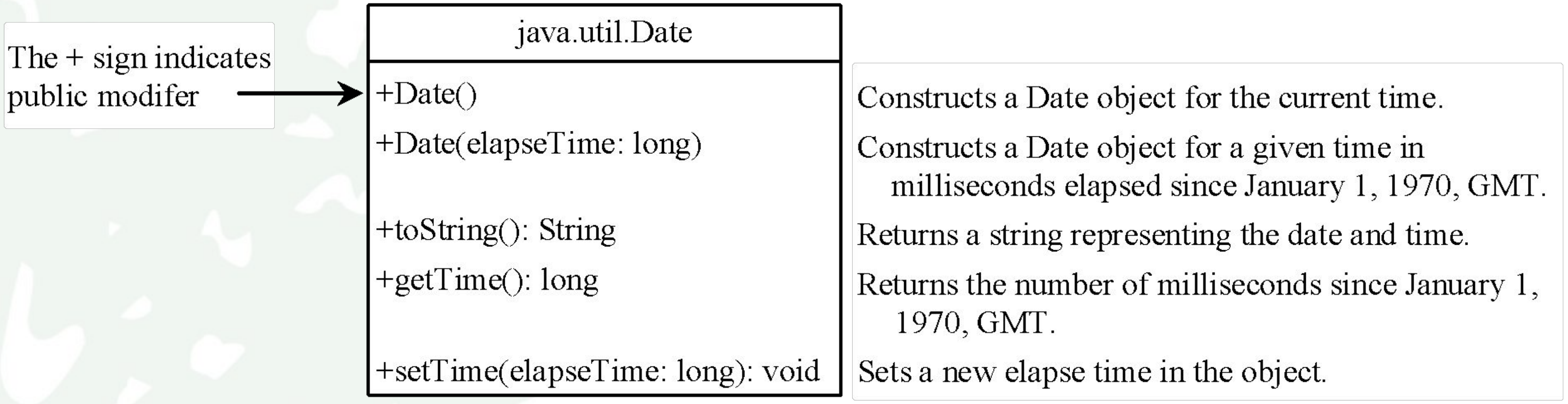
What’s the output?!



```
public class A {
  boolean x;

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

The Date Class

Java provides a system-independent encapsulation of date and time in the java.util.Date class. You can use the Date class to create an instance for the current date and time and use its toString method to return the date and time as a string.


The + sign indicates public modifer
java.util.Date
+Date()
+Date(elapseTime: long)
+toString(): String
+getTime(): long
+setTime(elapseTime: long): void
Constructs a Date object for the current time.
Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
Returns a string representing the date and time.
Returns the number of milliseconds since January 1, 1970, GMT.
Sets a new elapse time in the object.

The Date Class Example

For example, the following code

```
java.util.Date date = new java.util.Date();
System.out.println(date.toString());
```

displays a string like `Sun Mar 09 13:50:19 EST 2003`.

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.

The Random Class Example

If two <u>Random</u> objects have the same seed, they will generate identical sequences of numbers. For example, the following code creates two <u>Random</u> objects with the same seed 3.

```java
Random random1 = new Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
  System.out.print(random1.nextInt(1000) + " ");
Random random2 = new Random(3);
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
  System.out.print(random2.nextInt(1000) + " ");
```

```
From random1: 734 660 210 581 128 202 549 564 459 961
From random2: 734 660 210 581 128 202 549 564 459 961
```

The **Point2D** Class

Java API has a conveninent **Point2D** class in the **javafx.geometry** package for representing a point in a two-dimensional plane.

javafx.geometry.Point2D	
`+Point2D(x: double, y: double)`	Constructs a `Point2D` object with the specified *x*- and *y*-coordinates.
`+distance(x: double, y: double): double`	Returns the distance between this point and the specified point (*x*, *y*).
`+distance(p: Point2D): double`	Returns the distance between this point and the specified point p.
`+getX(): double`	Returns the *x*-coordinate from this point.
`+getY(): double`	Returns the *y*-coordinate from this point.
`+toString(): String`	Returns a string representation for the point.

TestPoint2D

Run

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.

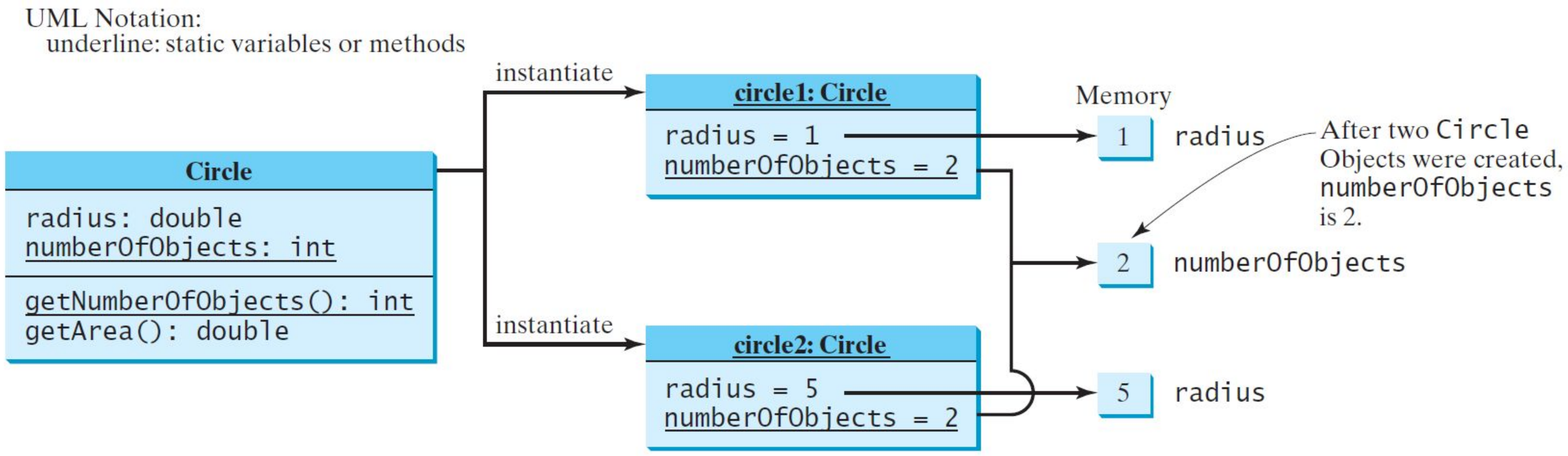
Static methods are not tied to a specific object.

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.



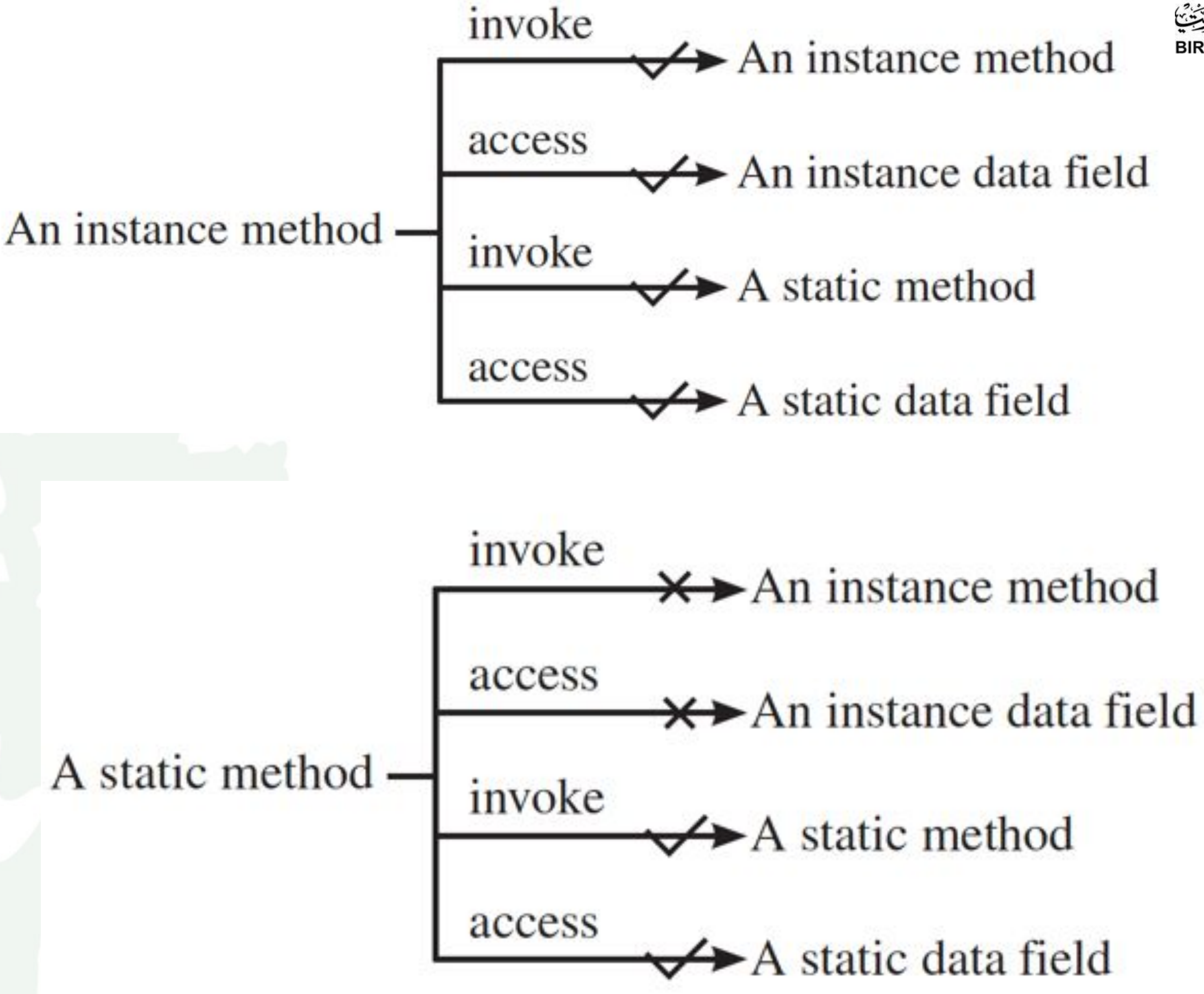
Example of Using Instance and Class Variables and Method

Objective: Demonstrate the roles of instance and class variables and their uses. This example adds a class variable numberOfObjects to track the number of Circle objects created.

CircleWithStaticMembers

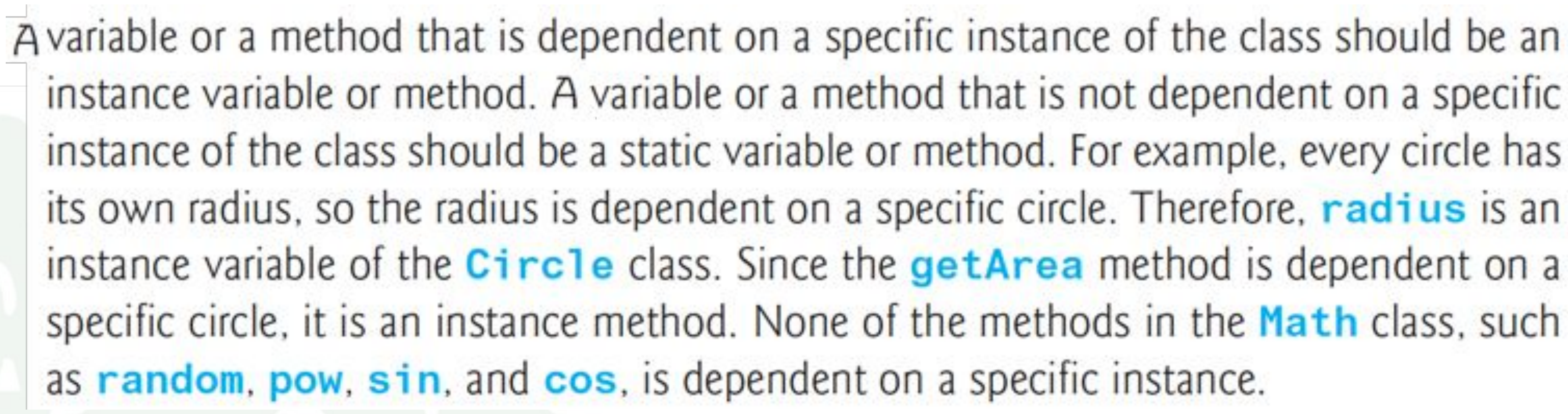
TestCircleWithStaticMembers

Run


invoke
An instance method
access
An instance data field
An instance method
invoke
A static method
access
A static data field
invoke
An instance method
access
An instance data field
A static method
invoke
A static method
access
A static data field

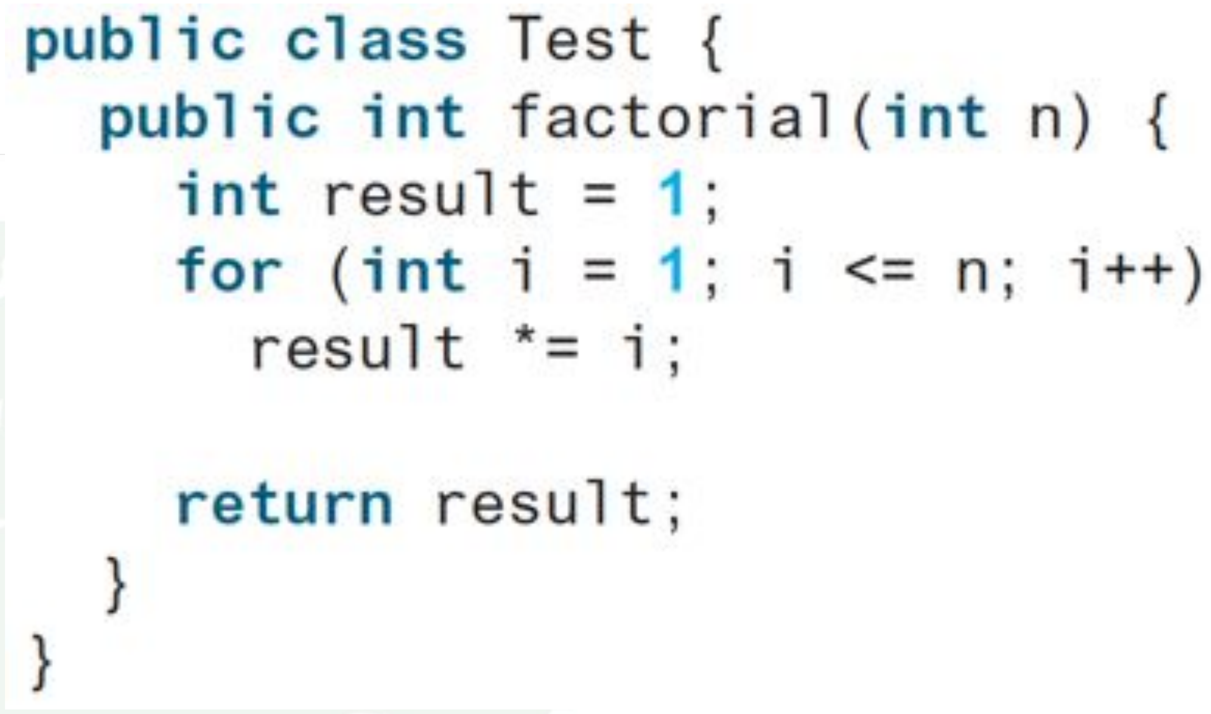
Is this Correct?!

```java
public class A {
  int i = 5;
  static int k = 2;

  public static void main(String[] args) {
    int j = i; // Wrong because i is an instance variable
    m1(); // Wrong because m1() is an instance method
  }

  public void m1() {
    // Correct since instance and static variables and methods
    // can be used in an instance method
    i = i + k + m2(i, k);
  }

  public static int m2(int i, int j) {
    return (int)(Math.pow(i, j));
  }
}
```



A variable or a method that is dependent on a specific instance of the class should be an instance variable or method. A variable or a method that is not dependent on a specific instance of the class should be a static variable or method. For example, every circle has its own radius, so the radius is dependent on a specific circle. Therefore, `radius` is an instance variable of the `Circle` class. Since the `getArea` method is dependent on a specific circle, it is an instance method. None of the methods in the `Math` class, such as `random`, `pow`, `sin`, and `cos`, is dependent on a specific instance.

What do you think?



```
public class Test {
  public int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

Which is correct?!

```
public class F {
  int i;
  static String s;

  void imethod() {
  }

  static void smethod() {
  }
}
```

Let f be an instance of F

```
System.out.println(f.i);
System.out.println(f.s);
f.imethod();
f.smethod();
System.out.println(F.i);
System.out.println(F.s);
F.imethod();
F.smethod();
```

Visibility Modifiers and Accessor/Mutator Methods

package-private or package-access

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;

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

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

```
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();
  }
}
```

```
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.

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

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

The default modifier on a class 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).

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

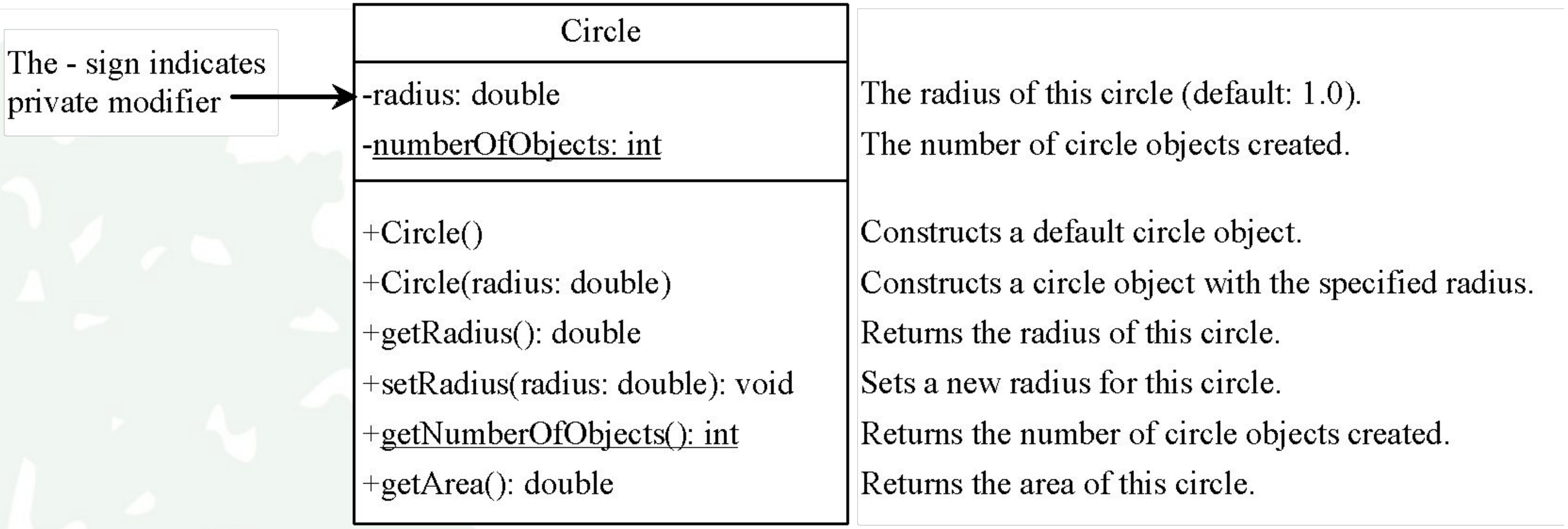
(b) This is wrong because **x** and **convert** are private in class **C**.

Why Data Fields Should Be private?

To protect data.

To make code easy to maintain.

Example of Data Field Encapsulation



-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.

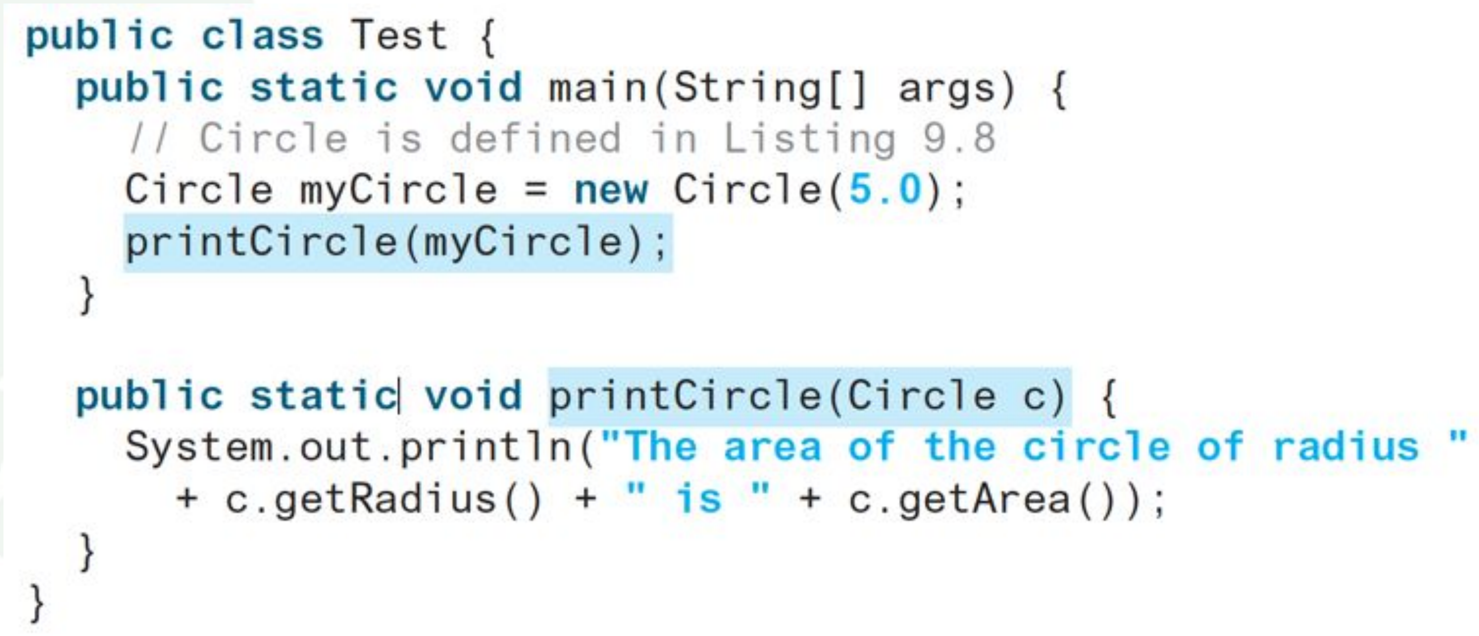
CircleWithPrivateDataFields

TestCircleWithPrivateDataFields

Run

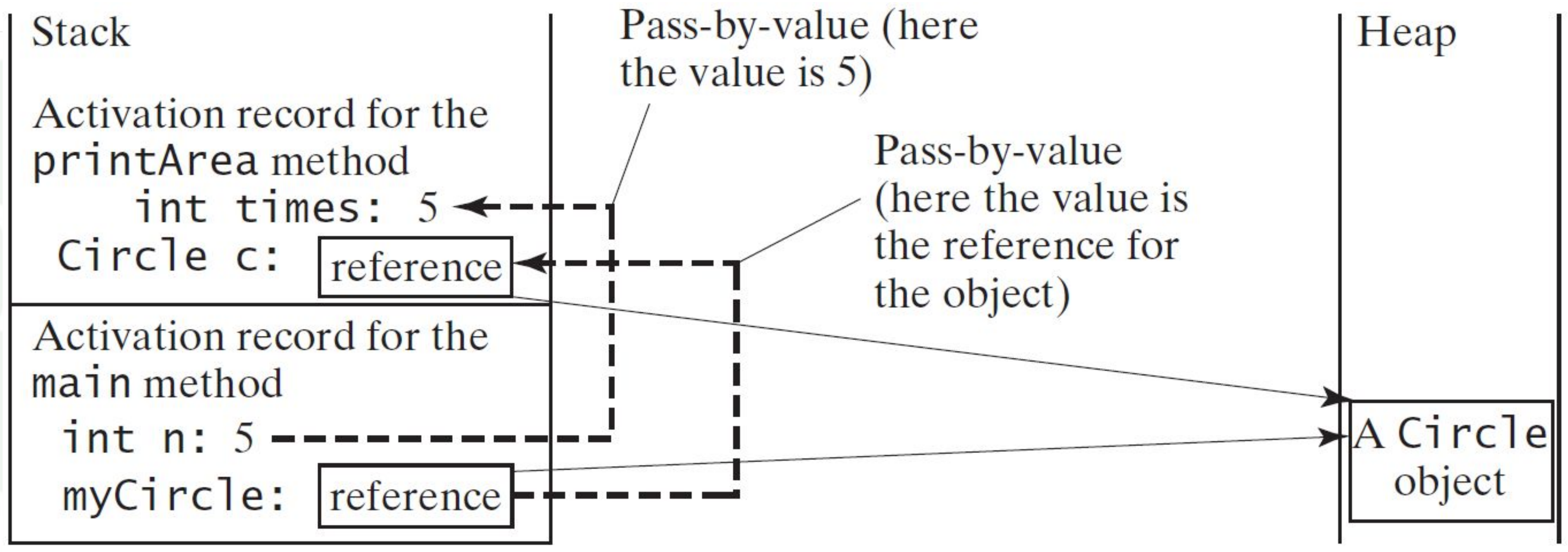
Passing Objects to Methods

- 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)

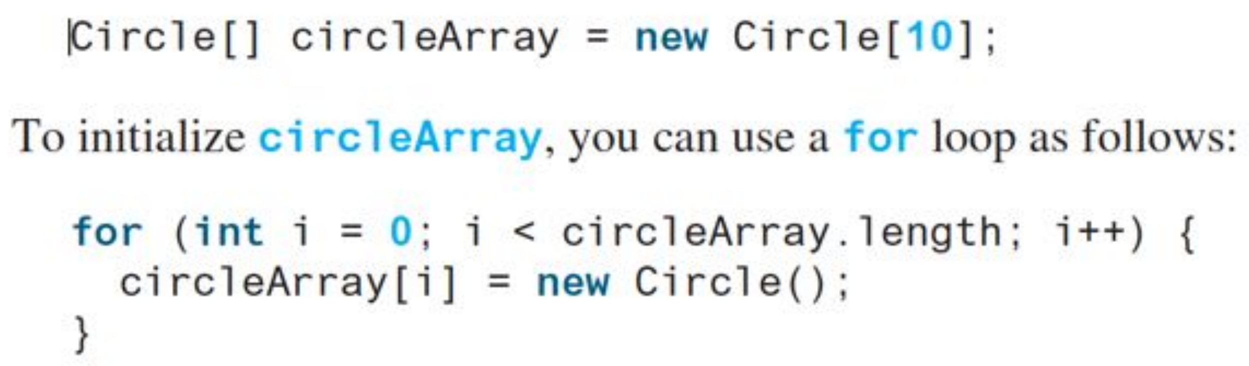


```
public class Test {
  public static void main(String[] args) {
    // Circle is defined in Listing 9.8
    Circle myCircle = new Circle(5.0);
    printCircle(myCircle);
  }

  public static void printCircle(Circle c) {
    System.out.println("The area of the circle of radius "
      + c.getRadius() + " is " + c.getArea());
  }
}
```

Passing Objects to Methods, cont.


Stack
Pass-by-value (here the value is 5)
Heap
Activation record for the printArea method
Pass-by-value (here the value is the reference for the object)
int times: 5
Circle c: reference
Activation record for the main method
int n: 5
myCircle: reference
A Circle object

Array of Objects



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

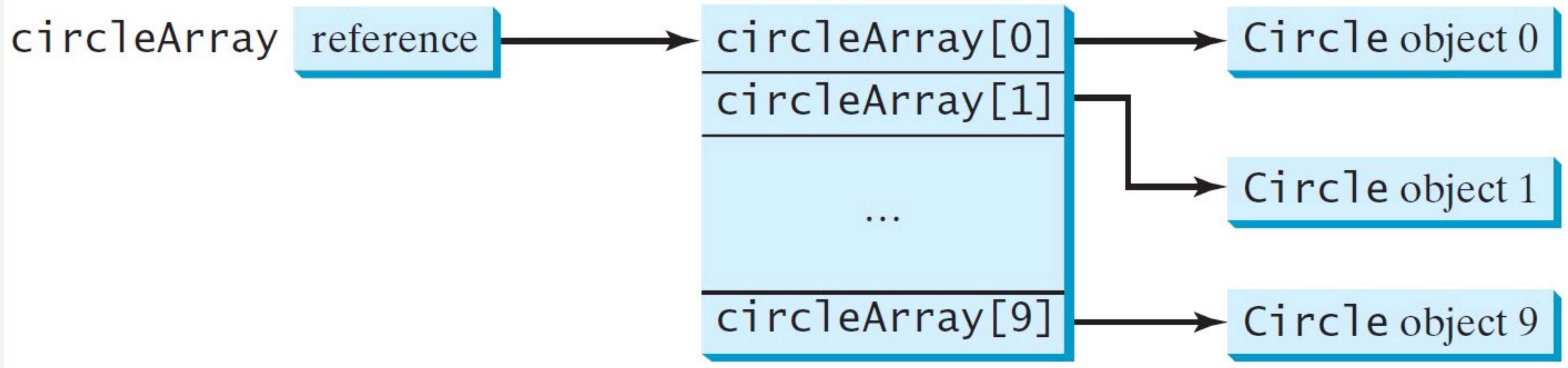
To initialize circleArray, you can use a for loop as follows:

```
for (int i = 0; i < circleArray.length; i++) {
  circleArray[i] = new Circle();
}
```

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];
```



Array of Objects, cont.

Summarizing the areas of the circles

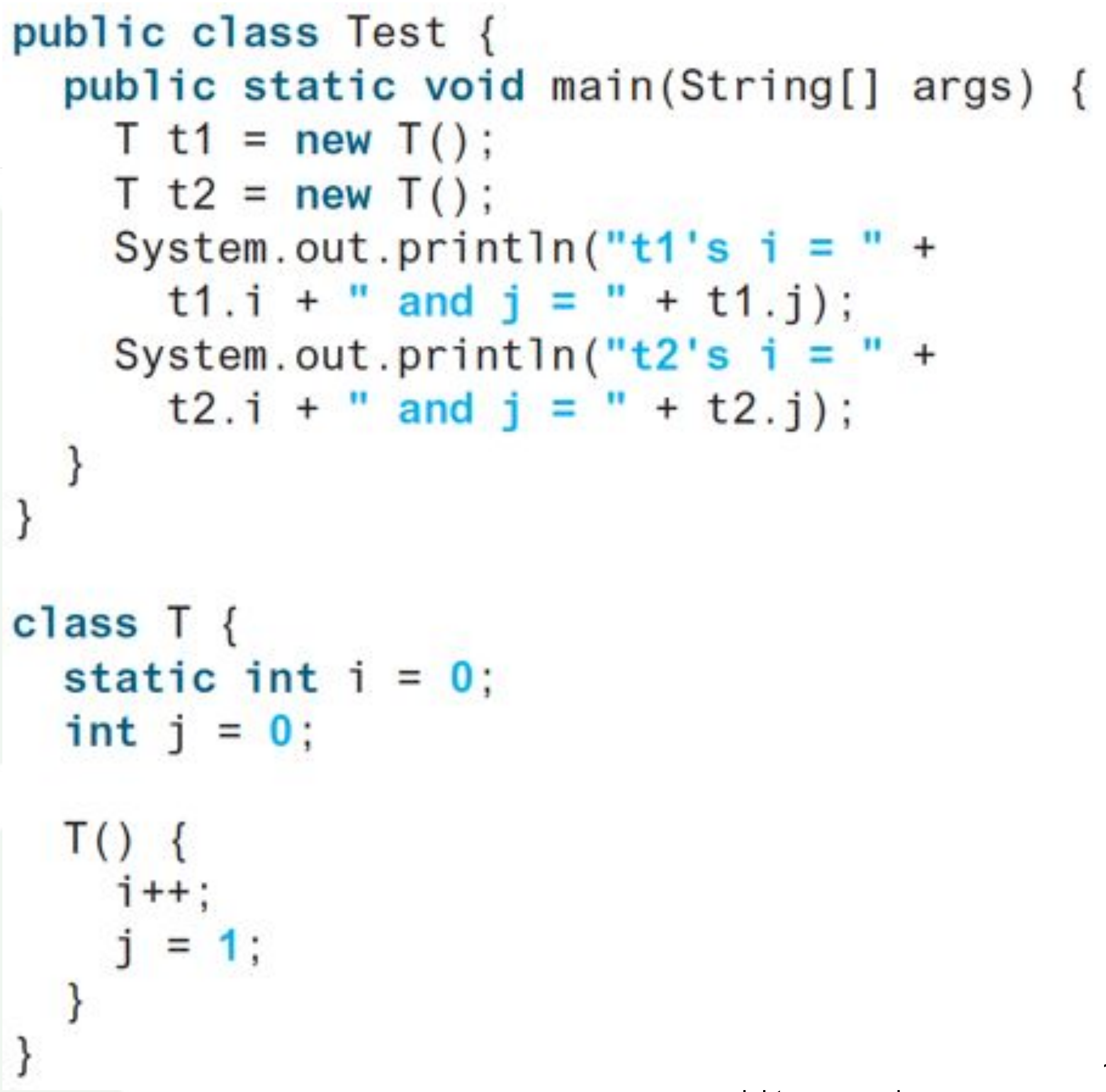
TotalArea

Run

```
public class Test {
  public static void main(String[] args) {
    Count myCount = new  Count();
    int times = 0;

    for (int i = 0; i < 100; i++)
      increment(myCount, times);

    System.out.println("count is " + myCount.count);
    System.out.println("times is " + times);
  }

  public static void increment(Count c, int times) {
    c.count++;
    times++;
  }
}
```

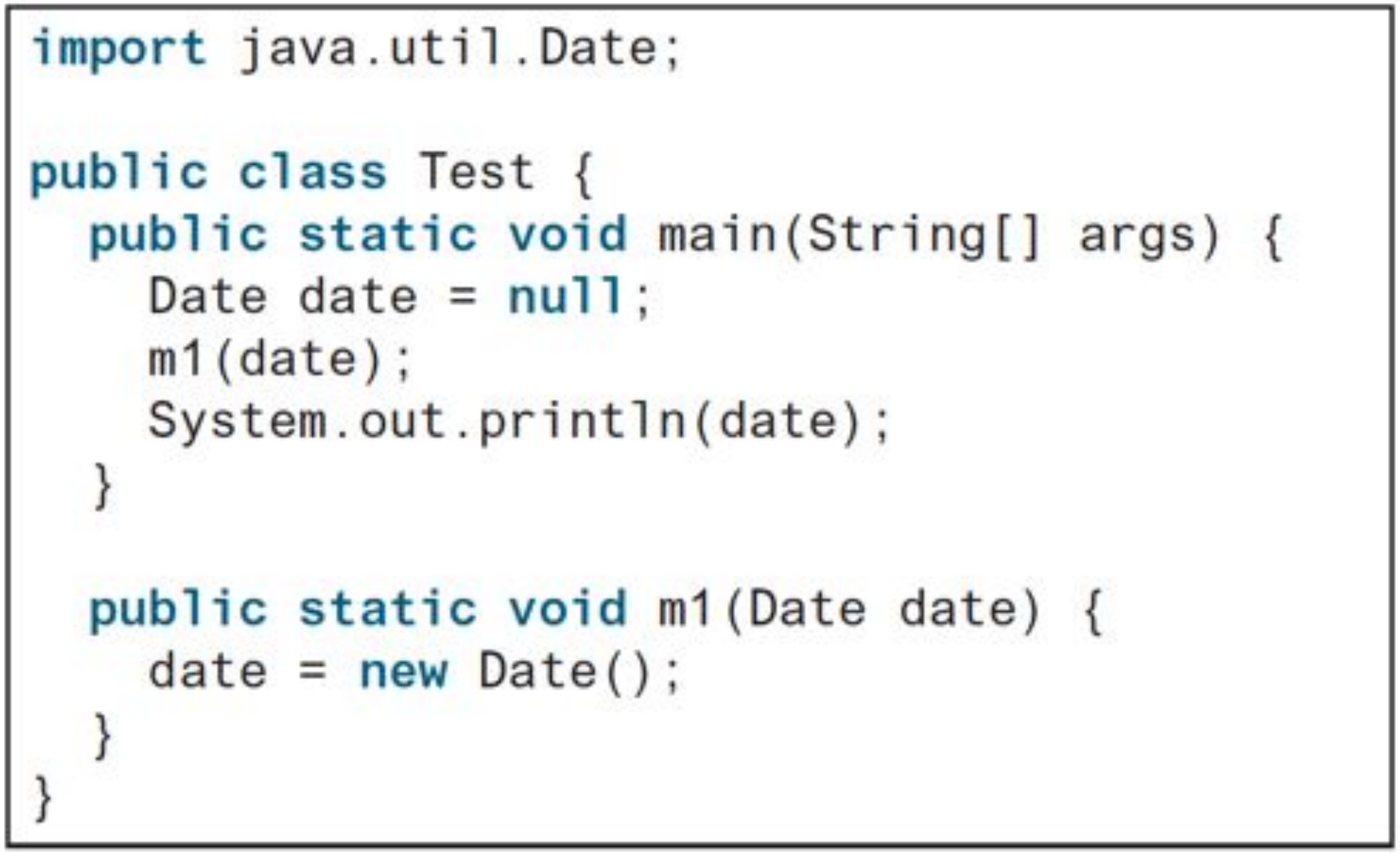
```
public class Count {
  public int count;

  public Count (int c) {
    count = c;
  }

  public Count () {
    count = 1;
  }
}
```

What is the output?



```
public class Test {
  public static void main(String[] args) {
    T t1 = new T();
    T t2 = new T();
    System.out.println("t1's i = " +
      t1.i + " and j = " + t1.j);
    System.out.println("t2's i = " +
      t2.i + " and j = " + t2.j);
  }
}

class T {
  static int i = 0;
  int j = 0;

  T() {
    i++;
    j = 1;
  }
}
```

What is the output?

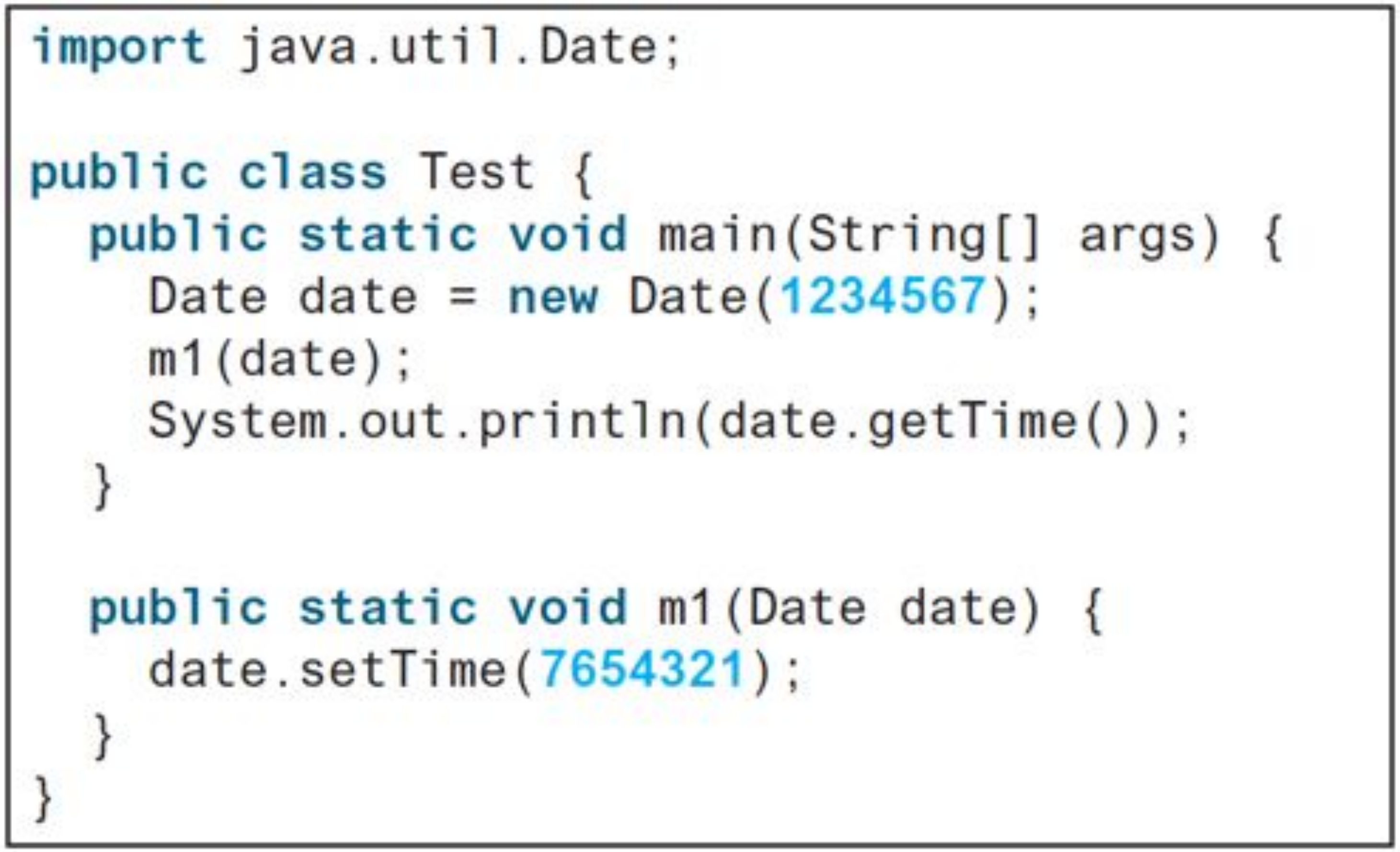


```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = null;
    m1(date);
    System.out.println(date);
  }

  public static void m1(Date date) {
    date = new Date();
  }
}
```

What is the output?

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = new Date(7654321);
  }
}
```

What is the output?



```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date.setTime(7654321);
  }
}
```

What is the output?

```java
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = null;
  }
}
```

Immutable Objects and Classes

If the contents of an object cannot be changed once the object is created, the object is called an *immutable object* 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

```
public class Student {
  private int id;
  private BirthDate birthDate;

  public Student(int ssn,
      int year, int month, int day) {
    id = ssn;
    birthDate = new BirthDate(year, month, day);
  }

  public int getId() {
    return id;
  }

  public BirthDate getBirthDate() {
    return birthDate;
  }
}
```

```
public class BirthDate {
  private int year;
  private int month;
  private int day;

  public BirthDate(int newYear,
      int newMonth, int newDay) {
    year = newYear;
    month = newMonth;
    day = newDay;
  }

  public void setYear(int newYear) {
    year = newYear;
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student(111223333, 1970, 5, 3);
    BirthDate date = student.getBirthDate();
    date.setYear(2010); // Now the student birth year is changed!
  }
}
```

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.

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.

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.

Reference the Hidden Data Fields

```
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

```
public class Circle {
  private double radius;

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

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

```
  public Circle() {
    this(1.0);
  }
```

this is used to invoke another constructor

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

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