Chapter 9

Objects and Classes

OO Programming in Java

Other than primitive data types (*byte*, *short*, *int*, *long*, *float*, *double*, *char*, *boolean*), everything else in Java is of type object.

Objects we already worked with:

```
String:     String name = new String("John Smith");

Scanner:    Scanner input = new Scanner(System.in);

Random:     Random generator = new Random(100);

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

What is an Object?

An *object* represents an entity in the real world that can be distinctly identified. For example, student, desk, circle, button, person, course, car, employee, department, store, computer, etc…

For instance, an object might represent a particular employee in a company. Each employee object handles the processing and management of data related to that employee object.

An object has a unique identity, state, and behaviors.

The *state* of an object consists of a set of *data fields* (*instance variables* or *properties*) with their current values.

The *behavior* of an object is defined by a set of methods defined in the class from which the object is created.

A ***class*** describes a set of similar objects.

Object Representation

In OO programming (e.g., Java), an object is associated with a memory space referenced by the object name.

The memory space is allocated when using operator **`new`** to create the object.

The object's memory space holds the values of the data fields (instance variables) of the object.

The class **constructor** method creates the object.

What is a Class?

A class is the blueprint (template) that defines objects of the same type, a set of similar object, such as students.

The class uses methods to define the behaviors of its objects.

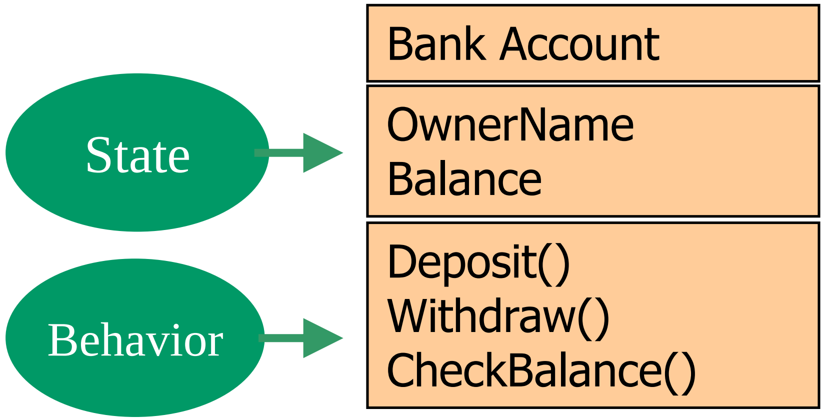
The class that contains the main method of a Java program represents the entire program

A class provides a special type of methods, known as constructors, which are invoked to construct (create) objects from the class.

Multiple objects can be created from the same class.

Example

A class
(the concept)



State

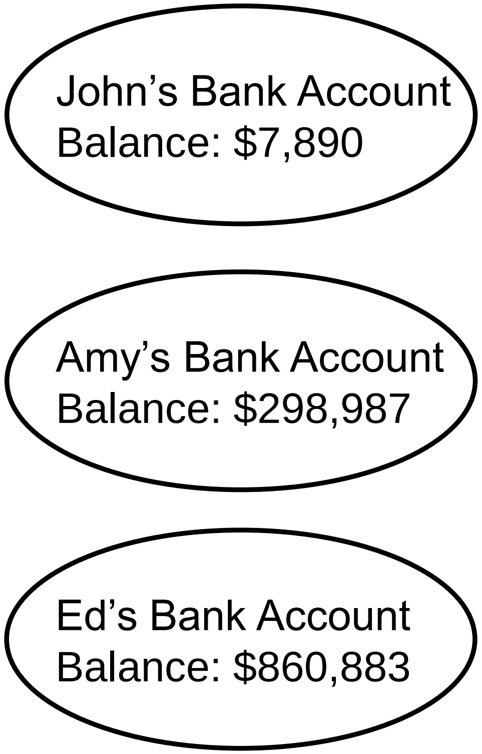
Behavior

Bank Account

OwnerName
Balance

Deposit()
Withdraw()
CheckBalance()

An object
(the implementation)



John’s Bank Account
Balance: $7,890

Amy’s Bank Account
Balance: $298,987

Ed’s Bank Account
Balance: $860,883

Multiple objects
from the same class

Writing Classes

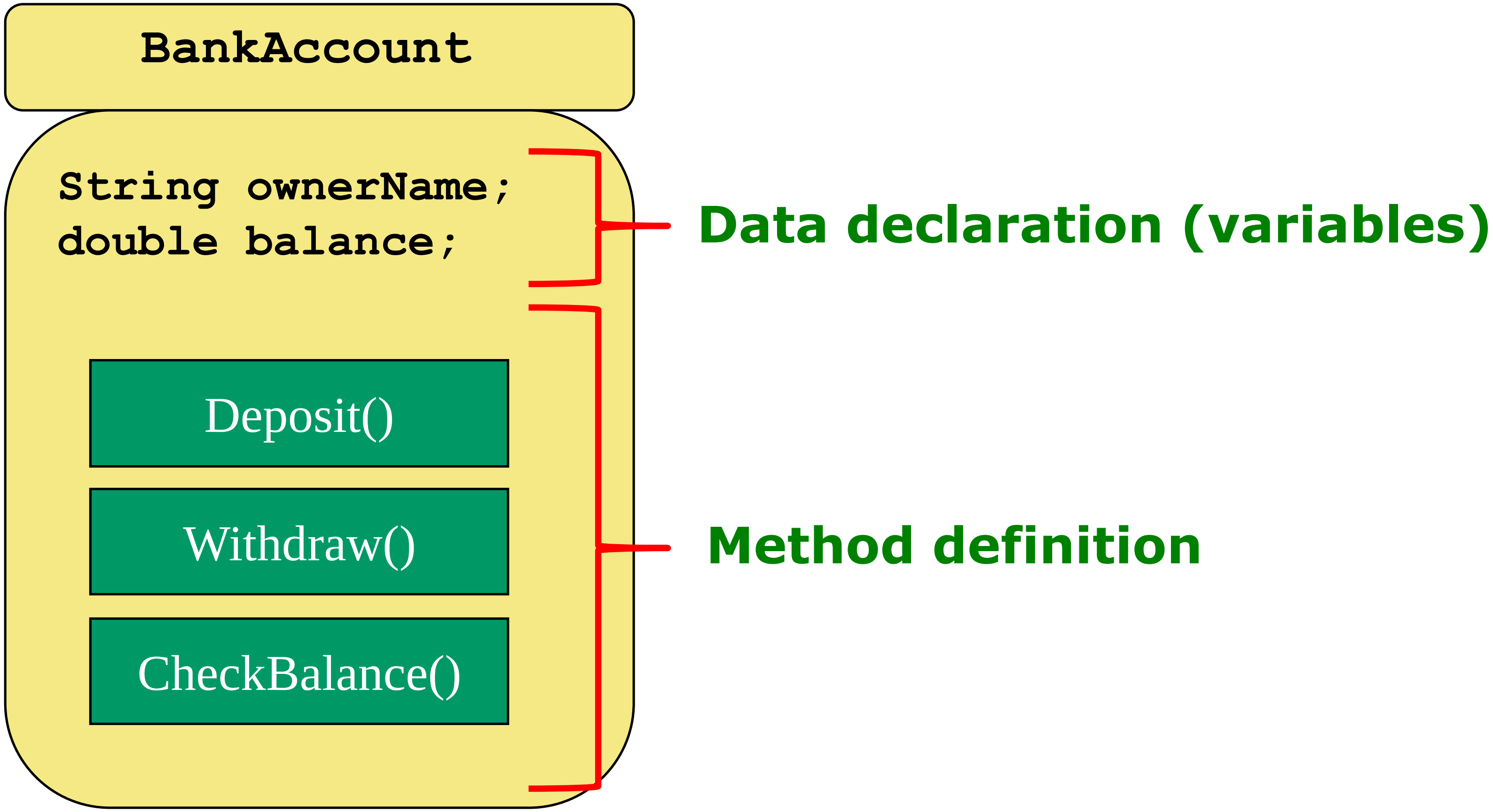
The programs we've written in previous examples have used classes defined in the Java standard class library.

Now, we will begin to design programs that rely on classes that we write ourselves.

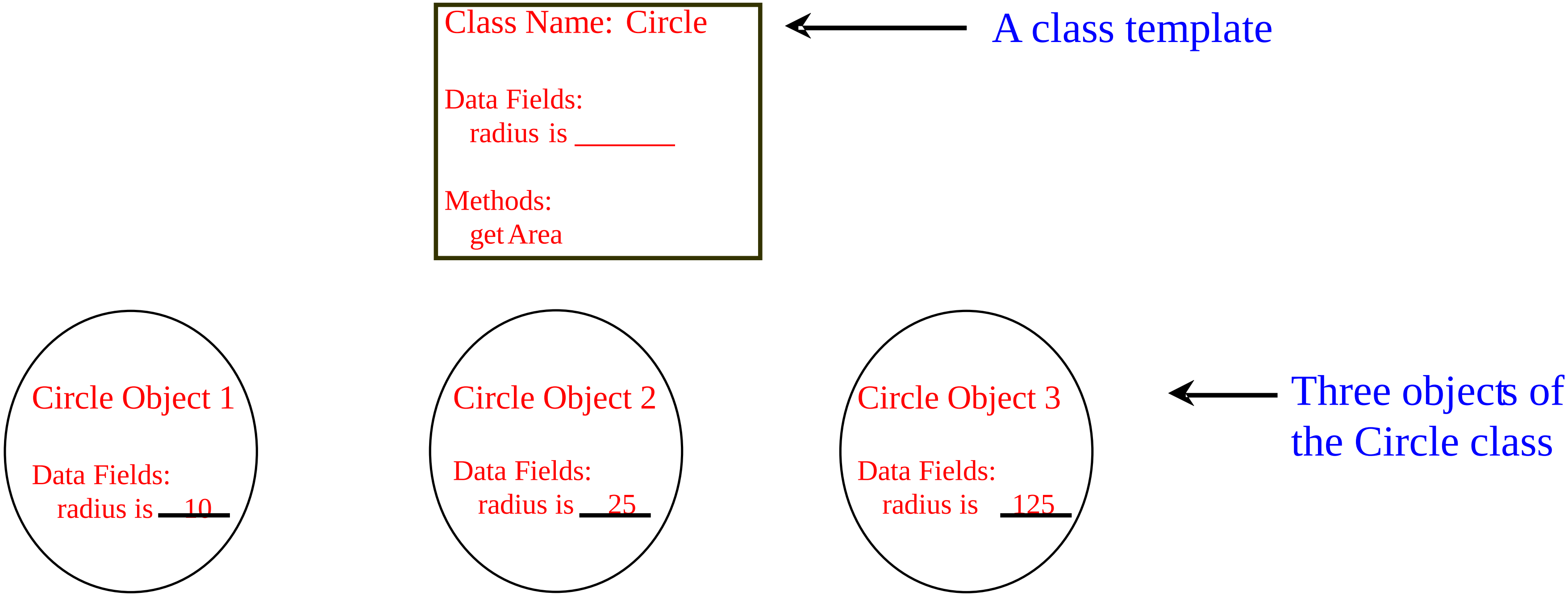
The class that contains the **`main`** method is just the starting point of a program. This is also known as the driver/test program

Writing Classes

A class can contain data declarations and method declarations.



Another Example



Class Name: Circle

Data Fields:
radius is _______

Methods:
get Area

A class template

Circle Object 1

Data Fields:
radius is 10

Circle Object 2

Data Fields:
radius is 25

Circle Object 3

Data Fields:
radius is 125

Three objects of
the Circle class

A class has both date fields (attributes/variables) and methods. The data fields represent the state of an object; while the methods represent the behavior of all objects.

Constructor Methods

The contractor method creates the object in the memory with the help of the Operating System.

Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

A class can have multiple versions of the constructor method, allowing the user to create the object in different ways.

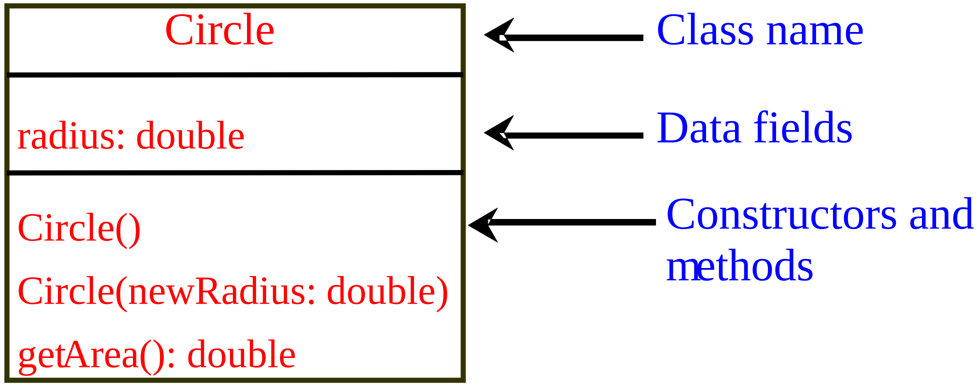
The constructor method must have same name as the class name.

Constructors do not have a return type, not even **`void`**.

A constructor with no parameters is called *no-arguments constructor.*

UML Class Diagram

(Unified Modeling Language)



Circle

radius: double

Circle()

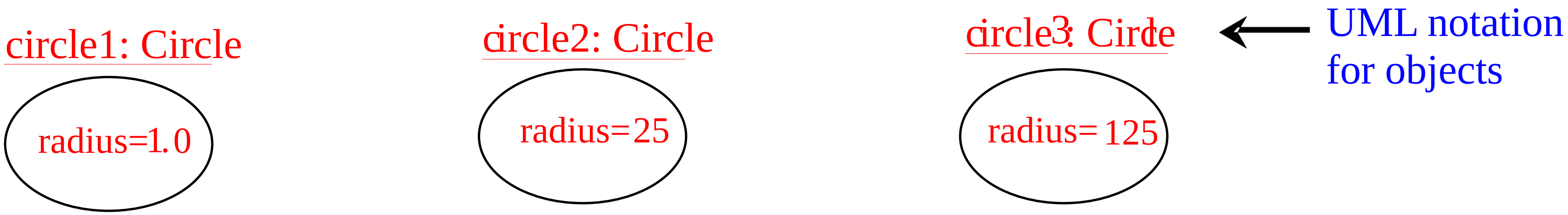
Circle(newRadius: double)

getArea(): double

Class name

Data fields

Constructors and methods



circle1: Circle

radius=1.0

circle2: Circle

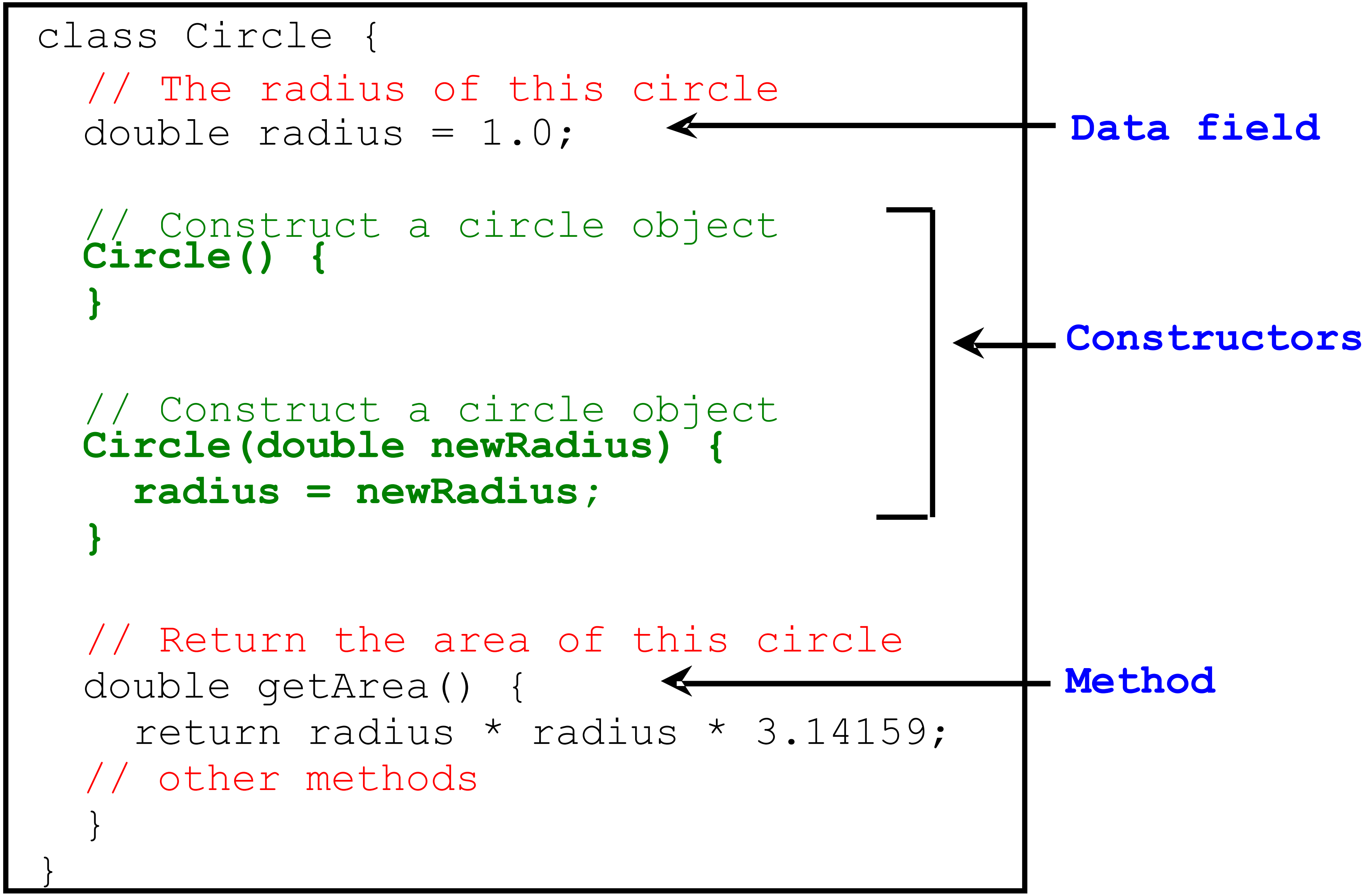
radius=25

circle3: Circle

radius=125

UML notation for objects

Class Circle Constructors



```
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;
  // other methods
  }
}
```

Data field

Constructors

Method

Creating Objects

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

To declare a reference variable, use the syntax:

```
          ClassName objectRefVar;
```

Example:

```
Circle circle1, circle2;    //reference variables
circle1 = new Circle();     //calls first constructor
circle2 = new Circle(25.0); //calls second constructor
```

OR

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(25.0);
```

Default Constructor

A class may be declared without constructors.

This constructor, called *a default constructor*, is provided automatically *only if no constructors are explicitly declared in the class.*

Accessing the Object

Referencing the object’s data:

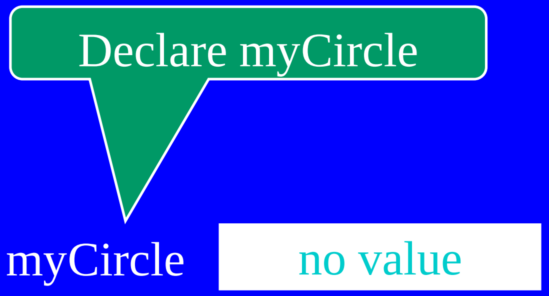
```
objectRefVar.data

double radius1 = circle1.radius; //data field
```

Invoking the object’s method:

```
objectRefVar.methodName(arguments)

double area1 = circle1.getArea(); //class method
```

Trace Code

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Declare myCircle

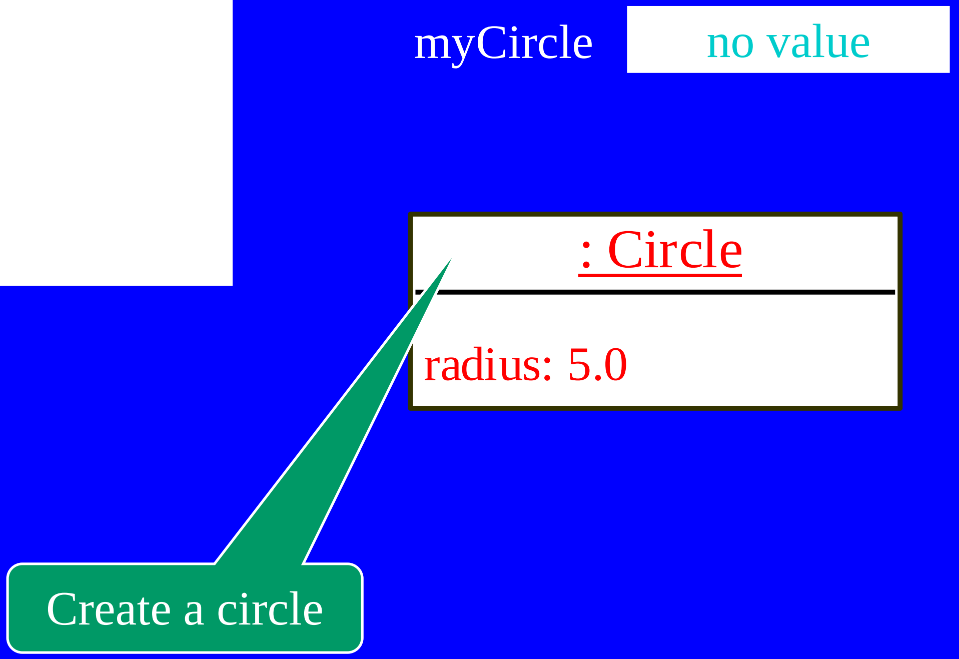
myCircle

no value

animation

Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

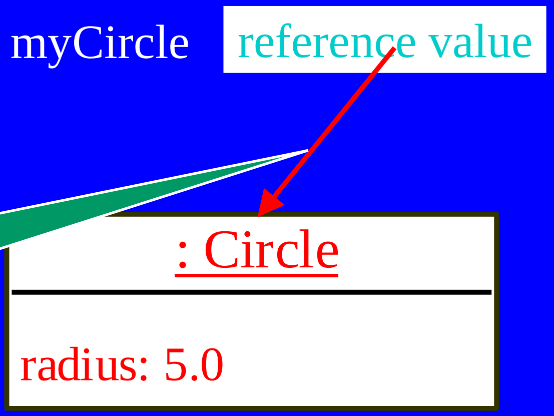


animation

Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

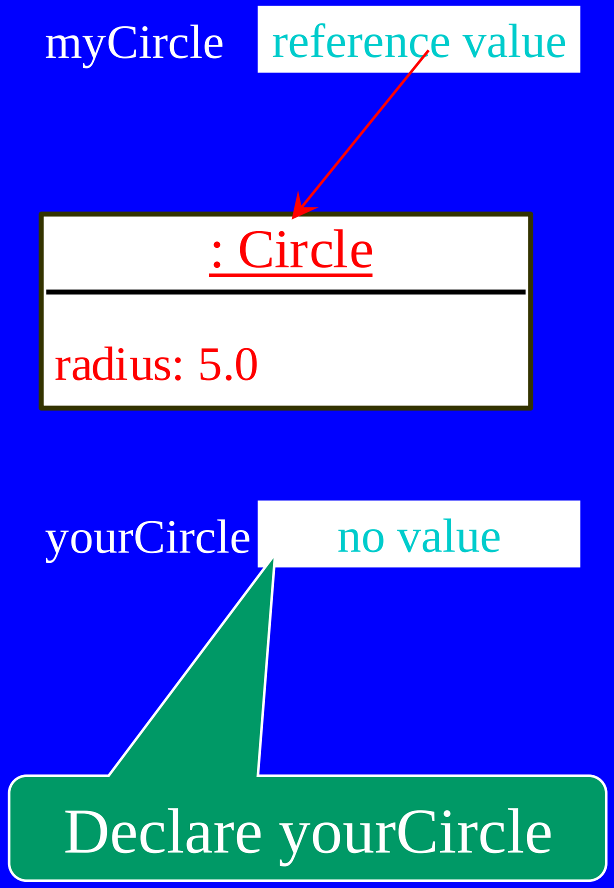
Assign object reference to myCircle



animation

Trace Code, cont.

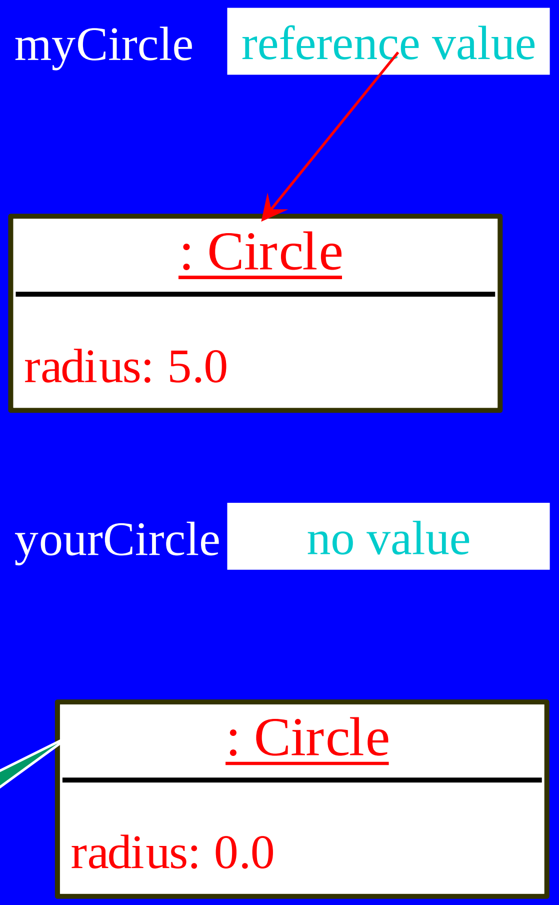
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



animation

Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



myCircle reference value

: Circle

radius: 5.0

yourCircle no value

: Circle

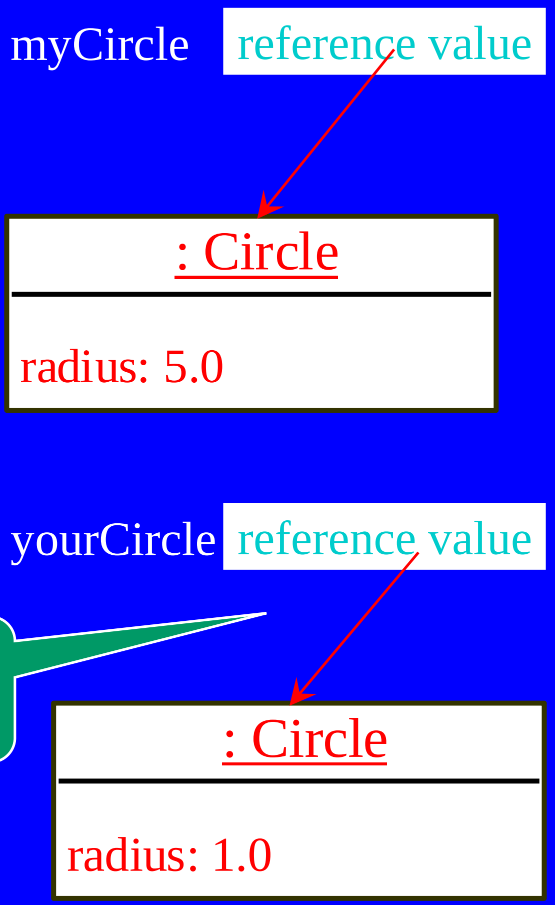
radius: 0.0

Create a new Circle object

animation

Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

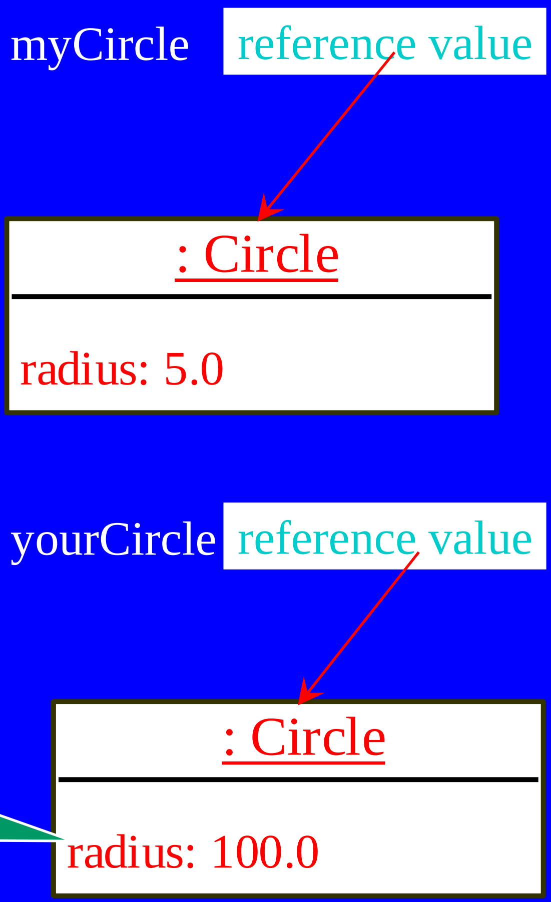


Assign object reference to yourCircle

animation

Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Change radius in yourCircle

Caution

Recall that we used

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

to invoke a method in the Math class.

Can you invoke `getArea()` using `circle1.getArea()`?

All the methods defined in the Math class are static (defined using the **`static`** keyword). However, method `getArea()` is non-static. It must be invoked from an object using this syntax:

```
    objectRefVar.methodName(arguments)
    (e.g., circle1.getArea())
```

Reference Data Fields

The data fields can be of reference types.

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

For example, Class Student contains a data field name of the type String (an array of characters).

```
public class Student {
  // data fields
  String name; //default value null. Why?
  int age;     //default value 0
  boolean isScienceMajor; //default value false
  char gender; //default value '\u0000', prints out as 00
}
```

Default Value for a Data Field

```
public class Test {
 public static void main(String[] args) {
   Student student1 = new Student(); //create student object
   System.out.println("name? " + student1.name);
   System.out.println("age? " + student1.age);
   System.out.println("isScienceMajor? " + student1.isScienceMajor);
   System.out.println("gender? " + student1.gender);
 }
}
```

Output:

```
name? null
age? 0
isScienceMajor? false
gender? 00
```

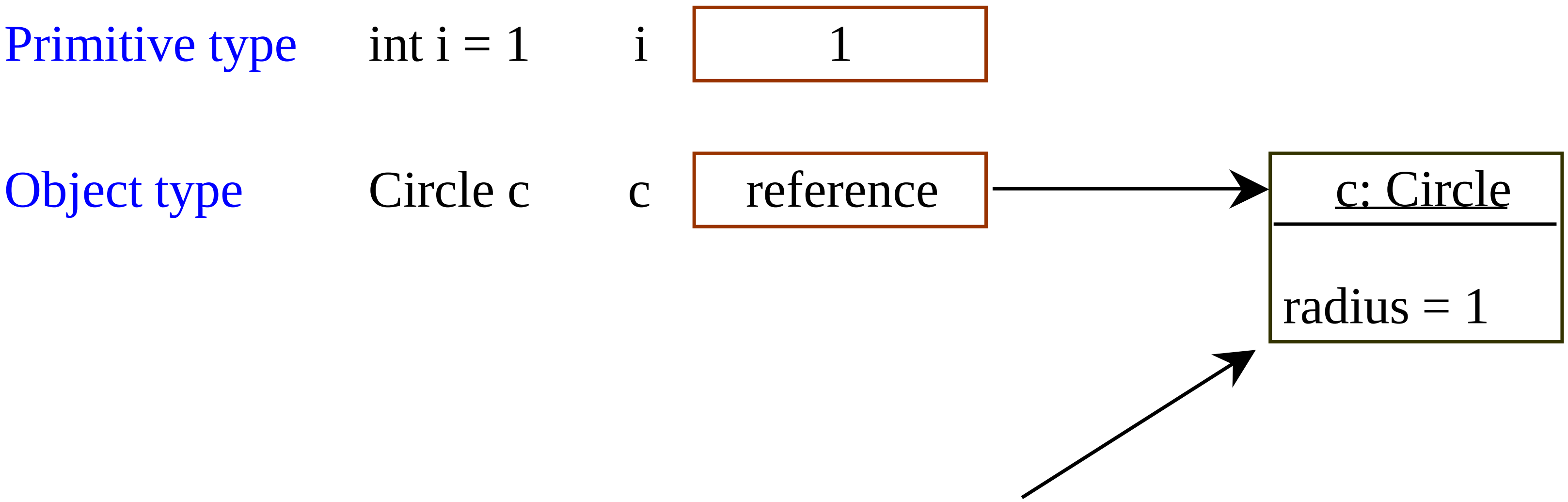
Default Values Inside Methods

Rule: Java assigns no default values to local variables inside a method. A method's local variables must be initialized.

```
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);
  }
}
```

Compilation error: variables not initialized

Primitive Type vs. Object Type



Primitive type int i = 1 i 1

Object type Circle c c reference

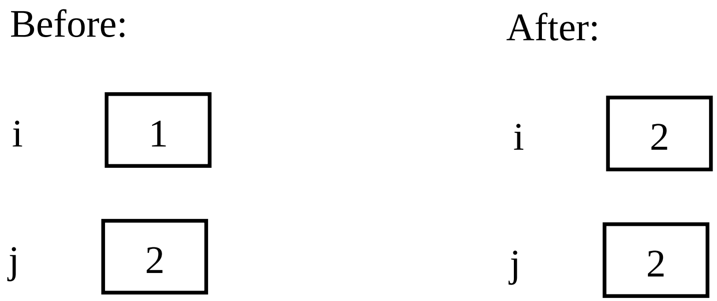
c: Circle

radius = 1

Created using: **`Circle c = new Circle(1.0);`**

Primitive Type vs. Object Type

Primitive type assignment i = j



Before:

i 1

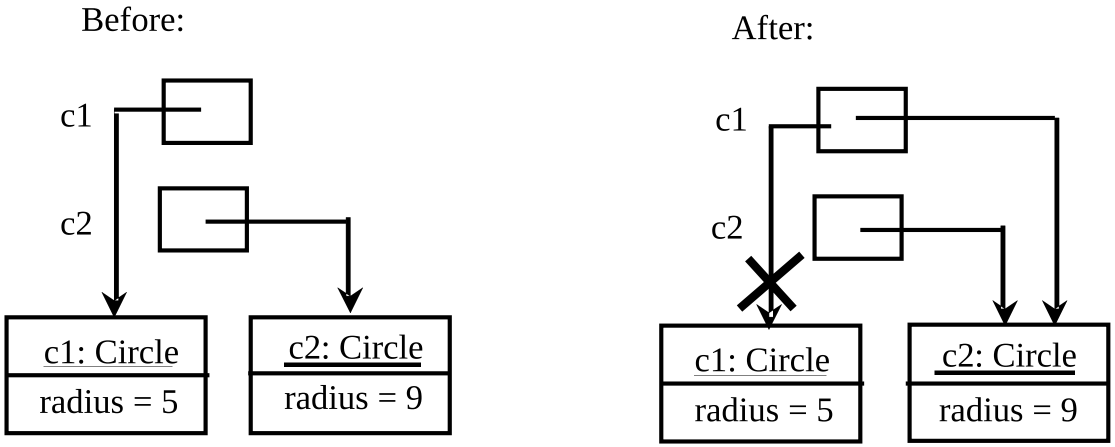
j 2

After:

i 2

j 2

Object type assignment c1 = c2



Before:

c1

c2

c1: Circle

radius = 5

c2: Circle

radius = 9

After:

c1

c2

c1: Circle

radius = 5

c2: Circle

radius = 9

Garbage Collection

On the previous slide, after the assignment statement

```
    c1 = c2;  //circle objects
```

`c1` points to the same object referenced by `c2`.

The object previously referenced by c1 is no longer referenced/accessible (i.e., 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 in the program.

Static Variables, Constants, and Methods

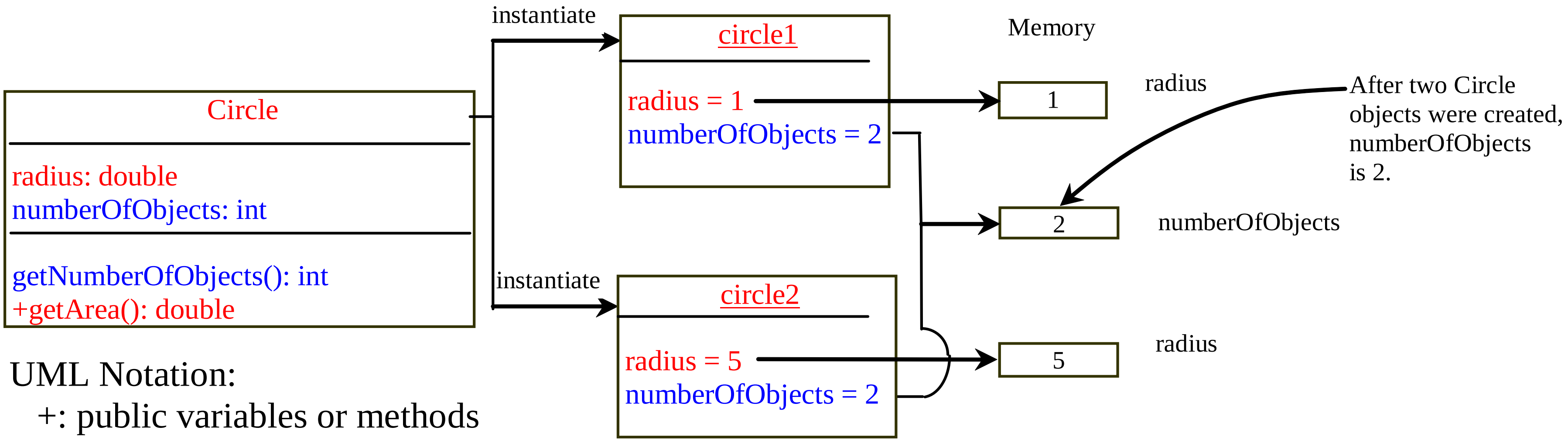
Static variables are shared by all the objects of the class.

Static methods are not tied to a specific object, applied to all objects of the class.

Static constants (`final` variables) are shared by all the objects of the class.

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

Static Variables, Constants, and Methods



UML Notation:

+: public variables or methods

blue: static variables or methods

Inner Classes

An Inner (nested) class is a class declared as a member of another class. Thus the inner class may access all members of the outer class.

An object of the inner class is cereated with each object of the outer class.

See example next slide.

Inner Classes

```
//Test inner classes
class OuterClass
{
   public void OuterMethod()
   { System.out.println("I am in the outer class method"); }

   class InnerClass  //inner class
   {
      public void InnerMethod()
      { System.out.println("I am in the inner class method"); }
   }
}
=========================================================================
class Driver {
   public static void main(String[] args) {
      //outer class object
      OuterClass outerObject = new OuterClass();
      outerObject.OuterMethod(); //method call

      //inner class object. Notice the syntax
      OuterClass.InnerClass innerObject = new OuterClass().new InnerClass();
      innerObject.InnerMethod(); //method call
   }
}
```

Visibility Modifiers

By default (no qualifier), the class variable or method can be accessed by any class in the same package, but not other packages.

`Public:`

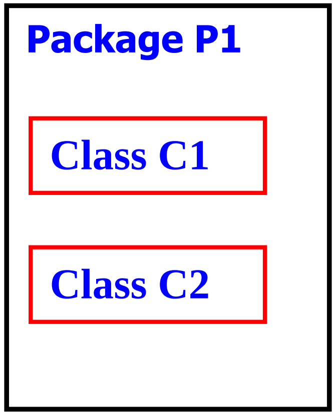
The class, data, or method is visible to any class in any package.

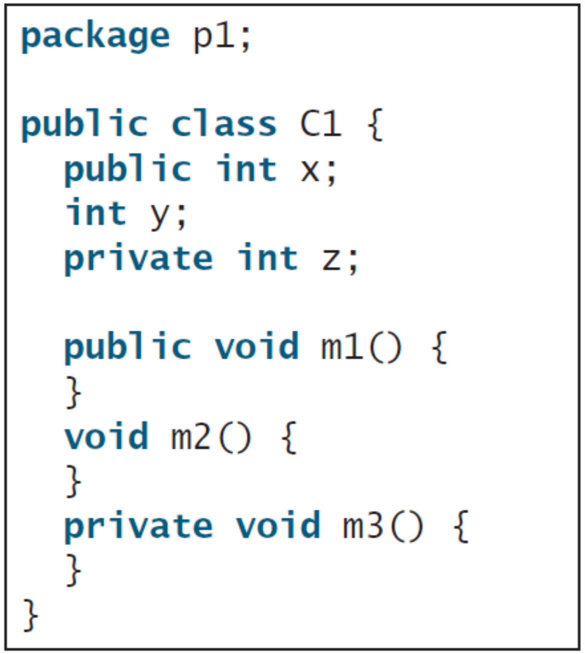
`Private:`

The data or methods can be accessed only by the declaring class.

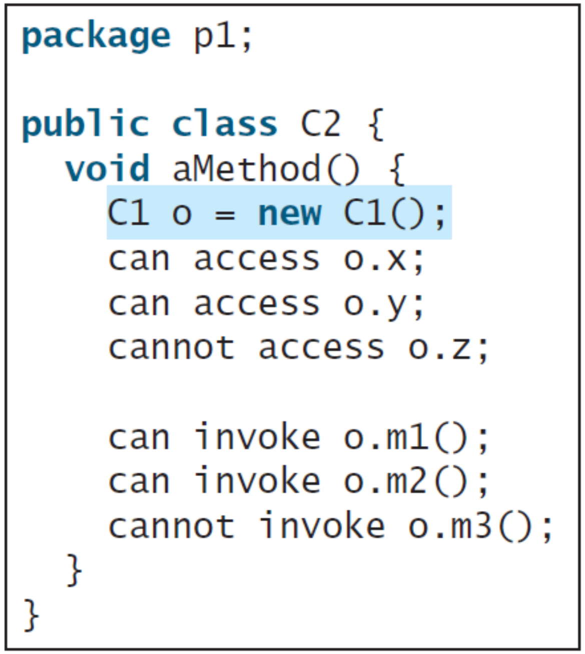
The **`get`** and **`set`** methods are used to read and modify private variables.

Visibility Modifiers Example - 1

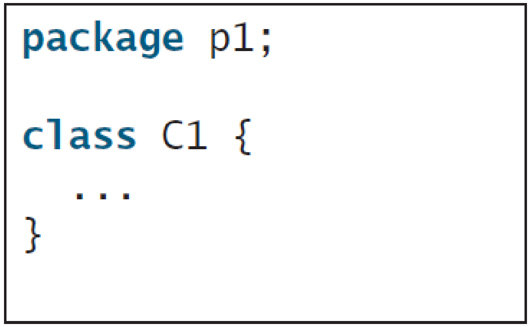




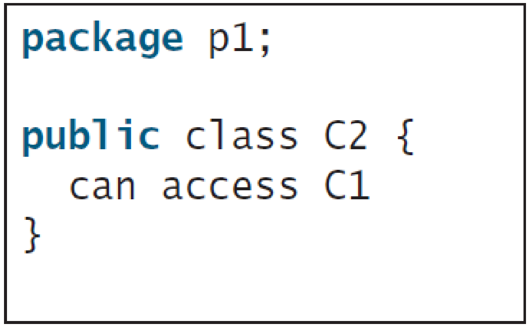
```
package p1;

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();
  }
}
```

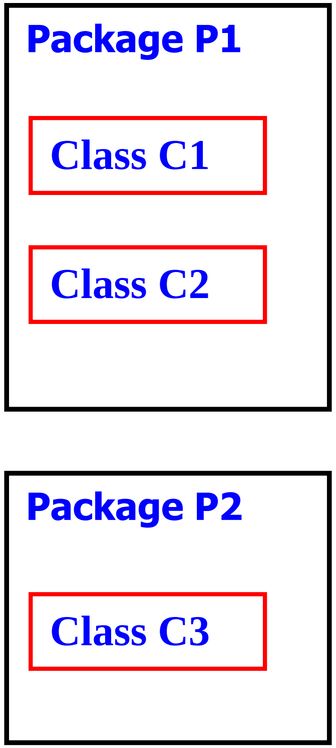


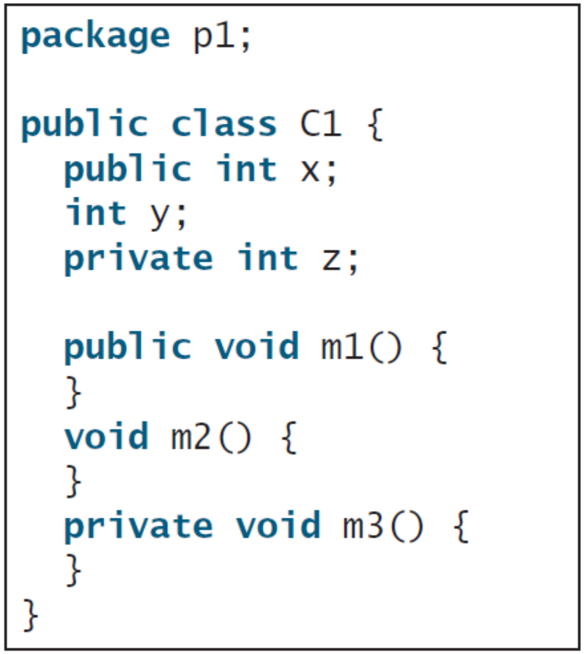
```
package p1;

class C1 {
  ...
}
```



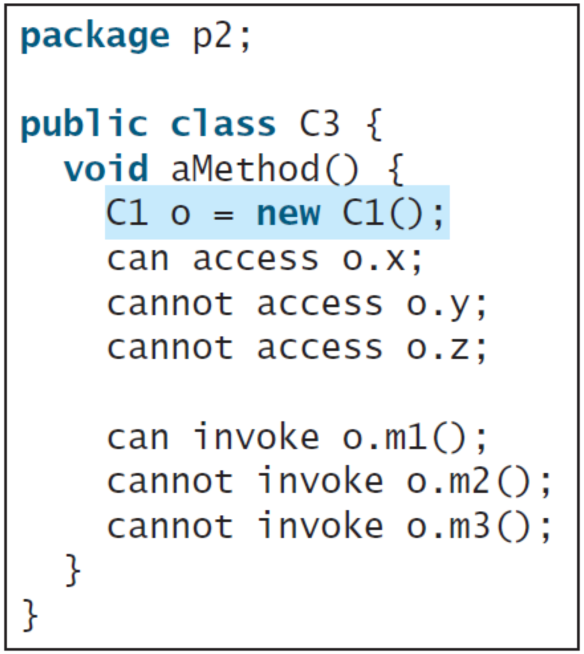
```
package p1;

public class C2 {
  can access C1
}
```

Visibility Modifiers Example - 2

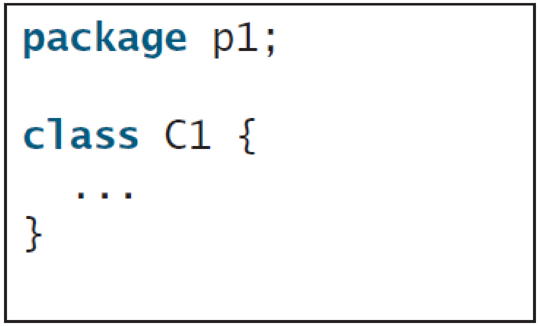




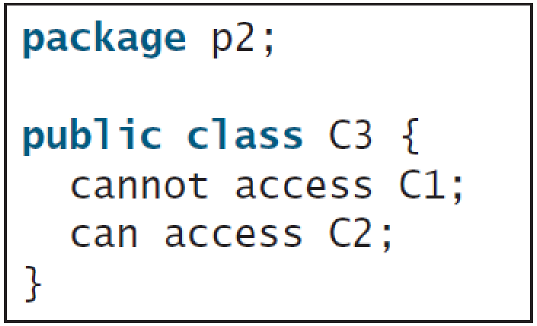
```
package p1;

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

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



```
package p2;

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

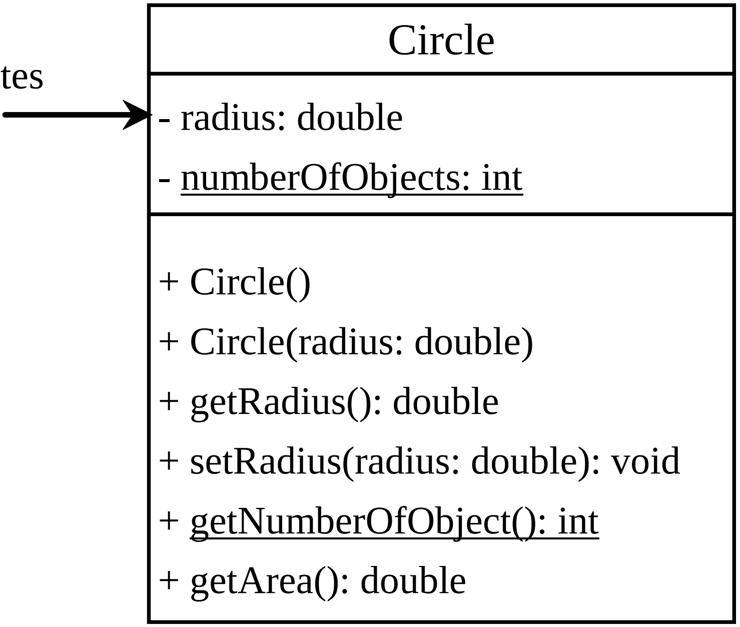
Data Field Encapsulation

Encapsulation is the idea of hiding the class internal details that are not required by clients/users of the class.

Why? To protect data and to make classes easy to maintain and update.

How? Always use private variables!

The - sign indicates
private modifier



Circle

- radius: double
- numberOfObjects: int

+ Circle()
+ Circle(radius: double)
+ getRadius(): double
+ setRadius(radius: double): void
+ getNumberOfObject(): int
+ getArea(): double

The radius of this circle (default: 1.0).

The number of circle objects created (**static**).

Constructs a default circle object.

Constructs a circle object with the specified radius.

Returns the radius of this circle.

Sets a new radius for this circle.

Returns the number of circle objects created.

Returns the area of this circle.

Visibility Modifiers - Comments - 1

Class members (variables or methods) that are declared with *public visibility* can be referenced/accessed anywhere in the program.

Class members that are declared with *private visibility* can be referenced/accessed only within that class.

Class members declared without a visibility modifier have *default (package) visibility* and can be referenced/accessed by any class in the same package.

Public variables violate encapsulation because they allow class clients to "reach in" and modify the values directly. *Therefore variables should not be declared with public visibility.*

Visibility Modifiers - Comments - 2

Methods that provide the object's services must be declared with *public visibility* so that they can be invoked by clients (users of the object).

Public methods are also called *service methods*.

A method created simply to assist a service method is called a *support method.*

Since a support method is not intended to be called by a client, it should be declared with *private visibility.*

Example - 1

```
public class CircleWithPrivateDataFields
{
   private double radius = 1;
   private static int numberOfObjects = 0;

   public CircleWithPrivateDataFields() { numberOfObjects++; }

   public CircleWithPrivateDataFields(double newRadius) {
     radius = newRadius;
     numberOfObjects++;
   }

   public double getRadius() { return radius; }

   public void setRadius(double newRadius) {
     radius = (newRadius >= 0) ? newRadius : 0; //no negative radius
   }

   public static int getNumberOfObjects() {return numberOfObjects; }

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

```java
public class TestCircleWithPrivateDataFields {

  public static void main(String[] args) { // Main method

    // Create a Circle with radius 10.0
    CircleWithPrivateDataFields myCircle =
      new CircleWithPrivateDataFields(10.0);

    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    // Increase myCircle's radius by 10%
    myCircle.setRadius(myCircle.getRadius() * 1.1);

    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());
  }
}
```

Note: variable radius cannot be directly accessed.
Only through the class methods!

```
Output:

The area of the circle of radius 10.0 is 314.1592653589793
The area of the circle of radius 11.0 is 380.132711084365
```

Passing Objects to Methods

Remember,

Passing by value for primitive types:
The actual value is copied into the formal parameter. Change to the actual parameters is local to the method.

Passing by value for reference types:
The reference value (memory address) is copied to the actual parameter, not the object itself. Any changes to the passed reference will be reflected on the object outside the method (similar to passing strings and arrays).

```
public class TestPassObject {

  public static void main(String[] args) {
  CircleWithPrivateDataFields myCircle = new
  CircleWithPrivateDataFields(1);

   // Print areas for radius 1, 2, 3, 4, and 5.
   int n = 5;
   printAreas(myCircle, n);

   // See myCircle.radius and times.
   System.out.println("\n" + "Radius is " + myCircle.getRadius());
   System.out.println("n is " + n);
   }

   // Print a table of areas for radius.
   public static void printAreas(CircleWithPrivateDataFields c,
                                 int times) {
     System.out.println("Radius\t\tArea");
     while (times >= 1) {
       System.out.println(c.getRadius() + "\t\t" + c.getArea());
       c.setRadius(c.getRadius() + 1);
       times = times - 1;
     }
   }

}
```

Array of Objects

Consider:

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

An array of objects is actually an *array of reference variables*.

Thus, invoking `circleArray[1].getArea()` involves two levels of referencing:

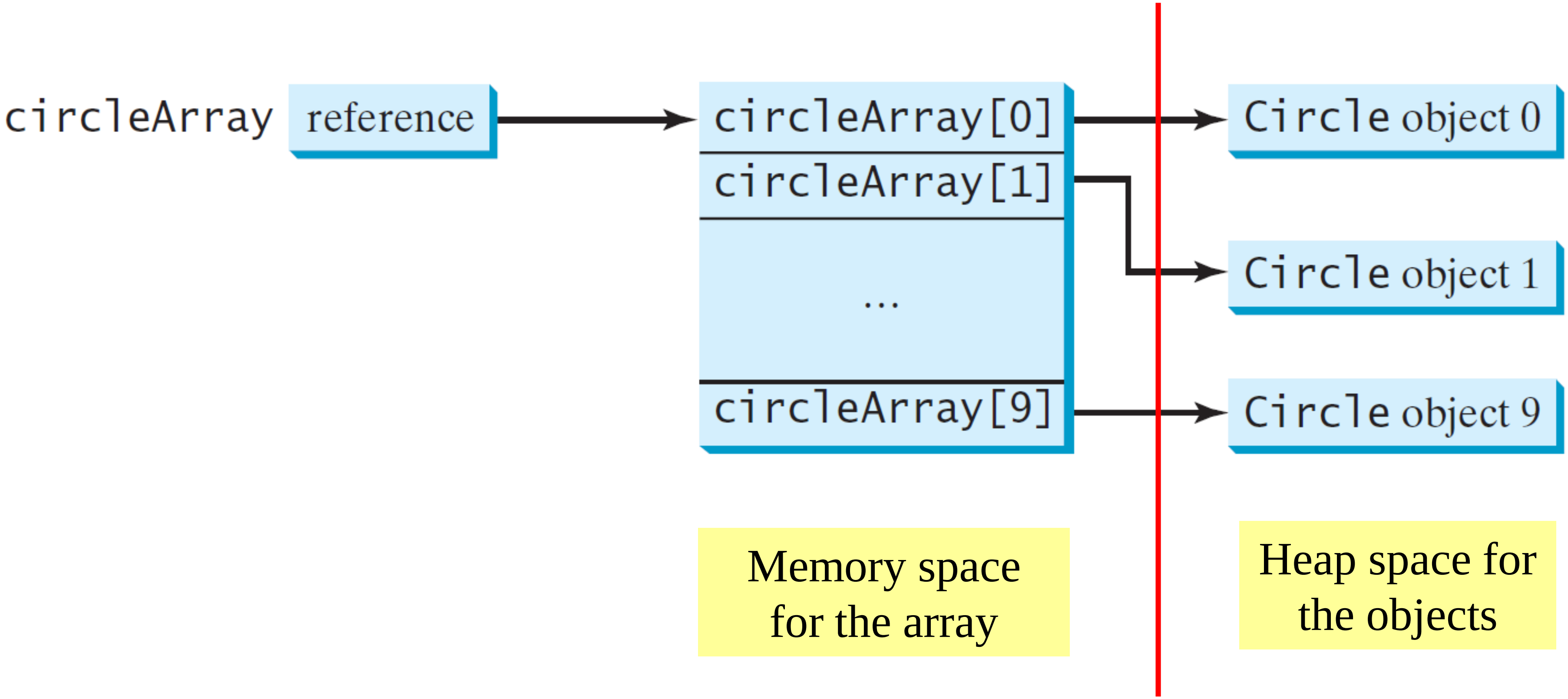
`circleArray` references the entire array

`circleArray[1]` references a Circle object

See next slide.

Array of Objects

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



Example

```
public class TotalArea {
  public static void main(String[] args) {

    CircleWithPrivateDataFields[] circleArray; //Declare circleArray
    circleArray = createCircleArray();//Create circleArray

    //Print circleArray and total areas of the circles
    printCircleArray(circleArray);
  }

  //Create an array of Circle objects
  public static CircleWithPrivateDataFields[] createCircleArray() {
    CircleWithPrivateDataFields[] circleArray =
      new CircleWithPrivateDataFields[5];

    for (int i = 0; i < circleArray.length; i++) {
      circleArray[i] =
        new CircleWithPrivateDataFields(Math.random() * 100);
    }
    return circleArray; //Return Circle array
  }

// next slide
//Print an array of circles and their total area
public static void printCircleArray(
     CircleWithPrivateDataFields[] circleArray)
{
    System.out.println("Radius" + "\t\t\t\t" + "Area");
    for (int i = 0; i < circleArray.length; i++) {
       System.out.println(circleArray[i].getRadius() + "\t\t" +
                        circleArray[i].getArea());
    }

    System.out.println("-----------------------------------------");
    //Compute and display the result
    System.out.println("The total areas of circles is\t" +
                       sum(circleArray));
}

public static double sum(  //Static method to add circle areas
     CircleWithPrivateDataFields[] circleArray) {

   double sum = 0; //Initialize sum

   for (int i = 0; i < circleArray.length; i++)//Add areas to sum
      sum = sum + circleArray[i].getArea();
   return sum;
  }
}
```

Output:

```
----jGRASP exec: java TotalArea

Radius                        Area
0.049319                      0.007642
81.879485                     21062.022854
95.330603                     28550.554995
92.768319                     27036.423936
46.794917                     6879.347364
-----------------------------------------
The total areas of circles is 83528.356790

----jGRASP: operation complete.
```

Immutable Objects and Classes

If the content of an object cannot be changed once it is created, the object is called an *immutable object* and its class is called an *immutable class*.

For example, If you delete the **`set`** method in the **`Circle`** class in Listing 8.10, the class would be immutable (not changeable) because radius is private and cannot be changed without a set method.

Note: A class with all variables being private and without mutators (*set methods*) is not necessarily immutable. For example, the following class Student has all private data fields and no mutators, but it is mutable (changeable).

Immutable Object 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!
  }
}
```

So, What Class is Immutable?

For a class to be immutable, it must mark all data fields (variables) private and provide no mutator (set) methods and no accessor (get) methods that would return a reference to a mutable (changeable) data field object.

Scope of Variables - Revisited

The scope of instance variables 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 point and continues to the end of the block that contains the variable. Example, int i=0; in a *for* loop.

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.

Common uses of keyword **`this`** include:

- referencing a class’s *hidden (private) data fields*.
- allowing a constructor to call another constructor in the same class.

Referencing Hidden Data Fields

```java
public class A {
  private int i = 5;
  private static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    A.k = k;
  }
}
```

```
Suppose that a1 and a2 are two objects of class A.

A a1 = new A(); A a2 = new A();

Invoking a1.setI(10) is to execute
   this.i = 10; //this refers to a1

Invoking a2.setI(45) is to execute
   this.i = 45; //this refers to a2
```

Calling Overloaded Constructors

```
public class Circle {
  private double radius;

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

  public Circle() {
    this(1.0);
  }

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

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

this is used to invoke another constructor

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

Class **Date** - Revisited

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/object for the current date and time and use the class toString method to return the date and time as a string.

Example:

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

Output:

```
Sat Nov 08 12:31:11 EST 2014
```

Class **Random** - Revisited

You have used **`Math.random()`** method 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.

Class **Random** - Revisited

Be Careful! If two Random objects have the same seed, they will generate identical sequences of numbers.

Example: create two Random objects with the same seed 3.

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

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

Class **Random** - Revisited

To avoid that, simply use the current time as the seed value.

```
Random random1 = new Random(); //current time is the seed
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
    System.out.print(random1.nextInt(1000) + "\t");
Random random2 = new Random(); //current time is the seed
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
    System.out.print(random2.nextInt(1000) + "\t");
```

```
From random1: 957   496   459   198   84    788   33    254   441   101
From random2: 583   672   320   735   261   122   956   489   303   120
```

End of Chapter 9