Objects and Classes

Recap: Object and Class

- *Classes* are constructs that define objects of the same type.
- a class provides a special type of methods, known as constructors, which are invoked to construct objects from the class.
- An *object*: represents an entity in the real world that can be distinctly identified
- An object has a unique identity, state, and behaviors.

Defining Classes

<modifier> class <className>

{

}

```
public class Rectangle
{
 int length ;       //Adding variables
 int width;

void getdata(int x, int y)      //Adding Method
 {
   length =x;
   width = y;
 }
}
```

```
class Rectangle
{
    int length ;
    int width ;

    void getdata(int x,int y)
    {
        length=x;
        width = y;
    }

    int rectArea()
    {
            int area   = length*width;
            return(area);
     }
}
```

Creating objects

- Objects in java are created using the **new** operator .
- The new operator creates an objects of the specified class and returns a reference to that objects.
- Ex:
- Rectangle rect1; //declare
- rect1 = new rectangle(); //instantiate

```
class Rectangle
{
   int length ;
   int width ;
   void getdata(int x,int y)
   {
      length=x;
      width = y;
   }

   int rectArea()
   {
         int area = length*width;
         return(area);
    }
}
```

The method rectangle is the default constructor of the class .

we can create any no. of objects of rectangle

Creating objects - Syntax

Syntax:

<**className**> <**variableName**> = **new** <**classConstructor**>

Eg. Rectangle rect1 = new rectangle();

Eg. Room1 obj = new Room1();

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

Accessing class member

- *Rectangle rect1 = new rectangle();*

  Rectangle rect2 = new rectangle();

  Rectangle rect8 = new rectangle();
- *Objectname.variable name ;*

```
class Rectangle
{
   int length  ;
   int width ;
   void getdata(int x,int y)
   {
      length=x;
      width = y;
   }

   int rectArea()
   {
         int area= length*width;
         return(area);
    }
}
```

- Objectname.methodname(parameter-list) ;

  rect8.getdata(20,30);

rect1.length =15;

rect3.length =23;

illustration of class and object

```
class Rect
{
 int length;
 int width;

void getdata(int x,int y)
{
   length=x;
   width=y;
}
int rectArea()
{
int area=length*width;
return(area);
}
}
```

```
class RectangleArea
{
public static void main(String[]args)
{
int area1,area2;
  Rect rect1=new Rect();
  Rect rect2=new Rect();
  rect1.length = 10;
  rect1.width = 2;
  area1=rect1.length*rect1.width;
 rect2.getdata(5,10);
 area2=rect2.rectArea();
System.out.println("area1="+area1);
System.out.println("area2="+area2);
}
}
```

Constructors

- All object that are created need initial values.
- Constructors play the role **of initializing objects**.
- Constructor initializes an object immediately on creation.
- Constructor is automatically called (invoked) immediately after the object is created.
- Constructor name is same as class name and have no return type not even void.

A constructor with **no parameters** is referred to as a ***no-arg constructor*.**

- If you don't include a constructor method in your class, the Java compiler will automatically include the **default constructor** in the compiled bytecode.

- A constructor method, like other methods, can take arguments.

- Usually a programmer would write a constructor with parameter variables so that the class can be instantiated with default values for the instance variables.

There are some rules that constructor methods must follow:

- The constructor method must always be public.
- The constructor method must *never* have a return type, not even void!
- The constructor method name must always match the class name **exactly**, including capitalization.

EX:

```
Public class Book{
                String title;
                double price;

Book(String bookTitle, double bookPrice) {
    title = bookTitle;
    price = bookPrice;
}
```

Example

```
Class Box
{
        int w,h,d;
        Box()    //contructor
        {
            w=10;
            h=10;
            d=10;
        }

public static void main(String str[])
{
Box b= new Box();
}
```

Replacement of getdata by a constructor method

```
class Rectangle
{
   int length ;
   int width  ;
void getdata(int x,int y)
{
    length=x;
    width = y;
}
int rectArea()
{
    int area = length*width;
    return(area);
  }
}
```

```
class Rectangle
{
   int length ;
   int width  ;
Rectangle (int x , int y)     // constructor  method
{
     length=x;
     width = y;
 }
int rectArea()
{
     int area = length*width;
  }
}
```

illustration of constructors

```
class Recta
  {
       static int length;
       static int width;

      void getdata(int x,int y)
        {
         length=x;
          width=y;
        }
         Recta(int x,int y)

  {

    length=x;

     width=y;

  }

Recta ()

  {

              }
        int rectArea()
        {
         int area=length*width;
          return(area);
        }
  }
```

```
class RectArea
{
public static void main(String[]args)
{
   int area1,area2;
   Recta rect1=new Recta();
   Recta rect2=new Recta(10,20);
   rect1.length = 1;
   rect1.width = 2;
   area1=rect1.length*rect1.width;
   rect2.getdata(5,10);
   area2=rect2.rectArea();
 System.out.println("area1="+area1);
System.out.println("area2="+area2);
}
}
```

Using the "this" Keyword

What if my constructor method's parameter variables were defined as:

```
public Book(String title, double price) { ... }
```

Would this compile? What about the statements inside the constructor? If we changed them, our constructor would be:

```
Public class Book{
        String title;
        double price;

public Book(String title, double price) {
   title = title;
   price = price;
}
```

How does Java know when we're referring to the instance variable title and when we're referring to the parameter variable title?? In fact, the program will compile with this constructor, but it will no longer work properly.

In the statement **`title = title;`**, the variable "title" will be referring to the parameter on both sides of the assignment operator. This means the title (and also price) instance variables won't get their proper values.

Using 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**.

In other words, "this object's title attribute gets the value in the title parameter." You can update your constructor method so that it now becomes:

```
1public Book(String title, double price) {
2    this.title = title;
3    this.price = price;
4}
```

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;
  }

  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 instance represented by this, which is normally omitted

The 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*.

The Default Constructor

Create a second room in your test class by using the default constructor:

```
Room newRoom = new Room();
```

What happens when you compile this program? You should get the following error:

```
TestRoom.java:13: cannot find symbol
symbol : constructor Room()
location: class Room
                Room r2 = new Room();
                          ^
1 error
```

Even though the arrow is pointing at "new", the problem is that Java doesn't recognize the constructor "Room()".

It's telling you that it can't find a constructor in your class that takes no arguments. But didn't we say earlier that Java creates a default constructor for you? Yes, but this is only true when you compile a class with *no* constructors.

Once you define a constructor in your class, Java will no longer create the default constructor for you automatically.

This means that if your class needs a default constructor, you will need to add one.

Most of your classes will need a default constructor.

The default constructor can be used to initialize your instance variables to default values.

For this reason, you should always make sure you include a default constructor in your class.

For the Room class, let's say that if the programmer wants to create a room object without an initial length and width, we'll initialize the length and width to 0:

```
public Room() {
  length = 0;
  width = 0;
}
```

If you don't have any code you want to put into a default constructor, but you need one anyway, you can make a blank one:

```
public Book() { }
```

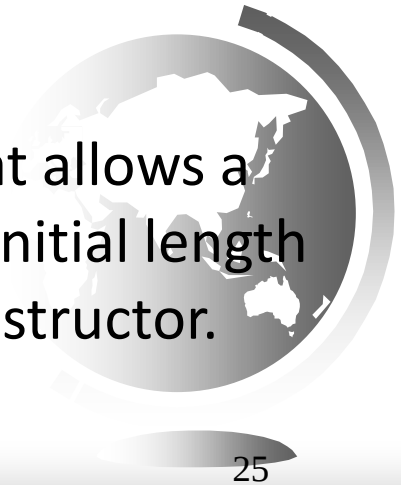
Exercise: Design a Room class

```
public class Room {
  public double length;
  public double width;

  public String getDimensions() {
    return length + " x " + width;
  }

  public double getArea() {
    double area = length * width;
    return area;
  }
}
```

Add a two-parameter constructor to the Room class that allows a programmer to instantiate a new Room object with an initial length and width. Modify your program to try out the new constructor.



Exercise: Design a Room class

```
public class Room {
  public double length;
  public double width;

  Room(double length, double width)
   {
     this.length=length;
     this.width=width;
   }

  public String getDimensions() {
    return length + " x " + width;
  }

  public double getArea() {
    double area = length * width;
    return area;
  }
}
```

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;
  }
}
```

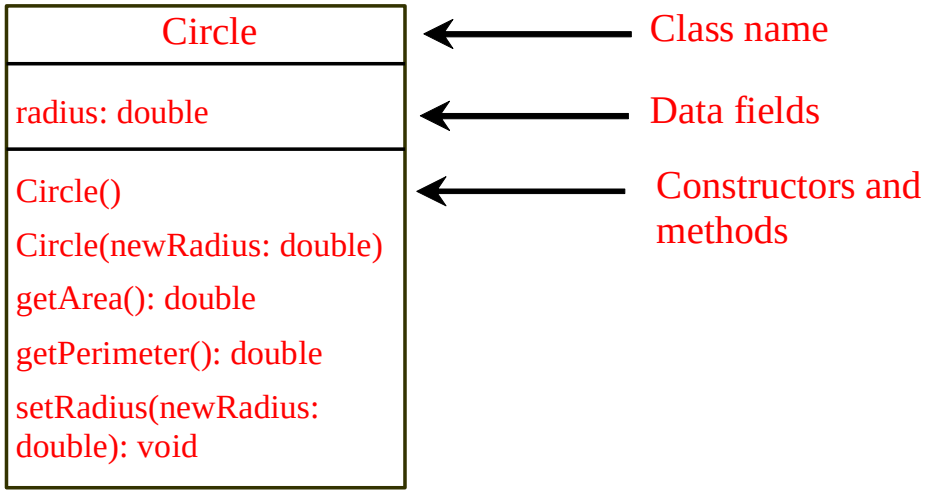
Data field

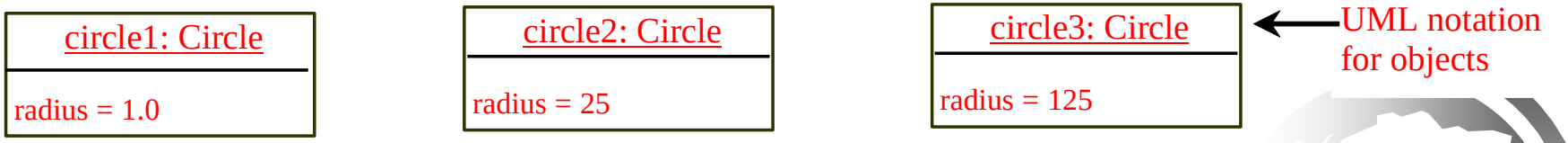
Constructors

Method

UML Class Diagram

UML Class Diagram





Example: Defining Classes and Creating Objects

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

TestSimpleCircle Run

www.cs.armstrong.edu/liang/intro11e/html/TestSimpleCircle.html

Difference between constructors and methods

- The important difference between methods is that

  constructors create and initialize objects that don't exist yet, while methods perform operations on objects that already exist.

- Constructors can't be called directly; they are called implicitly when the new keyword creates an object. Methods can be called directly on an object that has already been created with new.

- The definitions of constructors and methods look similar in code. They can take parameters, they can have modifiers (e.g. public), and they have method bodies in braces.
- Constructors must be named with the same name as the class name. They can't return anything, even void (the object itself is the implicit return).
- Methods must be declared to return something, although it can be void.

Modifiers: public and private

In Java, there are a set of **access modifiers**.

Types of access modifiers:

- Default – No keyword required
- Private
- Protected
- Public

- An access modifier defines the accessibility of a class, method, or data member.
- In other words, the access modifier will indicate where a programmer can reference or refer to a particular class, method, or data member.
- For example, the main() method in a Java program is defined as **public**. This means that the main method may be accessed (called, invoked) anywhere in the program's code, and also anywhere in another program's code.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Note: Read InstanceVariables.doc on SLATE

Visibility Modifiers and Accessor/Mutator Methods

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

- `public`

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

- `private`

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

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

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

- In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
private A(){}//private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
   A obj=new A();//Compile Time Error
 }
}
```

```
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.

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

In this example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output: Hello

```java
package p1;

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

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

```java
package p1;

public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```java
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

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

4) Protected

- The **protected access modifier** is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
- It provides more accessibility than the default modifer.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Output: Hello

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

In **Java**, **static** is a keyword used to describe how objects are managed in memory.
It **means** that the **static** object belongs specifically to the class, instead of instances of that class.

Variables, methods, and nested classes can be **static**. ... Instead, we can make the variable **static** and make it part of the class itself

Static Variables, Constants, and Methods

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

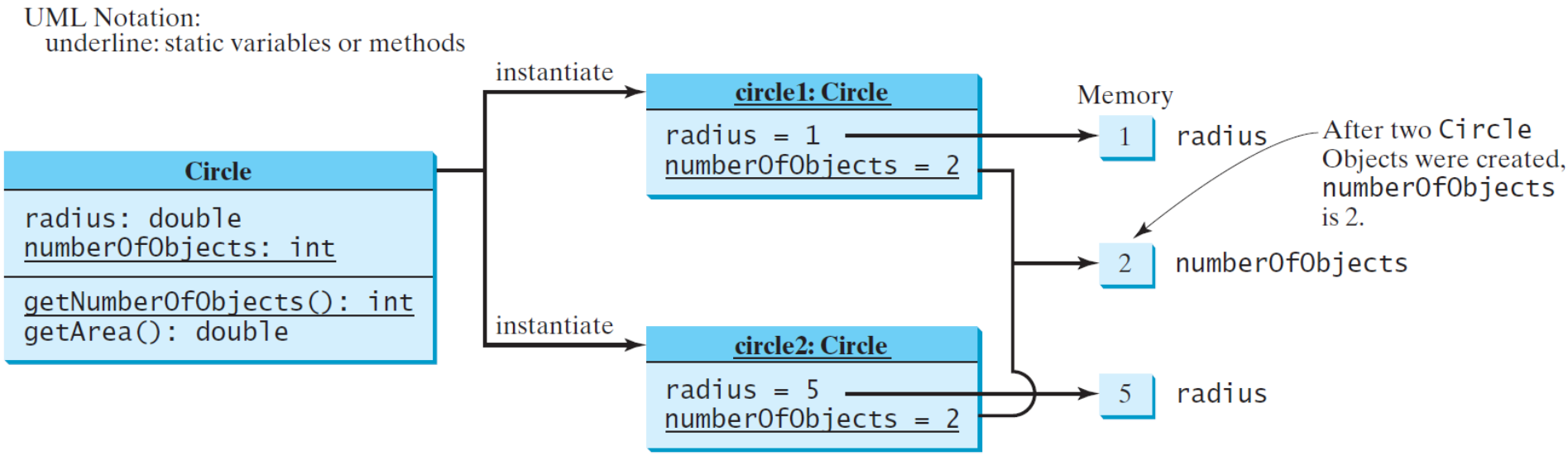
Static methods are not tied to a specific object.

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

www.cs.armstrong.edu/liang/intro11e/html/CircleWithStaticMembers.html

TestCircleWithStaticMembers

Run

www.cs.armstrong.edu/liang/intro11e/html/TestCircleWithStaticMembers.html

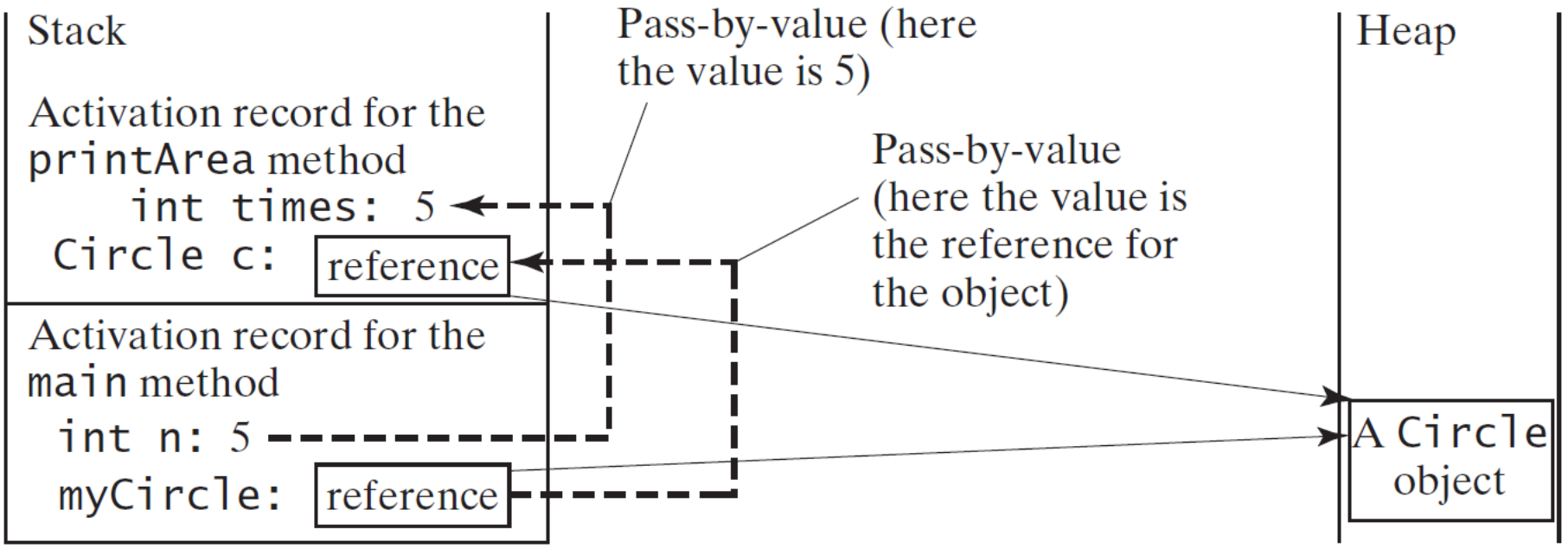
Passing Objects to Methods

- 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)

TestPassObject Run

www.cs.armstrong.edu/liang/intro11e/html/TestPassObject.html

Passing Objects to Methods, cont.



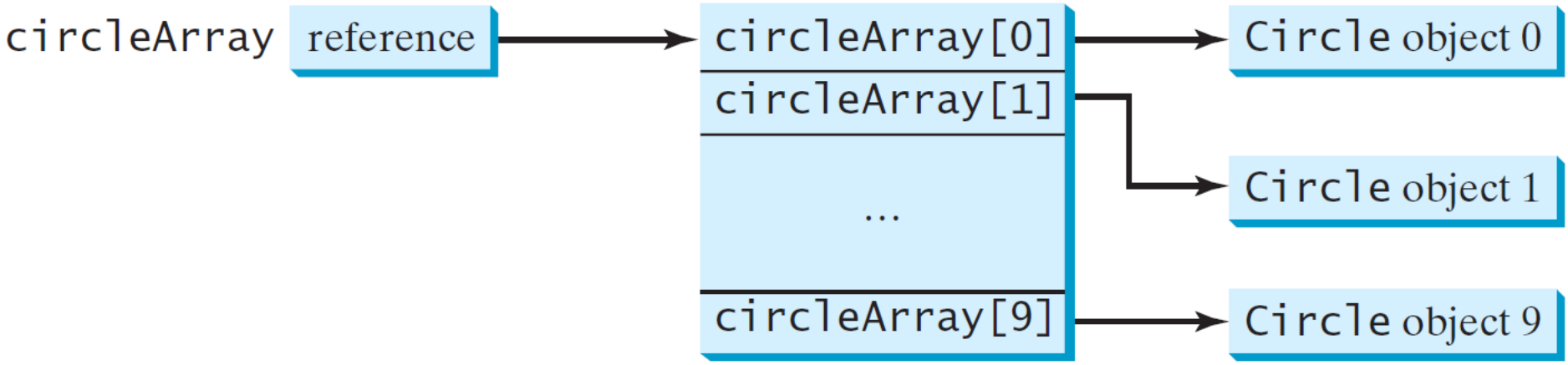
Array of Objects

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

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

Array of Objects, cont.

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



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!
  }
}
```

Array of Objects, cont.

Summarizing the areas of the circles

www.cs.armstrong.edu/liang/intro11e/html/TotalArea.html