Chapter 9 Objects and Classes

OO Programming Concepts

- **Object-oriented programming (OOP)**

- 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.

Objects

Class Name: Circle

Data Fields:
radius is _____

Methods:
getArea

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

An object consists of **two parts/members**:

- ***attributes/properties/***
  /data-fields /data-member/member-varialble: *state*
- ***methods/operations/member-function:*** *behavior*

Objects

Class Name: Circle

Data Fields:
radius is _____

Methods:
getArea

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

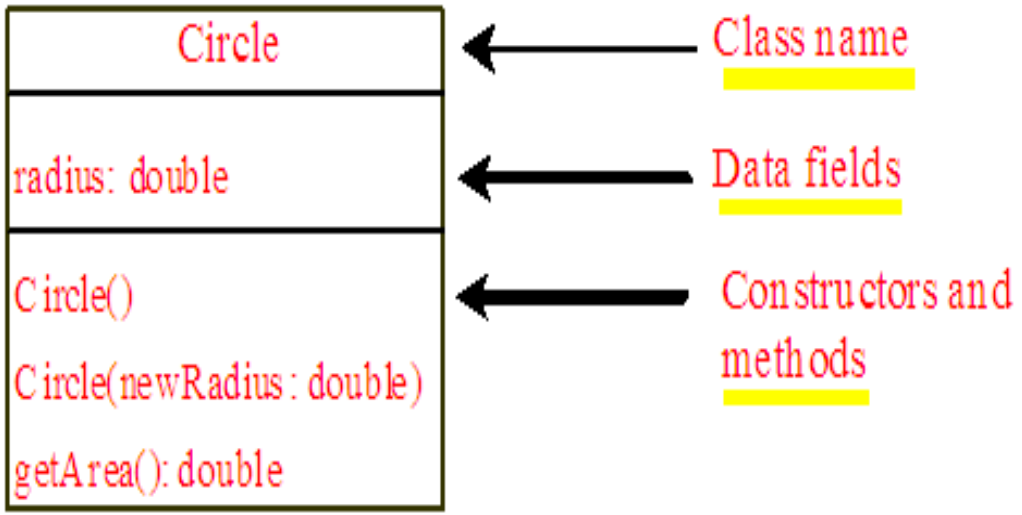
Three objects of the Circle class

```
Circle object1, object2, object3;
object1 = new Circle(10);
object2 = new Circle(25);
object3 = new Circle(125);

float a1 = object1.getArea();
```

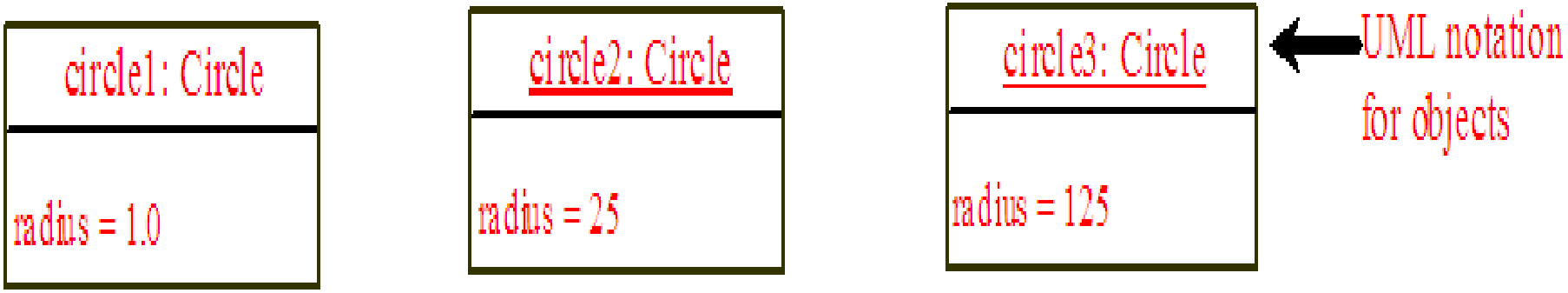
UML Diagram

Class Diagram: **className**

UML Class Diagram



Object Diagram: **objectName: className**



Classes

Class : template that define objects of the same type.

- **Constructor:**
  - a special method to construct objects from the class.
  - **ClassName(**parameterName: parameterType**)**
    - method name is the class name
    - **no return value**

The constructor is denoted as

```
ClassName(parameterName: parameterType)
```

The method is denoted as

```
methodName(parameterName: parameterType): returnType
```

- main class: the class that contains the **main** method
  - only one main class in a program
  - can be run

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;
  }
}
```

Data field

Constructors

Method

Can this class be run?

Creating Objects Using Constructors

Constructors

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

```
new ClassName();
```

Example:

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

Default Constructor

- **provided automatically *only* *<u>if no constructors are explicitly declared</u>* *in the class*.**
  - <u>A class may be declared without constructors</u>.
  - In this case, <u>a **no-arg** constructor</u> (i.e., a constructor with no parameters) with an **<u>empty body</u>** is <u>implicitly declared</u> in the class.

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









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

```
Circle()  {
}
```

Declaring Object Reference Variables

```
ClassName objectRefVar;
```

eg. `Circle myCircle;`

– Declares the variable **myCircle** to be of the **Circle** type
– The variable **myCircle** can reference a **Circle** object.

A class is a ***reference type***

–Meaning: a variable of the class type can reference an instance（e.g. object） of the class.

Declaring/Creating Objects in a Single Step

`ClassName objectRefVar = new ClassName();`

Assign object reference

Create an object

eg.

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

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

Accessing Objects

dot operator (**.**) *object member access operator*:

Referencing the object's data:

```
objectRefVar.data
```

e.g., `myCircle.radius`

Invoking the object's method:

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

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

animation

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;
```

myCircle | no value

: Circle

radius: 5.0

Create a circle

animation

Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

Assign object reference to myCircle

myCircle | reference value

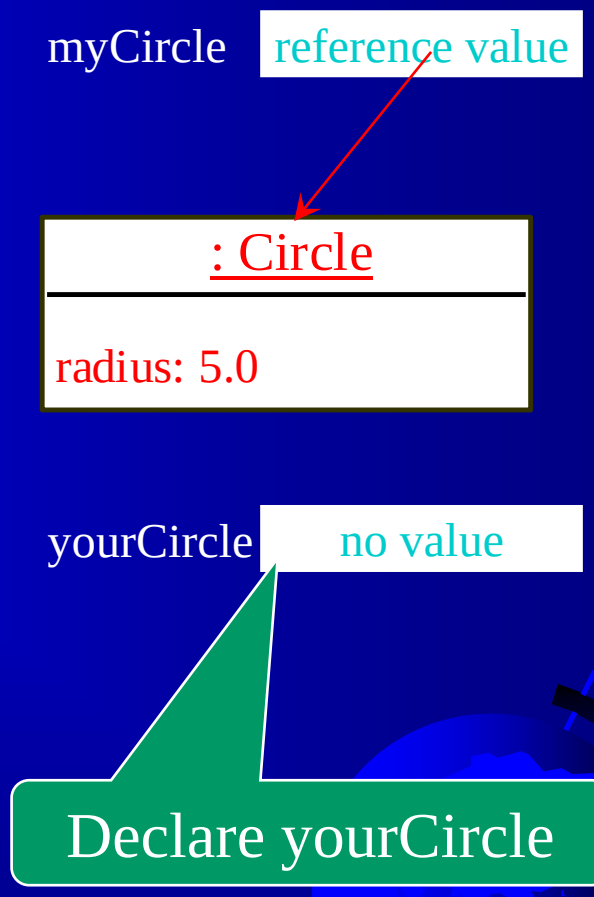
: Circle

radius: 5.0

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

Create a new Circle object

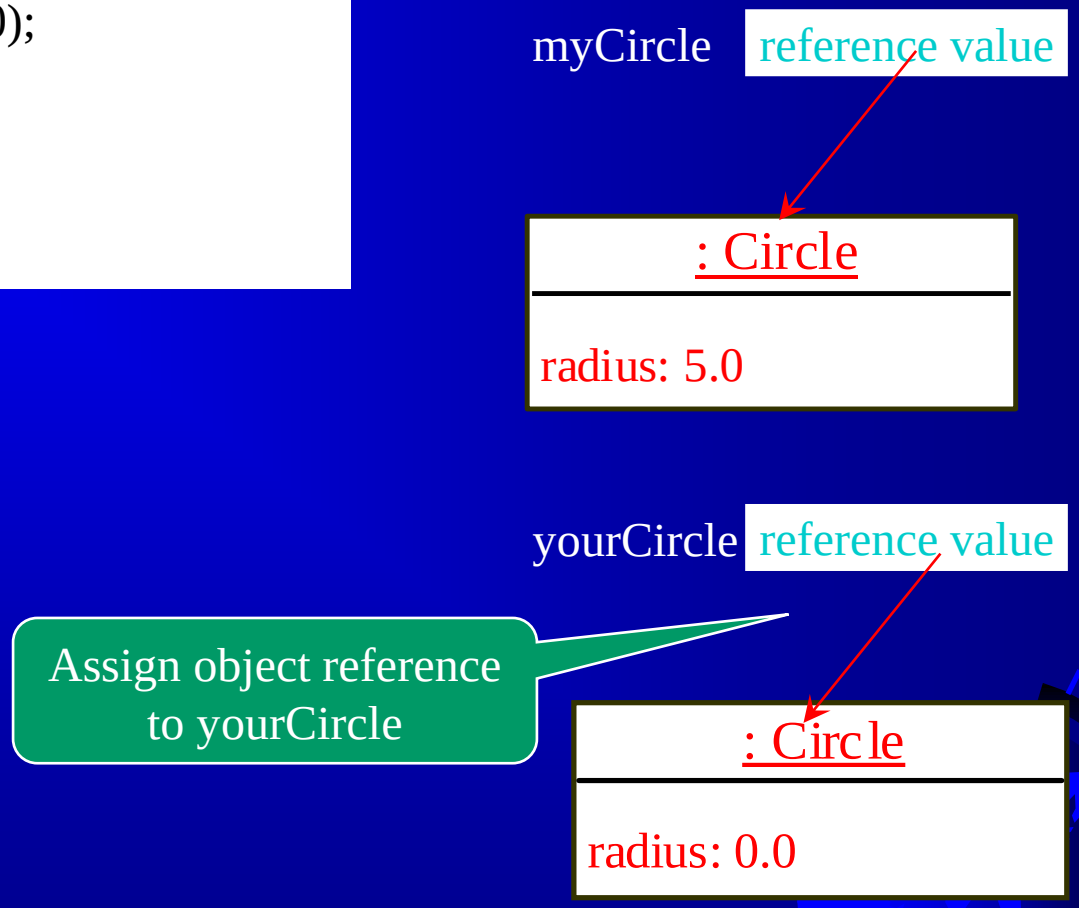
: Circle

radius: 0.0

animation

Trace Code, cont.

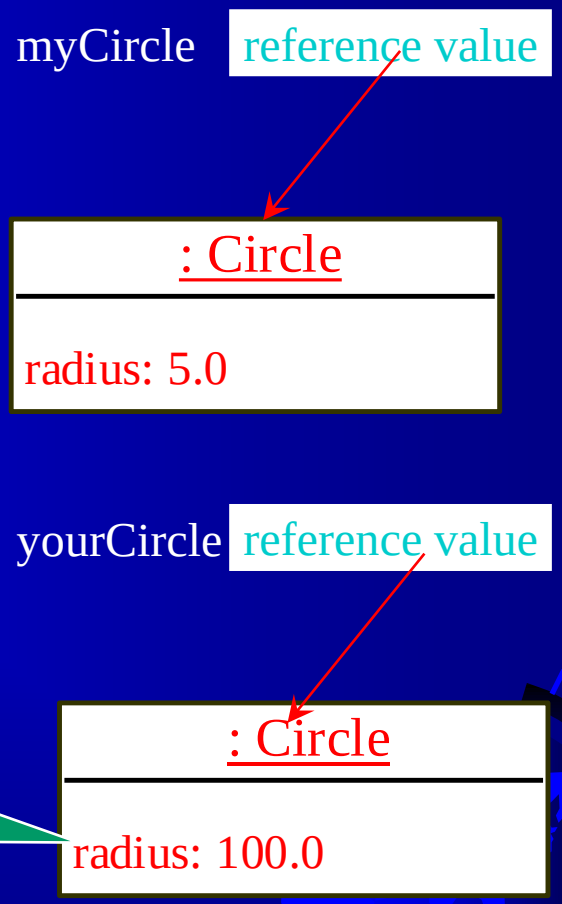
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



animation

Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Anonymous object

create an object <u>without assigning it to a variable</u>

Example：

- System.out.println("**Area is "** + **<u>new Circle(5).</u>**<u>getArea()</u>);

Examples

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

TestCircle1.java

main class

main method

create object

create object

create object

class **Circle1**

data field

no-arg constructor

second constructor

method

```
1  public class TestCircle1 {
2    /** Main method */
3    public static void main(String[] args) {
4      // Create a circle with radius 1.0
5      Circle1 circle1 = new Circle1();
6      System.out.println("The area of the circle of radius "
7        + circle1.radius + " is " + circle1.getArea() );
8
9      // Create a circle with radius 25
10     Circle1 circle2 = new Circle1(25);
11     System.out.println("The area of the circle of radius "
12       + circle2.radius + " is " + circle2.getArea());
13
14     // Create a circle with radius 125
15     Circle1 circle3 = new Circle1(125);
16     System.out.println("The area of the circle of radius "
17       + circle3.radius + " is " + circle3.getArea());
18
19     // Modify circle radius
20     circle2.radius = 100;
21     System.out.println("The area of the circle of radius "
22       + circle2.radius + " is " + circle2.getArea() );
23   }
24 }
25
26 // Define the circle class with two constructors
27 class Circle1 {
28   double radius ;
29
30   /** Construct a circle with radius 1 */
31   Circle1() {
32     radius = 1.0;
33   }
34
35   /** Construct a circle with a specified radius */
36   Circle1(double newRadius) {
37     radius = newRadius;
38   }
39
40   /** Return the area of this circle */
41   double getArea() {
42     return radius * radius * Math.PI;
43   }
44 }
```

NOTES

You can *put the two classes into one file*, but **only one class** in the file can be a **public** class.

- the public class name is the **file name**.
- eg., the file name is TestCircle1.java, since **TestCircle1** is public.

There are many ways to write Java programs.

- For instance, you can **combine the two classes into one class**

Circle1.java

```java
public class Circle1 {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 1.0
    Circle1 circle1 = new Circle1();
    System.out.println("The area of the circle of radius "
      + circle1.radius + " is " + circle1.getArea() );

    // Create a circle with radius 25
    Circle1 circle2 = new Circle1(25);
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());

    // Create a circle with radius 125
    Circle1 circle3 = new Circle1(125);
    System.out.println("The area of the circle of radius "
      + circle3.radius + " is " + circle3.getArea());

    // Modify circle radius
    circle2.radius = 100;
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());
  }

  double radius;

  /** Construct a circle with radius 1 */
  Circle1() {
    radius = 1.0;
  }

  /** Construct a circle with a specified radius */
  Circle1(double newRadius) {
    radius = newRadius;
  }

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

Example- TV class.

As another example, consider TV sets. Each TV is an object with states (current channel, current volume level, power on or off) and behaviors (change channels, adjust volume, turn on/off). You can use a class to model TV sets. The UML diagram for the class is shown in

gives a program that defines the TV class.

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 TV class models TV sets.

Define the **TV** class.

Uses the TV class to create objects

TestTV.java

```
1 public class TestTV {
2   public static void main(String[] args) {
3     TV tv1 = new TV();
4     tv1.turnOn();
5     tv1.setChannel(30);
6     tv1.setVolume(3);
7
8     TV tv2 = new TV();
9     tv2.turnOn();
10    tv2.channelUp();
11    tv2.channelUp();
12    tv2.volumeUp();
13
14    System.out.println("tv1's channel is " + tv1.channel
15      + " and volume level is " + tv1.volumeLevel );
```

Caution

ClassName.methodName(arguments)

- **static method**, which is defined using the static keyword.

e.g., **Math.pow** (3, 2.5) ;

objectName.methodName(arguments)

- **non-static method,** must be invoked from an object
- e.g., circle1.getArea();

TV.turnOn () ; ?

tv1.turnOn () ; ?

More explanations will be given in the section on "Static Variables, Constants, and Methods."

Reference Data Fields

Reference types of a variable/datafield

- can be any **Java class**
- eg. String str;
  Circle mycircle;

```
public class Student {
   String name;
   int age;
   boolean isScienceMajor;
   char gender;
}
```

Default Value for a Data Field

null for a reference type
- meaning: not reference any object

0 for a numeric type

false for a boolean type

'\u0000' for a char type

```
public class Student {
  String name;
  int age;
  boolean isScienceMajor;
  char gender;
}
```

```
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

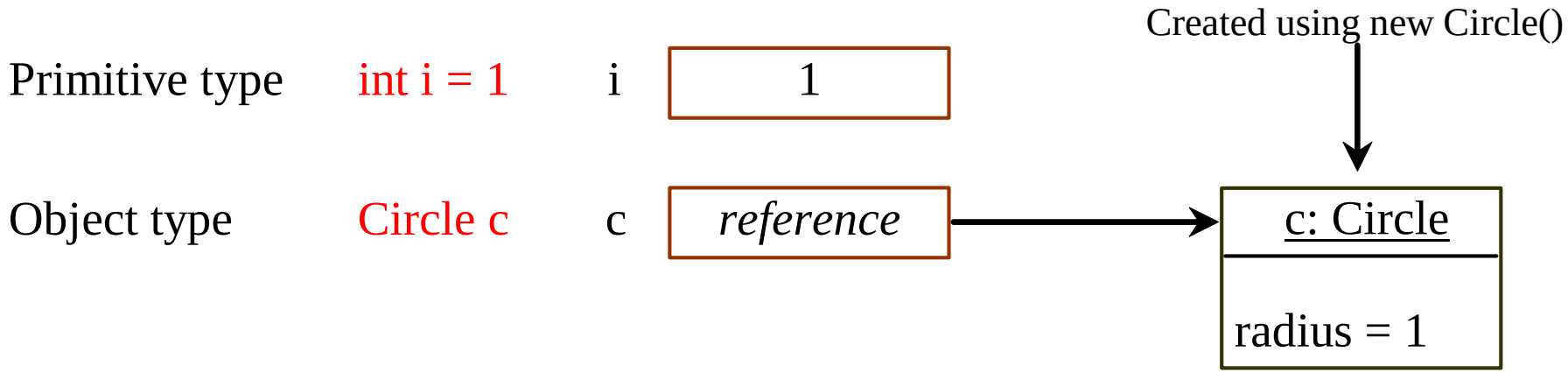
However, Java assigns ***no default value*** to a **local variable inside a method**.

=> local variables inside a method, **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

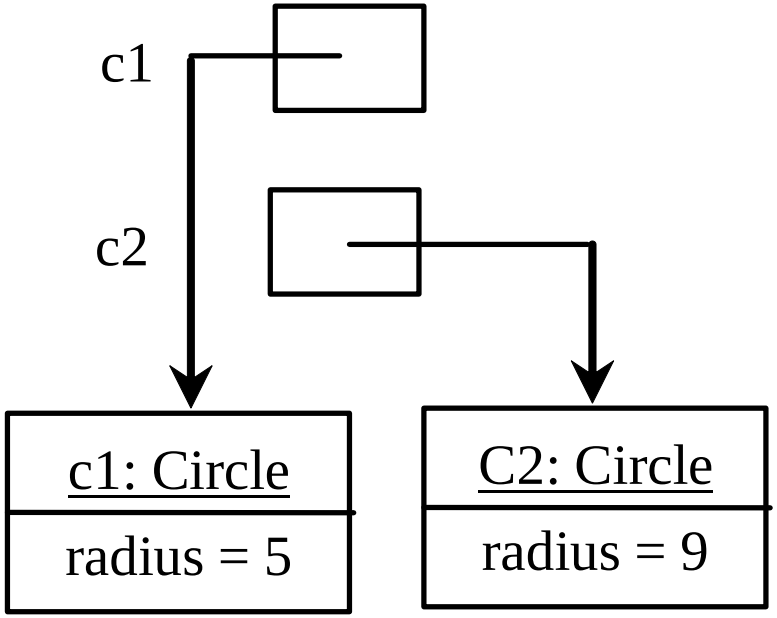
Differences between Variables of Primitive Data Types and Object Types



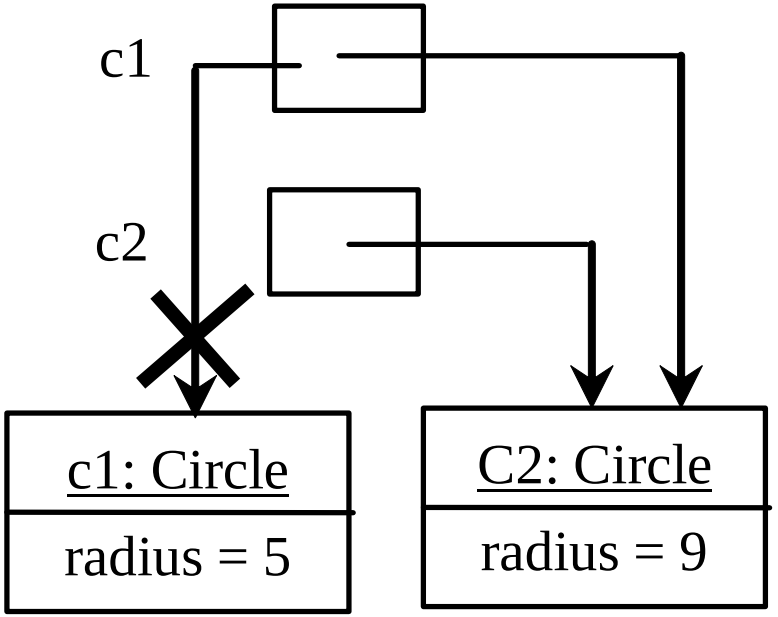
Copying Variables of Primitive Data Types and Object Types

Object type assignment **c1 = c2**

Before:



After:



Garbage Collection

Garbage is

- **Object** that is **not referenced by any variable**
- **The space is automatically collected** by JVM.

TIP:

- If you know that an object is **no longer needed**, you can also **explicitly assign null** to a reference variable for the object.

Using Classes from the **Java Library**

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/time; use its ***toString()*** method to return the date/time as a string.

The + sign indicates public modifer

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

The Date Class Example

```
java.util.Date date = new java.util.Date();
System.out.println("The elapsed time since Jan 1, 1970 is " +
  date.getTime() + " milliseconds");
System.out.println(date.toString());
```

create object

get elapsed time

invoke **toString**

displays a string like

```
The elapsed time since Jan 1, 1970 is 1100547210284 milliseconds
Mon Nov 15 14:33:30 EST 2004
```

The **Date** class has a <u>constructor</u>: **<u>Date (long elapseTime):</u>**

– create a **Date** object for a given time in <u>milliseconds</u>

The Random Class

Math.random() : random double value,
between 0.0 and 1.0 (excluding 1.0).

java.util.Random class: a more useful random number generator

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.

A Random object can be used to generate random values.

The Random Class Example

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

example, the following code creates two Random objects with the same seed, 3.

```
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) + " ");
```

The code generates the same sequence of random int values:

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

Instance Variables and Instance Methods

Instance variables belong to a specific instance (object)

Instance methods are invoked by an instance of the class.

--invocation: objectname.methodname();

eg. mycirle.getArea();

Static Variables, Constants, and Methods

Static variables: shared by all the instances of the ***class***.

Static methods: not tied to a specific object.

--invocation: Classname.methodname();

eg. Math.power(3, 4);

Static constants: final variables ,

shared by all the instances of the class.

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

declare static variable	`static int numberOfObjects;`
define static method	`static int getNumberObjects() { return numberOfObjects; }`

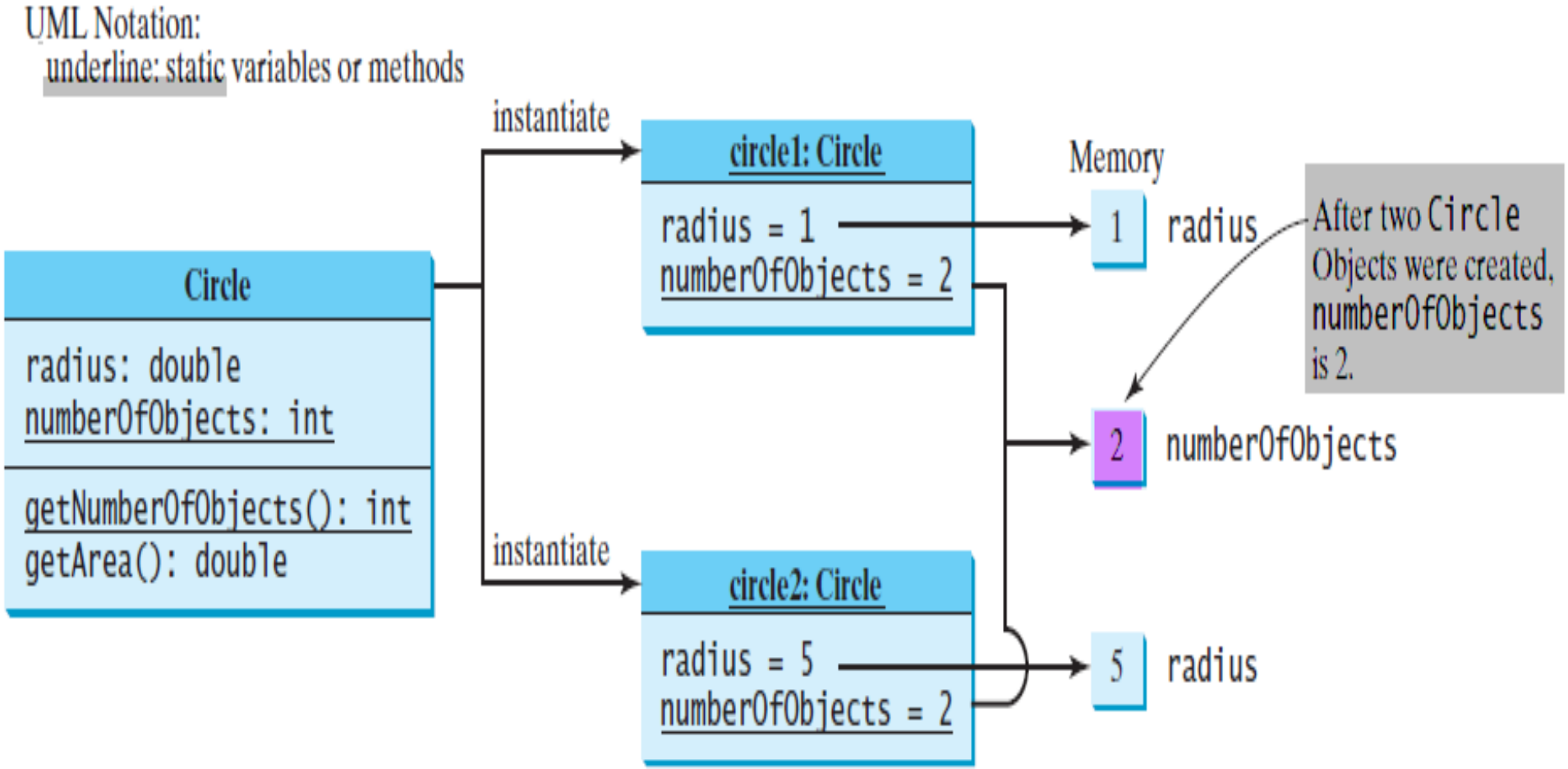
•in the Math class :

declare constant	`final static double PI = 3.14159265358979323846;`

Other Examples of **static methods**:

- the **main** method .
- all the methods in the **Math** class.
- **showMessageDialog** and **showInputDialog** in the **JOptionPane** class .

• Let us modify the **Circle** class by adding a static variable **numberOfObjects** to count the number of circle objects created



Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class.

Circle .java

```
1 public class Circle  {
2   /** The radius of the circle */
3   double radius;
4
5   /** The number of objects created */
6   static int numberOfObjects = 0;
7
8   /** Construct a circle with radius 1 */
9   Circle () {
10    radius = 1.0;
11    numberOfObjects++;
12  }
13
14  /** Construct a circle with a specified radius */
15  Circle (double newRadius) {
16    radius = newRadius;
17    numberOfObjects++;
18  }
19
20  /** Return numberOfObjects */
21  static int getNumberOfObjects() {
22    return numberOfObjects;
23  }
24
25  /** Return the area of this circle */
26  double getArea() {
27    return radius * radius * Math.PI;
28  }
29 }
```

static variable

increase by 1

increase by 1

static method

Circle .java

```
1 public class Circle2 {
2   /** The radius of the circle */
3   double radius;
4
5   /** The number of objects created */
6   static int numberOfObjects = 0;
7
8   /** Construct a circle with radius 1 */
9   Circle2() {
10    radius = 1.0;
11    numberOfObjects++;
12  }
13
14  /** Construct a circle with a specified radius */
15  Circle2(double newRadius) {
16    radius = newRadius;
17    numberOfObjects++;
18  }
19
20  /** Return numberOfObjects */
21  static int getNumberOfObjects() {
22    return numberOfObjects;
23  }
24
25  /** Return the area of this circle */
26  double getArea() {
27    return radius * radius * Math.PI;
28  }
29 }
```

static variable

increase by 1

increase by 1

static method

Instance methods (e.g., **getArea()**) and **instance data** (e.g., **radius**)

- belong to instances and can be used only after the instances are created.
- accessed via a reference variable/object name.

  i.e., objectname.methodname();

  eg. mycirle.getArea();

Static methods (e.g., **getNumberOfObjects()**) and **static data** (e.g., **numberOfObjects**)

- can be accessed from a reference variable or from their class name.

  i.e., Classname.methodname(); or objectname.methodname();

  eg. Cirle2.getNumberOfObjects();
  mycirle.getNumberOfObjects();

```java
public class TestCircle2 {
  /** Main method */
  public static void main(String[] args) {
    System.out.println("Before creating objects");
    System.out.println("The number of Circle objects is " +
      Circle2.numberOfObjects);

     // Create c1
     Circle2 c1 = new Circle2();

    // Display c1 BEFORE c2 is created
    System.out.println("\nAfter creating c1");
    System.out.println("c1: radius (" + c1.radius +
      ") and number of Circle objects (" +
      c1.numberOfObjects + ")");

    // Create c2
    Circle2 c2 = new Circle2(5);

    // Modify c1
    c1.radius = 9;

    // Display c1 and c2 AFTER c2 was created
    System.out.println("\nAfter creating c2 and modifying c1");
    System.out.println("c1: radius (" + c1.radius +
      ") and number of Circle objects (" +
      c1.numberOfObjects + ")");
    System.out.println("c2: radius (" + c2.radius +
      ") and number of Circle objects (" +
      c2.numberOfObjects + ")");
  }
}
```

static variable

```
Before creating objects
The number of Circle objects is 0

After creating c1
c1: radius (1.0) and number of Circle objects (1)

After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

ClassName.staticMethodName(…)
ClassName.staticVariable

- This improves readability, because the user can easily recognize the static method and data in the class.

In static method :

- Instance variables and methods **can not be used**

In instance method：

- Static variables and methods can be used

```java
public class Foo {
  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));
  }
}
```

Instance or Static?

How to decide whether a variable/method should be an instance one or a static one?

Instance one: belongs to a specific instance

- **E.g. radius** is an instance variable of the **Circle** class. **getArea** method is an instance method.

Static one: shared by all instances of the class

- E.g. methods in the **Math** class, such as **random**, **pow**, **sin**, and **cos**
- The **main** method is static and can be invoked directly from a class.

It is a common **design error** to define an instance method that should have been defined static.

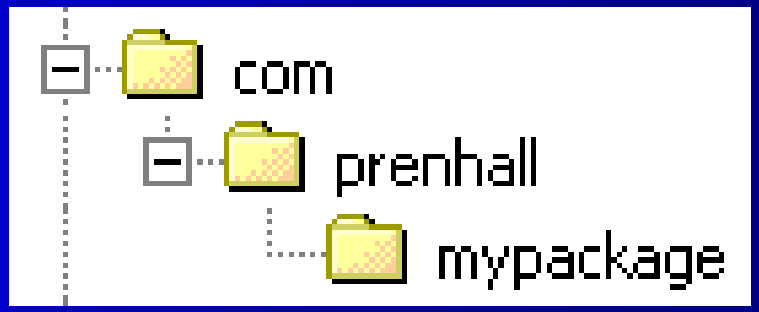
```
public class Test {
  public static int factorial(int n)
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

For example, the method should be defined static, because it is independent of any specific instance.

Package

A package corresponds to a **directory** in the *file system.* used to organize classes.

e.g. package com.prehall.mypackage;



package packageName;

– as the first statement in the program:

If a class is defined without the package statement, it is said to be placed in the ***default package*** (i.e., ***current directory***).

- **`package-private / package-access:`**

By default, class/data/method

can be accessed by any class in the **same package**

`public`

class/data/method : visible to any class in **any package**.

`private`

data/methods : accessed only by the **declaring class.**

get() , set() : read and modify private properties.

```java
package p1;

class C1 {
  ...
}
```

```java
package p1;

public class C2 {
  can access C1
}
```

```java
package p2;

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

A nonpublic class has package-access.

Illustrate how a data/method in class **C1,** can be accessed from a class **C2** in the same package , and from a class **C3** in a different package.

```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();
  }
}
```

public modifier enables <u>unrestricted</u> access.

default modifier restricts access to <u>within a package</u>,

private modifier: restricts access to <u>within a class</u>,

- **visibility modifier specifies how** class members (data/methods) can be accessed **from outside** the class.
- **no restriction** on accessing them **from inside** the class.

```java
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() {
    return x ? 1 :  1;
  }
}
```

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

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

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

An object can access its private members if it is defined in its own class.

Caution:

The **private** modifier applies only to the **class members**.

The **public** modifier can apply to **class members / class**.

Note:

In most cases, the **constructor** should be **public**.

private constructor prohibits the user from creating an object of the class

– in ***java.lang.Math*** **:**

```
private Math() {
}
```

There is **no** reason to create an **object** from the **Math** class, because **all of its data fields and methods are static**.

Why Data Fields Should Be private?

- To protect data.
- To make class easy to maintain.

- To prevent direct modifications of data fields, you should declare the data fields private. This is known as *data field encapsulation*.

- To access data field, use a *get* method (getter, accessor) to return its value.

```
public returnType getPropertyName()
```

- To update data field, use a *set* method (setter, mutator) to set a new value.

```
public void setPropertyName(dataType propertyValue)
```

Example of Data Field Encapsulation

The - sign indicates private modifier

Circle	
-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.

The `Circle` class encapsulates circle properties and provides get/set and other methods.

Circle3.java

```java
public class Circle3 {
  /** The radius of the circle */
  private double radius = 1;

  /** The number of the objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public Circle3() {
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  public Circle3(double newRadius) {
    radius = newRadius;
    numberOfObjects++;
  }

  /** Return radius */
  public double getRadius() {
    return radius;
  }

  /** Set a new radius */
  public void setRadius(double newRadius) {
    radius = (newRadius >= 0) ? newRadius : 0;
  }

  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  public double getArea() {
    return radius * radius * Math.PI;
  }
}
```

- Since these methods are the only ways to read and modify radius, you have total control over how the **radius** property is accessed.

You cannot use **myCircle.radius** in the client program.

TestCircle3.java

```
public class TestCircle3 {
  /** Main method */
  public static void main(String[] args) {
    // Create a Circle with radius 5.0
    Circle3 myCircle = new Circle3(5.0);
    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());

    System.out.println("The number of objects created is "
      + Circle3.getNumberOfObjects());
  }
}
```

invoke public method

invoke public method

invoke public method

Passing Objects to Methods

Like passing an array, passing an object is actually passing the **reference** of the object.

Passing by value

- primitive type value
- reference type value : the reference to object/array

TestPassObject.java

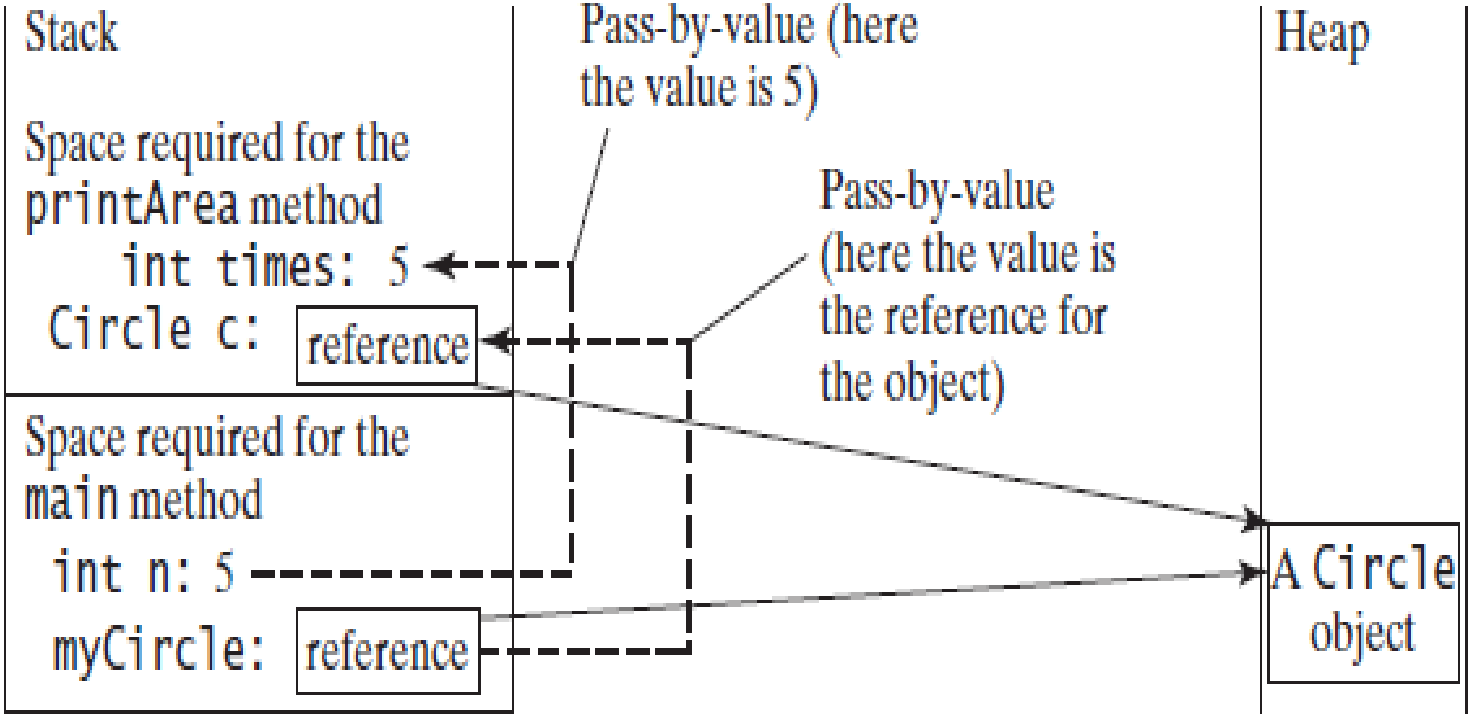
```java
public class TestPassObject {
  /** Main method */
  public static void main(String[] args) {
    // Create a Circle object with radius 1
    Circle3 myCircle = new Circle3(1);

    // 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(Circle3 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--;
    }
  }
}
```

TestPassObject.java

```java
public class TestPassObject {
  /** Main method */
  public static void main(String[] args) {
    // Create a Circle object with radius 1
    Circle3 myCircle = new Circle3(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(Circle3 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--;
    }
  }
}
```

```
Radius                                  Area
 1.0                                     3.141592653589793
 2.0                                    12.566370614359172
 3.0                                    29.274333882308138
 4.0                                    50.2654824574366 9
 5.0                                    79.53981633974483
Radius is 6.0
n is 5
```

Passing Objects to Methods, cont.



The value of `n` is passed to `times`, and the reference of `myCircle` is passed to `c` in the `printAreas` method.

Array of Objects

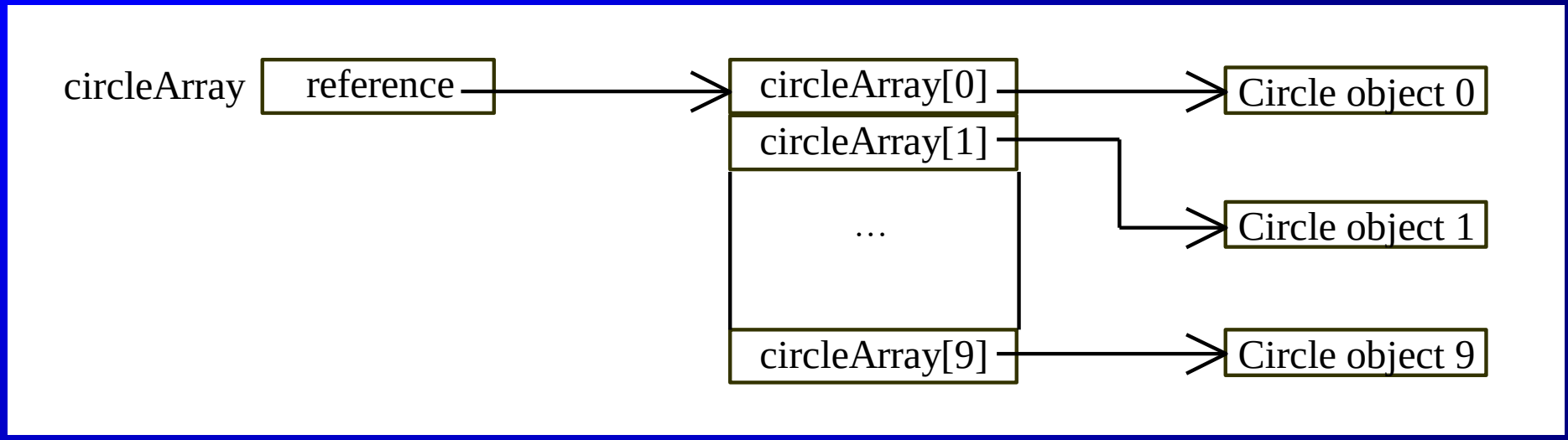
Declares and creates an array of ten **Circle** objects:

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

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

- as shown in the next figure.

Array of Objects

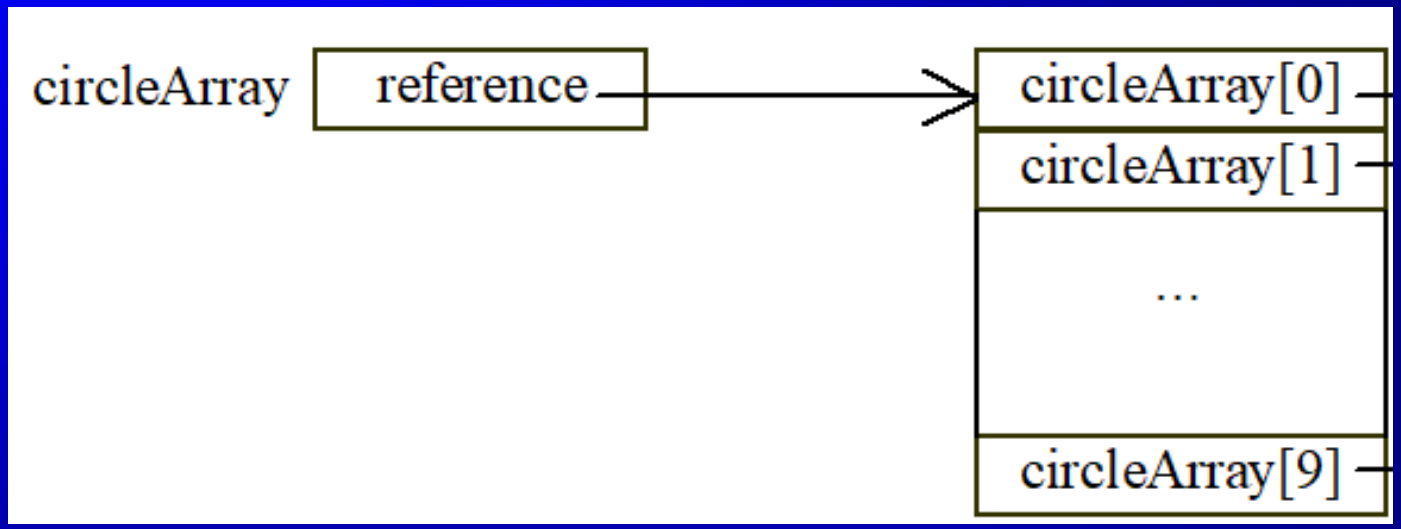


invoking circleArray[1].getArea() involves two levels of referencing

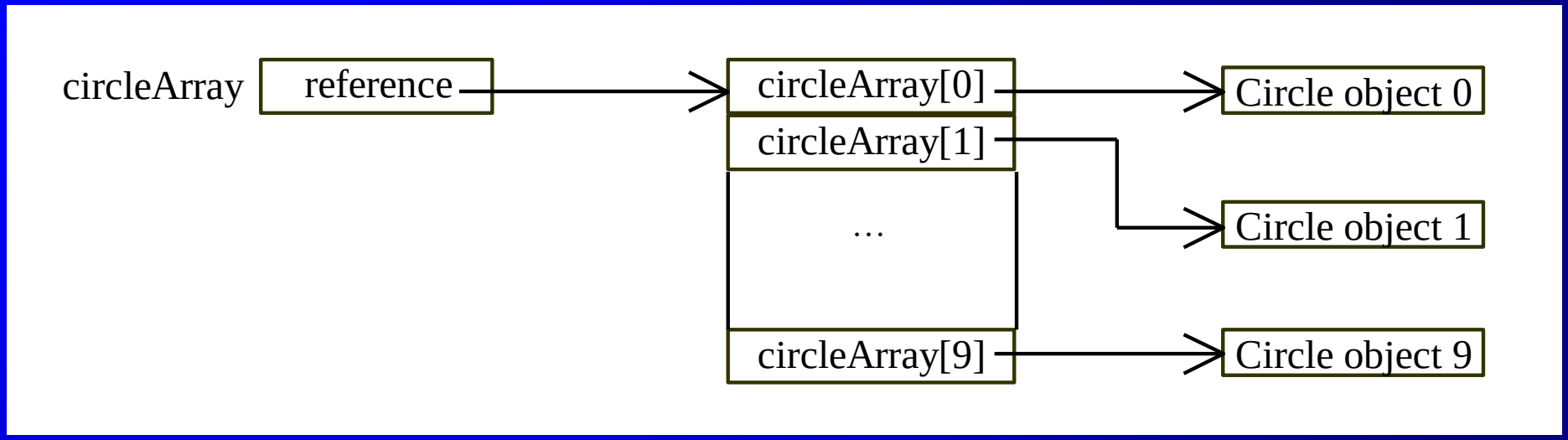
- circleArray references to the entire array.
- circleArray[1] references to a Circle object 1.

Array of Objects

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



Each element in the array is a reference variable with a default value of **null**.



To initialize the **circleArray**, you can use a **for** loop:

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

Example

Summarizing the areas of an array of circles.

- The program creates **circleArray**, an array composed of five **Circle** objects;
- it then initializes circle radii with random values
- and displays the total area of the circles in the array.

TotalArea.java

Circle class

array of objects

return array of objects

pass array of objects

```java
public class TotalArea {
  /** Main method */
  public static void main(String[] args) {
    // Declare circleArray
    Circle3[] circleArray;

    // Create circleArray
    circleArray = createCircleArray();

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

  /** Create an array of Circle objects */
  public static Circle3[] createCircleArray() {
    Circle3[] circleArray = new Circle3[5];

    for (int i = 0; i < circleArray.length; i++) {
      circleArray[i] = new Circle3(Math.random() * 100);
    }

    // Return Circle array
    return circleArray;
  }

  /** Print an array of circles and their total area */
  public static void printCircleArray(Circle3[] circleArray) {
    System.out.printf("%-30s%-15s\n", "Radius", "Area");
    for (int i = 0; i < circleArray.length; i++) {
      System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
        circleArray[i].getArea());
    }

    System.out.println("----------------------------------------");

    // Compute and display the result
    System.out.printf("%-30s%-15f\n", "The total area of circles is",
      sum(circleArray));
  }
```

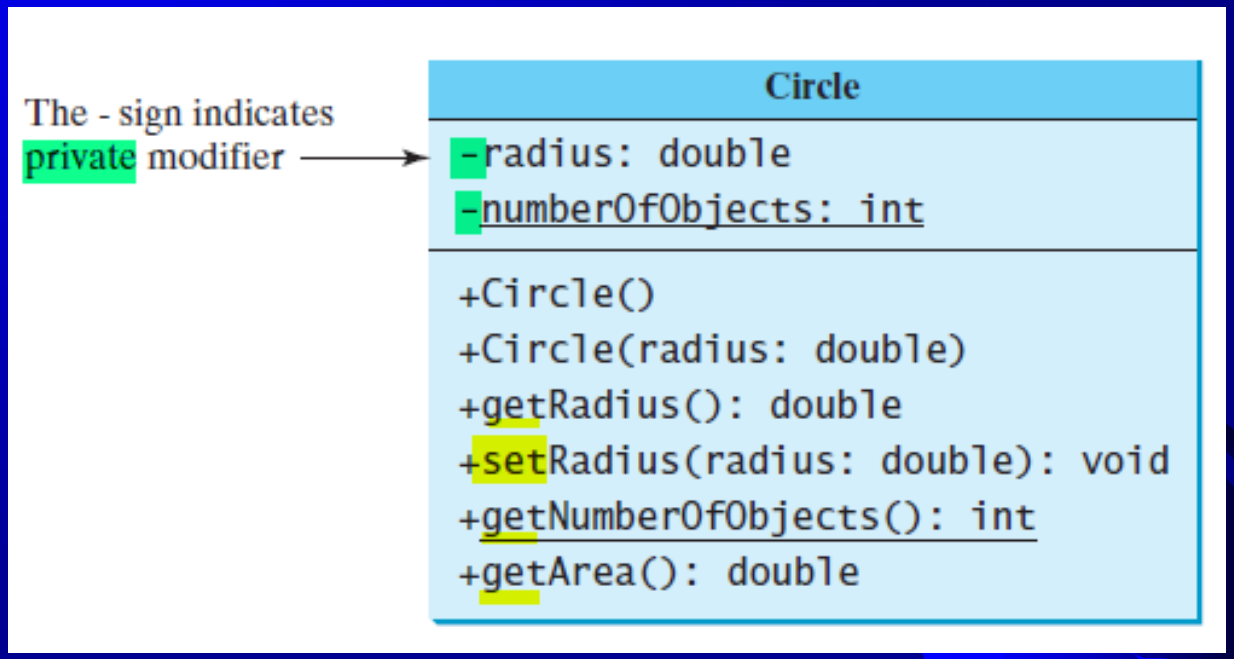
```
 Radius                          Area
70.577708                      15648.941866
44.152266                       6124.291736
24.867853                       1942.792644
 5.680718                        101.380949
36.734246                       4239.280350
----------------------------------------
The total area of circles is   28056.687544
```

pass array of objects

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*.**

e.g. If you delete the set method in the Circle class in the preceding example, the class would be immutable because radius is private and cannot be changed without a set method.



- But, a class with all private data fields and without setters is not necessarily immutable.

```java
public class Stdudent {
  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;
  }
}
```

```java
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;
  }
}
```

```java
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!
  }
}
```

Data field **birthDate** is returned using the **getBirthDate()** method. This is **a reference to** a **BirthDate** object. Through this reference, the content for **birthDate** can be changed.

What Class is Immutable?

Immutable class:

1. all data fields are private

2. has no setter methods

3. has no getter methods that would return a reference to a mutable object.

Scope of Variables

Instance variables : *data fields*

- Scope : the entire class body.
- Declared: anywhere inside a class.

local variable: *declared inside a block* (e.g.method)

- Scope: from its declaration and continues to the end of the block.
- must be initialized explicitly before it can be used.

```
public class Circle {
  public double findArea() {
    return radius * radius * Math.PI;
  }

  private double radius = 1;
}
```

(a) variable radius and method findArea() can be declared in any order

```
public class Foo {
  private int i;
  private int j = i + 1;
}
```

(b) i has to be declared before j because j's initial value is dependent on i.

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*. For example, in the following program, x is defined as an instance variable and as a local variable in the method.

```
public class Foo {
  private int x = 0; // Instance variable
  private int y = 0;

  public Foo() {
  }

  public void p() {
    int x = 1; // Local variable
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

```
x = 1
y = 0
```

What is the printout for f.p(), where f is an instance of Foo?

The this Keyword

<u>this</u> : a reference that refers to an object itself.

– Two common usage: (next slides)

Reference Data Fields

- **this.instanceVariable**
- **ClassName.staticVariable.**

```java
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
```

Calling Overloaded Constructor

```
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 <u>invoke another constructor</u>

Every instance variable belongs to an instance represented by this, which is <u>normally omitted</u>

Supplements:
Two classes in one file

```
1
2  public class T {
3      public static void main(String[] args) {
4          C c = new C ();
5          c = new C (1);
6      }
7  }
8
9  class C {
10     C () {}
11     C (int a) {}
12 }
```

Supplements:
Rarely used

```
public class T {
    public static void main(String[] args) {
        T t = new T ();
        C c = t.new C ();
        c = t.new C (1);
    }

    class C {
        C () {}
        C (int a) {}
    }
}
```