



CS 213 – Programming Language 2

Dr. Ahmed Hesham Mostafa

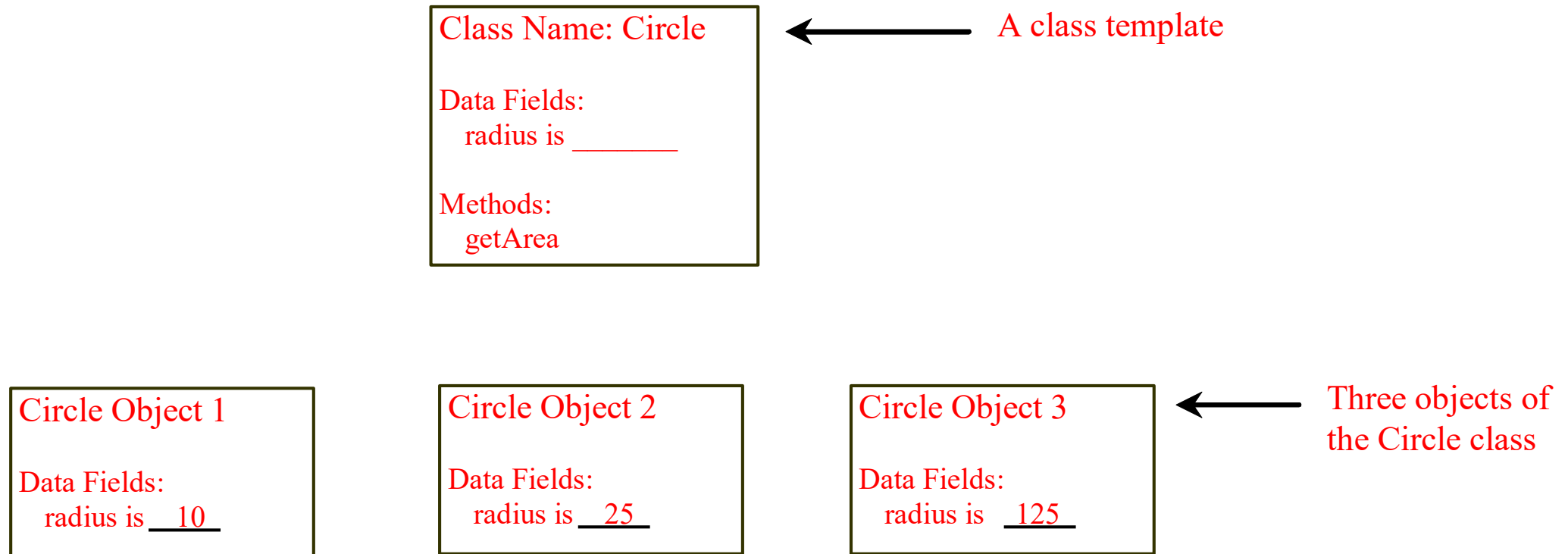
Lecture 3 – OOP Part 1

OO Programming Concepts

- Object-oriented programming (OOP) involves programming using objects.
- An *object* represents an entity in the real world that can be uniquely identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.
- An object has a unique identity, state, and behavior.
- The *state* of an object consists of a set of *data fields* (also known as *properties*) with their current values.
- The *behavior* of an object is defined by a set of methods.

Objects

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



Classes

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

Why classes not struct

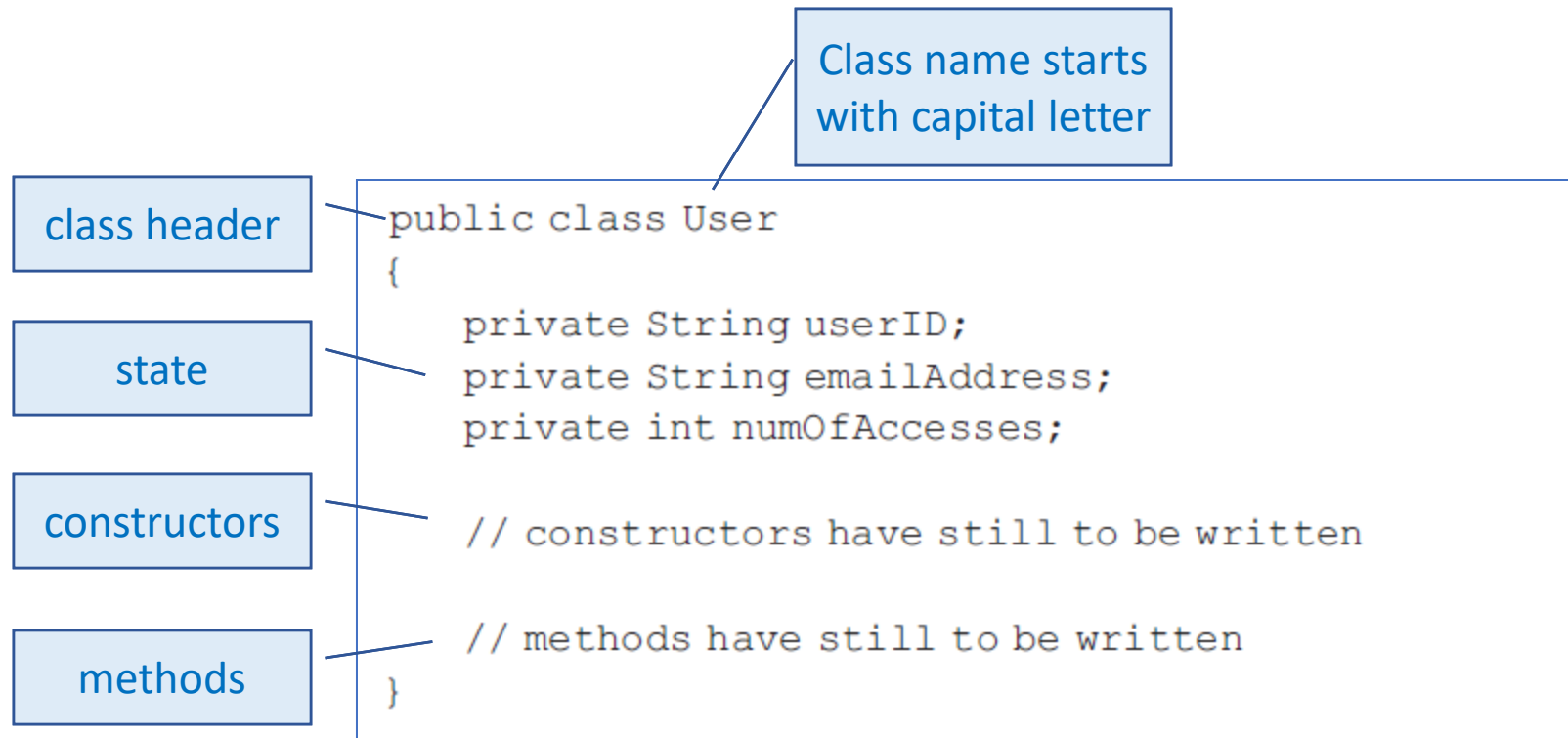
- Struct data variables and methods are defined as public by default.
- Class data variables and method are defined as private by default (in C++).
- Classes can have relationship between each other.
- Classes can have concepts like overloading, inheritance, and polymorphism.
- Classes have a new access levels for its' members called protected, default.
- Classes Provide public interface, while hiding the implementation details (**Encapsulation**)
- Class: when implementation is hidden, any change do not affect programmers that use the object.

Defining classes and constructing objects

- **A class** is a definition of a category of objects.
- A class defines:
 - **the items of data** that make up the state of an object (**reference variables**), including their type: for example, whether they are integers, strings, or types defined by other classes;
 - **the 'behavior'** of objects of the class, by which we mean the **methods** that can be invoked on objects of the class.
- **Class members**: The instance variables and methods of a class or object are known collectively as its members.
- Classes **are like templates or blueprints** for objects. They do not make objects!
- There are usually three steps to using an object:
 1. Define a class.
 2. Construct an object.
 3. Store an object reference.

Example

(1) Define a class



Example (cont.)

```
User john;
```

```
john = new User();
```

2. Construct an object

3. Store an object reference.

```
User john = new User();
```

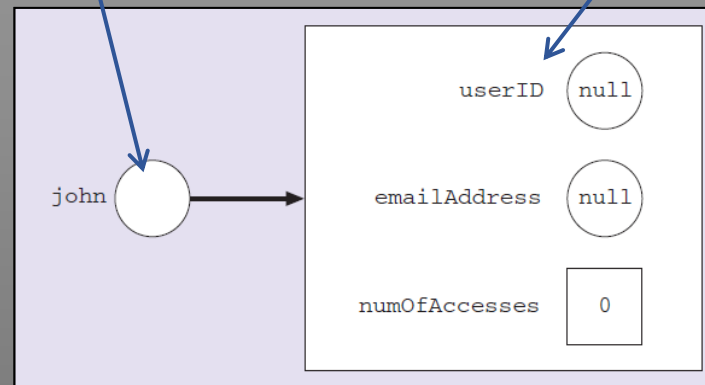
Store an object reference: storing a reference to a User object. The general statement is:

ClassName **objectName**;

In the above example, the declaration creates a reference variable, “john” of the type “User”

Construct an object. We call this **invoking a constructor**

After execution →

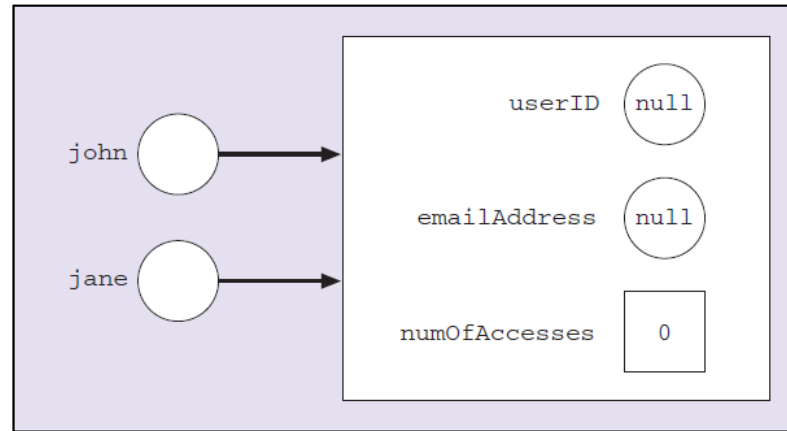


The keyword **null** represents a reference that does not point at any object.

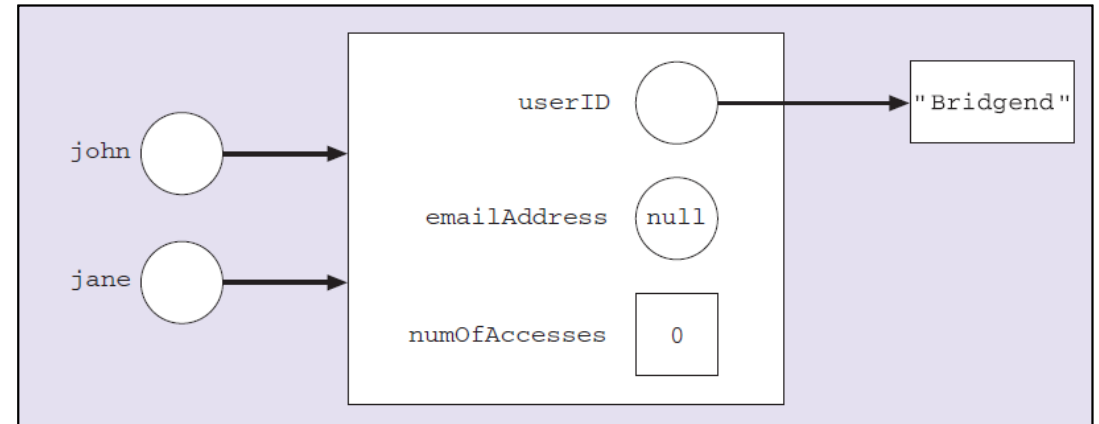
Reference variables and assignment

- If we write:

```
jane = john;
```



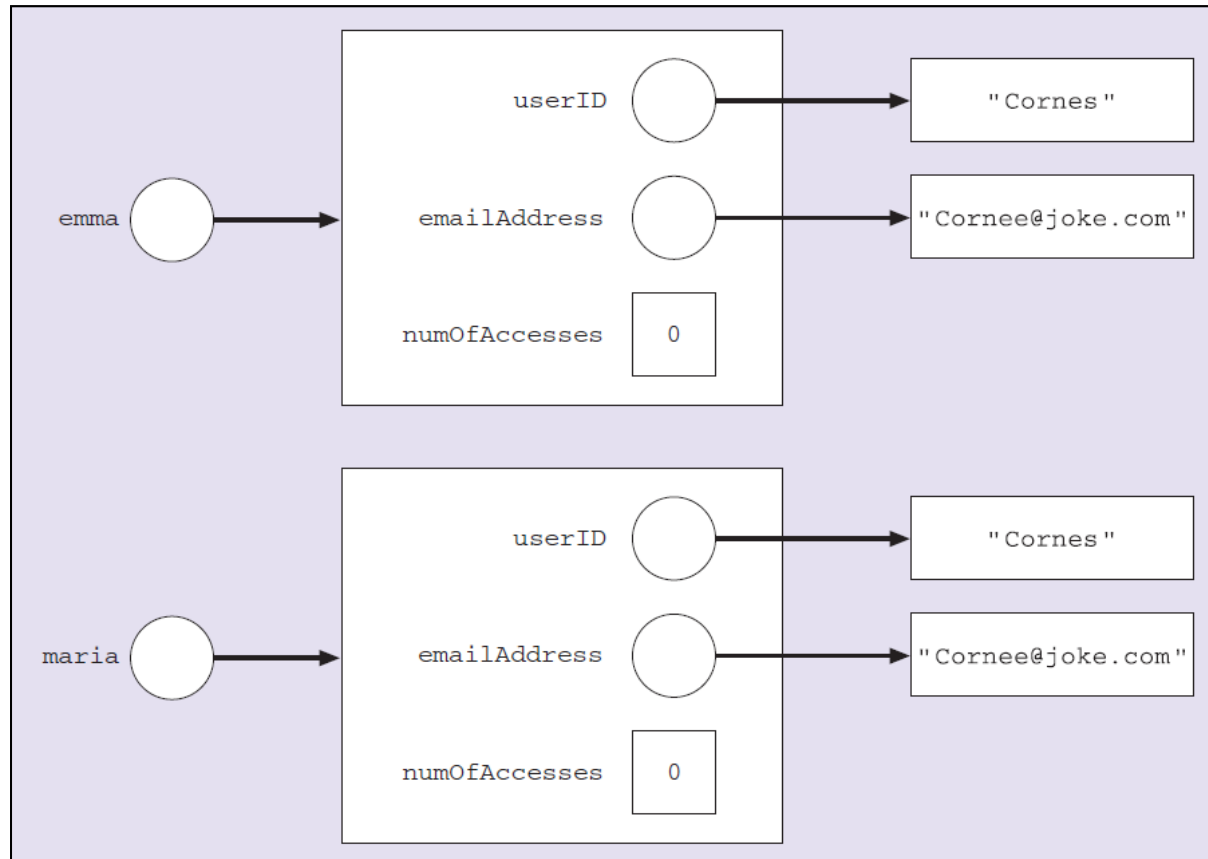
- If we make the userID variable
- in this object reference the string "Bridgend"



Reference variables and assignment (cont.)

- If we write:

```
User emma = new User("Cornes", "cornee@joke.com");  
User maria = new User("Cornes", "cornee@joke.com");
```



Classes

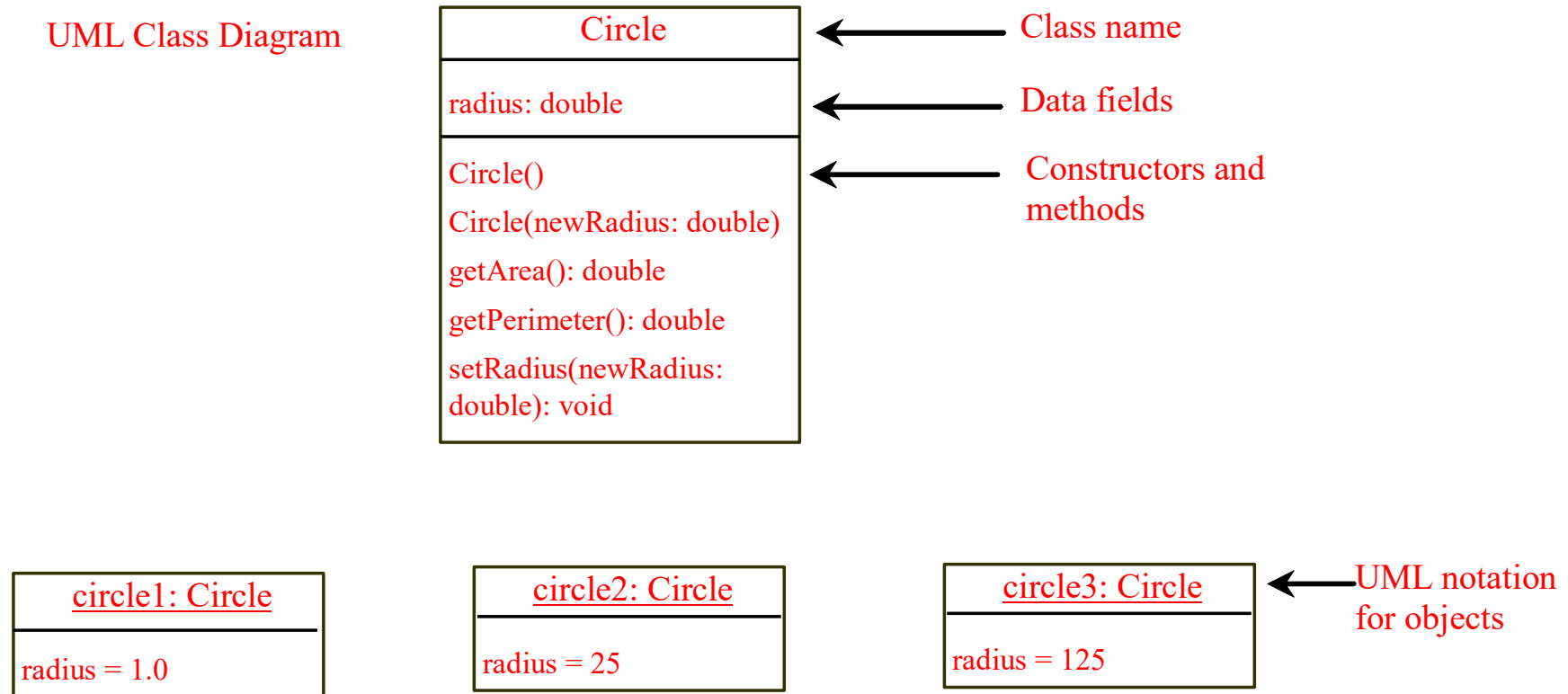
```
class Circle {  
    /** The radius of this circle */  
    double radius = 1.0;  
  
    /** Construct a circle object */  
    Circle() {  
    }  
  
    /** Construct a circle object */  
    Circle(double newRadius) {  
        radius = newRadius;  
    }  
  
    /** Return the area of this circle */  
    double getArea() {  
        return radius * radius * 3.14159;  
    }  
}
```

← Data field

← Constructors

← Method

UML Class Diagram



// Define the circle class with two constructors

```
class SimpleCircle {  
    double radius;  
    /** Construct a circle with radius 1 */  
    SimpleCircle() {  
        radius = 1;  
    }  
    /** Construct a circle with a specified radius */  
    SimpleCircle(double newRadius) {  
        radius = newRadius;  
    }  
    /** Return the area of this circle */  
    double getArea() {  
        return radius * radius * Math.PI;  
    }  
    /** Return the perimeter of this circle */  
    double getPerimeter() {  
        return 2 * radius * Math.PI;  
    }  
    /** Set a new radius for this circle */  
    void setRadius(double newRadius) {  
        radius = newRadius;  
    }  
}
```

Example: Defining Classes and Creating Objects

Example: Defining Classes and Creating Objects

```
public class TestSimpleCircle {  
    /** Main method */  
    public static void main(String[] args) {  
        // Create a circle with radius 1  
        SimpleCircle circle1 = new SimpleCircle();  
        System.out.println("The area of the circle of radius "  
            + circle1.radius + " is " + circle1.getArea());  
  
        // Create a circle with radius 25  
        SimpleCircle circle2 = new SimpleCircle(25);  
        System.out.println("The area of the circle of radius "  
            + circle2.radius + " is " + circle2.getArea());  
  
        // Create a circle with radius 125  
        SimpleCircle circle3 = new SimpleCircle(125);  
        System.out.println("The area of the circle of radius "  
            + circle3.radius + " is " + circle3.getArea());  
  
        // Modify circle radius  
        circle2.radius = 100; // or circle2.setRadius(100)  
        System.out.println("The area of the circle of radius "  
            + circle2.radius + " is " + circle2.getArea());  
    }  
}
```

The area of the circle of radius 1.0 is 3.141592653589793
The area of the circle of radius 25.0 is 1963.4954084936207
The area of the circle of radius 125.0 is 49087.385212340516
The area of the circle of radius 100.0 is 31415.926535897932

Constructors

- Constructors are a special kind of methods that are invoked to construct objects.

```
Circle() {  
}
```

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

Constructors, cont.

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

- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

Constructors, cont.

- Remember that a constructor cannot be abstract, final, synchronized, and static. We cannot override a constructor.
- There are two types of constructor in Java:
- Default Constructor (also known as a no-argument constructor)
- Parameterized Constructor

Creating Objects Using Constructors

```
new ClassName() ;
```

Example:

```
new Circle() ;
```

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

Default Constructor

- A class may be defined without constructors. In this case, a no-arg constructor with an empty body is implicitly defined in the class. This constructor, called *a default constructor*, is provided automatically *only if no constructors are explicitly defined in the class*.

Declaring Object Reference Variables

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

To declare a reference variable, use the syntax:

```
ClassName objectRefVar;
```

Example:

```
Circle myCircle;
```

Declaring/Creating Objects in a Single Step

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

Example:

Assign object reference Create an object

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

Accessing Object's Members

❑ Referencing the object's data:

`objectRefVar.data`

e.g., `myCircle.radius`

❑ Invoking the object's method:

`objectRefVar.methodName (arguments)`

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

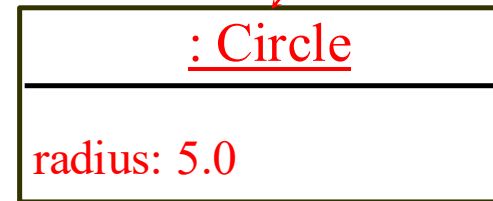
Trace Code

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

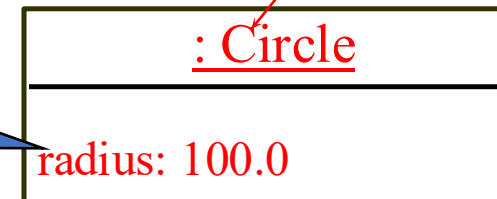
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle reference value



Change radius in
yourCircle

Caution

- Recall that you use `Math.methodName(arguments)` (e.g., `Math.pow(3, 2.5)`)
- to invoke a method in the `Math` class. Can you invoke `getArea()` using `SimpleCircle.getArea()`? The answer is no. All the methods used before this chapter are **static methods**, which are defined using the `static` keyword.
- However, `getArea()` is non-static. It must be invoked from an object using `objectRefVar.methodName(arguments)` (e.g., `myCircle.getArea()`).
- More explanations will be given in the section on “Static Variables, Constants, and Methods.”

Reference Data Fields

- The data fields can be of reference types. For example, the following Student class contains a data field name of the String type.

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

Default Value for a Data Field

- If a data field of a reference type does not reference any object, the data field holds a special literal value, null.
- The default value of a data field is null for a reference type
- 0 for a numeric type
- false for a boolean type
- '\u0000' for a char type.
- However, Java assigns no default value to a local variable inside a method.

Default Value for a Data Field

```
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);  
    }  
}
```

name? null
age? 0
isScienceMajor? false
gender?

Default Value for a Data Field

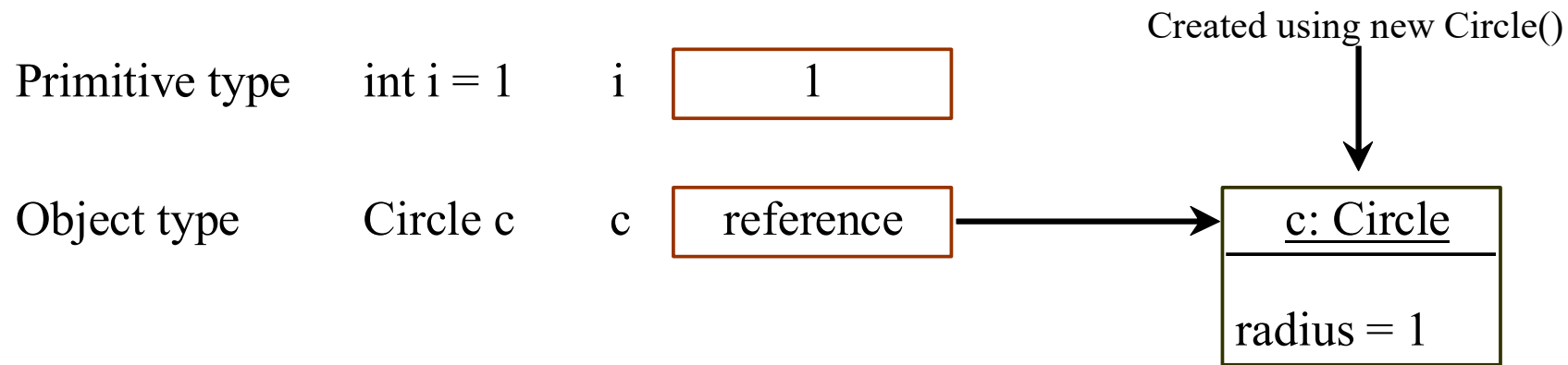
- Java assigns no default value to a local variable inside a method.

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



Compile error: variable not initialized

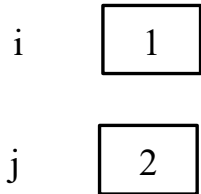
Differences between Variables of Primitive Data Types and Object Types



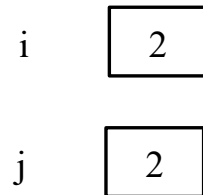
Copying Variables of Primitive Data Types and Object Types

Primitive type assignment $i = j$

Before:

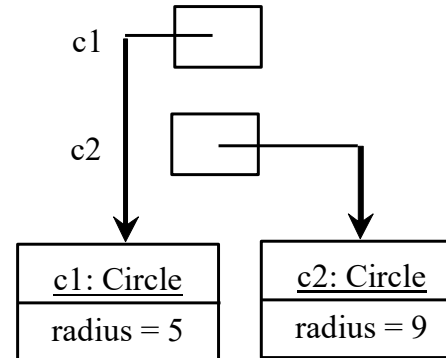


After:

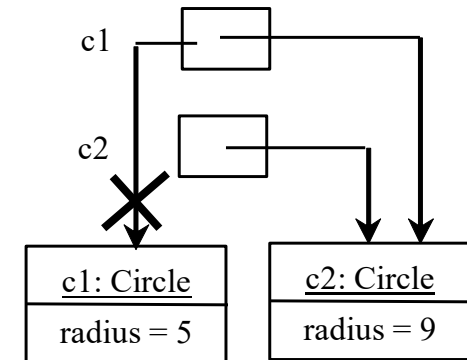


Object type assignment $c1 = c2$

Before:



After:



Garbage Collection

- As shown in the previous figure, after the assignment statement `c1 = c2`, `c1` points to the same object referenced by `c2`. The object previously referenced by `c1` is no longer referenced. This object is known as garbage. Garbage is automatically collected by JVM.

Static Variables, Constants, and Methods

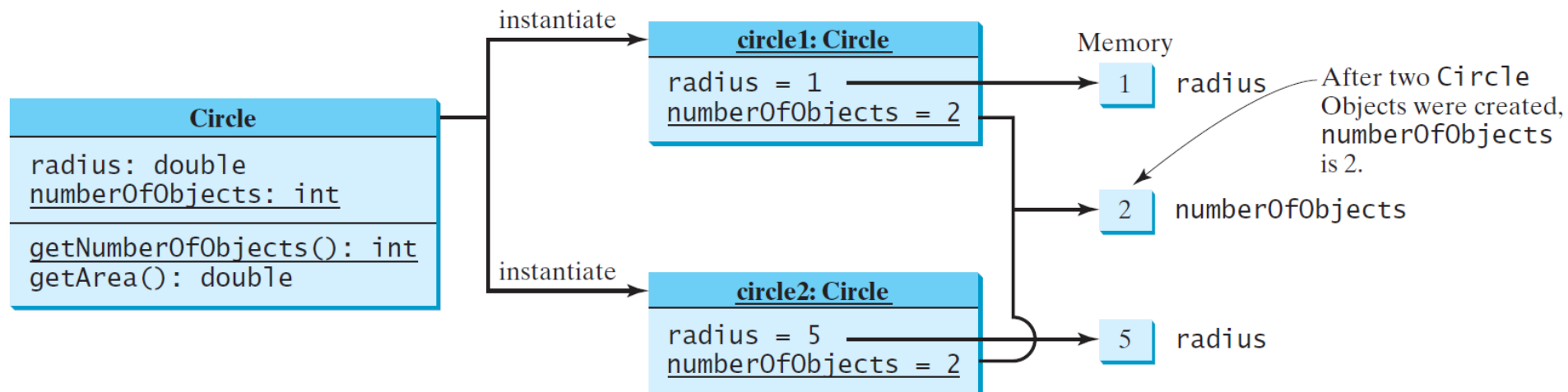
Static variables are shared by all the instances of the class.

Static methods are not tied to a specific object. Static constants are final variables shared by all the instances of the class.

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

UML Notation:

underline: static variables or methods



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.

```
public class CircleWithStaticMembers {  
    double radius;  
    static int numberOfObjects = 0;  
    CircleWithStaticMembers() {  
        radius = 1.0;  
        numberOfObjects++;  
    }  
    CircleWithStaticMembers(double newRadius) {  
        radius = newRadius;  
        numberOfObjects++;  
    }  
    static int getNumberOfObjects() {  
        return numberOfObjects;  
    }  
    double getArea() {  
        return radius * radius * Math.PI;  
    }  
}
```

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.

Example of Using Instance and Class Variables and Method

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Before creating objects");  
        System.out.println("The number of Circle objects is " + CircleWithStaticMembers.numberOfObjects);  
        CircleWithStaticMembers c1 = new CircleWithStaticMembers();  
        System.out.println("\nAfter creating c1");  
        System.out.println("c1: radius (" + c1.radius + ") and number of Circle objects (" + c1.numberOfObjects + ")");  
        CircleWithStaticMembers c2 = new CircleWithStaticMembers(5);  
        c1.radius = 9;  
        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 + ")");  
    }  
}
```

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)

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.

Visibility Modifiers and Accessor/Mutator Methods

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

```
package p1;

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

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

```
package p1;

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

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

```
package p2;

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

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

Visibility Modifiers and Accessor/Mutator Methods

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

```
package p1;  
  
class C1 {  
    ...  
}
```

```
package p1;  
  
public class C2 {  
    can access C1  
}
```

```
package p2;  
  
public class C3 {  
    cannot access C1;  
    can access C2;  
}
```

NOTE

- An object cannot access its private members, as shown in (b). It is OK, however, if the object is declared in its own class, as shown in (a).

```
public class C {  
    private boolean x;  
  
    public static void main(String[] args) {  
        C c = new C();  
        System.out.println(c.x);  
        System.out.println(c.convert());  
    }  
  
    private int convert() {  
        return x ? 1 : -1;  
    }  
}
```

(a) This is okay because object `c` is used inside the class `C`.

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

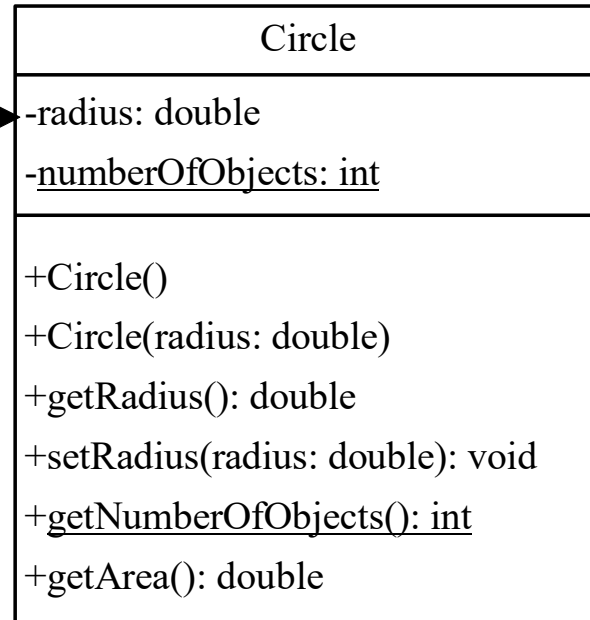
(b) This is wrong because `x` and `convert` are private in class `C`.

Why Data Fields Should Be private?

- To protect data.
- **Data Encapsulation**

Example of Data Field Encapsulation

The - sign indicates
private modifier



The radius of this circle (default: 1.0).

The number of circle objects created.

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.

Example of Data Field Encapsulation

```
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;
    }
    public static int getNumberOfObjects() {
        return numberOfObjects;
    }
    public double getArea() {
        return radius * radius * Math.PI;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        CircleWithPrivateDataFields myCircle =
            new CircleWithPrivateDataFields(5.0);
        System.out.println("The area of the circle of radius "
            + myCircle.getRadius() + " is " + myCircle.getArea());
        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 "
            + CircleWithPrivateDataFields.getNumberOfObjects());
    }
}
```

The area of the circle of radius 5.0 is 78.53981633974483
The area of the circle of radius 5.5 is 95.03317777109125
The number of objects created is 1

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)

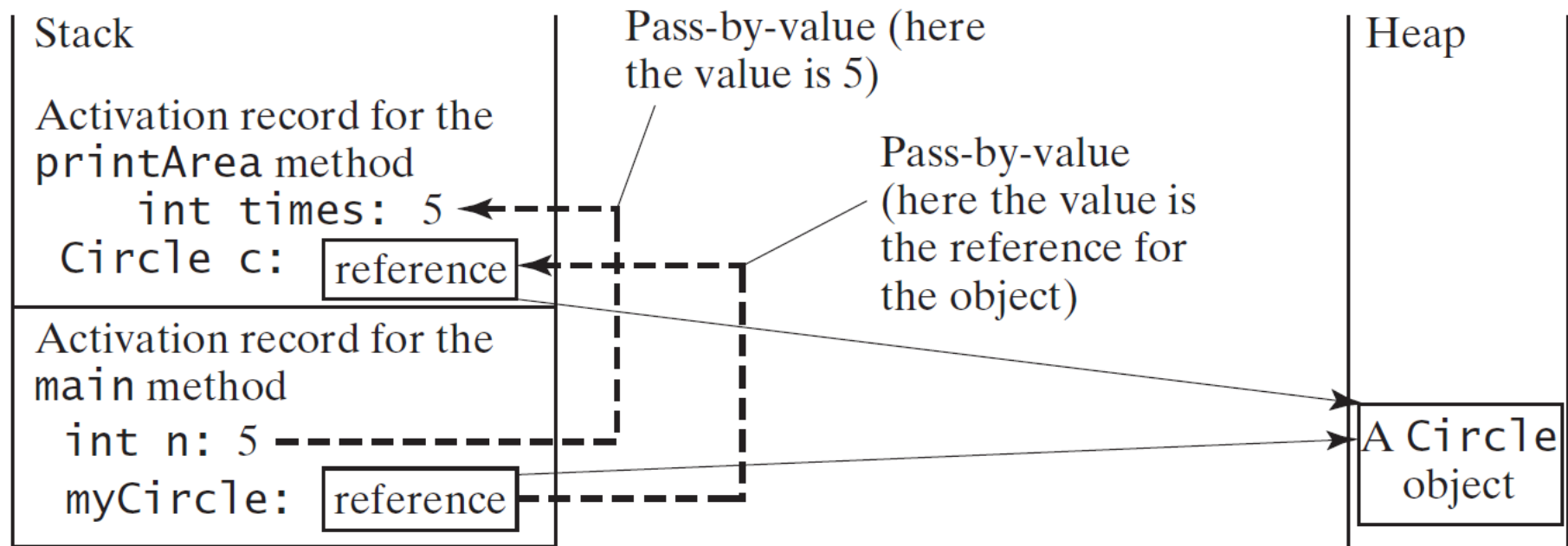
Passing Objects to Methods Example

```
public class Main {  
    public static void main(String[] args) {  
        CircleWithPrivateDataFields myCircle = new CircleWithPrivateDataFields(1);  
        int n = 5;  
        printAreas(myCircle, n);  
        System.out.println("\n" + "Radius is " + myCircle.getRadius());  
        System.out.println("n is " + n);  
    }  
    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--;  
        }  
    }  
}
```

Radius	Area
1.0	3.141592653589793
2.0	12.566370614359172
3.0	28.274333882308138
4.0	50.26548245743669
5.0	78.53981633974483

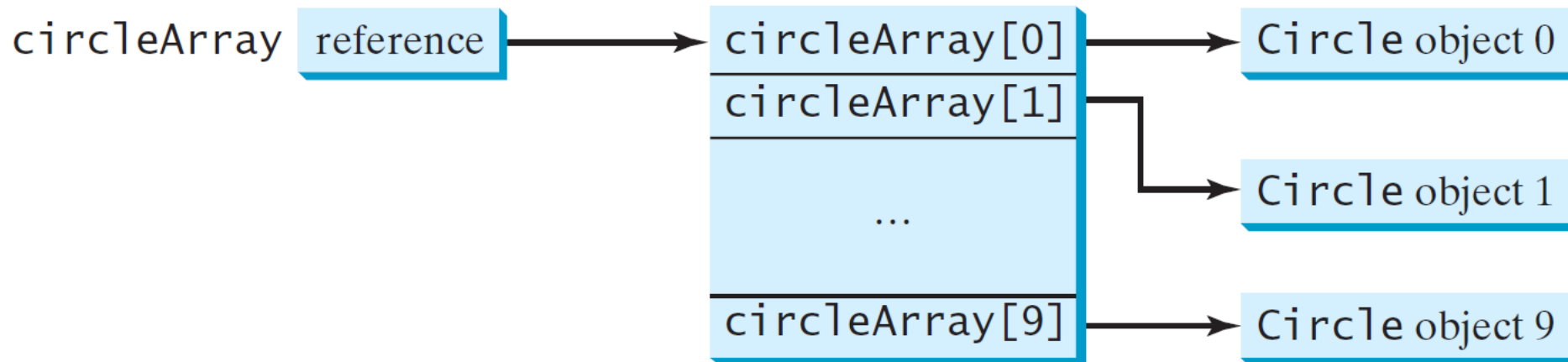
Radius is 6.0
n is 5

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



```

public class Main {
    public static void main(String[] args) {
        CircleWithPrivateDataFields[] circleArray;
        circleArray = createCircleArray();
        printCircleArray(circleArray);
    }
    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;
    }
    public static void printCircleArray(
        CircleWithPrivateDataFields[] 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("-----");
        System.out.printf("%-30s%-15f\n", "The total areas of circles is",
            sum(circleArray));
    }
}

```



```

public static double sum(
    CircleWithPrivateDataFields[] circleArray) {
    double sum = 0;
    for (int i = 0; i < circleArray.length; i++)
        sum += circleArray[i].getArea();
    return sum;
}

```

Radius	Area
70.799357	15747.387180
20.283846	1292.559252
79.881382	20046.613256
32.226883	3262.770072
16.435723	848.647837

The total areas of circles is 41197.977597	

Information hiding: **Getter and setter methods**

- A **getter method** (also called an '**accessor**' method):
 - It is one that accesses some attribute of an object.
 - A getter method **should not change** the state of the object.
- A **setter method** (also called a '**mutator**' method)
 - It is one that changes the state of an object by setting the value of an instance variable.
 - Setter methods **do not** normally **return** a value.

Information hiding: getter and setter methods

- **Example:** Getter and setter methods for the User class

```
public class User
{
    private String userID;
    private String emailAddress;
    private int numOfAccesses;

    // constructors have still to be written

    // setter methods
    public void setUserID (String usIdVal)
    {
        userID = usIdVal;
    }

    public void setEmailAddress (String emailAddressVal)
    {
        emailAddress = emailAddressVal;
    }
}
```

```
// getter methods
public String getUserID ()
{
    return userID;
}

public String getEmailAddress ()
{
    return emailAddress;
}

public int getNumOfAccesses ()
{
    return numOfAccesses;
}

// update number of accesses by this user
public void updateAccesses ()
{
    numOfAccesses++;
}
}
```

Information hiding: Member access by dot notation (.)

- The keyword **public** in front of a member in the class User indicates that it may be accessed using dot notation with a User reference variable.
- In contrast, because we have used the keyword **private** in front of the User instance variables, there is no access outside the class itself to these variables.

- This is NOT allowed → `john.numOfAccesses = -1; // not allowed`

- This is allowed → `john.updateAccesses(); // allowed`

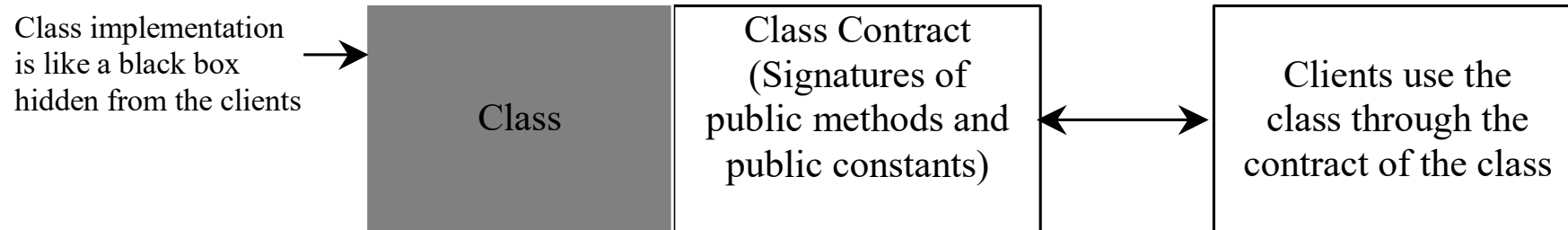
The this Keyword

- ❑ 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. (Calling Overloaded Constructor)

```
public class three {  
    private int a,b,c;  
    public three(int a,int b,int c){  
        this.a=a;  
        this.b=b;  
        this.c=c;  
    }  
    public three(){  
        this(0,0,0);  
    }  
    public three(int a){  
        this(a,0,0);  
    }  
    public three(int a,int b){  
        this(a,b,0);  
    }  
}
```

Class Abstraction and Encapsulation

Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.



Designing the Loan Class

Loan	
-annualInterestRate: double	The annual interest rate of the loan (default: 2.5).
-numberOfYears: int	The number of years for the loan (default: 1)
-loanAmount: double	The loan amount (default: 1000).
-loanDate: Date	The date this loan was created.
+Loan()	Constructs a default Loan object.
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)	Constructs a loan with specified interest rate, years, and loan amount.
+getAnnualInterestRate(): double	Returns the annual interest rate of this loan.
+getNumberOfYears(): int	Returns the number of the years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+getLoanDate(): Date	Returns the date of the creation of this loan.
+setAnnualInterestRate(annualInterestRate: double): void	Sets a new annual interest rate to this loan.
+setNumberOfYears(numberOfYears: int): void	Sets a new number of years to this loan.
+setLoanAmount(loanAmount: double): void	Sets a new amount to this loan.
+getMonthlyPayment(): double	Returns the monthly payment of this loan.
+getTotalPayment(): double	Returns the total payment of this loan.

Loan

TestLoanClass

```
public class Loan {
    private double annualInterestRate;
    private int numberOfYears;
    private double loanAmount;
    private java.util.Date loanDate;

    /** No-arg constructor */
    public Loan() {
        this(2.5, 1, 1000);
    }

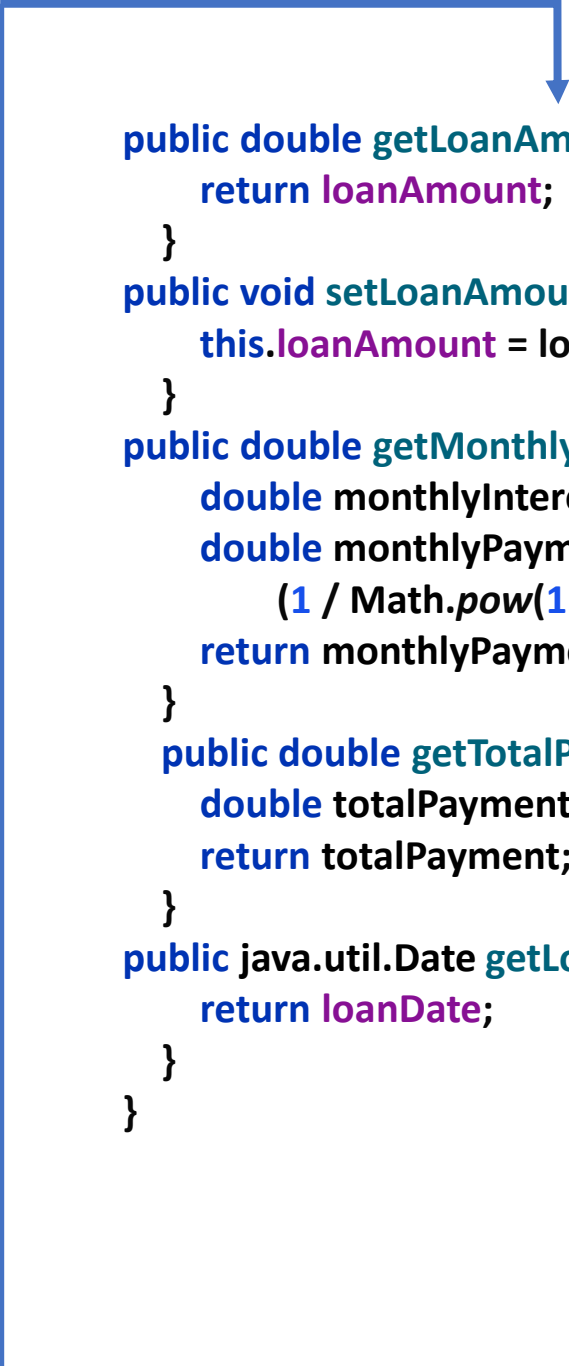
    public Loan(double annualInterestRate, int numberOfYears,
        double loanAmount) {
        this.annualInterestRate = annualInterestRate;
        this.numberOfYears = numberOfYears;
        this.loanAmount = loanAmount;
        loanDate = new java.util.Date();
    }

    public double getAnnualInterestRate() {
        return annualInterestRate;
    }

    public void setAnnualInterestRate(double annualInterestRate) {
        this.annualInterestRate = annualInterestRate;
    }

    public int getNumberOfYears() {
        return numberOfYears;
    }

    public void setNumberOfYears(int numberOfYears) {
        this.numberOfYears = numberOfYears;
    }
}
```



```
public double getLoanAmount() {
    return loanAmount;
}

public void setLoanAmount(double loanAmount) {
    this.loanAmount = loanAmount;
}

public double getMonthlyPayment() {
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
        (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
    return monthlyPayment;
}

public double getTotalPayment() {
    double totalPayment = getMonthlyPayment() * numberOfYears * 12;
    return totalPayment;
}

public java.util.Date getLoanDate() {
    return loanDate;
}
}
```

```
import java.util.Scanner;
```

```
public class Main {
```

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

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

```
        System.out.print(
```

```
            "Enter annual interest rate, for example, 8.25: ");
```

```
        double annualInterestRate = input.nextDouble();
```

```
        System.out.print("Enter number of years as an integer: ");
```

```
        int numberOfYears = input.nextInt();
```

```
        System.out.print("Enter loan amount, for example, 120000.95: ");
```

```
        double loanAmount = input.nextDouble();
```

```
        Loan loan =
```

```
            new Loan(annualInterestRate, numberOfYears, loanAmount);
```

```
        System.out.printf("The loan was created on %s\n" +
```

```
            "The monthly payment is %.2f\nThe total payment is %.2f\n",
```

```
            loan.getLoanDate().toString(), loan.getMonthlyPayment(),
```

```
            loan.getTotalPayment());
```

```
    }
```

```
}
```

Enter annual interest rate, for example, 8.25: 22

Enter number of years as an integer: 5

Enter loan amount, for example, 120000.95: 120000

The loan was created on Tue Nov 08 19:00:49 EET 2022

The monthly payment is 3314.27

The total payment is 198856.17

Example: The Course Class

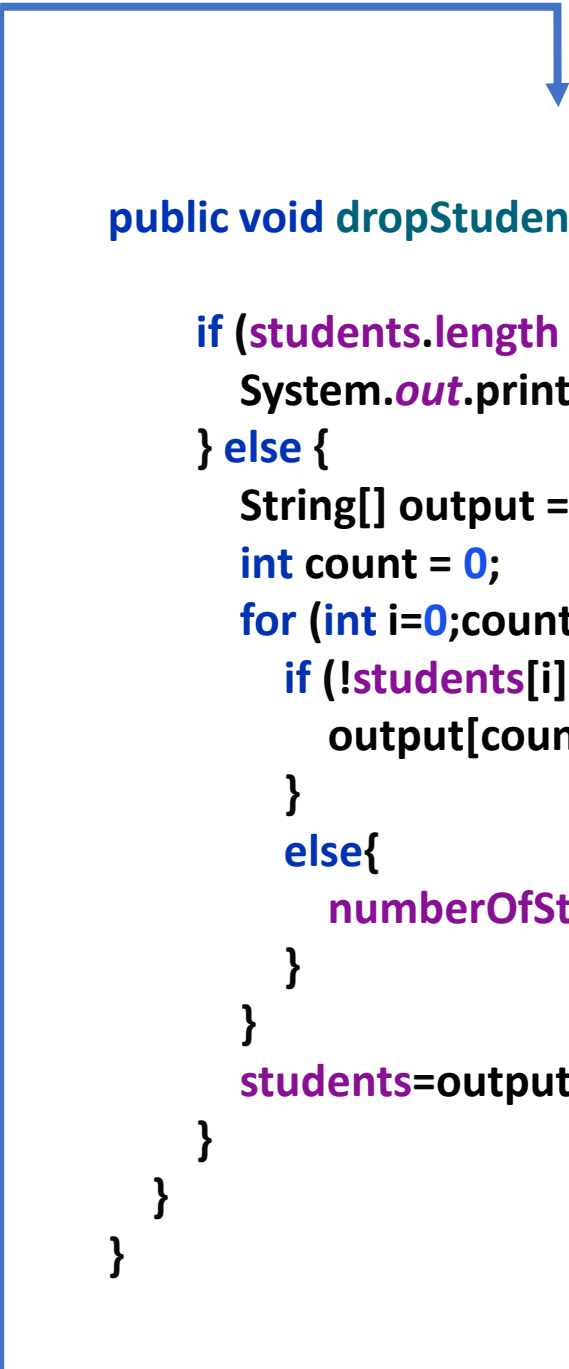
Course	
-courseName: String	The name of the course.
-students: String[]	An array to store the students for the course.
-numberOfStudents: int	The number of students (default: 0).
+Course(courseName: String)	Creates a course with the specified name.
+getCourseName(): String	Returns the course name.
+addStudent(student: String): void	Adds a new student to the course.
+dropStudent(student: String): void	Drops a student from the course.
+getStudents(): String[]	Returns the students in the course.
+getNumberOfStudents(): int	Returns the number of students in the course.

Course

TestCourse


```
public class Course {
    private String courseName;
    private String[] students = new String[100];
    private int numberOfStudents;

    public Course(String courseName) {
        this.courseName = courseName;
    }
    public void addStudent(String student) {
        students[numberOfStudents] = student;
        numberOfStudents++;
    }
    public String[] getStudents() {
        return students;
    }
    public int getNumberOfStudents() {
        return numberOfStudents;
    }
    public String getCourseName() {
        return courseName;
    }
}
```



```
public void dropStudent(String student) {

    if (students.length <= 0) {
        System.out.println("No student registered for this course");
    } else {
        String[] output = new String[students.length - 1];
        int count = 0;
        for (int i=0;count<numberOfStudents;i++) {
            if (!students[i].equals(student)) {
                output[count++] =students[i] ;
            }
            else{
                numberOfStudents--;
            }
        }
        students=output;
    }
}
```

```

public class Main {
    public static void main(String[] args) {
        Course course1 = new Course("Data Structures");
        Course course2 = new Course("Database Systems");
        course1.addStudent("Jones");
        course1.addStudent("Smith");
        course1.addStudent("Kennedy");
        course2.addStudent("Jones");
        course2.addStudent("Smith");
        System.out.println("Number of students in course1: "
            + course1.getNumberOfStudents());
        String[] students = course1.getStudents();
        for (int i = 0; i < course1.getNumberOfStudents(); i++)
            System.out.print(students[i] + ", ");
        course1.dropStudent("Kennedy");
        System.out.println();
        System.out.println("Course1 after Removeing a student");
        System.out.println("Number of students in course1: "
            + course1.getNumberOfStudents());
        students = course1.getStudents();
        for (int i = 0; i < course1.getNumberOfStudents(); i++)
            System.out.print(students[i] + ", ");

        System.out.println();
        System.out.print("Number of students in course2: "
            + course2.getNumberOfStudents());
    }
}

```

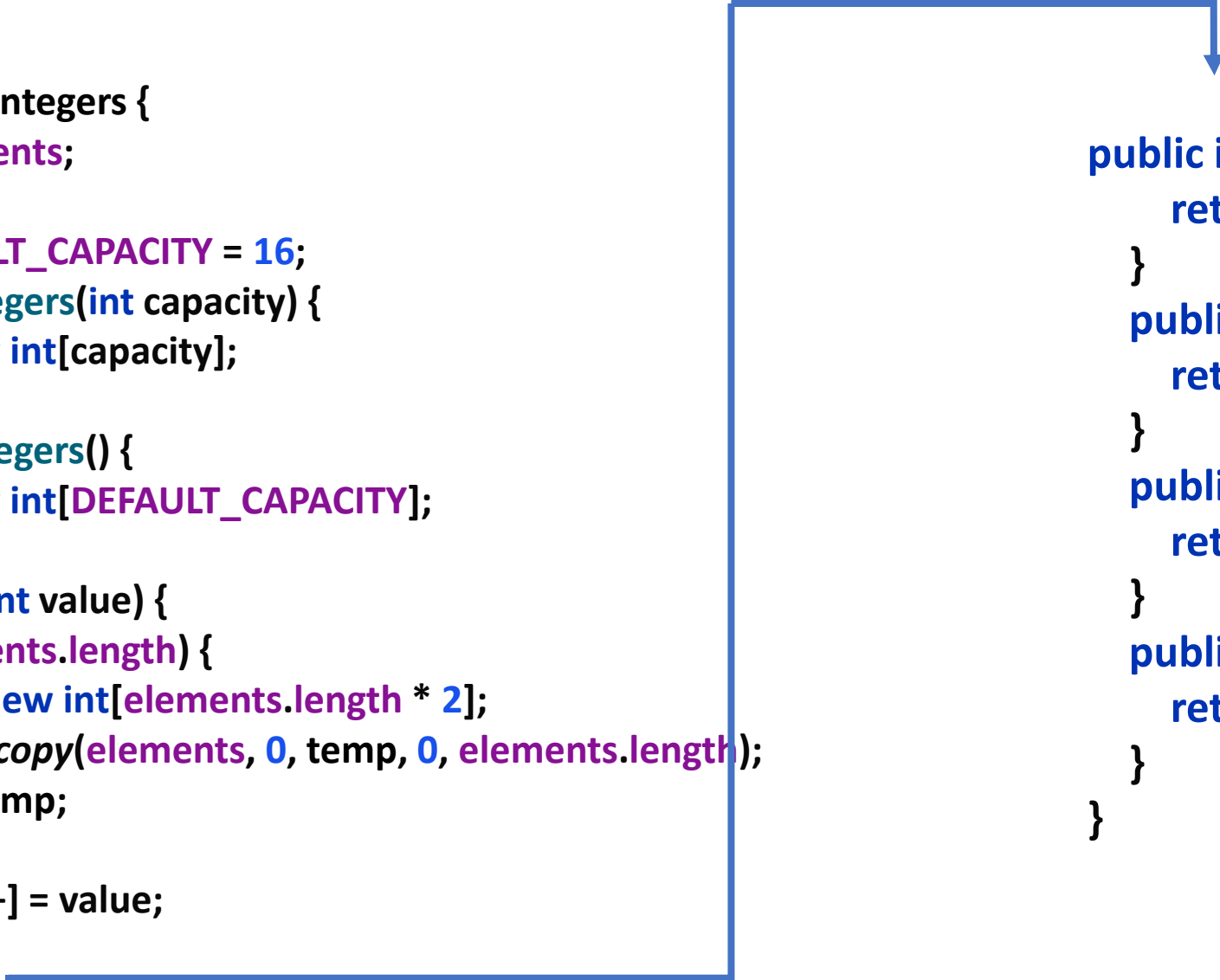
Number of students in course1: 3
 Jones, Smith, Kennedy,
 Course1 after Removeing a student
 Number of students in course1: 2
 Jones, Smith,
 Number of students in course2: 2

Example: The StackOfIntegers Class

StackOfIntegers	
-elements: int[]	An array to store integers in the stack.
-size: int	The number of integers in the stack.
+StackOfIntegers()	Constructs an empty stack with a default capacity of 16.
+StackOfIntegers(capacity: int)	Constructs an empty stack with a specified capacity.
+empty(): boolean	Returns true if the stack is empty.
+peek(): int	Returns the integer at the top of the stack without removing it from the stack.
+push(value: int): int	Stores an integer into the top of the stack.
+pop(): int	Removes the integer at the top of the stack and returns it.
+getSize(): int	Returns the number of elements in the stack.

TestStackOfIntegers

```
public class StackOfIntegers {
    private int[] elements;
    private int size;
    private int DEFAULT_CAPACITY = 16;
    public StackOfIntegers(int capacity) {
        elements = new int[capacity];
    }
    public StackOfIntegers() {
        elements = new int[DEFAULT_CAPACITY];
    }
    public void push(int value) {
        if (size >= elements.length) {
            int[] temp = new int[elements.length * 2];
            System.arraycopy(elements, 0, temp, 0, elements.length);
            elements = temp;
        }
        elements[size++] = value;
    }
}
```

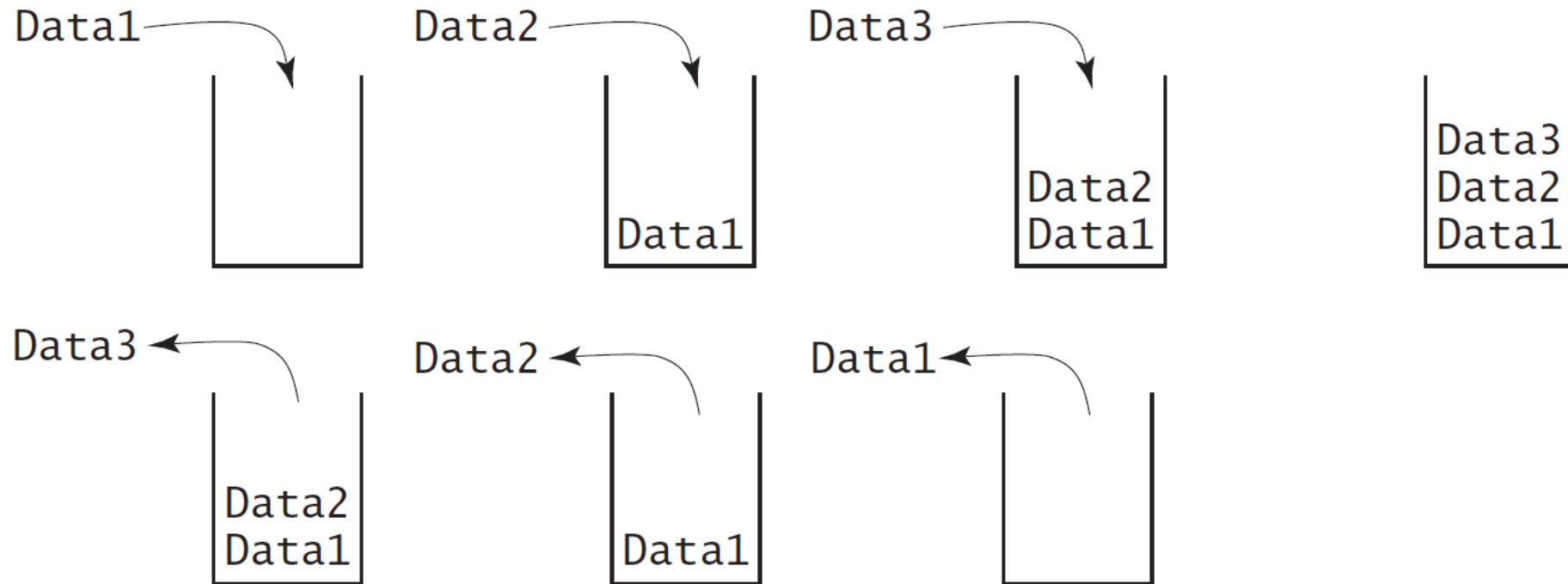


```
public int pop() {
    return elements[--size];
}
public int peek() {
    return elements[size - 1];
}
public boolean empty() {
    return size == 0;
}
public int getSize() {
    return size;
}
}
```

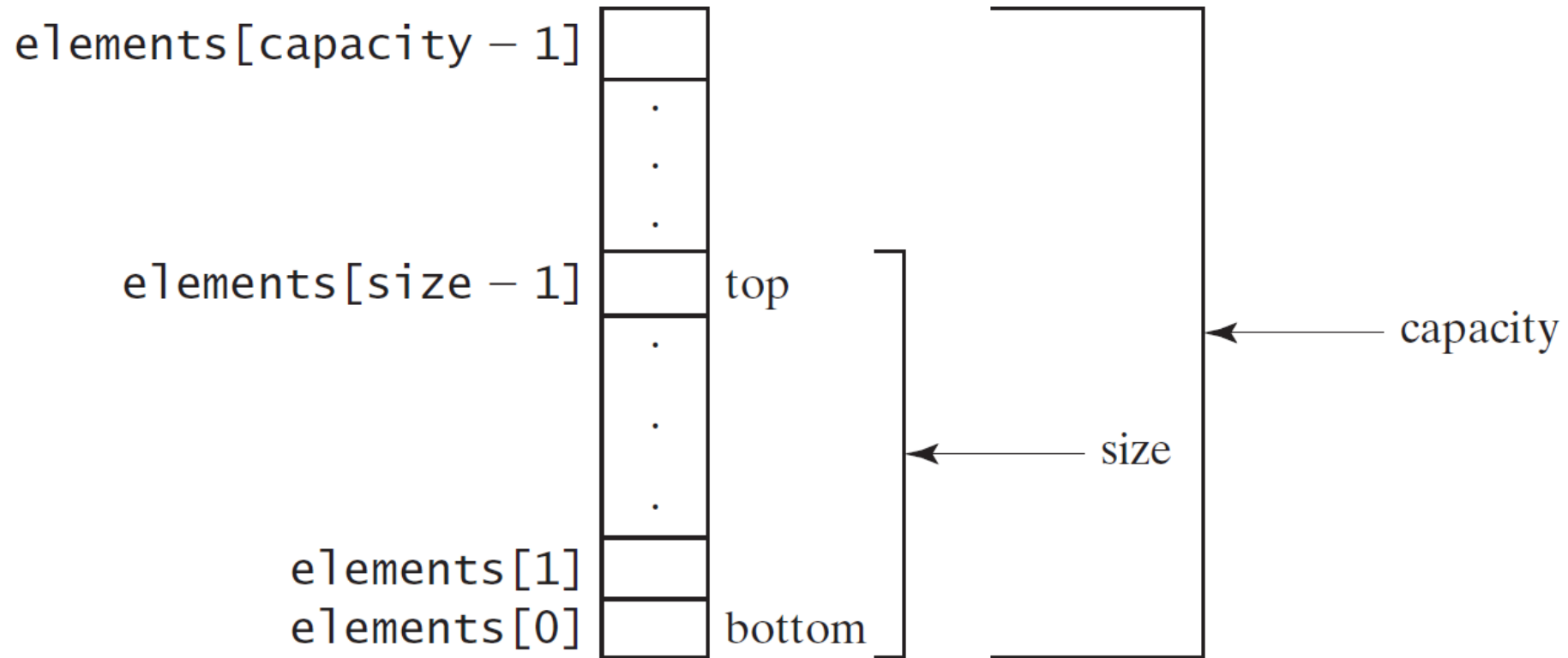
```
public class Main {  
    public static void main(String[] args) {  
        StackOfIntegers stack = new StackOfIntegers();  
        for (int i = 0; i < 10; i++)  
            stack.push(i); // Push i to the stack  
        while (!stack.empty()) // Test if stack is empty  
            System.out.print(stack.pop() + " "); // Remove and return from stack  
    }  
}
```

9 8 7 6 5 4 3 2 1 0

Designing the StackOfIntegers Class



Implementing StackOfIntegers Class



StackOfIntegers

Class Relationships

Association

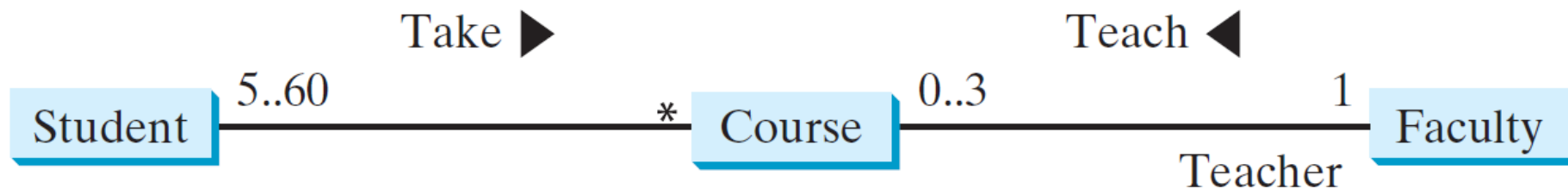
Aggregation

Composition

Inheritance (Chapter 13)

Association

- Association: is a general binary relationship that describes an activity between two classes.
- Association in Java is a connection or relation between two separate classes that are set up through their objects.
- Association relationship indicates how objects know each other and how they are using each other's functionality.
- It can be one-to-one, one-to-many, many-to-one and many-to-many.
- This UML diagram shows that a student may take any number of courses, a faculty member may teach at most three courses, a course may have from 5 to 60 students, and a course is taught by only one faculty member.



Association

- Each class involved in an association may specify a **multiplicity**, which is placed at the side of the class to specify how many of the class's objects are involved in the relationship in UML.
- A multiplicity could be a number or an interval that specifies how many of the class's objects are involved in the relationship.
- The character * means an unlimited number of objects, and the interval m..n indicates that the number of objects is between m and n
- For example : each student may take any number of courses, and each course must have at least 5 and at most 60 students. Each course is taught by only one faculty member, and a faculty member may teach from 0 to 3 courses per semester.

Association

- In Java code, you can implement associations by using data fields and methods.
- The relation “a student takes a course” is implemented using the addCourse method in the Student class and the addStudent method in the Course class.
- The relation “a faculty teaches a course” is implemented using the addCourse method in the Faculty class and the setFaculty method in the Course class.
- The Student class may use a list to store the courses that the student is taking, the Faculty class may use a list to store the courses that the faculty is teaching
- Course class may use a list to store students enrolled in the course and a data field to store the instructor who teaches the course.

Association

```
public class Student {  
    private Course[]  
        courseList;  
  
    public void addCourse(  
        Course c) { ... }  
}
```

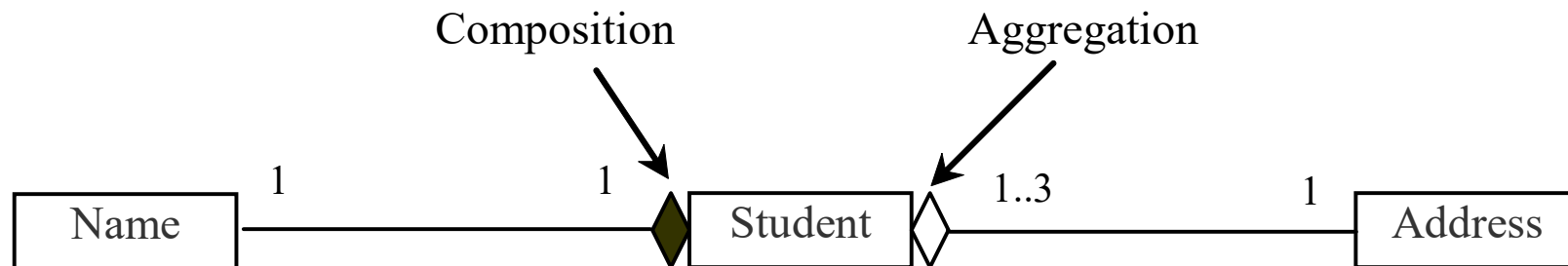
```
public class Course {  
    private Student[]  
        classList;  
    private Faculty faculty;  
  
    public void addStudent(  
        Student s) { ... }  
  
    public void setFaculty(  
        Faculty faculty) { ... }  
}
```

```
public class Faculty {  
    private Course[]  
        courseList;  
  
    public void addCourse(  
        Course c) { ... }  
}
```

FIGURE 10.5 The association relations are implemented using data fields and methods in classes.

Object Composition

- Composition is actually a special case of the aggregation relationship. **Aggregation models *has-a* relationships** and represents an ownership relationship between two objects.
- The owner object is called an *aggregating object* and its class an *aggregating class*.
- The subject object is called an *aggregated object* and its class an *aggregated class*.



Aggregation or Composition

- We refer aggregation between two objects as composition if the existence of the aggregated object is dependent on the aggregating object.
- In other words, if a relationship is composition, the aggregated object cannot exist on its own
- For example, “a student has a name” is a composition relationship between the Student class and the Name class because Name is dependent on Student,
- “a student has an address” is an aggregation relationship between the Student class and the Address class because an address can exist by itself. Composition implies exclusive ownership.
- One object owns another object. When the owner object is destroyed, the dependent object is destroyed as well. (The class (owner) who create (construct) the object)

Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationship in Figure 10.6 can be represented as follows:

```
public class Name {  
    ...  
}
```

Aggregated class

```
public class Student {  
    private Name name;  
    private Address address;  
  
    ...  
}
```

Aggregating class

```
public class Address {  
    ...  
}
```

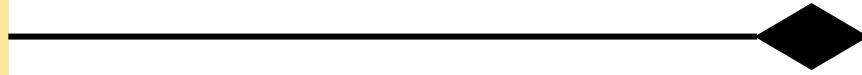
Aggregated class

Composition in Java Example

```
public class Job {  
    private String role;  
    private long salary;  
    private int id;  
  
    public String getRole() {  
        return role;  
    }  
    public void setRole(String role) {  
        this.role = role;  
    }  
    public long getSalary() {  
        return salary;  
    }  
    public void setSalary(long salary) {  
        this.salary = salary;  
    }  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
}
```

```
public class Person {  
  
    //composition has-a relationship  
    private Job job;  
  
    public Person(){  
        this.job=new Job();  
        job.setSalary(1000L);  
    }  
    public long getSalary() {  
        return job.getSalary();  
    }  
}
```

```
public class TestPerson {  
  
    public static void main(String[] args) {  
        Person person = new Person();  
        long salary = person.getSalary();  
    }  
}
```

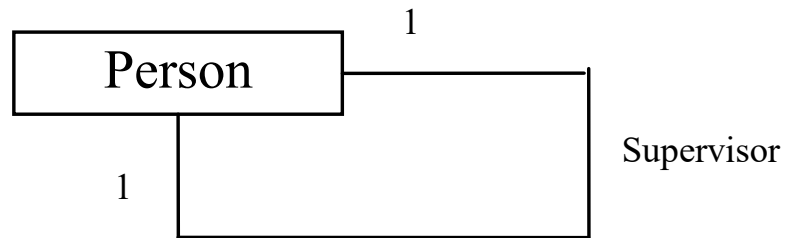


Aggregation or Composition

Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both compositions.

Aggregation Between Same Class

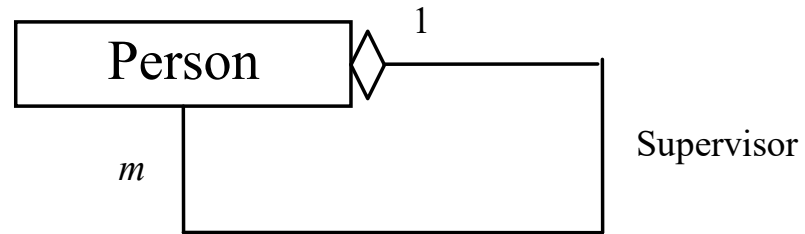
Aggregation may exist between objects of the same class. For example, a person may have a supervisor.



```
public class Person {  
    // The type for the data is the class itself  
    private Person supervisor;  
    ...  
}
```

Aggregation Between Same Class

What happens if a person has several supervisors?

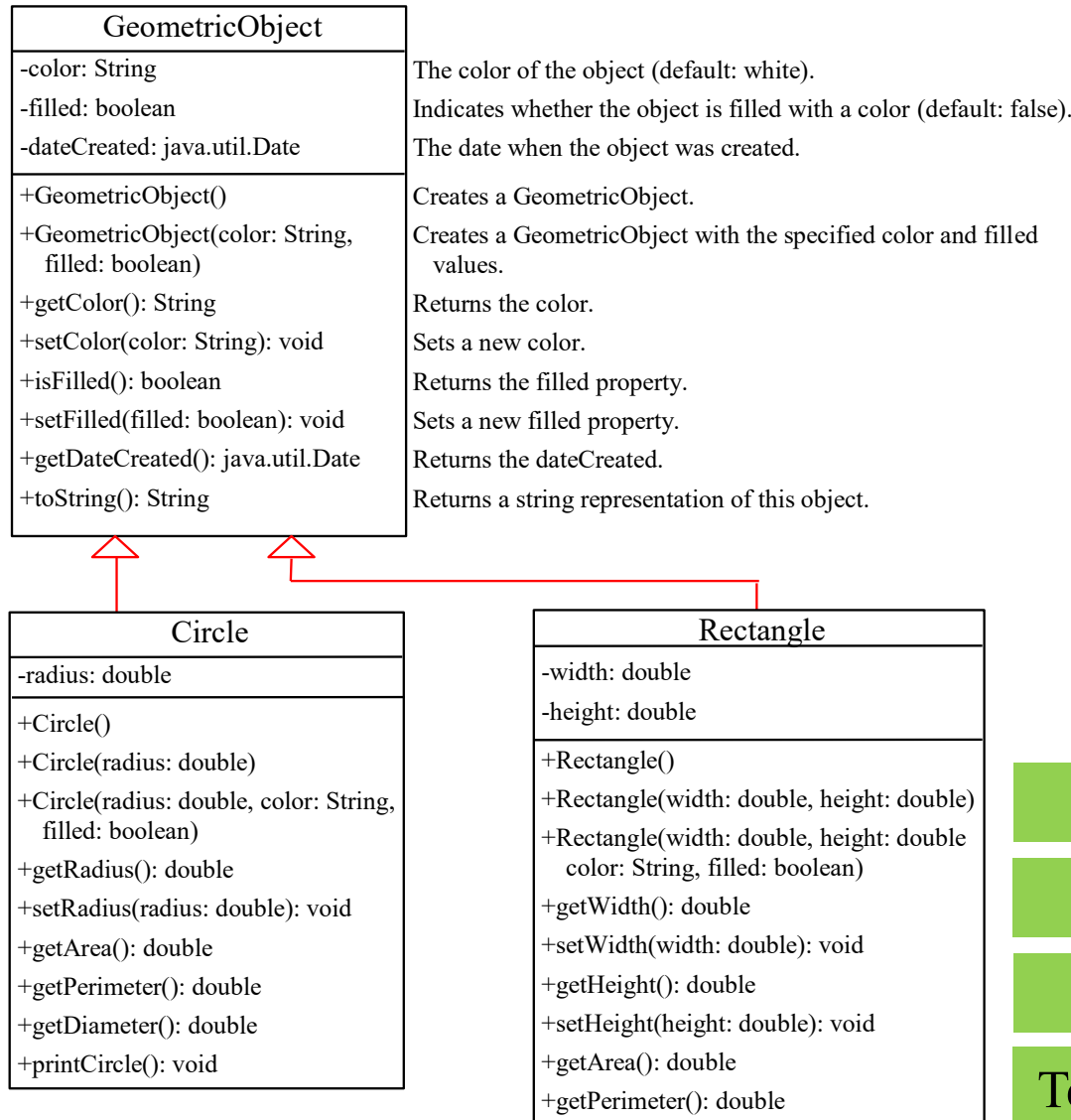


```
public class Person {  
    ...  
    private Person[] supervisors;  
}
```

Inheritance and Polymorphism

- Suppose you will define classes to model circles, rectangles, and triangles.
- These classes have many common features.
- What is the best way to design these classes so to avoid redundancy?
The answer is to use inheritance.

Superclasses and Subclasses



GeometricObject

Circle

Rectangle

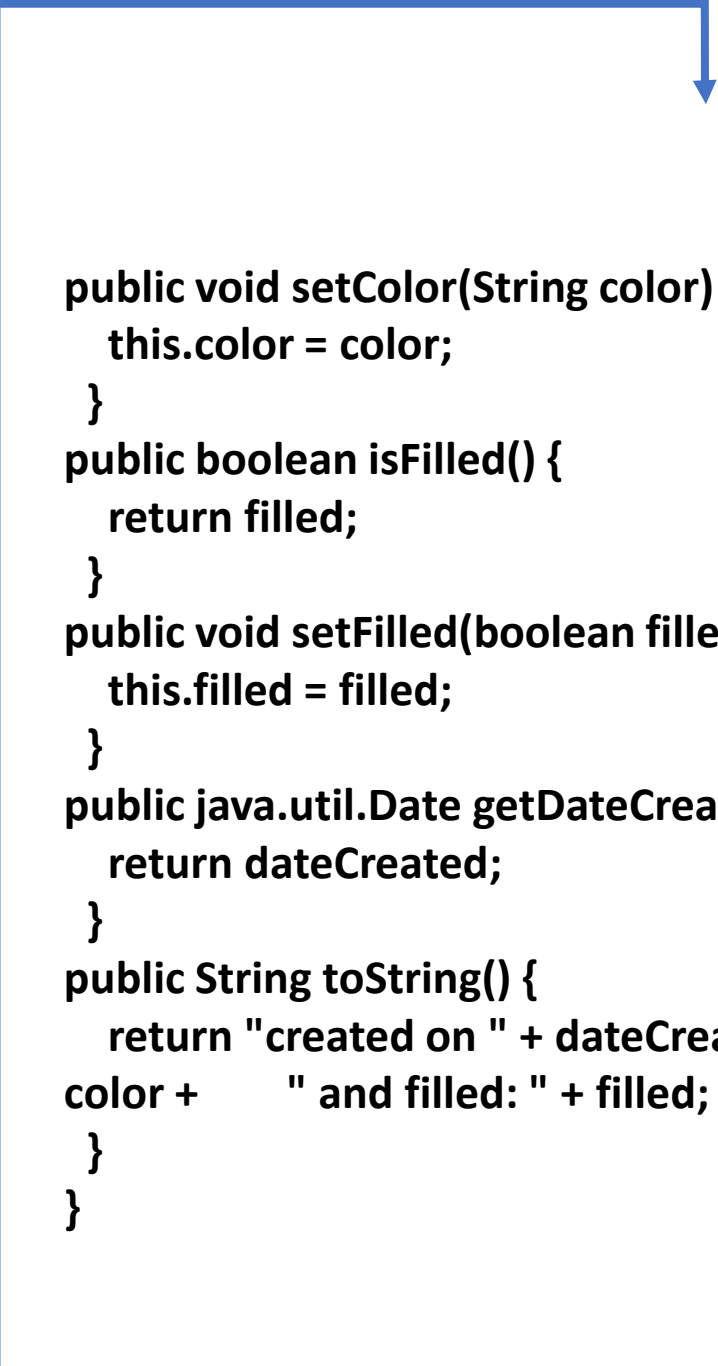
TestCircleRectangle

```
public class SimpleGeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    /** Construct a default geometric object */
    public SimpleGeometricObject() {
        dateCreated = new java.util.Date();
    }


    /** Construct a geometric object with the specified color
     * and filled value */
    public SimpleGeometricObject(String color, boolean filled) {
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }

    /** Return color */
    public String getColor() {
        return color;
    }
}
```



```
    public void setColor(String color) {
        this.color = color;
    }
    public boolean isFilled() {
        return filled;
    }
    public void setFilled(boolean filled) {
        this.filled = filled;
    }
    public java.util.Date getDateCreated() {
        return dateCreated;
    }
    public String toString() {
        return "created on " + dateCreated + "\n color: " +
            color + " and filled: " + filled;
    }
}
```

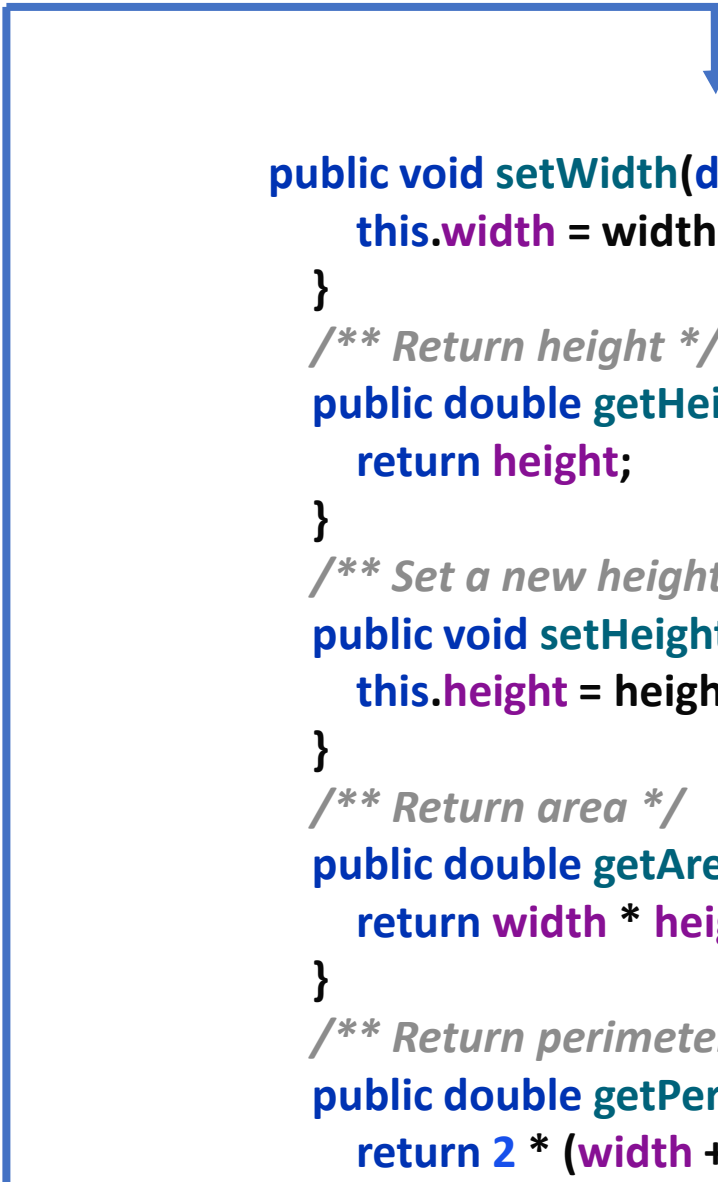
```
public class Circle extends SimpleGeometricObject {  
    private double radius;  
    public Circle() {  
    }  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
    public Circle(double radius, String color, boolean filled) {  
        this.radius = radius;  
        setColor(color);  
        setFilled(filled);  
    }  
    public double getRadius() {  
        return radius;  
    }  
    public void setRadius(double radius) {  
        this.radius = radius;  
    }  
}
```



```
    public double getArea() {  
        return radius * radius * Math.PI;  
    }  
  
    /** Return diameter */  
    public double getDiameter() {  
        return 2 * radius;  
    }  
  
    /** Return perimeter */  
    public double getPerimeter() {  
        return 2 * radius * Math.PI;  
    }  
  
    /** Print the circle info */  
    public void printCircle() {  
        System.out.println("The circle is created " +  
            getDateCreated() +  
                " and the radius is " + radius);  
    }  
}
```

```
public class Rectangle extends SimpleGeometricObject {
    private double width;
    private double height;

    public Rectangle() {
    }
    public Rectangle(
        double width, double height) {
        this.width = width;
        this.height = height;
    }
    public Rectangle(
        double width, double height, String color, boolean filled) {
        this.width = width;
        this.height = height;
        setColor(color);
        setFilled(filled);
    }
    public double getWidth() {
        return width;
    }
```



```
        public void setWidth(double width) {
            this.width = width;
        }
        /** Return height */
        public double getHeight() {
            return height;
        }
        /** Set a new height */
        public void setHeight(double height) {
            this.height = height;
        }
        /** Return area */
        public double getArea() {
            return width * height;
        }
        /** Return perimeter */
        public double getPerimeter() {
            return 2 * (width + height);
        }
    }
}
```



```
public class Main {  
    public static void main(String[] args) {  
        Circle circle = new Circle(1);  
        System.out.println("A circle " + circle.toString());  
        System.out.println("The color is " + circle.getColor());  
        System.out.println("The radius is " + circle.getRadius());  
        System.out.println("The area is " + circle.getArea());  
        System.out.println("The diameter is " + circle.getDiameter());  
  
        Rectangle rectangle = new Rectangle(2, 4);  
        System.out.println("\nA rectangle " + rectangle.toString());  
        System.out.println("The area is " + rectangle.getArea());  
        System.out.println("The perimeter is " + rectangle.getPerimeter());  
    }  
}
```

A circle created on Sat Nov 19 20:31:22 EET 2022

color: white and filled: false

The color is white

The radius is 1.0

The area is 3.141592653589793

The diameter is 2.0

A rectangle created on Sat Nov 19 20:31:22 EET 2022

color: white and filled: false

The area is 8.0

The perimeter is 12.0

Super Keyword in Java

- The `super` keyword in Java is a reference variable used to refer to the immediate parent class object. It plays a crucial role in inheritance and polymorphism, allowing subclasses to access and utilize the properties and methods of their parent classes.
- **Characteristics and Usage**
- **Accessing Parent Class Variables**
- When a subclass and its parent class have the same variable names, the `super` keyword can be used to differentiate and access the parent class's variable. For example:

```
class Vehicle {  
    int maxSpeed = 120;  
}
```

```
class Car extends Vehicle {  
    int maxSpeed = 180;
```

```
    void display() {  
        System.out.println("Maximum Speed: " + super.maxSpeed);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Car small = new Car();  
        small.display();  
    }  
}
```

Output:
Maximum Speed: 120

- **Invoking Parent Class Methods**
- The `super` keyword can also be used to call methods from the parent class, especially when the subclass has overridden those methods. For instance:

Output:

```
This is student class  
This is person class
```

Here, `super.message()` calls the `message()` method from the `Person` class

```
class Person {  
    void message() {  
        System.out.println("This is person class");  
    }  
}
```

```
class Student extends Person {  
    void message() {  
        System.out.println("This is student class");  
    }  
}
```

```
void display() {  
    message();  
    super.message();  
}
```

```
public class Test {  
    public static void main(String args[]) {  
        Student s = new Student();  
        s.display();  
    }  
}
```

- **Calling Parent Class Constructors**
- The `super` keyword can be used to invoke the parent class's constructor. This is particularly useful for initializing inherited properties. For example:

Output:

Person class Constructor
Student class Constructor

```
class Person {  
    Person() {  
        System.out.println("Person class Constructor");  
    }  
}  
  
class Student extends Person {  
    Student() {  
        super();  
        System.out.println("Student class Constructor");  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Student s = new Student();  
    }  
}
```

Are superclass's Constructor Inherited?

No. They are not inherited.

They are invoked explicitly or implicitly.

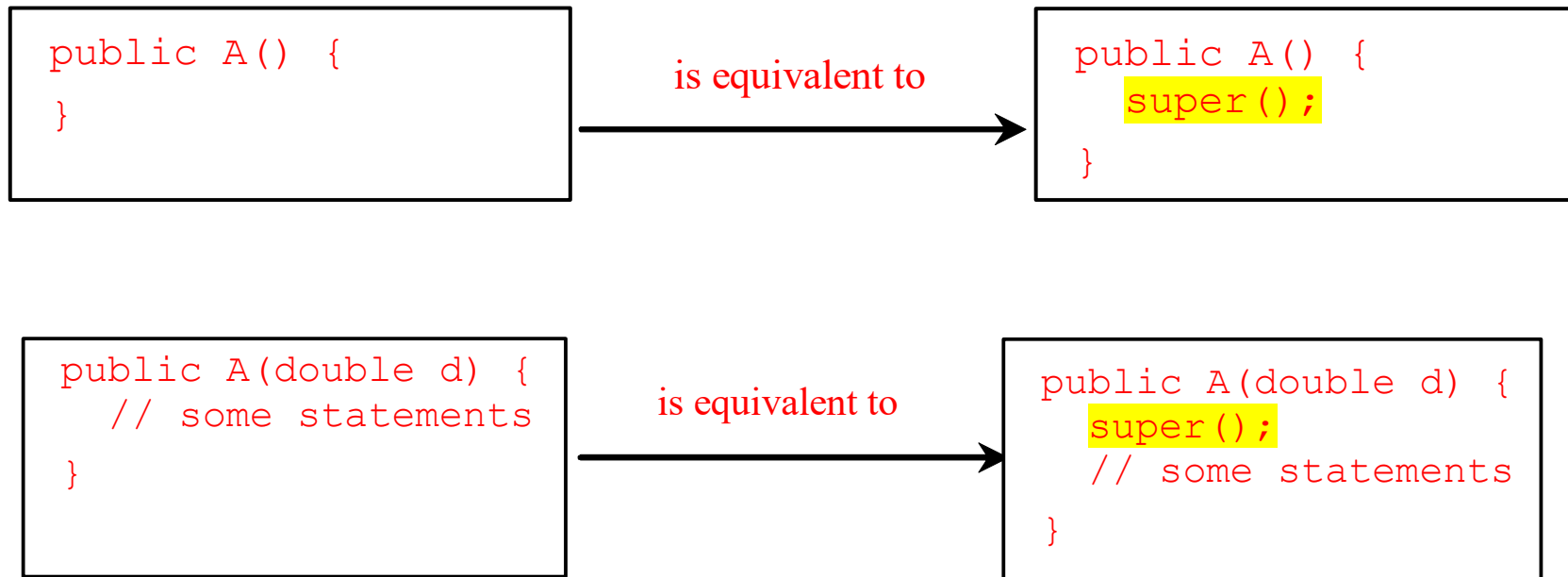
Explicitly using the **super** keyword.

A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass.

They can only be invoked from the subclasses' constructors, using the keyword super. *If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,



Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- ❑ To call a superclass constructor
- ❑ To call a superclass method

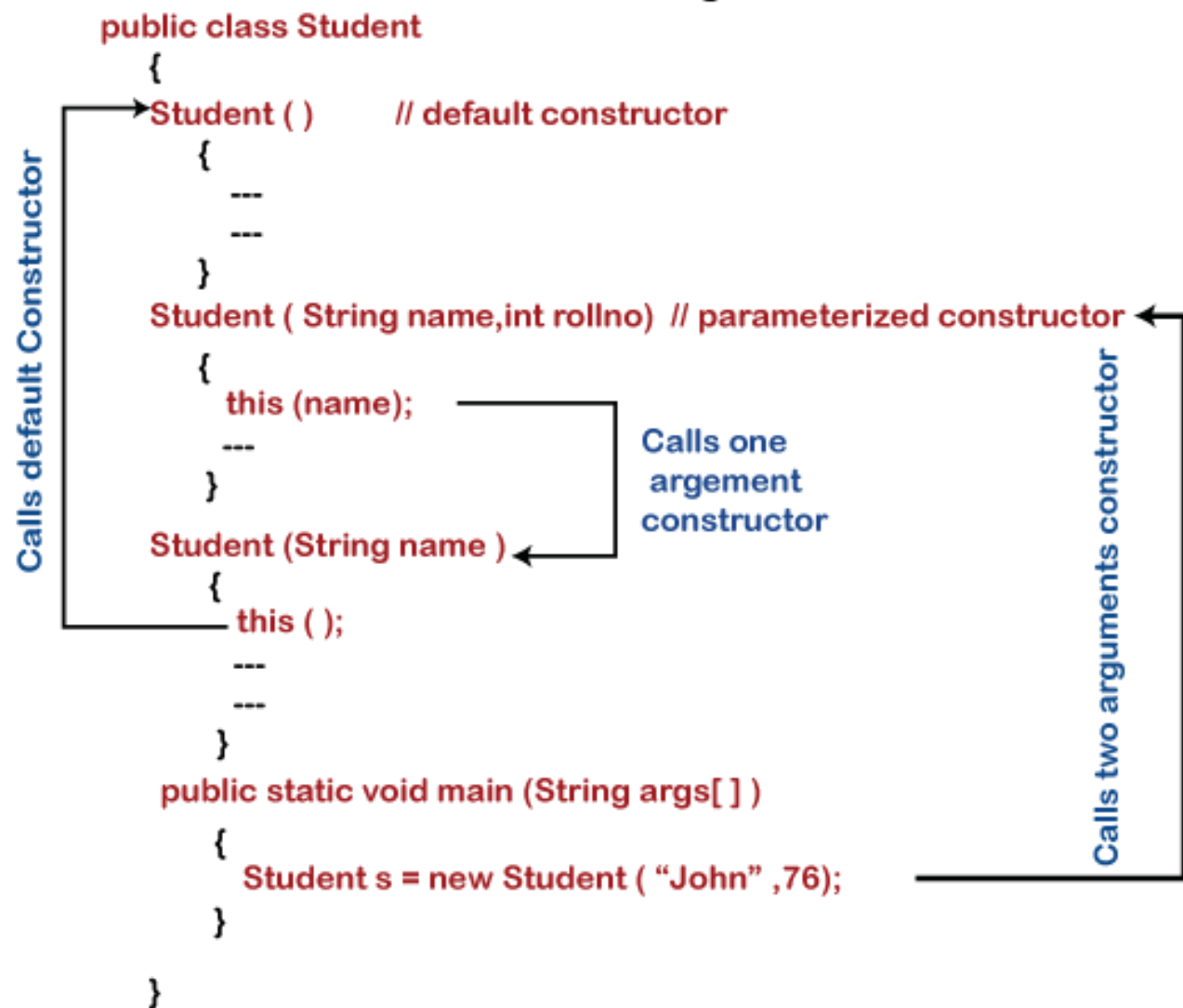
CAUTION

You must use the keyword super to call the superclass constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

Constructor Chaining

- In constructor chain, a constructor is called from another constructor in the same class this process is known as constructor chaining. It occurs through inheritance. When we create an instance of a derived class, all the constructors of the inherited class (base class) are first invoked, after that the constructor of the calling class (derived class) is invoked.
- We can achieve constructor chaining in two ways:
- Within the same class: If the constructors belong to the same class, we use this
- From the base class: If the constructor belongs to different classes (parent and child classes), we use the super keyword to call the constructor from the base class.
- Remember that changing the order of the constructor does not affect the output.

Constructor Chaining in Java



Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is known as *constructor chaining*.

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

1. Start from the
main method

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

2. Invoke Faculty
constructor

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
}
```

```
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}
```

```
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
}
```

```
    public Employee(String s) {  
        System.out.println(s);  
    }  
}
```

```
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

3. Invoke Employee's no-arg constructor

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

4. Invoke Employee(String)
constructor

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

5. Invoke Person() constructor

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

6. Execute println

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

7. Execute println

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```



8. Execute println

Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

9. Execute println

The Need of Constructor Chaining

- Suppose, there are five tasks to perform. There are two ways to perform these tasks, either implement all the tasks in a single constructor or create separate tasks in a single constructor.
- By using the constructor chaining mechanism, we can implement multiple tasks in a single constructor. So, whenever we face such types of problems, we should use constructor chaining. We can make the program more readable and understandable by using constructor chaining.

Rules of Constructor Chaining

- An expression that uses **this** keyword must be the first line of the constructor.
- **Order** does not matter in constructor chaining.
- There must exist at least one constructor that does not use **this**

Constructor Calling form another Constructor

- The calling of the constructor can be done in two ways:
- **By using this() keyword:** It is used when we want to call the current class constructor within the same class.
- **By using super() keyword:** It is used when we want to call the superclass constructor from the base class.
- Note: In the same constructor block, we cannot use this() and super() simultaneously.

Constructor Chaining Examples

Calling Current Class Constructor

- We use `this()` keyword if we want to call the current class constructor within the same class. The use of `this()` is mandatory because JVM never put it automatically like the `super()` keyword. Note that `this()` must be the first line of the constructor. There must exist at least one constructor without `this()` keyword.

- **Syntax:**

`this();` or `this(parameters list);`

- For example:

`this();`

`this("Javatpoint");`

-

In the above example, we have created an instance of the class without passing any parameter. It first calls the default constructor and the default constructor redirects the call to the parameterized one because of this(). The statements inside the parameterized constructor are executed and return back to the default constructor. After that, the rest of the statements in the default constructor is executed and the object is successfully initialized. The following is the calling sequence of the constructor:

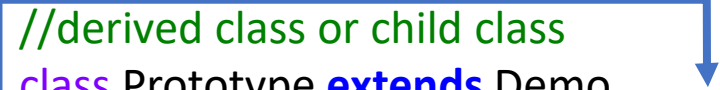
```
public class ConstructorChain
{
    //default constructor
    ConstructorChain()
    {
        this("Javatpoint");
        System.out.println("Default constructor called.");
    }
    //parameterized constructor
    ConstructorChain(String str)
    {
        System.out.println("Parameterized constructor called");
    }
    //main method
    public static void main(String args[])
    {
        //initializes the instance of example class
        ConstructorChain cc = new ConstructorChain();
    }
}
```

```
Parameterized constructor called
Default constructor called
```

Calling Super Class Constructor

- Sometimes, we need to call the superclass (parent class) constructor from the child class (derived class) in such cases, we use the `super()` keyword in the derived class constructor. It is optional to write `super()` because JVM automatically puts it. It should always write in the first line. We get a syntax error if we try to call a superclass constructor in the child class.
- **Syntax:**
`super();` or `super(Parameter List);`
- **`super();`**: It calls the no-argument or default constructor of the superclass.
- **`super(parameters);`**: It invokes the superclass parameterized constructor.
- Remember that the superclass constructor cannot be inherited in the subclass. It can be called from the subclass constructor by using the `super` keyword.

```
//parent class or base class
class Demo
{
//base class default constructor
Demo()
{
this(80, 90);
System.out.println("Base class default constructor called");
}
//base class parameterized constructor
Demo(int x, int y)
{
System.out.println("Base class parameterized constructor
called");
}
}
```



```
//derived class or child class
class Prototype extends Demo
{
//derived class default constructor
Prototype()
{
this("Java", "Python");
System.out.println("Derived class default constructor called");
}
//derived class parameterized constructor
Prototype(String str1, String str2)
{
super();
System.out.println("Derived class parameterized constructor
called");
} }
public class ConstructorChaining
{
//main method
public static void main(String args[])
{
//initializes the instance of example class
Prototype my_example = new Prototype();
} }
```

Defining a Subclass

A subclass inherits from a superclass. You can also:

- ☐ Add new properties
- ☐ Add new methods
- ☐ Override the methods of the superclass

Calling Superclass Methods

You could rewrite the printCircle() method in the Circle class as follows:

```
public void printCircle() {  
    System.out.println("The circle is created " +  
        super.getDateCreated() + " and the radius is " + radius);  
}
```

Overriding Methods in the Superclass

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

```
public class Circle extends GeometricObject {  
    // Other methods are omitted  
  
    /** Override the toString method defined in GeometricObject */  
    public String toString() {  
        return super.toString() + "\nradius is " + radius;  
    }  
}
```

Method Overloading

- Method overloading in java is a feature that allows a class to have more than one method with the same name, but with different parameters.
- Java supports method overloading through two mechanisms:
 1. By changing the number of parameters
 2. By changing the data type of parametersOverloading by changing the number of parameters A method can be overloaded by changing the number of parameters.
- Method overloading cannot be done by changing the return type of methods.

NOTE

An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

NOTE

Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

Overriding vs. Overloading

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

The Object Class and Its Methods

Every class in Java is descended from the `java.lang.Object` class. If no inheritance is specified when a class is defined, the superclass of the class is `Object`.

```
public class Circle {  
    ...  
}
```

Equivalent

```
public class Circle extends Object {  
    ...  
}
```

The toString() method in Object

The toString() method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object.

```
Loan loan = new Loan();  
System.out.println(loan.toString());
```

The code displays something like **Loan@15037e5** . This message is not very helpful or informative. Usually you should override the toString method so that it returns a digestible string representation of the object.

Thanks



References

- Java: How To Program, Early Objects, 11th edition
- Cay S. Horstmann, Big Java: Late Objects
- Introduction to Java Programming and Data Structures, Comprehensive Version 12th Edition, by Y. Liang (Author), Y. Daniel Liang
- Java documentation <https://docs.oracle.com/javase>
- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - TestSimpleCircle.java \(pearsoncmg.com\)](#)
- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - CircleWithStaticMembers.java \(pearsoncmg.com\)](#)
- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - TestCircleWithStaticMembers.java \(pearsoncmg.com\)](#)
- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - CircleWithPrivateDataFields.java \(pearsoncmg.com\)](#)
- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - TestCircleWithPrivateDataFields.java \(pearsoncmg.com\)](#)
- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - TestPassObject.java \(pearsoncmg.com\)](#)

References

- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - TestPassArray.java \(pearsoncmg.com\)](#)
- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - TestMax.java \(pearsoncmg.com\)](#)
- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - Weather.java \(pearsoncmg.com\)](#)
- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - TotalScore.java \(pearsoncmg.com\)](#)
- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - GradeExam.java \(pearsoncmg.com\)](#)
- [Introduction to Java Programming and Data Structures, 12E, Y. Daniel Liang - CheckSudokuSolution.java \(pearsoncmg.com\)](#)
- [Binary Search Animation by Y. Daniel Liang \(pearsoncmg.com\)](#)