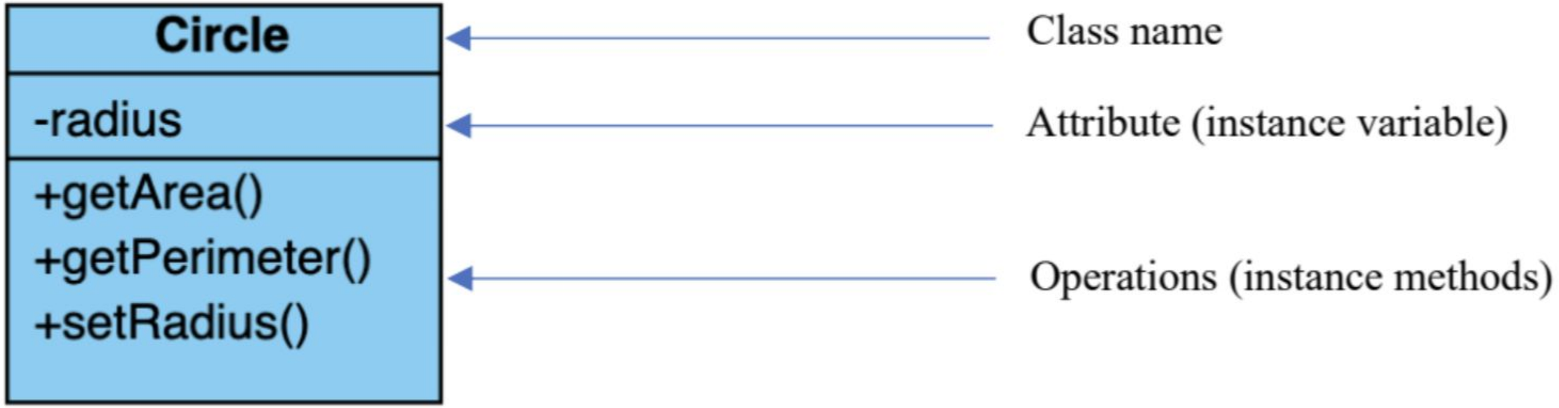
CS213 Recitation 2

- Class and object
- Constructors
- Accessing Objects
- Java library classes
- The Object Class
- Static variables, constants, and methods
- Visibility Modifiers
- Data Encapsulation
- Passing Objects to Methods
- Array of Objects
- Immutable Objects and Classes
- Scope of Variables

Class and object

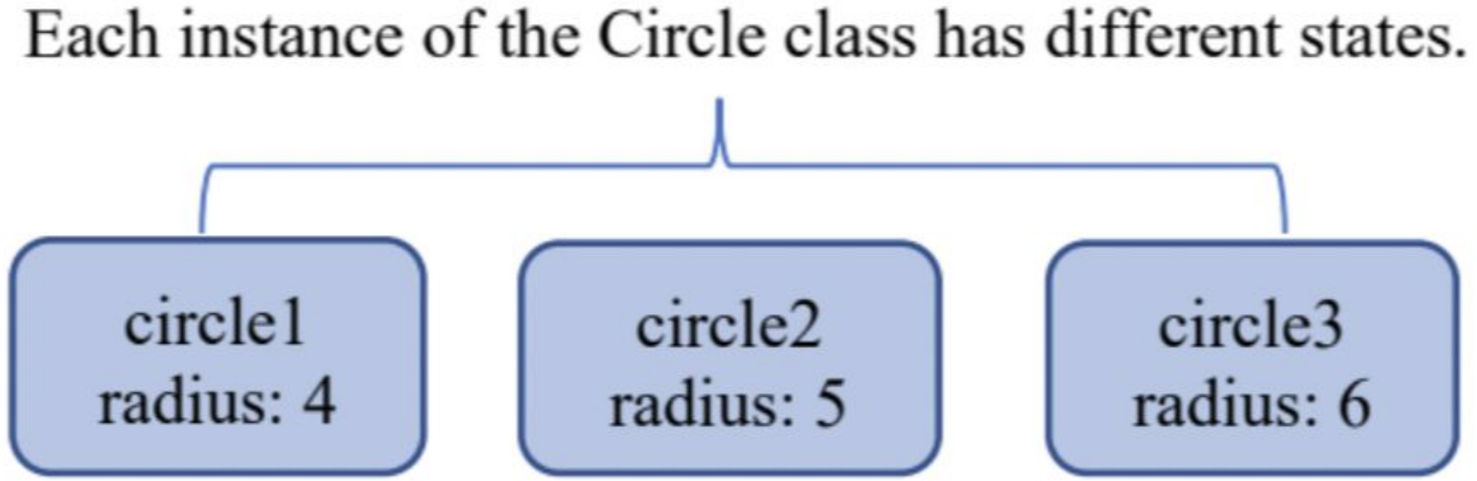
A class defines the properties and behaviors for objects. OOP involves programming using objects. An object represents an entity in the real world that can be distinctly identified, tangible or intangible.

The following picture is an example of a class:



A class is a template, an object is an instance of a class. You can create many instances of a class and every instance has its own “state”.

As shown in the following pictures, we can get several circle objects after we defined a circle class.

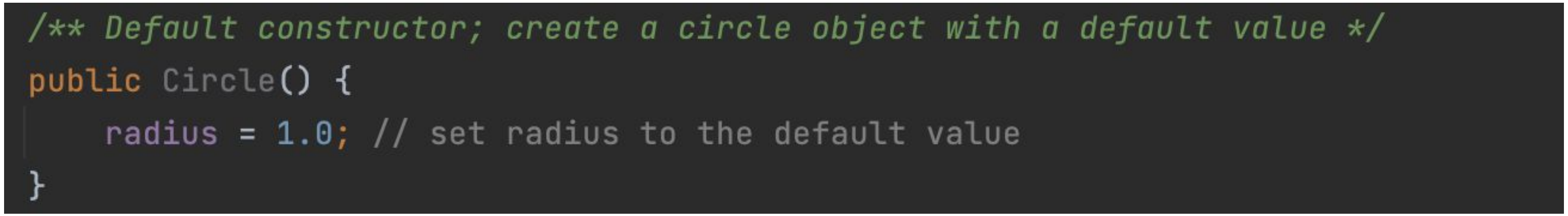


Constructors

A constructor is invoked to create (instantiate) an object using the “new” operator. Constructors are a special kind of method. They have three peculiarities:

1. A constructor must have the same name as the class itself.
2. Constructors do not have a return type—not even void.
3. Constructors are invoked using the “new” operator when an object is created.

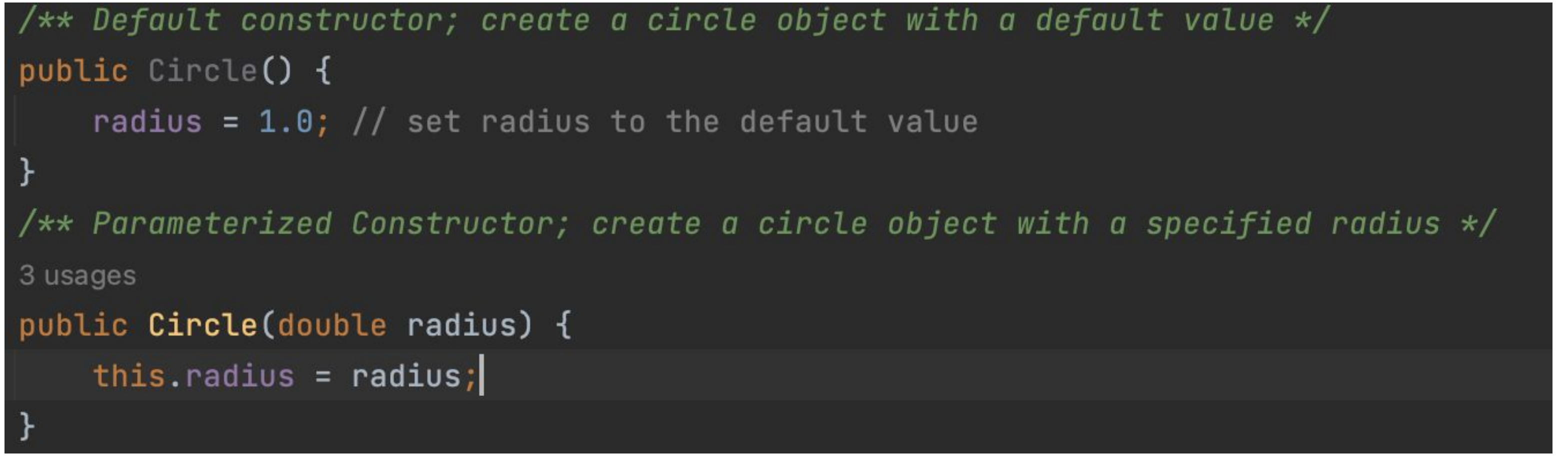
Constructors play the role of initializing objects.



```
/** Default constructor; create a circle object with a default value */
public Circle() {
    radius = 1.0; // set radius to the default value
}
```

In one class, it can have multiple constructors (have the same name but different signatures). There are 3 kinds of constructors.

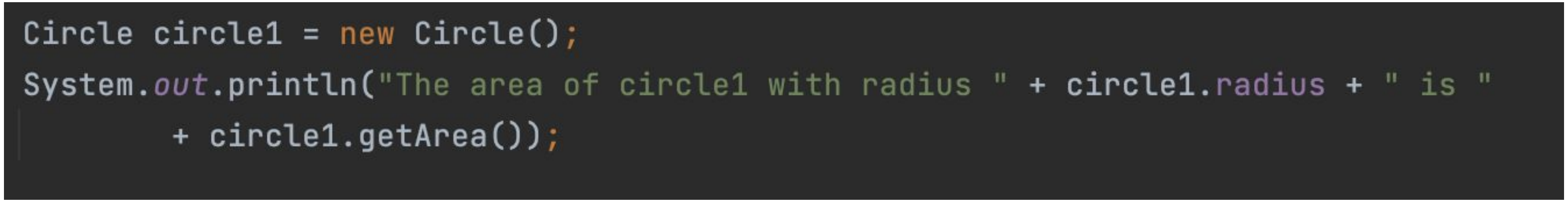
1. Default constructor – no-parameter constructors.
2. Parameterized constructor – a various numbers of parameters are defined.
3. Copy constructor – to clone an object; a single parameter with the class type is defined.



```java
/** Default constructor; create a circle object with a default value */
public Circle() {
    radius = 1.0; // set radius to the default value
}
/** Parameterized Constructor; create a circle object with a specified radius */
3 usages
public Circle(double radius) {
    this.radius = radius;
}
```

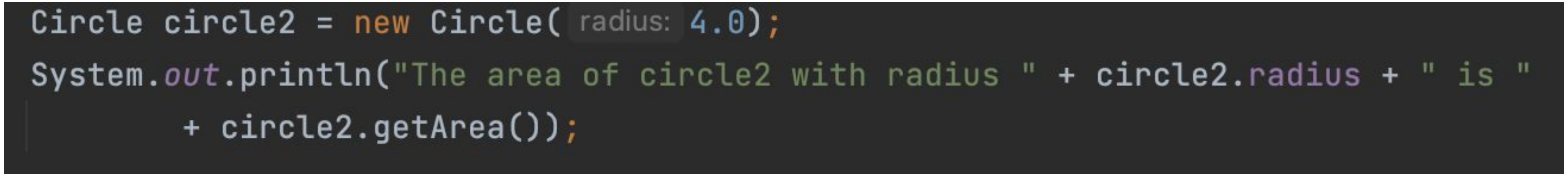
To construct an object from a class, invoke a constructor of the class using the “new” operator as follows: new ClassName(arguments);

For example, new Circle() creates an object of the Circle class using the first constructor (default constructor) defined in the Circle class



```java
Circle circle1 = new Circle();
System.out.println("The area of circle1 with radius " + circle1.radius + " is "
        + circle1.getArea());
```

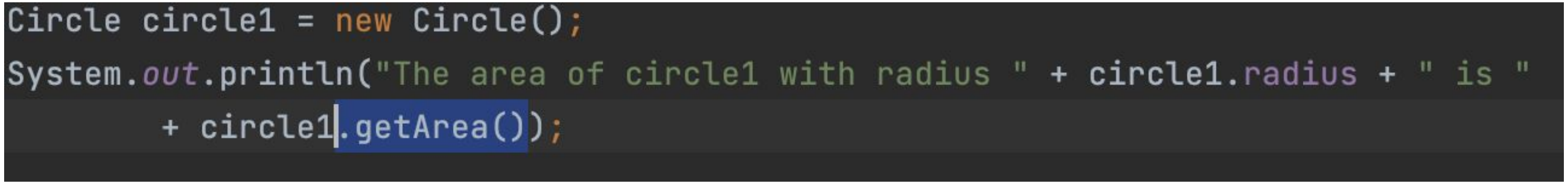
New Circle(25.0) creates an object using the second constructor (parameterized constructor) defined in the Circle class.



```java
Circle circle2 = new Circle( radius: 4.0);
System.out.println("The area of circle2 with radius " + circle2.radius + " is "
        + circle2.getArea());
```

Accessing Objects

An object’s data and methods can be accessed through the dot (.) operator via the object’s reference variable.



```
Circle circle1 = new Circle();
System.out.println("The area of circle1 with radius " + circle1.radius + " is "
        + circle1.getArea());
```

Objects are accessed via the object’s reference variables (circle1 in the picture), which contain references to the objects. A class is essentially a programmer-defined type. A class is a reference type, which means that a variable of the class type can reference an instance of the class.

```
class Car {
    String color;
    int speed;
    void accelerate() {
        speed += 10;
    }
    void display() {
        System.out.println("The " + color + " car is moving at " + speed + " km/h.");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.color = "Red";
        myCar.speed = 50;
        myCar.display();
        myCar.accelerate();
        myCar.display();
    }
}
```

```
The Red car is moving at 50 km/h.
The Red car is moving at 60 km/h.
```

Java library classes

In object-oriented programming, a class library is a collection of prewritten classes or coded templates, any of which can be specified and used by a programmer when developing an application program.

For example, you can use the Date class to process date information, and use the Random class to generate random numbers.

```java
import java.util.Random;
import java.util.Date;

public class Demo {
    public static void main(String[] args) {
        // Using the Random class to generate random numbers
        Random random = new Random();

        // Generating a random integer between 0 and 99
        int randomInt = random.nextInt(100);
        System.out.println("Random integer between 0 and 99: " + randomInt);

        // Using the Date class to represent the current time
        Date currentDate = new Date();

        // Displaying the current date and time
        System.out.println("Current date and time: " + currentDate);
    }
}
```

```
Random integer between 0 and 99: 45
Current date and time: Wed Sep 19 14:58:52 CST 2023
```

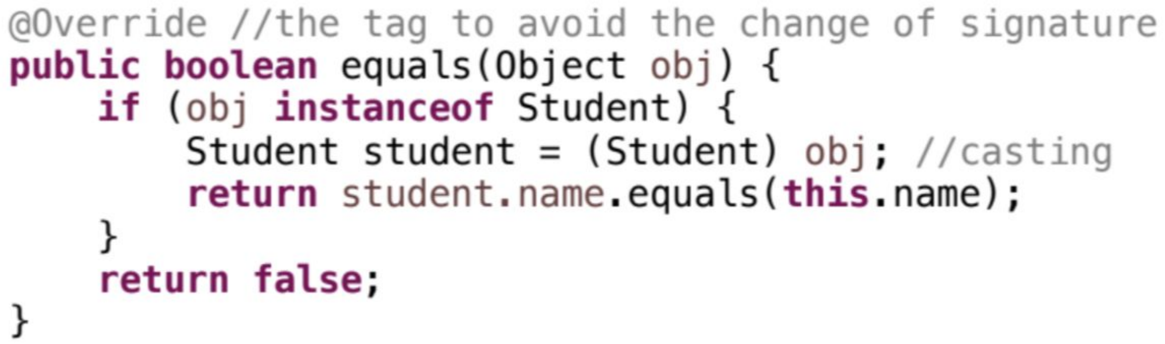
The Object Class

Class Object is the root of the Java class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class, especially the equals() and toString() method.

The default method of equals() is to compares the object references or the memory location where the objects are stored in the heap. The default method of toString() returns a string consisting of the name of the class of which the object is an instance, with the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object.

We can also “Overriding” them in subclasses. We should notice that “Overriding” is different from “overloading”.

Below is an example of overriding:



```
@Override //the tag to avoid the change of signature
public boolean equals(Object obj) {
    if (obj instanceof Student) {
        Student student = (Student) obj; //casting
        return student.name.equals(this.name);
    }
    return false;
}
```

Static variables, constants, and methods

A static variable is shared by all objects of the class. Static variables store values for the variables in a common memory space, if one object changes the value of a static variable, all objects of the same class are affected.

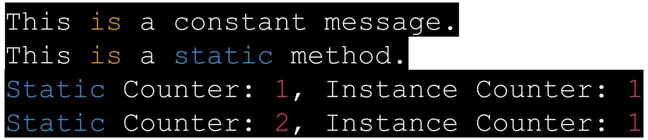
A static method cannot access instance members (i.e., instance data fields and methods) of the class.

Static methods can be called without creating an instance of the class.

An instance method can invoke an instance or static method and access an instance or static data field. A static method can invoke a static method and access a static data field. However, a static method of a class cannot invoke an instance method or access an instance data field without creating an object, since instance methods and instance data fields must be associated with a specific object.

Static/or non-static	Invoke instance methods	Access instance variables	Invoke static methods	Access static variables
Instance methods	√	√	√	√
Static methods	X	X	√	√

```java
public class Demo {
    // Static variable: It belongs to the class and not to any particular instance. It will be shared among all the instances of the class.
    public static int staticCounter = 0;
    // Constants: They are declared using the 'final' keyword and are usually static. Their value cannot be changed once assigned.
    public static final String CONSTANT_MESSAGE = "This is a constant message.";
    // Instance variable for demonstration
    private int instanceCounter = 0;

    // Constructor: increments the static and instance counters each time a new instance is created.
    public Demo() {
        staticCounter++;
        instanceCounter++;
    }

    // Static method: It belongs to the class and not to any particular instance. It can be called without creating an instance of the class.
    public static void displayStaticMethod() {
        System.out.println("This is a static method.");
    }

    // Method to display counters
    public void displayCounters() {
        System.out.println("Static Counter: " + staticCounter + ", Instance Counter: " + instanceCounter);
    }

    public static void main(String[] args) {
        System.out.println(Demo.CONSTANT_MESSAGE);
        Demo.displayStaticMethod();
        Demo demo1 = new Demo();
        Demo demo2 = new Demo();
        demo1.displayCounters();
        demo2.displayCounters();
    }
}
```

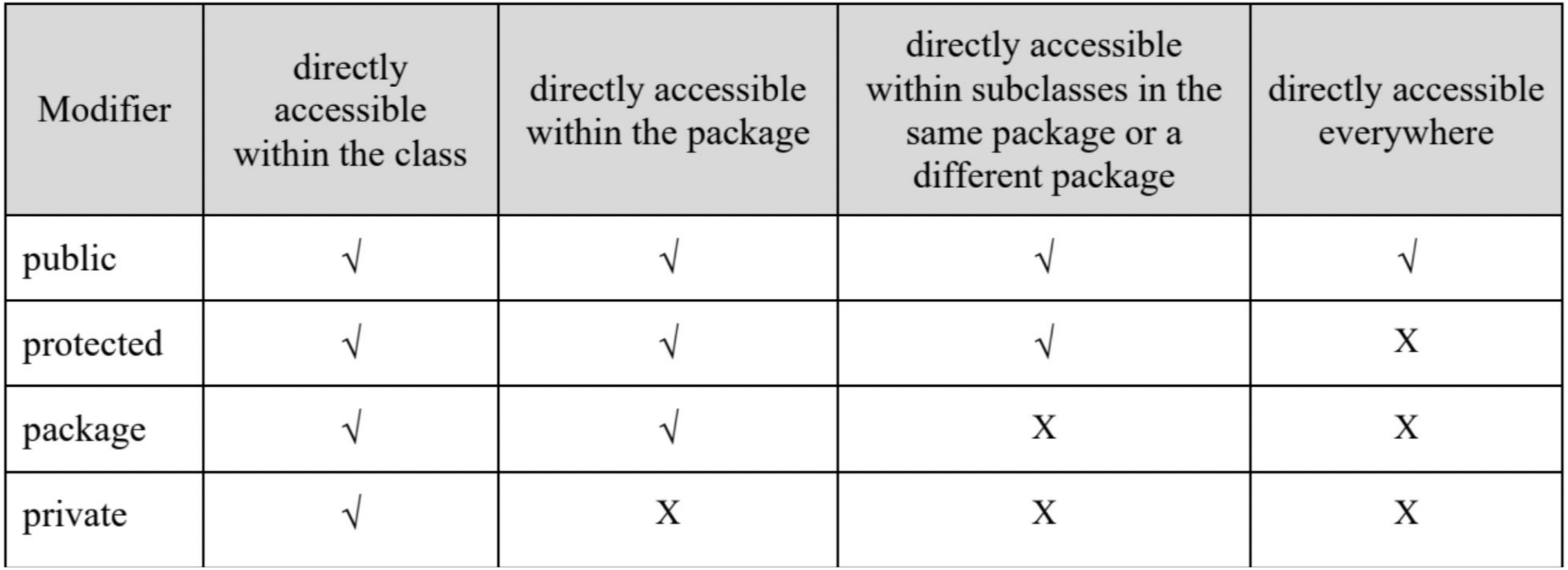


```
This is a constant message.
This is a static method.
Static Counter: 1, Instance Counter: 1
Static Counter: 2, Instance Counter: 1
```

Visibility Modifiers

Visibility modifiers can be used to specify the visibility of a class and its members

Java provides four modifiers for class members. Below is their accessibility:



Modifier	directly accessible within the class	directly accessible within the package	directly accessible within subclasses in the same package or a different package	directly accessible everywhere
public	√	√	√	√
protected	√	√	√	X
package	√	√	X	X
private	√	X	X	X

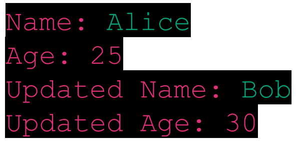
Data Encapsulation

To prevent direct modifications of data fields from other classes, you should always declare the data fields as private using the private modifier. This is known as data encapsulation.

A private data field cannot be accessed by an object from outside the class that uses the private data field. However, a client class often needs to retrieve and modify a data field. To make a private data field accessible, provide a “getter” method to return its value. To enable a private data field to be updated, provide a “setter” method to set a new value. A getter method is also referred to as an accessor and a setter to a mutator.

```java
class Person {
    // Private attributes - Data Encapsulation
    private String name;
    private int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Public getter for name
    public String getName() {
        return name;
    }

    // Public setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Public getter for age
    public int getAge() {
        return age;
    }

    // Public setter for age
    public void setAge(int age) {
        if (age > 0) {  // Simple validation to ensure age is positive
            this.age = age;
        }
    }
}
```

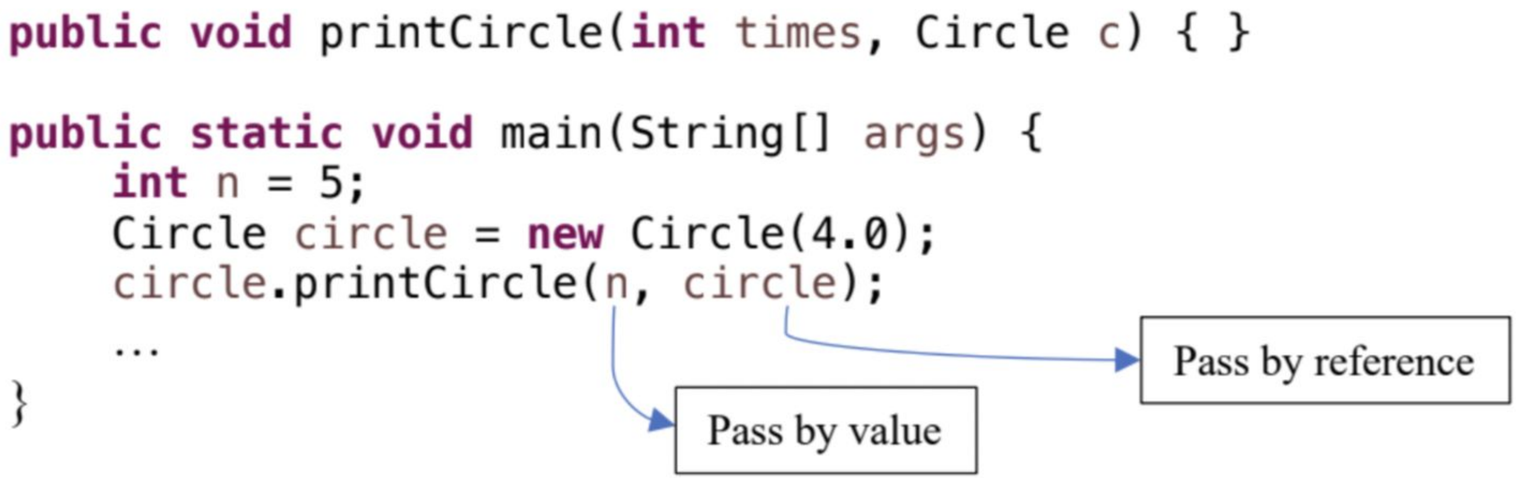
```java
public class EncapsulationDemo {
    public static void main(String[] args) {
        Person person = new Person("Alice", 25);

        // Accessing attributes through getters and setters
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());

        // Modifying attributes through setters
        person.setName("Bob");
        person.setAge(30);

        System.out.println("Updated Name: " + person.getName());
        System.out.println("Updated Age: " + person.getAge());
    }
```



```
Name: Alice
Age: 25
Updated Name: Bob
Updated Age: 30
```

Passing Objects to Methods

Passing an object to a method is to pass the reference of the object. You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object. For example, The following code passes the circle object as an argument to the printCircle() method:

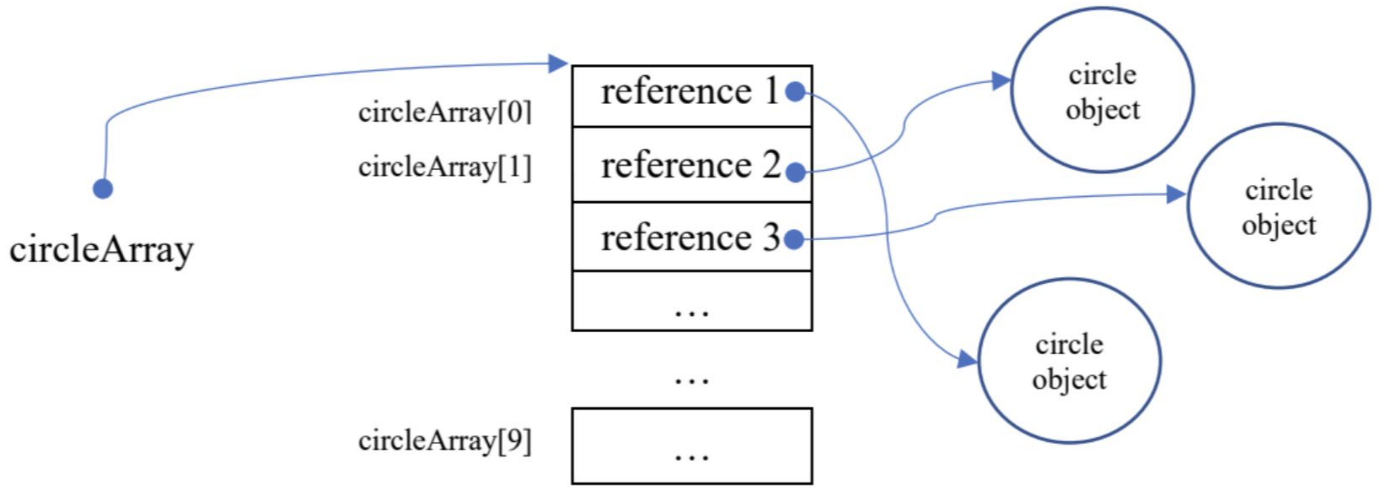


```
public void printCircle(int times, Circle c) { }

public static void main(String[] args) {
    int n = 5;
    Circle circle = new Circle(4.0);
    circle.printCircle(n, circle);
    ...
}
```

Array of Objects

An array of objects is actually an array of references.

For example, if we run this code: Circle[] circleArray = new Circle[10];
We will get this:



Note that an array occupies a block of consecutive memory addresses; however, the memory addresses that are storing the actual circle objects are not necessarily consecutive. When an array of objects is created using the new operator, each element in the array is a reference variable with a default value of null.

Immutable Objects and Classes

We call an object whose contents cannot be changed once the object has been created as immutable object and its class as immutable class.

For a class to be immutable, it must meet the following requirements:
1. All data fields must be private.
2. There can’t be any mutator methods for data fields.
3. No accessor methods can return a reference to a data field that is mutable.

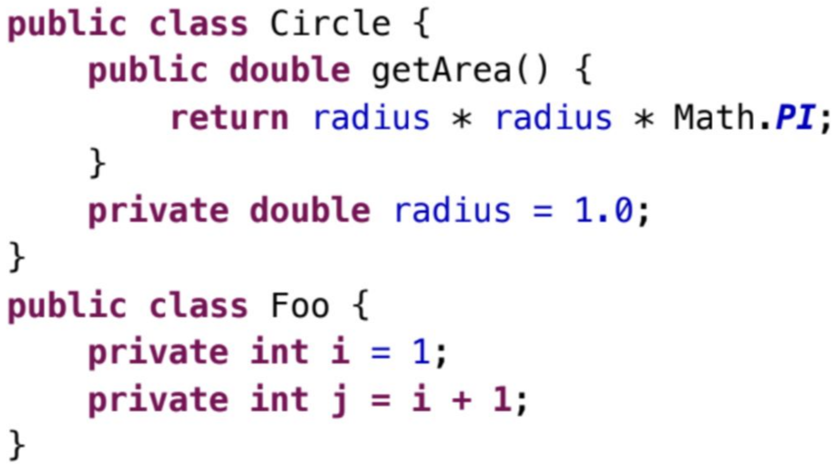
```java
/**
 * This is an example of an immutable class.
 * Once an instance of this class is created, its state cannot be modified.
 */
public final class ImmutablePerson {

    // Declare the attributes as private and final to ensure they cannot be modified.
    private final String name;
    private final int age;

    // Constructor to set the values of the attributes.
    public ImmutablePerson(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter method for name. No setter method to ensure immutability.
    public String getName() {
        return name;
    }

    // Getter method for age. No setter method to ensure immutability.
    public int getAge() {
        return age;
    }
}
```

```java
// Example usage
public static void main(String[] args) {
    ImmutablePerson person = new ImmutablePerson("John", 30);
    System.out.println("Name: " + person.getName());
    System.out.println("Age: " + person.getAge());

    // The following lines of code would produce compilation errors
    // because the ImmutablePerson class is designed to be immutable.
    // person.name = "Jane";
    // Error: name has private access in ImmutablePerson
    // person.age = 31;
    // Error: age has private access in ImmutablePerson
}
```

Scope of Variables

The scope of instance and static variables is the entire class, regardless of where the variables are declared. Instance and static variables in a class are referred to as the class's variables or data fields. A class's variables and methods can appear in any order in the class. The exception is when a data field is initialized based on a reference to another data field. In such cases, the other data field must be declared first.

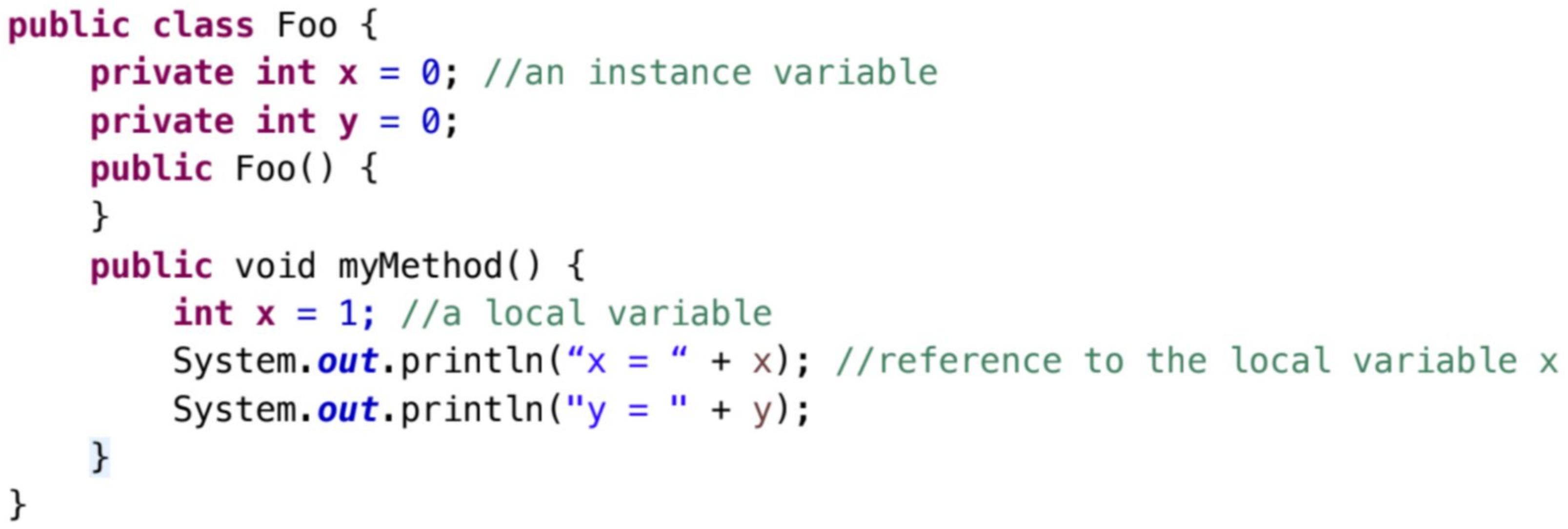
For example, the getArea() method in the Circle class can be declared before the data field radius; however, the integer i must be declared before the integer j.



```java
public class Circle {
    public double getArea() {
        return radius * radius * Math.PI;
    }
    private double radius = 1.0;
}
public class Foo {
    private int i = 1;
    private int j = i + 1;
}
```

A variable defined inside a method is referred to as a local variable. If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is hidden.

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



```java
public class Foo {
    private int x = 0; //an instance variable
    private int y = 0;
    public Foo() {
    }
    public void myMethod() {
        int x = 1; //a local variable
        System.out.println(“x = “ + x); //reference to the local variable x
        System.out.println("y = " + y);
    }
}
```