Advanced Programming

Object-Oriented Programming

ThS. Trần Thị Thanh Nga
Khoa CNTT, Trường ĐH Nông Lâm TPHCM
Email: ngattt@hcmuaf.edu.vn

OOP

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

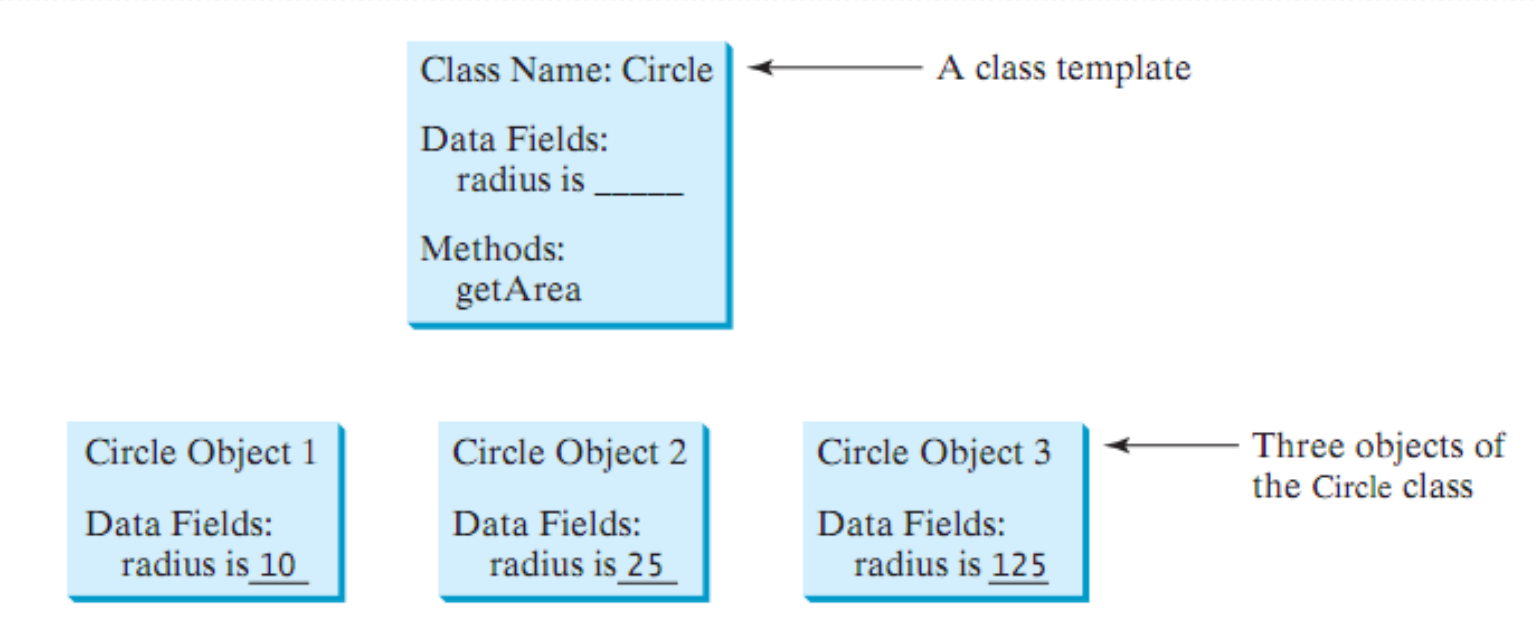
State and behavior

- The *state* of an object (also known as its **properties** or **attributes**) is represented by *data fields* with their current values.
  - A **circle object** has a data field *radius*, which is the property that characterizes a circle.
- The *behavior* of an object is defined by *methods*. To invoke a *method* on an object is to ask the object to perform an action.
  - Define a method named **getArea()** for circle objects. A circle object may invoke **getArea()** to return its area.

Class, Object, Instance

- A **class** is a template, blueprint, or contract that defines what an **object's** *data fields* and *methods* will be.
- An **object** is an **instance** of a class. You can create many instances of a class.

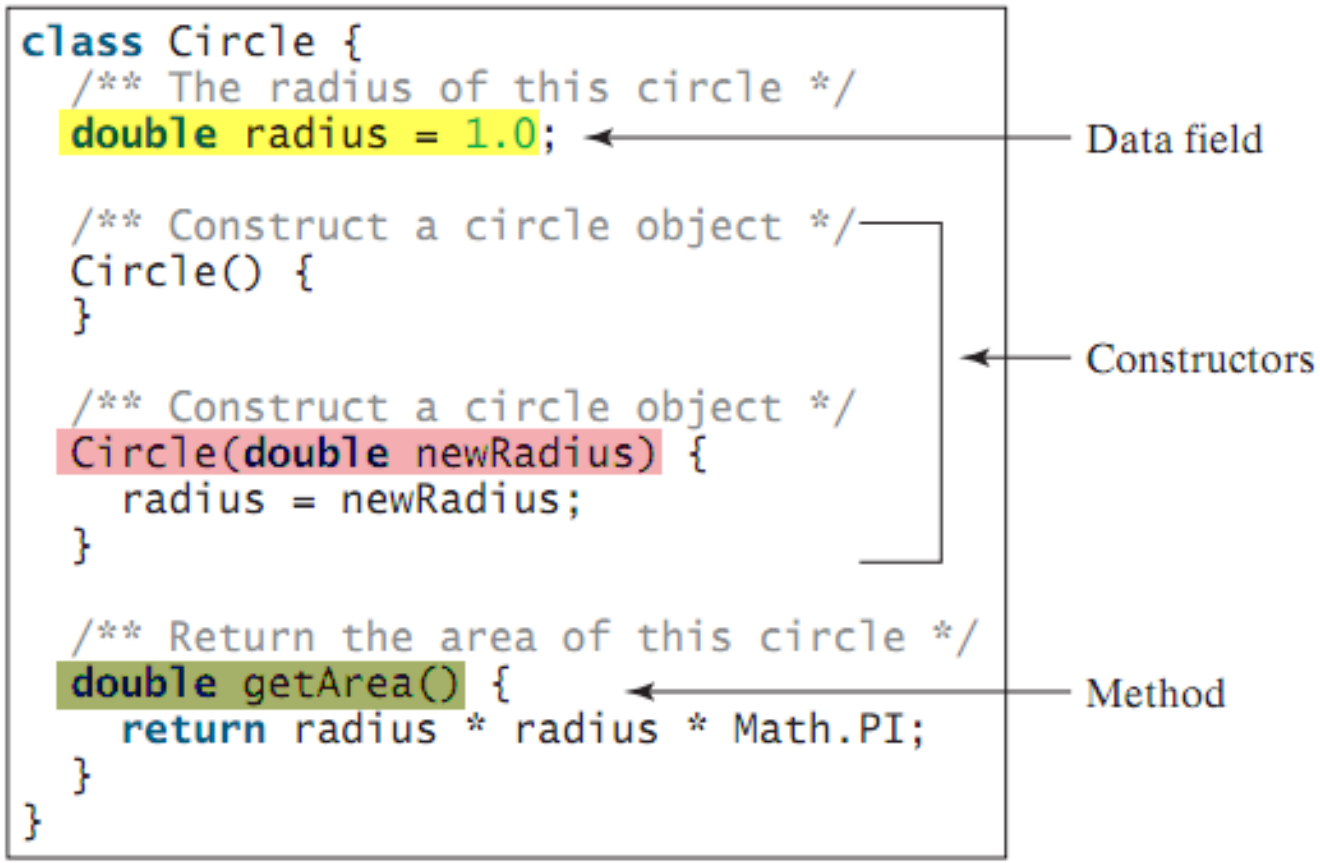
Class, Object, Instance



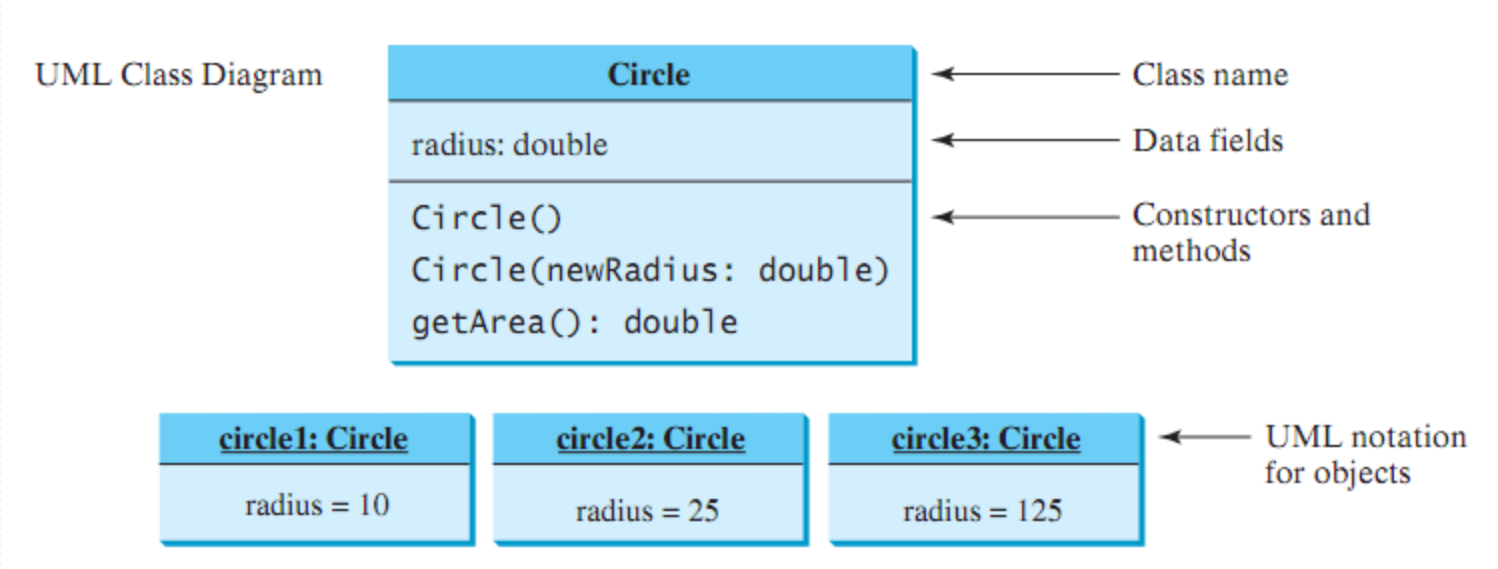
Constructor

- A Java class uses *variables* to define *data fields* and *methods* to define *actions*.
- A class provides *methods of a special type*, known as **constructors**, which are invoked to create a new object.
- A **constructor** can perform any action, but constructors are designed to perform *initializing actions*, such as initializing the data fields of objects.

Class example



Class diagram



Constructing Objects Using Constructors

- Constructors have three peculiarities:
  - A constructor 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.
- It is a common mistake to put the *void* keyword in front of a constructor.
  - public **void** Circle() {}

➔ **Circle()** is a *method*, not a *constructor*.

Accessing Object via Reference Variables

- Newly created objects are allocated in the memory. They can be accessed via reference variables.

Reference Variables and Reference Types

- Objects are accessed via object *reference variables*, which contain references to the object.
  - Syntax: **ClassName** *objectRefVar*;
- A class is a *reference type*, which means that a variable of the class type can reference an *instance of the class*.
  - **Circle** *myCircle*;
- Creates an object and assigns its reference to *myCircle*:
  - myCircle = new Circle();

Reference Variables and Reference Types

- You can write a single statement that combines the declaration of an object reference variable:

  ClassName objectRefVar = new ClassName();

  - the *creation* of an object: **Circle** *myCircle*;
  - the *assigning* of an object reference to the variable:

    myCircle = new Circle();

Accessing an Object's Data and Methods

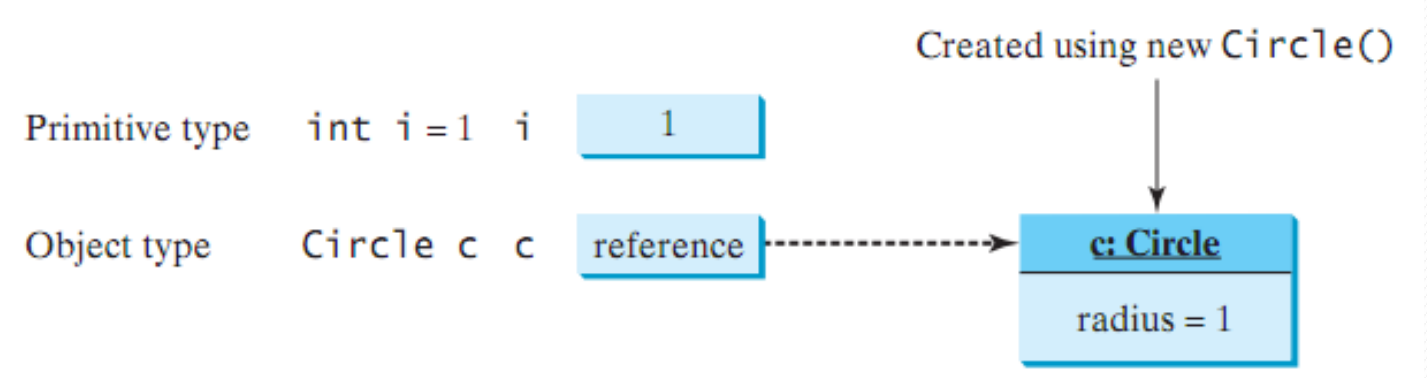
- After an object is created, its data can be accessed and its methods invoked using the dot operator (.), also known as the *object member access* operator.
  - **objectRefVar.dataField** references a data field in the object.
  - **objectRefVar.method(arguments)** invokes a method on the object.

Reference Data Fields and the null Value

- The data fields can be of **reference types**.
  - Student class contains a data field *name* of the **String** type.
  - **String** is a predefined Java class.
- If a data field of a reference type does not reference any object, the data field holds a special Java value, **null**.
  - **null** is a literal just like *true* and *false*. While true and false are Boolean literals, null is a literal for a *reference type*.

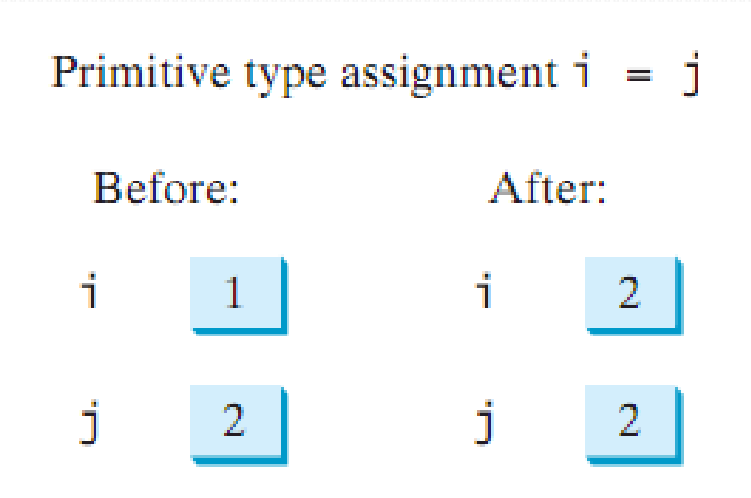
Differences Between Variables of Primitive Types and Reference Types

- When you declare a variable, you are telling the compiler what *type* of value the variable can hold.
  - For a variable of a primitive type, the value is of the primitive type.
  - For a variable of a reference type, the value is a *reference* to where an object is located.



Differences Between Variables of Primitive Types and Reference Types

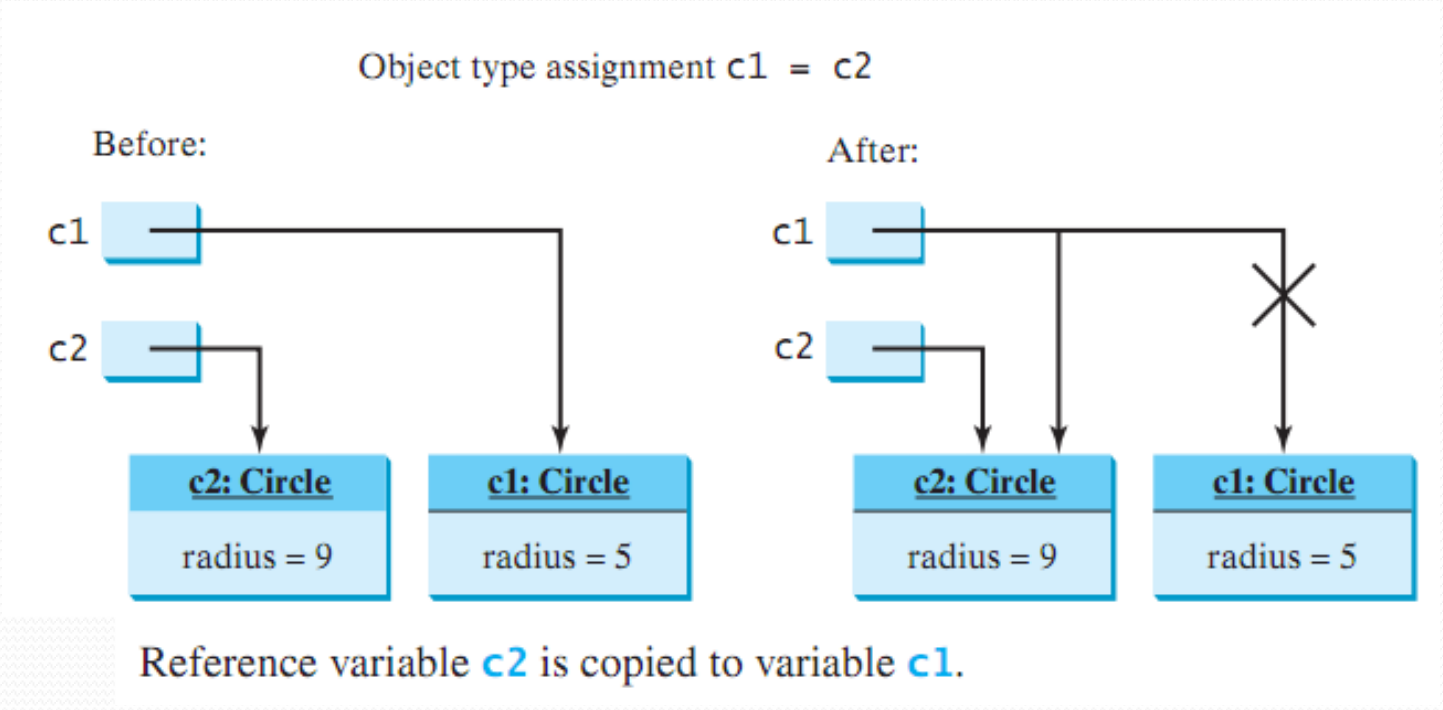
- When you assign one variable to another, the other variable is set to the same value.
  - For a variable of a primitive type, the real value of one variable is assigned to the other variable.



Primitive variable **j** is copied to variable **i**

Differences Between Variables of Primitive Types and Reference Types

- For a variable of a reference type, the reference of one variable is assigned to the other variable.



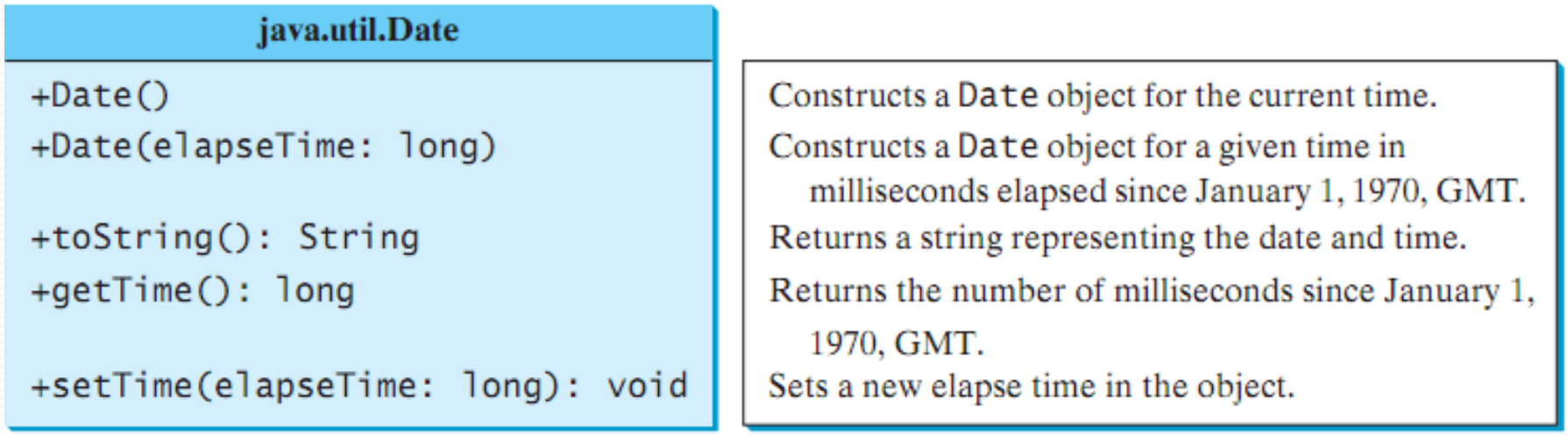
Reference variable `c2` is copied to variable `c1`.

Using Classes from the Java Library

- **Date** class
- **Random** class

The Date class

- Java provides a system-independent encapsulation of date and time in java.util.Date class



The Date class

- Java provides a system-independent encapsulation of date and time in java.util.Date class

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

The Random class

- You have used **Math.random()** to obtain a random double value between 0.0 and 1.0.
- Another way to generate random numbers is to use the **java.util.Random** class, which can generate a random **int**, **long**, **double**, **float**, and **boolean** value.

java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (exclusive).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (exclusive).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (exclusive).
+nextBoolean(): boolean	Returns a random boolean value.

The Random class

- When you create a **Random** object, you have to specify a *seed* or use the *default seed*.
- The no-arg constructor creates a **Random** object using the current elapsed time as its seed.
- If two **Random** objects have the same seed, they will generate identical sequences of numbers.

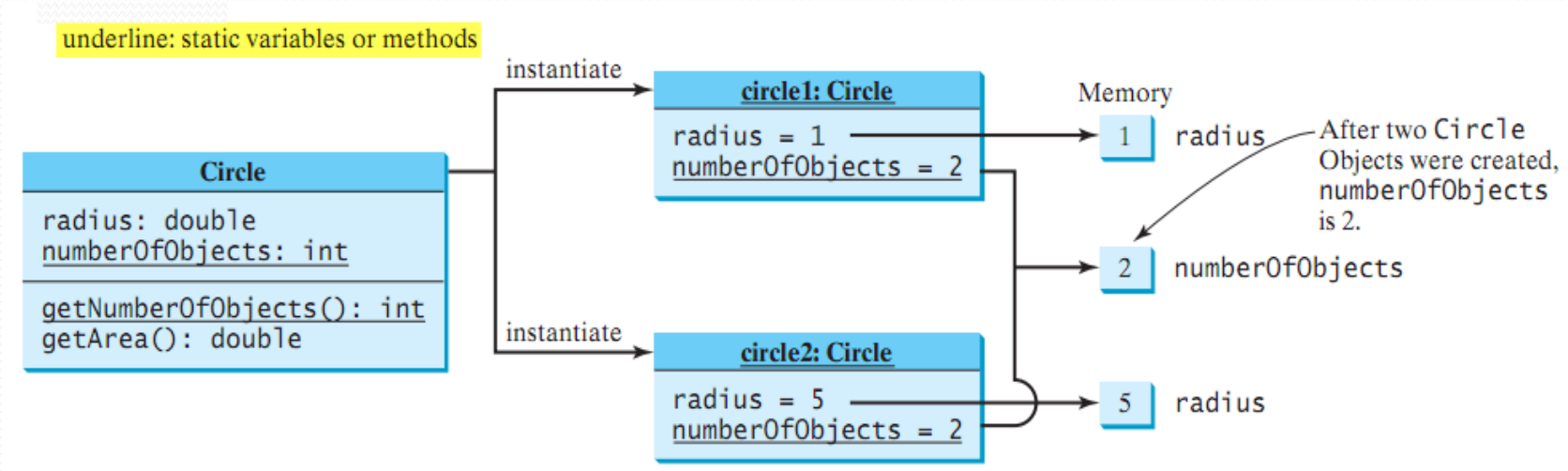
The Random class

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

The code generates the same sequence of random `int` values:

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

Static Variables, Constants, and Methods



Static Variables, Constants, and Methods

- The data field **radius** in the Circle class is known as an *instance variable*. An *instance variable* is tied to a specific instance of the class; it is **not shared** among objects of the same class.

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
```

  - The **radius** in **circle1** is *independent* of the **radius** in **circle2** and is stored in a different memory location.
  - Changes made to **circle1**'s radius do not affect **circle2**'s radius, and vice versa.

Static Variables, Constants, and Methods

- If you want all the instances of a class to *share data*, use *static variables*, also known as *class variables*.
- *Static variables* store values for the variables in a common memory location.
- Because of this common location, if one object changes the value of a static variable, *all objects of the same class are affected*.
- Static methods can be called without creating an instance of the class

```
static int numberOfObjects;

static int getNumberObjects() {
  return  numberOfObjects;
}
```

Example

```
public class Circle {
        double radius;
        static int numberOfObjects = 0; //Number of objects created
         public Circle() {
                radius = 1.0;
                numberOfObjects++;
        }
         public Circle(double newRadius) {
                numberOfObjects++;
        }
        static int getNumberOfObjects() {
                return numberOfObjects;
        }
        public double getArea() {
                return radius * radius * Math.PI;
        }
}
```

Visibility Modifiers

- You can use the **public** visibility modifier for classes, methods, and data fields to denote that they can be accessed from any other classes.
- If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as *package-private* or *package-access*.

Visibility Modifiers

- The **private** modifier makes methods and data fields accessible only from within its own class.

```
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

- If a class is not defined **public**, it can be accessed only within the same package

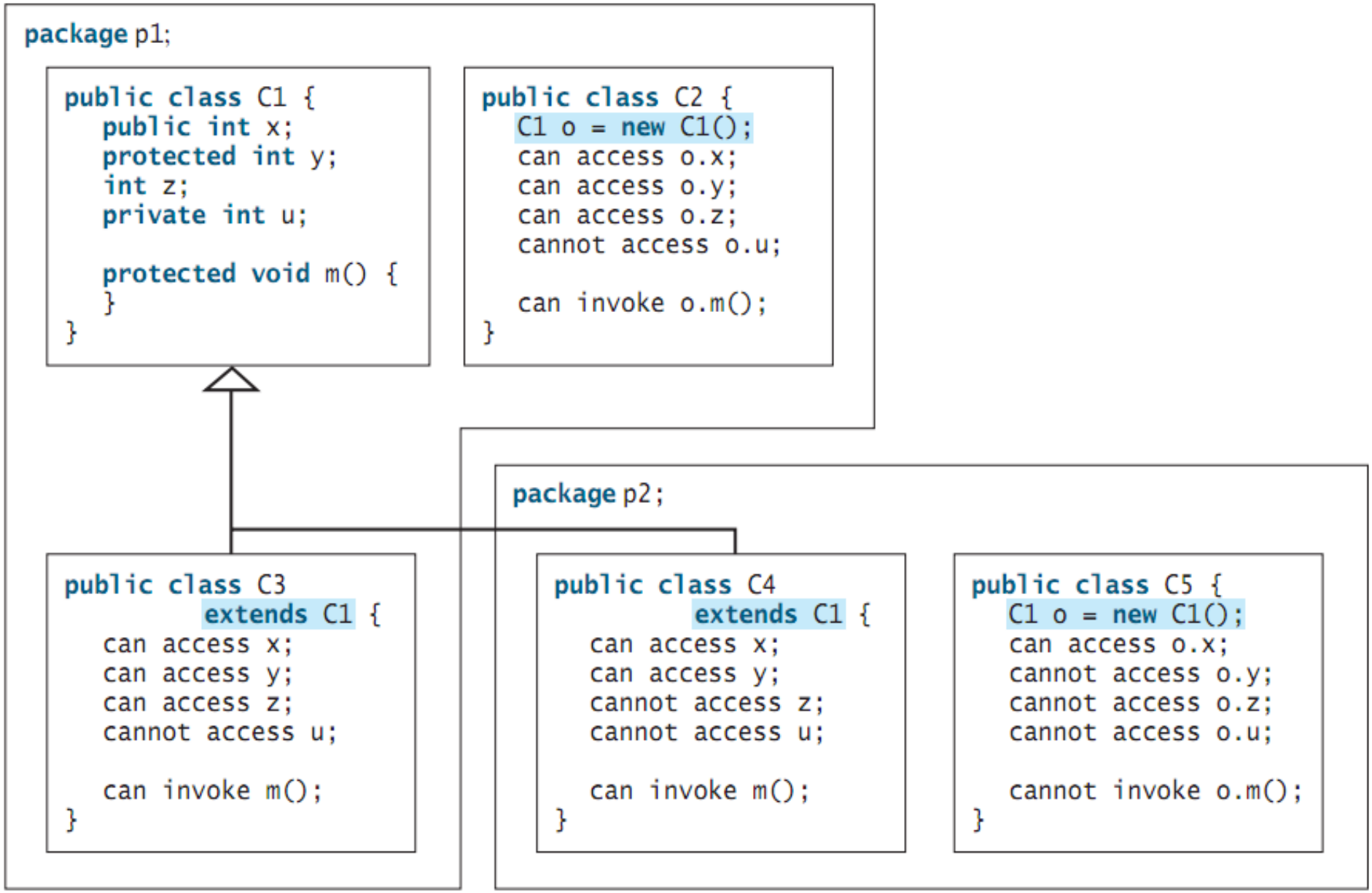
```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

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

Visibility Modifiers

- To allow subclasses to access data fields or methods defined in the superclass, but *not allow nonsubclasses* to access these data fields and methods ➔ use the **protected** keyword.
- A **protected** *data field* or *method* in a superclass can be accessed in its subclasses.


package p1;
public class C1 {
public int x;
protected int y;
int z;
private int u;
protected void m() {
}
}
public class C2 {
C1 o = new C1();
can access o.x;
can access o.y;
can access o.z;
cannot access o.u;
can invoke o.m();
}
package p2;
public class C3
extends C1 {
can access x;
can access y;
can access z;
cannot access u;
can invoke m();
}
public class C4
extends C1 {
can access x;
can access y;
cannot access z;
cannot access u;
can invoke m();
}
public class C5 {
C1 o = new C1();
can access o.x;
cannot access o.y;
cannot access o.z;
cannot access o.u;
cannot invoke o.m();
}

Datafield Encapsulation

- The data fields **radius** and **numberOfObjects** in the **Circle2** class can modified directly
  - myCircle.radius = 5
  - Circle.numberOfObjects = 1
- This is not a good practice—for two reasons:
  - First, data may be tampered with.
    - For example, **numberOfObjects** is to count the number of objects created, but it may be mistakenly set to an arbitrary value (e.g., **Circle.numberOfObjects** = 10).
  - Second, the class becomes difficult to maintain and vulnerable to bugs.
    - Suppose you want to modify the **Circle** class to ensure that the **radius** is nonnegative after other programs have already used the class. You have to change not only the **Circle** class but also the programs that use it, because the clients may have modified the radius directly (e.g., myCircle.radius = -5).

Datafield Encapsulation

- To prevent direct modifications of data fields, you should declare the data fields **private**, using the **private** modifier. This is known as *data field encapsulation*.
- A **private** data field cannot be accessed by an object from outside the class. But often a client needs to retrieve and modify a data field.
  - To make a private data field accessible, provide a **get** method to return its value:

    public *returnType* **getPropertyName**()
  - To enable a private data field to be updated, provide a **set** method to set a new value.

    public *void* **setPropertyName**(*dataType* propertyValue)

Passing Objects to Methods

- You can pass **objects** to methods. Passing an object is actually passing the **reference** of the object.

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

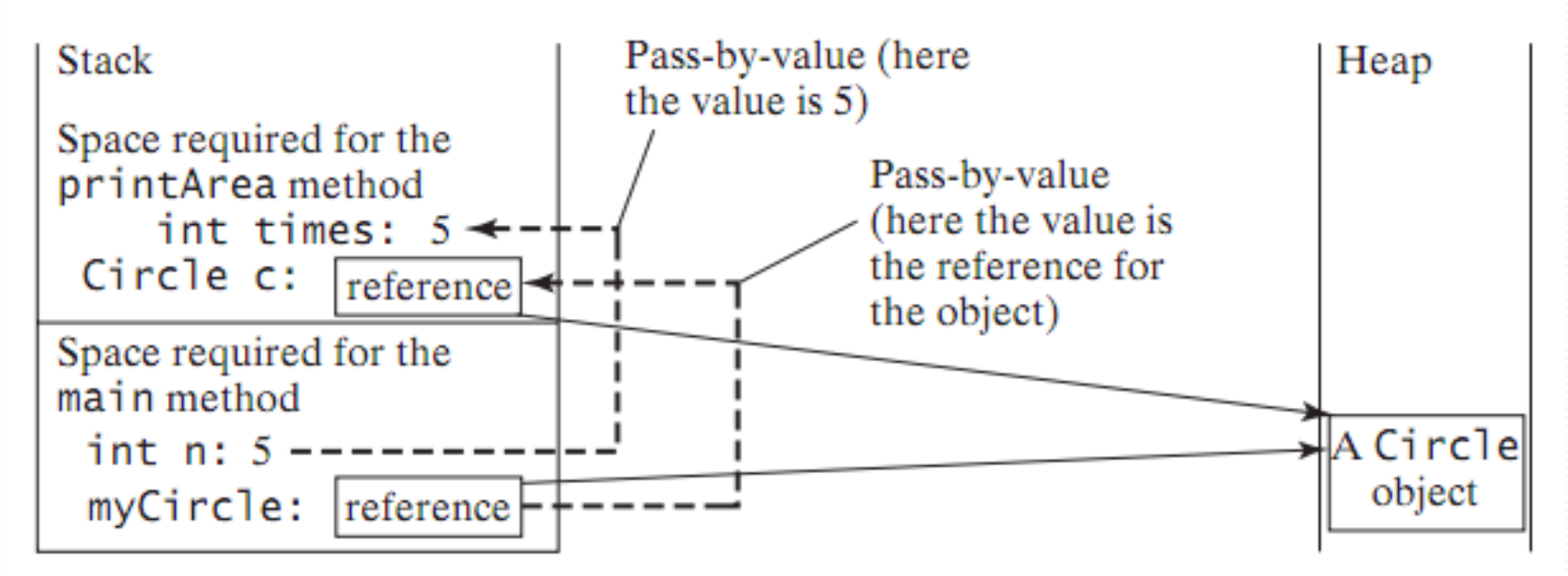
Example

```
public class Circle {
        double radius;
        static int numberOfObjects = 0; //Number of objects created
         public Circle() {
                radius = 1.0;
                numberOfObjects++;
        }
         public Circle(double newRadius) {
                numberOfObjects++;
        }
        static int getNumberOfObjects() {
                return numberOfObjects;
        }
        public double getArea() {
                return radius * radius * Math.PI;
        }
}
```

Passing Objects to Methods

- When passing an argument of a reference type, the reference of the object is passed.
  - **c** contains a reference for the object that is also referenced via **myCircle**.
  - Changing the properties of the object through **c** inside the **printAreas** method has the same effect as doing so outside the method through the variable **myCircle**.

Passing Objects to Methods



Array of objects

- You can create arrays of objects.

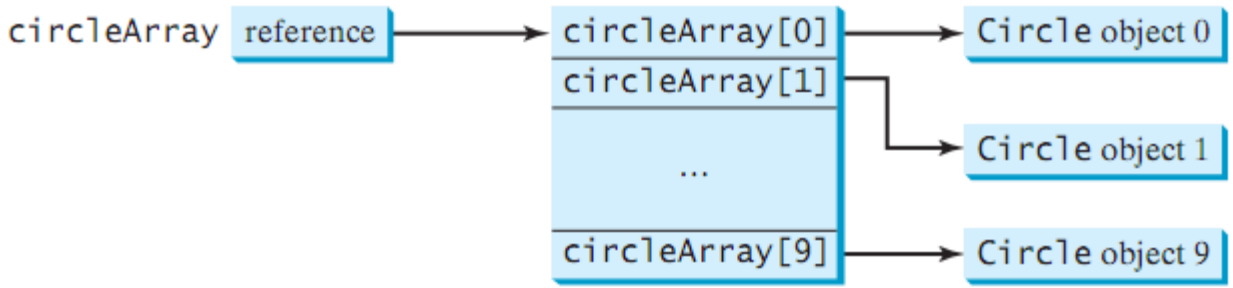
```
Circle[] circleArray = new Circle[10];
```

- To initialize the circleArray, you can use a for loop like this one:

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

Array of objects

- An array of objects is actually an array of reference variables. Invoking circleArray[1].getArea() involves two levels of referencing.
  - circleArray references the entire array.
  - circleArray[1] references a Circle object.



In an array of objects, an element of the array contains a reference to an object.

Example

```
public class TotalArea {
   public static void main(String[] args) {
        Circle[] circleArray;
        circleArray = createCircleArray();
        printCircleArray(circleArray);
        createCircleArray();
   }

   /** Create an array of Circle objects */
   public static Circle[] createCircleArray() {
        Circle[] circleArray = new Circle[5];
        for (int i = 0; i < circleArray.length; i++) {
                ircleArray[i] = new Circle(Math.random() * 100);
        }
        return circleArray;
   }
```

```
    public static void printCircleArray(Circle[] circleArray) {
        System.out.printf("%-30s%-15s\n", "Radius", "Area");
        for (int i = 0; i < circleArray.length; i++) {
                System.out.printf("%-30f%-15f\n",
        circleArray[i].getRadius(), circleArray[i].getArea());
        }
        System.out.println("————————————————————————");
        // Compute and display the result
        System.out.printf("%-30s%-15f\n", "The total area of
circles is",sum(circleArray));
    }
    public static double sum(Circle[] circleArray) {
        double sum = 0; // Initialize sum
        for (int i = 0; i < circleArray.length; i++)
        sum += circleArray[i].getArea();
        return sum;
    }
}
```

Immutable Objects and Classes

- Normally, you create an object and allow its contents to be changed later.
- Occasionally it is desirable to create an object whose contents *cannot be changed*, once the object is created. We call such an object an **immutable object** and its class an **immutable class**.
  - The String class is **immutable**.
  - If you deleted the **set** method in the **Circle** class, the class would be immutable, because radius is *private* and *cannot be changed* without a **set** method.

Immutable Objects and Classes

- If a class is **immutable**, then all its data fields must be *private* and it *cannot contain public* **set** methods for any data fields.
- A class with all private data fields and no mutators is not necssarily immutable.
  - For example, the following Student class has all private data fields and *no* **set** methods, but it is *not an immutable class*.

```
public class Student {
        private int id;
        private String name;
        private java.util.Date dateCreated;
        public Student(int ssn, String newName) {
                id = ssn;
                name = newName;
                dateCreated = new java.util.Date();
        }
        public int getId() {
                return id;
        }
        public String getName() {
                return name;
        }
        public java.util.Date getDateCreated() {
                return dateCreated;
        }
}
```

Immutable Objects and Classes

```
public class TestStudent {
   public static void main(String[] args) {
       Student student = new Student(111223333, "John");
       java.util.Date dateCreated = student.getDateCreated();
       dateCreated.setTime(200000);
       // Now dateCreated field is changed!
  }
}
```

Immutable Objects and Classes

- For a class to be immutable, it must meet the following requirements:
  - all data fields private;
  - no mutator methods;
  - no accessor method that returns a reference to a data field that is mutable.

The scope of Variables

- Instance and static variables in a class are referred to as the *class's variables* or *data fields*.
- A variable defined inside a method is referred to as a *local variable*.
- The scope of a class's variables is the entire class, regardless of where the variables are declared. A class's variables and methods can appear in any order in the class.

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

  private double radius = 1;
}
```

The scope of Variables

- The exception is when a data field is initialized based on a reference to another data field ➔ the other data field must be declared first.

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

The scope of Variables

- You can declare a *class's variable* only once, but you can declare the same variable name in a method many times in different nonnesting blocks.
  - If a local variable has the same name as a class's variable, the local variable takes precedence , the class's variable with the same name *is hidden*.

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

  public Foo() {
  }

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

The this Reference

- The **this** keyword is the name of a reference that refers to a *calling object* itself.

The this Reference

- The line **this.i** = **i** means "assign the value of parameter i to the data field i of the **calling object**."
- The keyword **this** refers to the object that invokes the instance

```
public class Foo {
  int i = 5;
  static double k = 0;

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

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

```
Suppose that f1 and f2 are two objects of Foo.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2
```

The this Reference

```
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

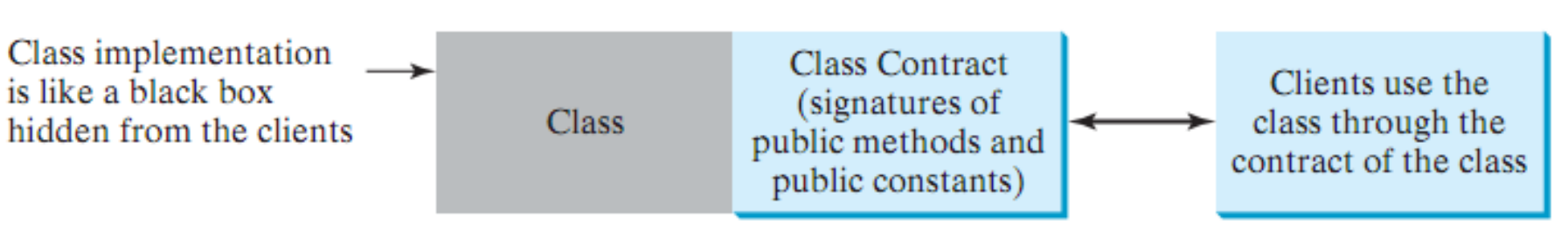
Every instance variable belongs to an instance represented by this, which is normally omitted

Class Abstraction and Encapsulation

- **Class abstraction** is the separation of class *implementation* from the *use* of a class.
  - The creator of a class describes it and lets the user know *how it can be used*.
- The collection of methods and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the **class's contract**.

Class Abstraction and Encapsulation

- *The user of the class* does not need to know *how the class is implemented*. The details of implementation are encapsulated and hidden from the user.
  - This is known as **class encapsulation**.



Example: Building a computer system

- Your personal computer has many components—a CPU, memory, disk, motherboard, fan,…
- Each component can be viewed as an object that has properties and methods.
- To get the components to work together, you need know only how each component is used and how it interacts with the others. You don't need to know how the components work internally.
- The internal implementation is encapsulated and hidden from you. You can build a computer without knowing how a component is implemented.

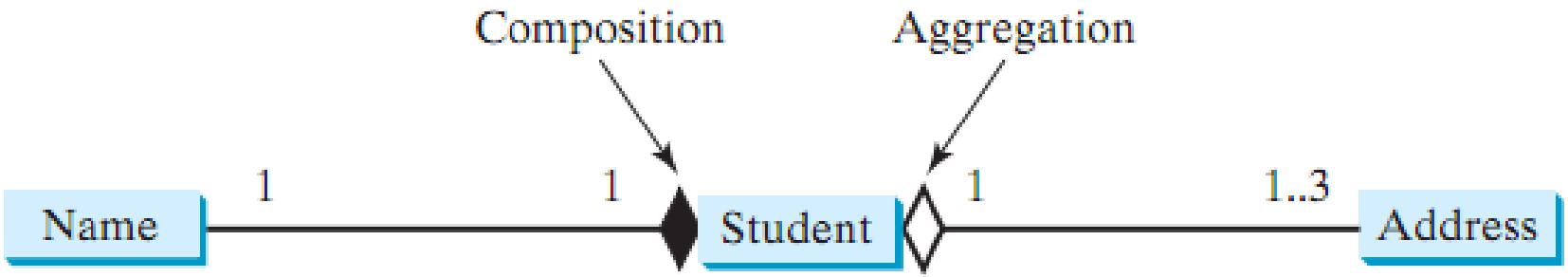
Object Composition

- An object can *contain* another object. The relationship between the two is called **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**.

Object Composition

- If an object is exclusively owned by an aggregating object, the relationship between them is referred to as **composition**.
  - "*a student has a name*" is a **composition** relationship between the **Student** class and the **Name** class.
  - "*a student has an address*" is an **aggregation** relationship between the **Student** class and the **Address** class, since an address may be shared by several students.

Object Composition



A student has a name and an address.

Object Composition

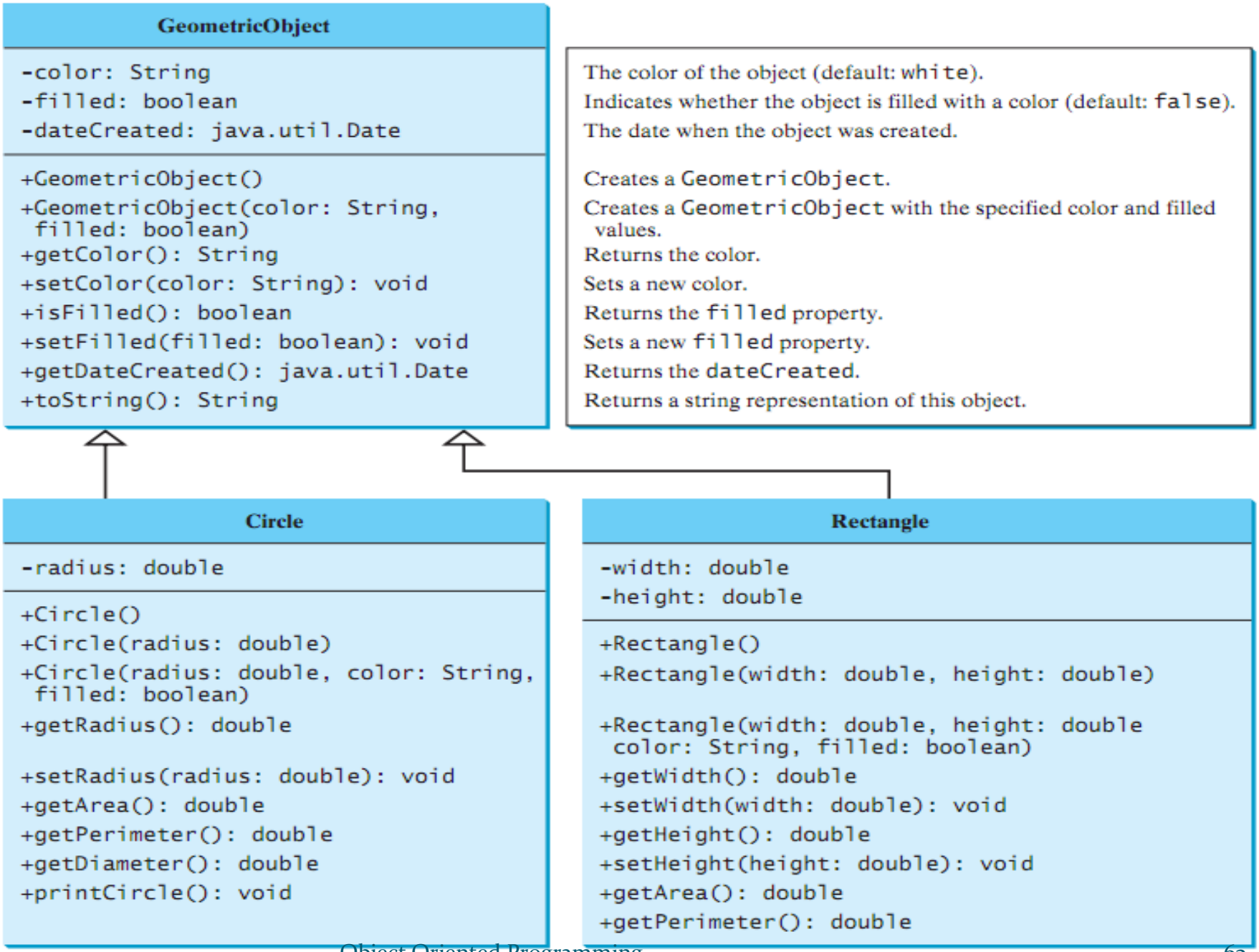
- Each class involved in a relationship may specify a *multiplicity*. A *multiplicity* could be a number or an interval that specifies how many objects of the class are involved in the relationship.
- The character * means an unlimited number of objects, and the interval m..n means that the number of objects should be between m and n, inclusive.
- In above example:
  - Each student has only one address, and each address may be shared by up to 3 students.
  - Each student has one name, and a name is unique for each student.

Inheritance

- Object-oriented programming allows you to **derive** new classes from existing classes. This is called **inheritance**.
- **Inheritance** is an important and powerful feature in Java for reusing software.
- Example:
  - Suppose you are to 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 and make the system easy to comprehend and easy to maintain?

Superclass and subclass

- A class C1 extended from another class C2 is called a **subclass**, and C2 is called a **superclass**.
  - A **superclass** is also referred to as a *parent* class, or a **base** class.
- A **subclass** as a *child* class, an *extended* class, or a **derived** class.
  - A **subclass** inherits accessible *data fields* and *methods* from its **superclass** and may also *add new data fields and methods*.


GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

Calling Superclass Constructors

- Syntax:
  - **super()** invokes the *no-arg constructor* of its superclass
  - **super(arguments)** invokes the superclass constructor that matches the arguments.
- The statement **super()** or **super(arguments)** must appear in the *first line* of the subclass constructor.

```
public Circle(double radius, String color, boolean filled) {
        super(color, filled);
        this.radius = radius;
}
```

Overriding Methods

- A **subclass** inherits methods from a **superclass**.
  - The **subclass** modify the implementation of a method defined in the **superclass**.
  - This is referred to a **method overriding**.

Overriding Methods

- An **instance method** can be overridden only if it is *accessible*.
  - a **private method** cannot be overridden, because it is not accessible outside its own class.
- A **static method** can be inherited. 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.
  - The hidden **static methods** can be invoked using the syntax **SuperClassName.staticMethodName**.

Overriding vs. Overloading

- Overloading means to define multiple methods with the *same name* but *different signatures*.
  - The method is already defined in the superclass.
  - The method must be defined in the subclass using the same signature and the same return type.

Override example

```
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);
  }
}
```

Overload example

```
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 toString() Method

- 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** by default.

```
public class ClassName {
  ...
}
```

Equivalent

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

- You can use the methods provided by the Object class in your classes.

The Object Class and Its toString() Method

- Invoking **toString**() on an object returns a string that describes the object.
  - It returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal.
  - Example:

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

  - Result: Loan@15037e5.

The Object Class and Its toString() Method

- You should override the **toString** method so that it returns a descriptive string representation of the object.

```
public String toString() {
      return "created on " + dateCreated + "\ncolor: "
                          +color  + " and filled: " + filled;
}
```

Polymorphism

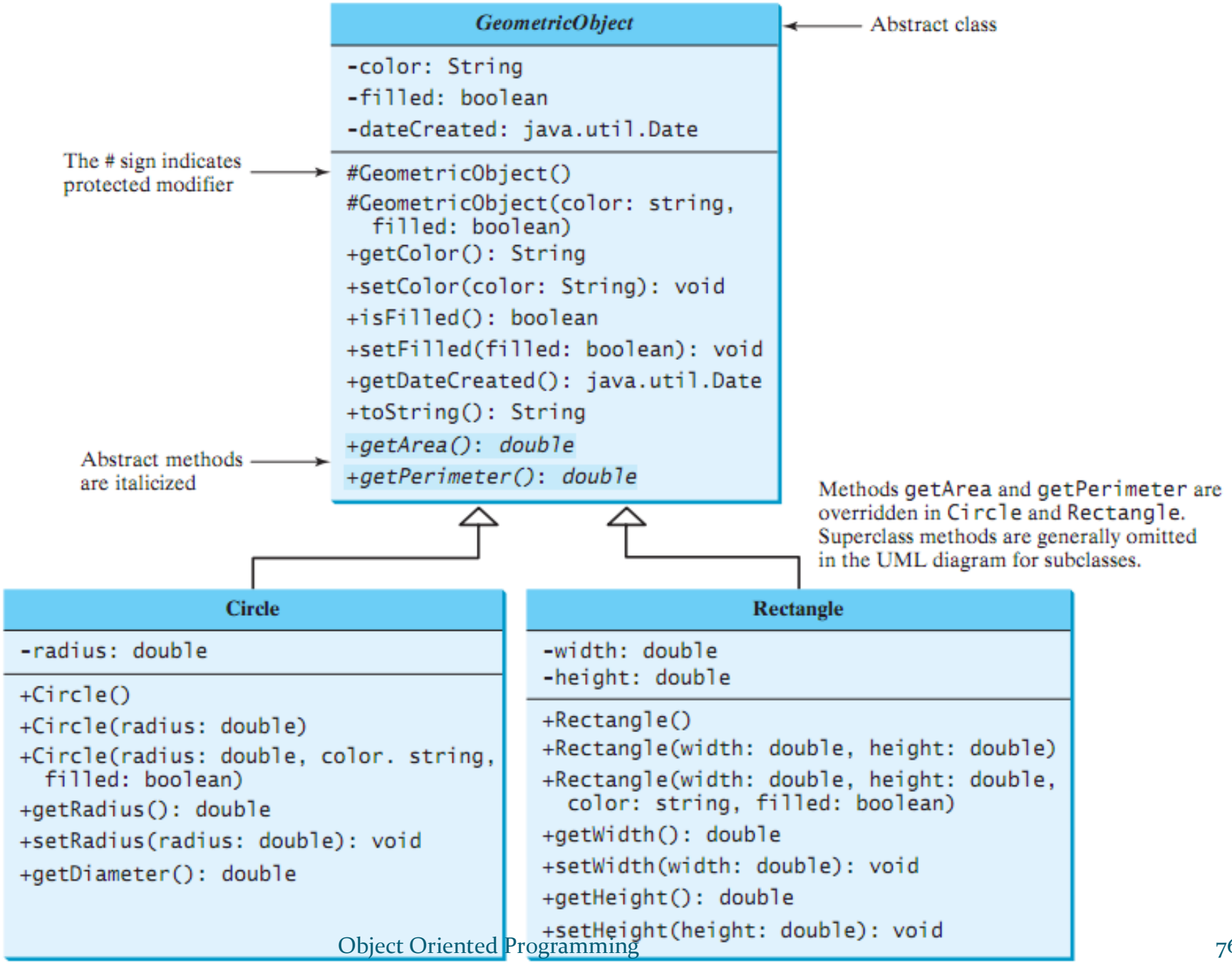
- Define two useful terms: subtype and supertype.
  - A class defines a type.
  - A type defined by a subclass is called a subtype.
  - A type defined by its superclass is called a supertype.
- Example: Circle is a subtype of GeometricObject and GeometricObject is a supertype for Circle.

Polymorphism

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        // Display circle and rectangle properties
        displayObject(new Circle(1, "red", false));
        displayObject(new Rectangle(1, 1, "black", true));
    }
    /** Display geometric object properties */
    public static void displayObject(GeometricObject object) {
        System.out.println("Created on " +
                object.getDateCreated() + ". Color is " +
                object.getColor());
    }
}
```

Abstract Classes

- Class design should ensure that a superclass contains *common features* of its subclasses.
  - Sometimes a superclass is so abstract that it cannot have any specific instances.
- Such a class is referred to as an **abstract class**.


GeometricObject
Abstract class
-color: String
-filled: boolean
-dateCreated: java.util.Date
The # sign indicates protected modifier
#GeometricObject()
#GeometricObject(color: string, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
Abstract methods are italicized
+getArea(): double
+getPerimeter(): double
Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color. string, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double, color: string, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void

Abstract classes

- **Abstract classes** are like regular classes, but you **cannot** create instances of abstract classes using the **new** operator.
- An **abstract method** is defined without implementation. Its implementation is provided by the subclasses.
- A class that contains **abstract methods** must be defined **abstract**.

Abstract classes

- The **constructor** in the abstract class is defined **protected**, because it is used only by subclasses.
- When you create an instance of a concrete subclass, *its superclass's constructor is invoked* to initialize data fields defined in the superclass.

Interesting Points on Abstract Classes

- In a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented.
  - Abstract methods are nonstatic.
- An *abstract class* cannot be instantiated using the **new** operator, but you can still define its constructors, which are invoked in the constructors of its subclasses.
- A class that **contains** abstract methods must be abstract.
  - It is possible to define an abstract class that contains **no abstract methods.**
  - This class is used as a base class for defining a new subclass

Interesting Points on Abstract Classes

- You **cannot** create an instance from an abstract class using the **new** operator, but an abstract class can be used as a **data type**.

```
GeometricObject[] objects = new GeometricObject[10];
```

- You can create an instance of GeometricObject and assign its reference to the array like this:

```
objects[0] = new Circle();
```

Interfaces

- An interface is a classlike construct that contains only **constants** and **abstract methods**.
- In many ways an **interface** is similar to an **abstract class**, but its intent is to specify common **behavior** for objects.
  - For example, using appropriate interfaces, you can specify that the objects are *comparable*, *edible*, and/or *cloneable*.

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

Interfaces

- As with an **abstract class**, you cannot create an instance from an **interface** using the **new** operator, but in most cases you can use an interface more or less the same way you use an abstract class.
  - For example, you can use an interface as a data type for a reference variable.

Interfaces

- Since all data fields are **public final static** and all methods are **public abstract** in an interface, Java allows these modifiers to be omitted.

```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T {
  int K = 1;

  void p();
}
```

Interfaces vs. Abstract Classes

Interfaces vs. Abstract Classes

	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods.

Interfaces vs. Abstract Classes

- Java allows only **single inheritance** for class extension but allows **multiple extensions** for interfaces.

```
public class NewClass extends BaseClass
    implements Interface1, ..., InterfaceN {
  ...
}
```

- An interface can inherit other interfaces using the **extends** keyword. Such an interface is called a **subinterface**.

```
public interface NewInterface extends Interface1, ..., InterfaceN {
  // constants and abstract methods
}
```

Interfaces vs. Abstract Classes

- A class implementing **NewInterface** must implement the abstract methods defined in **NewInterface**, **Interface1**, and **InterfaceN**.
  - An **interface** can *extend* other interfaces but not classes.
  - A **class** can *extend* its superclass and *implement* multiple interfaces.

Example

```
abstract class Animal {
  public abstract String howToEat();
}
```

Two subclasses of `Animal` are defined as follows:

```
class Chicken extends Animal {
  public String howToEat() {
    return "Fry it";
  }
}

class Duck extends Animal {
  public String howToEat() {
    return "Roast it";
  }
}
```

```
public static void main(String[] args) {
  Animal animal = new Chicken();
  eat(animal);

  animal = new Duck();
  eat(animal);
}

public static void eat(Animal animal) {
  animal.howToEat();
}
```

```
public static void main(String[] args) {
  Edible stuff = new Chicken();
  eat(stuff);

  stuff = new Duck();
  eat(stuff);

  stuff = new Broccoli();
  eat(stuff);
}

public static void eat(Edible stuff) {
  stuff.howToEat();
}
```

```
interface Edible {
  public String howToEat();
}

class Chicken implements Edible {
  public String howToEat() {
    return "Fry it";
  }
}

class Duck implements Edible {
  public String howToEat() {
    return "Roast it";
  }
}
```

Sorting an Array of Objects

- This example presents a static generic method for *sorting an array of comparable objects*.
- The objects are instances of the **Comparable** interface, and they are compared using the **compareTo** method.
- The method can be used to sort an array of **any** objects as long as their classes implement the **Comparable** interface.

Sorting an Array of Objects

```
public class GenericSort {
   public static void main(String[] args) {
      // Create an Integer array
      Integer[] intArray = { new Integer(2), new Integer(4),
            new Integer(3) };
      // Create a Double array
      Double[] doubleArray = { new Double(3.4), new
            Double(1.3), new Double(-22.1) };
      // Create a Character array
      Character[] charArray = { new Character('a'), new
            Character('J'), new Character('r') };
      // Create a String array
      String[] stringArray = { "Tom", "John", "Fred" };
```

Sorting an Array of Objects

```
        // Sort the arrays
        Arrays.sort(stringArray);
        Arrays.sort(charArray);
        Arrays.sort(doubleArray);
        Arrays.sort(intArray);
        // Display the sorted arrays
        System.out.print("Sorted Integer objects: ");
        printList(intArray);
        System.out.print("Sorted Double objects: ");
        printList(doubleArray);
        System.out.print("Sorted Character objects: ");
        printList(charArray);
        System.out.print("Sorted String objects: ");
        printList(stringArray);
}
```

Sorting an Array of Objects

```
        /** Print an array of objects */
        public static void printList(Object[] list) {
                for (int i = 0; i < list.length; i++)
                        System.out.print(list[i] + " ");
        System.out.println();
}
```

Sorting an Array of Objects

- Now suppose we want to use the sort method of the Arrays class to sort an array of **Employee** objects.
- Then the Employee class must implement the **Comparable** interface.

Sorting an Array of Objects

- To make a class implement an interface:
  - You declare that your class intends to implement the given interface:

    class Employee **implements** **Comparable**
  - You supply definitions for all methods in the interface.

Sorting an Array of Objects

- Let's suppose that we want to compare employees by their SALARY:

```
public int compareTo(Employee other) {
        if (salary < other.salary)
                return -1;
        if (salary > other.salary)
                return 1;
        return 0;
}
```

Sorting an Array of Objects

- Let's suppose that we want to compare employees by their NAME:

```
public int compareTo(Employee other) {
        return name.compareToIgnoreCase(other.name);
}
```

Sorting an Array of Objects

```
public class EmployeeSortTest {
   public static void main(String[] args) {
      Employee[] staff = new Employee[3];
      staff[0] = new Employee("Tony Tester", 3800);
      staff[1] = new Employee("Harry Hacker", 3500);
      staff[2] = new Employee("Carl Cracker", 7500);

      Arrays.sort(staff);
      for (Employee e : staff)
      System.out.println("name = " + e.getName() + ",
            salary = " + e.getSalary());
   }
}
```

Sorting an Array of Objects

```
class Employee implements Comparable<Employee> {
	private String name;
	private double salary;
	public Employee(String n, double s) {
		name = n;
		salary = s;
	}
	public String getName() {
		return name;
	}
	public double getSalary() {
		return salary;
	}
```

Sorting an Array of Objects

```
    /*Compares employees by NAME
    public int compareTo(Employee other) {
        return name.compareToIgnoreCase(other.name);
    }
    /* Compares employees by salary*/
     public int compareTo(Employee other) {
        if (salary < other.salary)
                return -1;
        if (salary > other.salary)
                return 1;
        return 0;
    }
}
```

Reference

- **Introduction to Java Programming** 8th , Y. Daniel Liang.