Introduction to Information Systems and Programming

Objects and Classes

[some materials adopted from Liang, Introduction to Java Programming]

Objectives

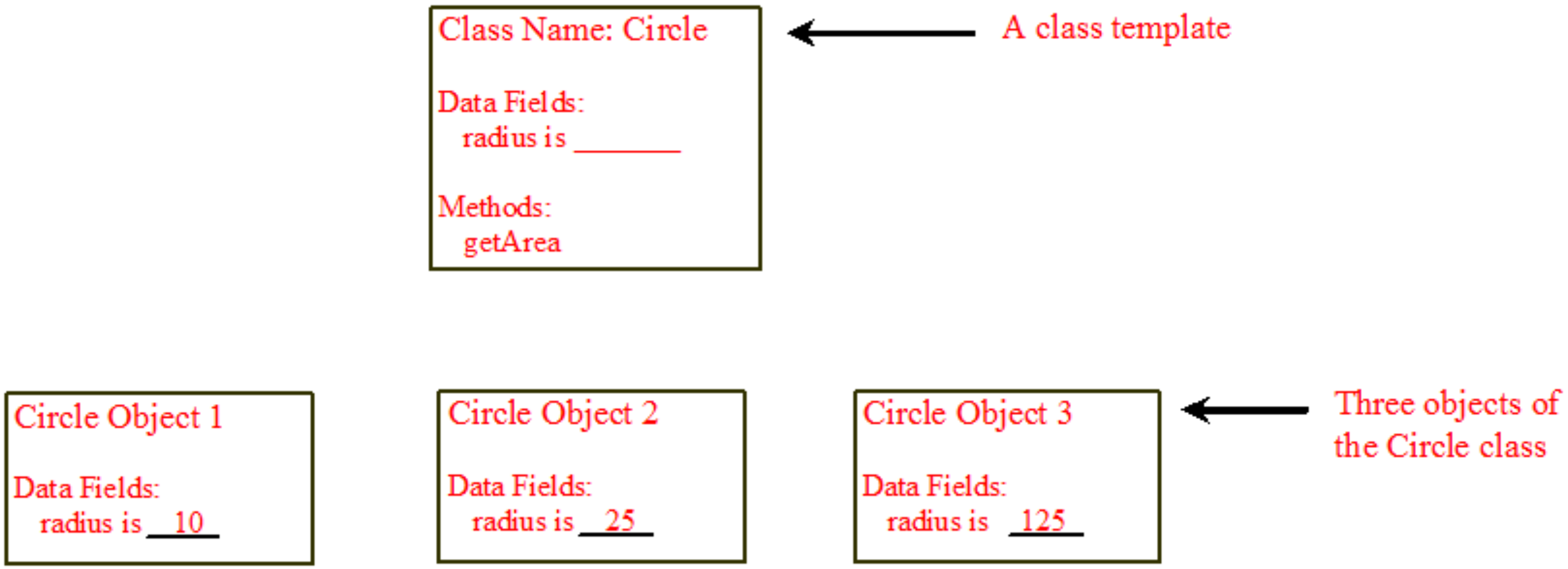
- Class and Objects in Java
- Constructor
- Static modifier
- Scope of Variables
- Access modifier: public, default, private
- Encapsulation
- Passing object as argument

Object-Oriented (OO) Programming

- Object-oriented Programming is a **programming paradigm, the way of thinking where software are designed around the concept of object**
  - vs. *procedural programming* (step-by-step statements, procedure call)
  - vs. *functional programming* (data are passed to pure functions, no changing state)
- OOP Program generally consists of **interaction between objects**
- OOP Principles (also what makes a programming language an OOP language) : **Abstraction, encapsulation, inheritance, polymorphism**

Class vs Object

- Class: **Template** to create object. **Attributes** and **methods** are defined here.
- Object represents an **entity**, an instantiation of a class
- An object has a unique identity, state, and behavior
  - State = data fields = properties = attributes
  - Behavior = method = action



Class Name: Circle

Data Fields:
radius is _______

Methods:
getArea

A class template

Circle Object 1

Data Fields:
radius is __10__

Circle Object 2

Data Fields:
radius is __25__

Circle Object 3

Data Fields:
radius is __125__

Three objects of the Circle class

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;
```

Mini quiz:

int x = 5; → is *x* a reference variable?

Instantiating Object

Template:

```
new ClassName();
```

Example:

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

Variable Declaration + Instantiating Object

Template:

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

Example:

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

Accessing Attribute and Invoking Method

- Referencing the object’s data/attribute:

```
objectRefVar.data
```

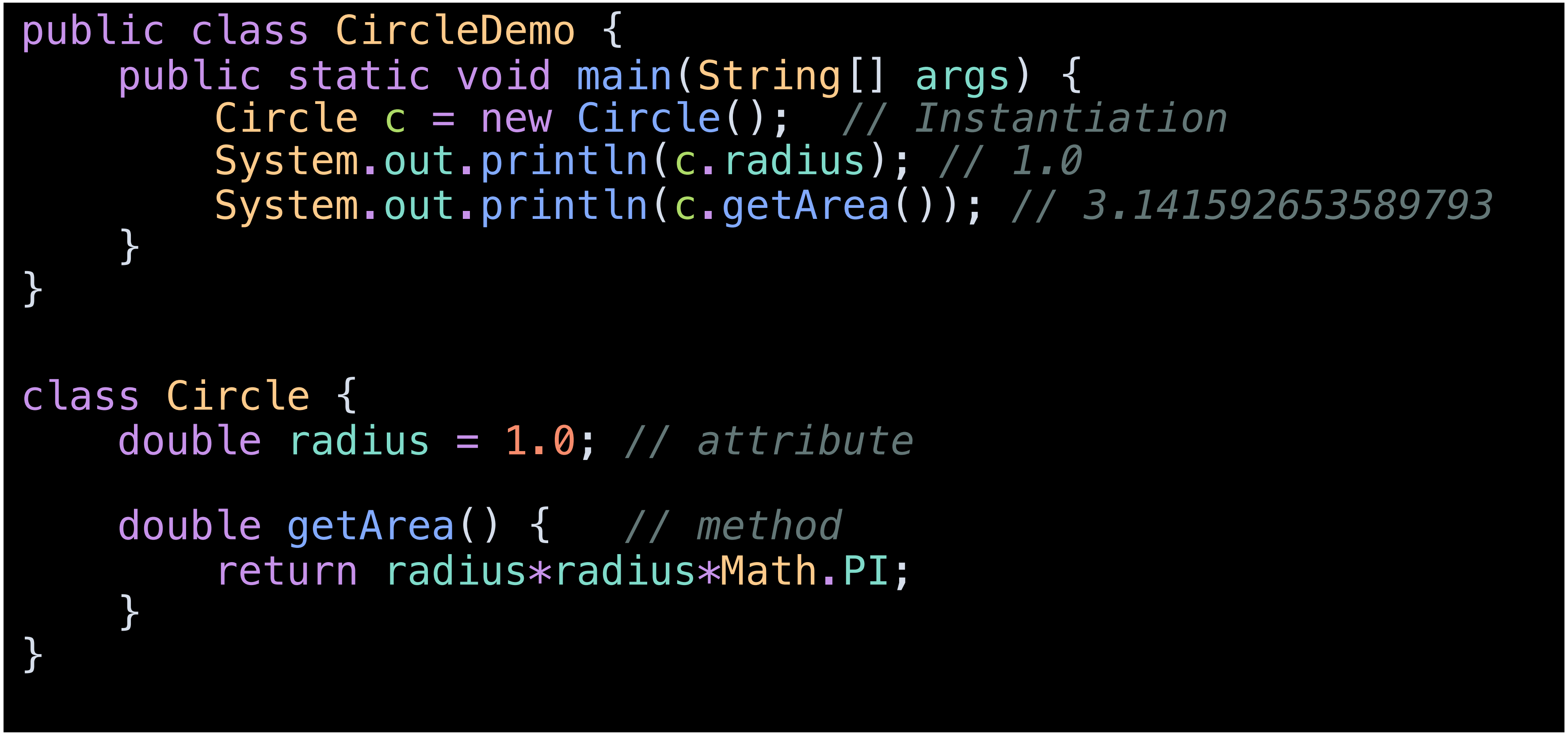
e.g., `myCircle.radius`

- Invoking the instance method:

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

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

Class Definition, Object Instantiation, Accessing Instance Attribute, Invoking Method



```java
public class CircleDemo {
    public static void main(String[] args) {
        Circle c = new Circle();  // Instantiation
        System.out.println(c.radius); // 1.0
        System.out.println(c.getArea()); // 3.141592653589793
    }
}

class Circle {
    double radius = 1.0; // attribute

    double getArea() {   // method
        return radius*radius*Math.PI;
    }
}
```

What if I want to instantiate an object with a radius other than 1.0?

We need to define a constructor!

Constructor

- Constructors: special type of method (*remember __init__ in Python?*), invoked to **construct objects** from a class
- Constructor is invoked using the **new** keyword
- To define constructor in Java, write the same name as the class name
- **No** return type (not even **void**)
- Can take **parameters**
- Constructor without parameter is called **no-arg constructor**
- Can be **overloaded**

What is method overloading?

Constructor

- Constructors: special type of method (*remember __init__ in Python?*), invoked to **construct objects** from a class
- Constructor is invoked using the **new** keyword
- To define constructor in Java, write the same name as the class name
- **No** return type (not even **void**)
- Can take **parameters**
- Constructor without parameter is called **no-arg constructor**
- Can be **overloaded**

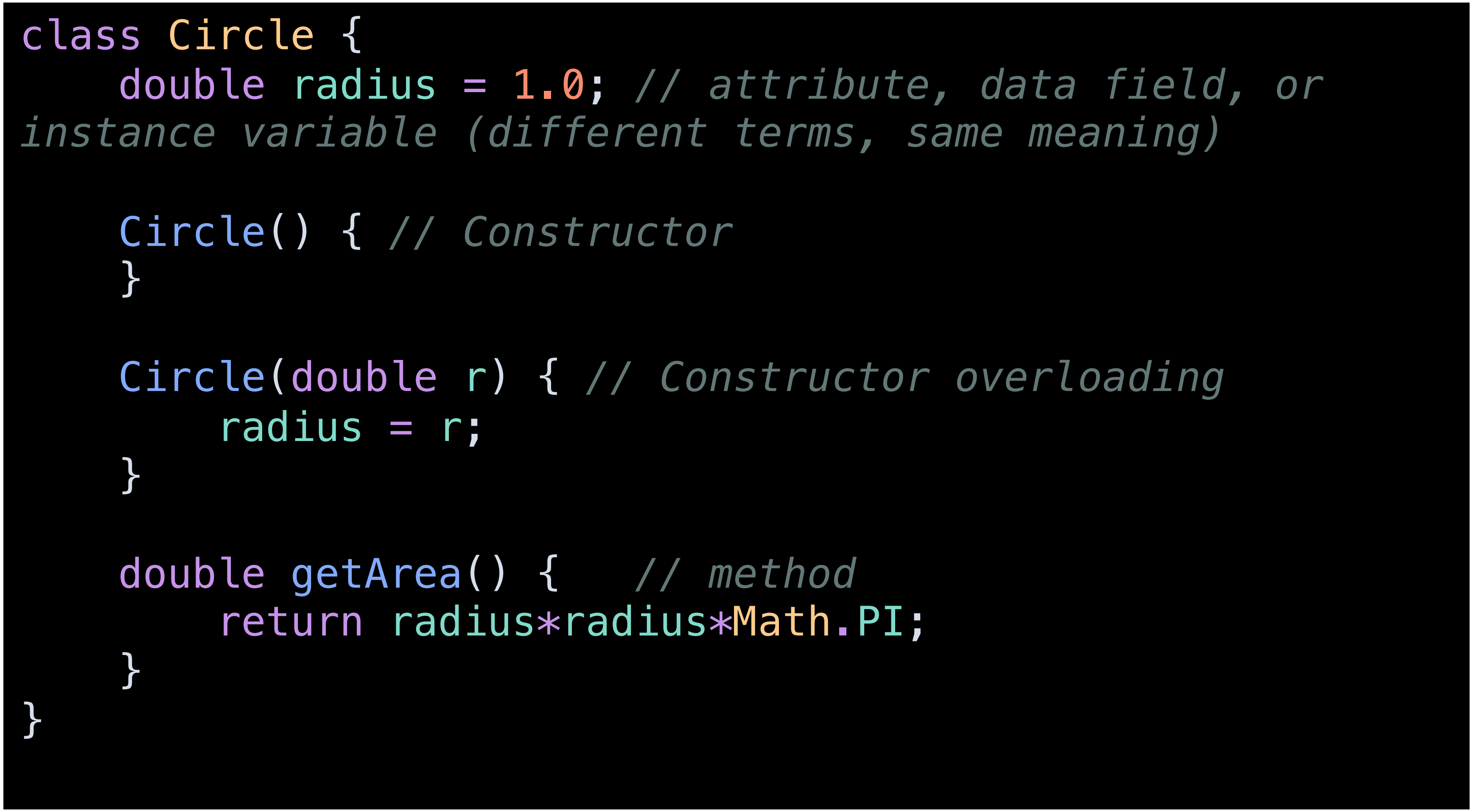
What is method overloading?

Method overloading = ***same method name,***

different set of parameters

Using Constructor

Example



```
class Circle {
    double radius = 1.0; // attribute, data field, or
instance variable (different terms, same meaning)

    Circle() { // Constructor
    }

    Circle(double r) { // Constructor overloading
        radius = r;
    }

    double getArea() {   // method
        return radius*radius*Math.PI;
    }
}
```

Default Constructor

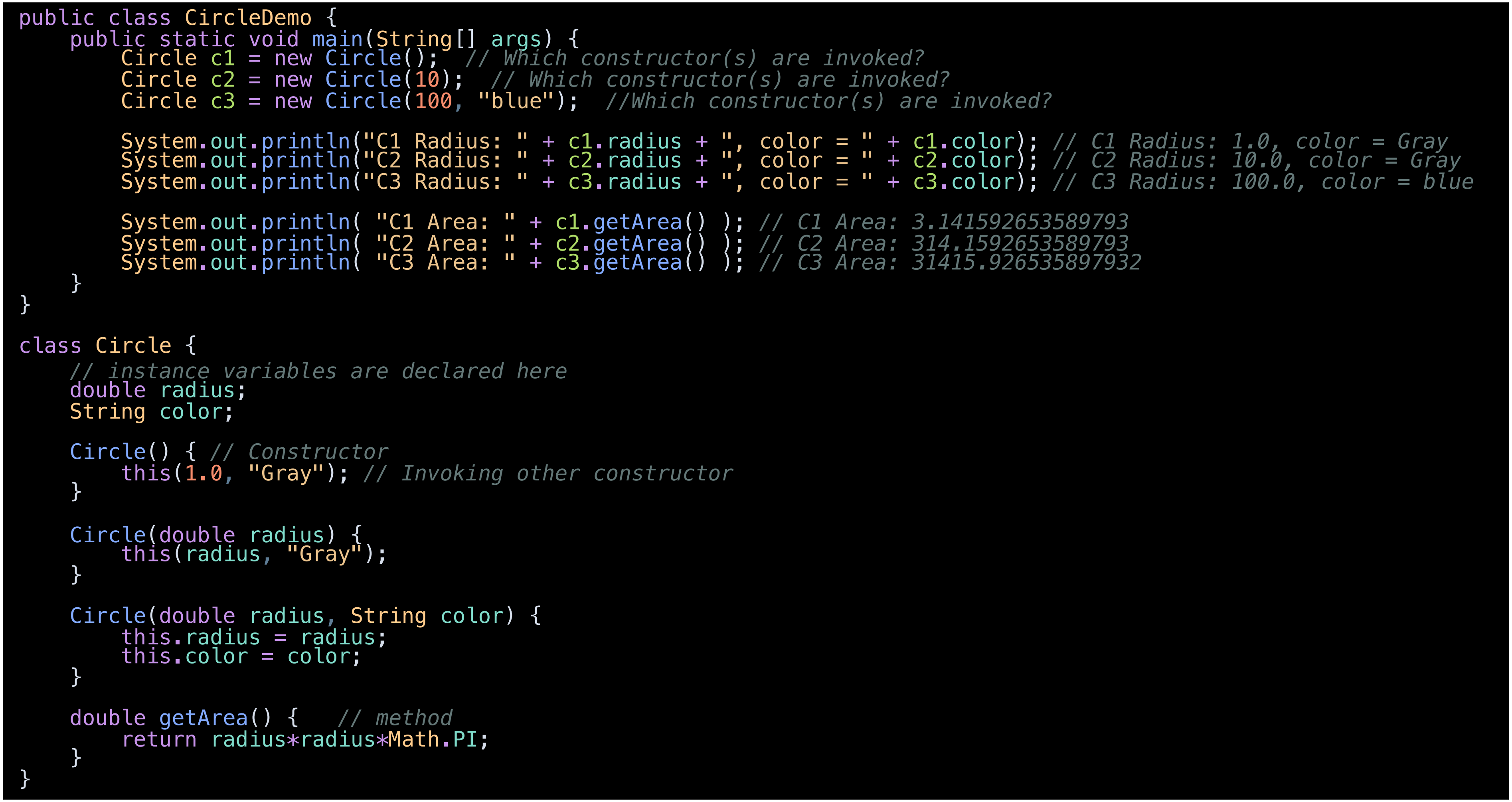
- A class may be defined without constructors. In this case, a no-arg constructor with an empty body is implicitly declared in the class. This constructor, called *a default constructor*, is provided automatically *only if no constructors are explicitly defined in the class*.

this **Keyword**

- Remember *self* in Python? It is called *this* in java
- It refers to the current object.
- Also help to avoid variable name conflict
- *this* also can be used to invoke constructor from the other constructor

see code at the next slide

Putting Everything Together

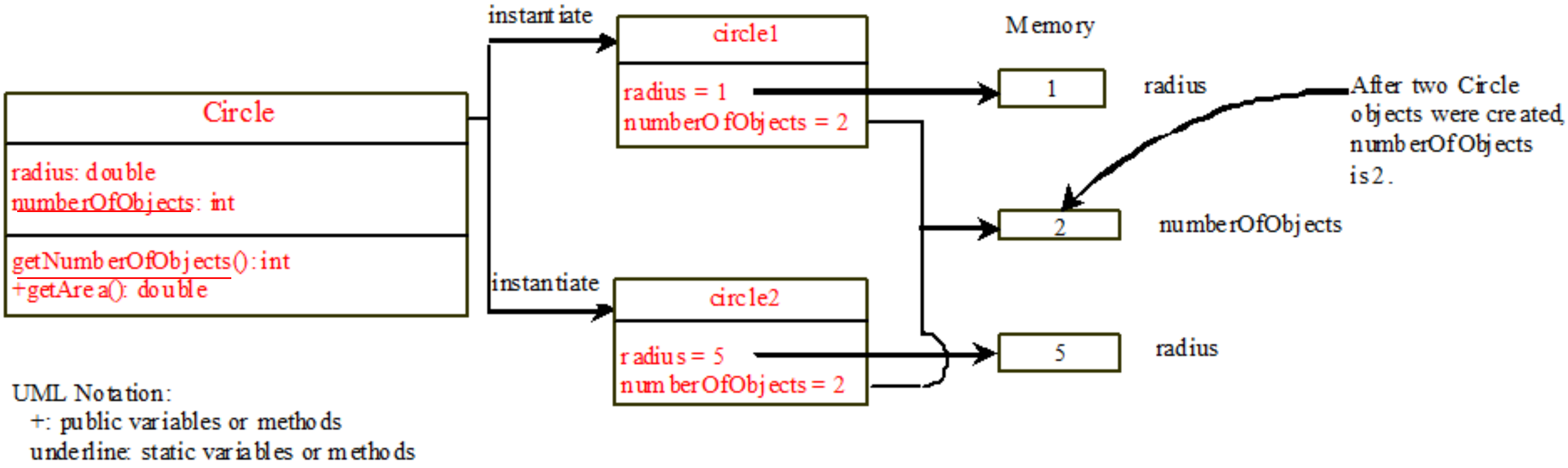


```java
public class CircleDemo {
    public static void main(String[] args) {
        Circle c1 = new Circle();  // Which constructor(s) are invoked?
        Circle c2 = new Circle(10);  // Which constructor(s) are invoked?
        Circle c3 = new Circle(100, "blue");  //Which constructor(s) are invoked?

        System.out.println("C1 Radius: " + c1.radius + ", color = " + c1.color); // C1 Radius: 1.0, color = Gray
        System.out.println("C2 Radius: " + c2.radius + ", color = " + c2.color); // C2 Radius: 10.0, color = Gray
        System.out.println("C3 Radius: " + c3.radius + ", color = " + c3.color); // C3 Radius: 100.0, color = blue

        System.out.println( "C1 Area: " + c1.getArea() ); // C1 Area: 3.141592653589793
        System.out.println( "C2 Area: " + c2.getArea() ); // C2 Area: 314.1592653589793
        System.out.println( "C3 Area: " + c3.getArea() ); // C3 Area: 31415.926535897932
    }
}

class Circle {
    // instance variables are declared here
    double radius;
    String color;

    Circle() { // Constructor
        this(1.0, "Gray"); // Invoking other constructor
    }

    Circle(double radius) {
        this(radius, "Gray");
    }

    Circle(double radius, String color) {
        this.radius = radius;
        this.color = color;
    }

    double getArea() {   // method
        return radius*radius*Math.PI;
    }
}
```

Static Modifier

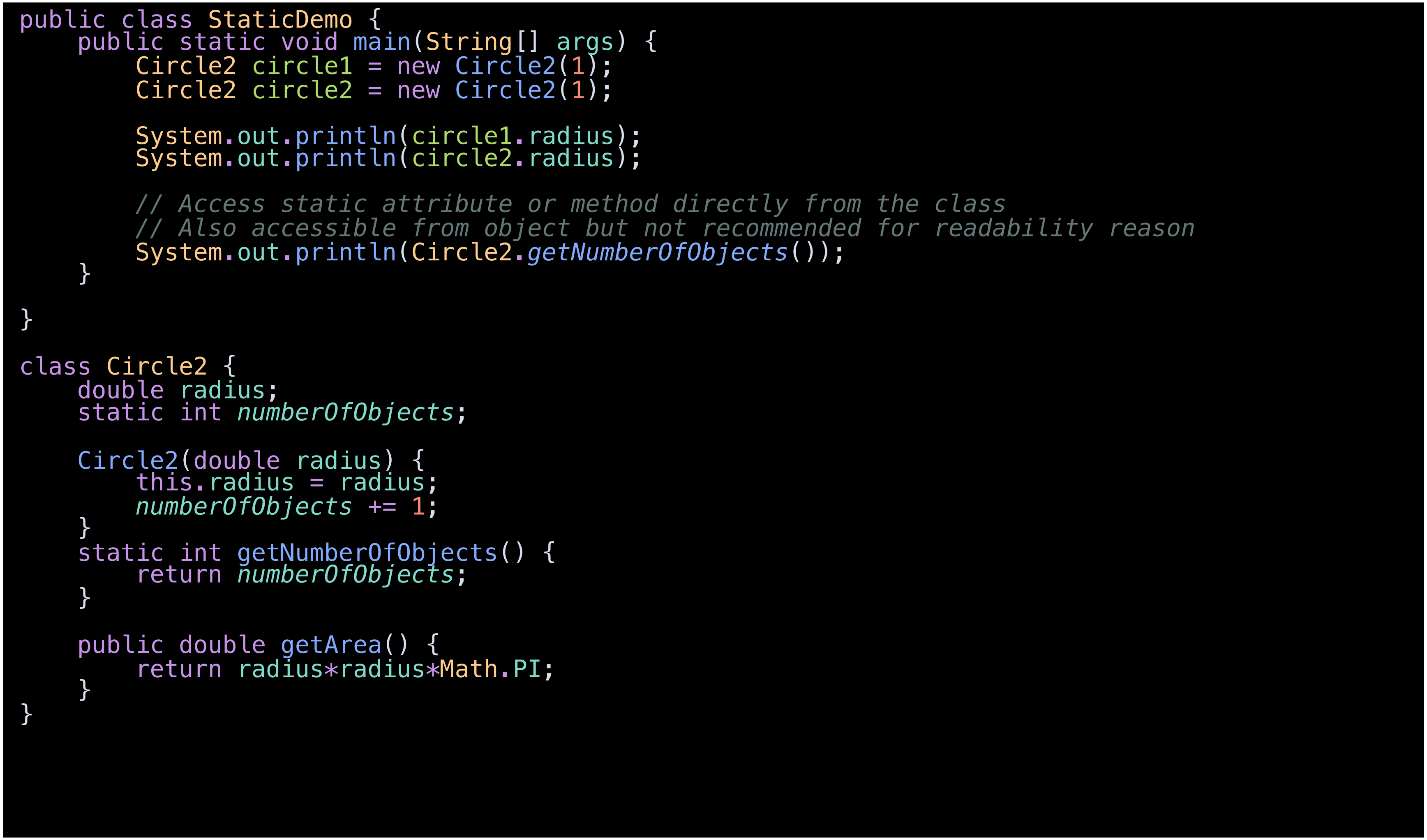
- *Static variables* are **shared** by all the instances of the class
- *Static method* can be called **without** creating an instance of the class
- Static variable or method are **not** tied to a specific object

Static variables, methods



Static Modifier

Code Example



```java
public class StaticDemo {
    public static void main(String[] args) {
        Circle2 circle1 = new Circle2(1);
        Circle2 circle2 = new Circle2(1);

        System.out.println(circle1.radius);
        System.out.println(circle2.radius);

        // Access static attribute or method directly from the class
        // Also accessible from object but not recommended for readability reason
        System.out.println(Circle2.getNumberOfObjects());
    }

}

class Circle2 {
    double radius;
    static int numberOfObjects;

    Circle2(double radius) {
        this.radius = radius;
        numberOfObjects += 1;
    }
    static int getNumberOfObjects() {
        return numberOfObjects;
    }

    public double getArea() {
        return radius*radius*Math.PI;
    }
}
```

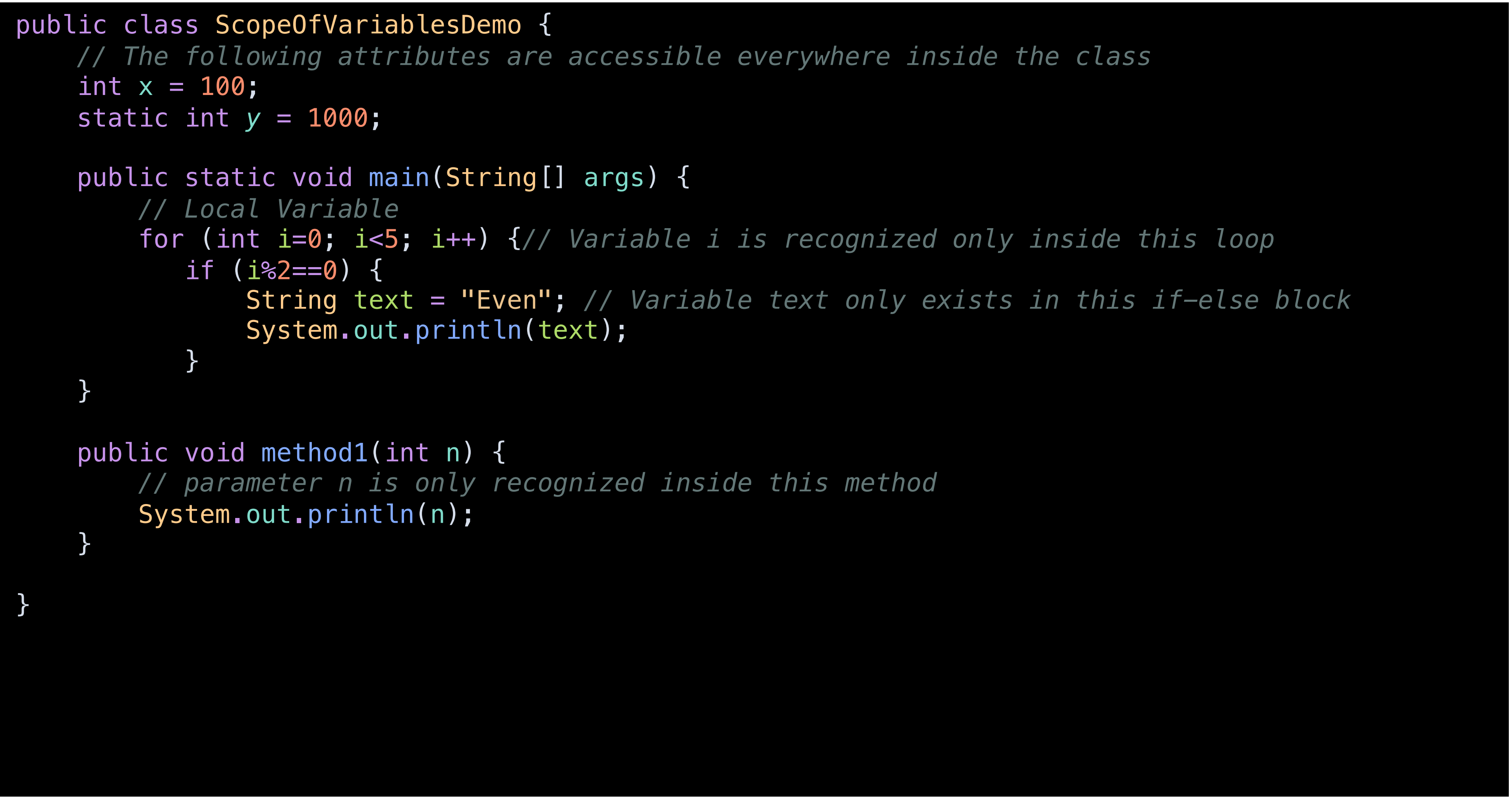
Scope of Variables

- **Local Variable**
  Variables defined inside method, if-else, or looping.

- **Parameter**
  Entire body of method

- **Data field / attribute (Static & non-static)**
  Entire body of class

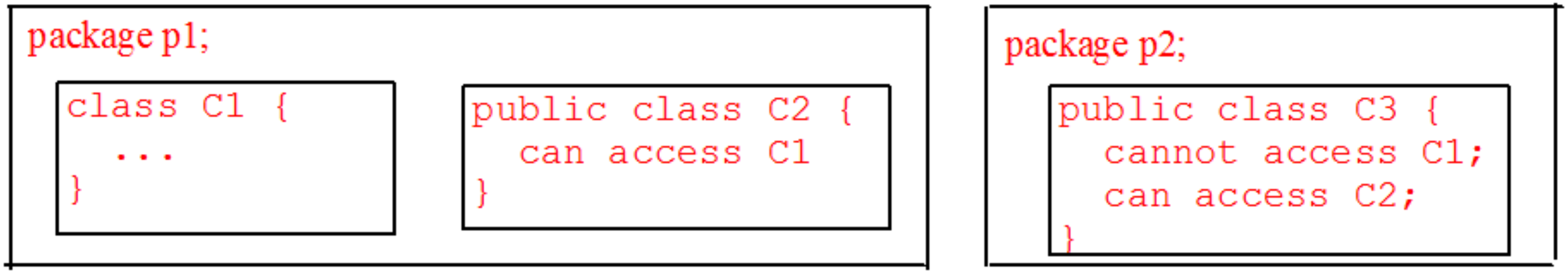
Demo: Scope of Variables



```java
public class ScopeOfVariablesDemo {
    // The following attributes are accessible everywhere inside the class
    int x = 100;
    static int y = 1000;

    public static void main(String[] args) {
        // Local Variable
        for (int i=0; i<5; i++) {// Variable i is recognized only inside this loop
           if (i%2==0) {
               String text = "Even"; // Variable text only exists in this if-else block
               System.out.println(text);
           }
    }

    public void method1(int n) {
        // parameter n is only recognized inside this method
        System.out.println(n);
    }

}
```

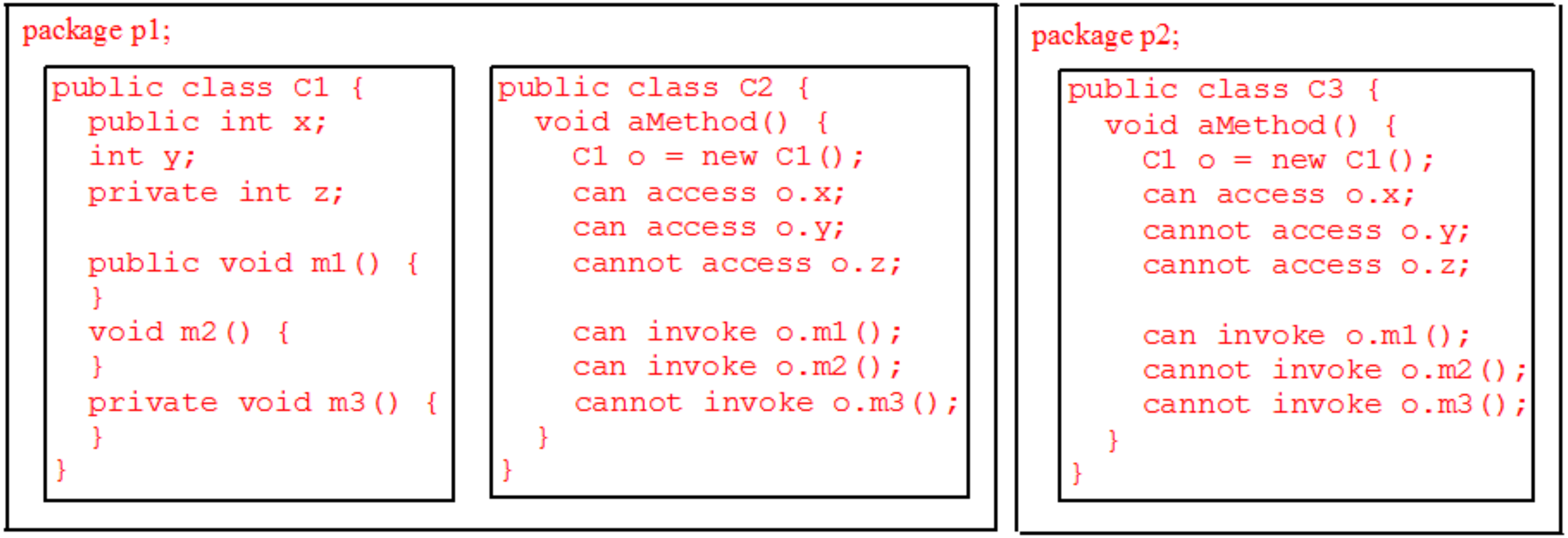
Visibility modifiers: public, default, and private

- public: visible to any class in any package
- Default (no access modifier defined): package private, can be accessed by any class in the same package
- private: visible only by the declaring class



```
package p1;

class C1 {
  ...
}

public class C2 {
  can access C1
}
```

```
package p2;

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

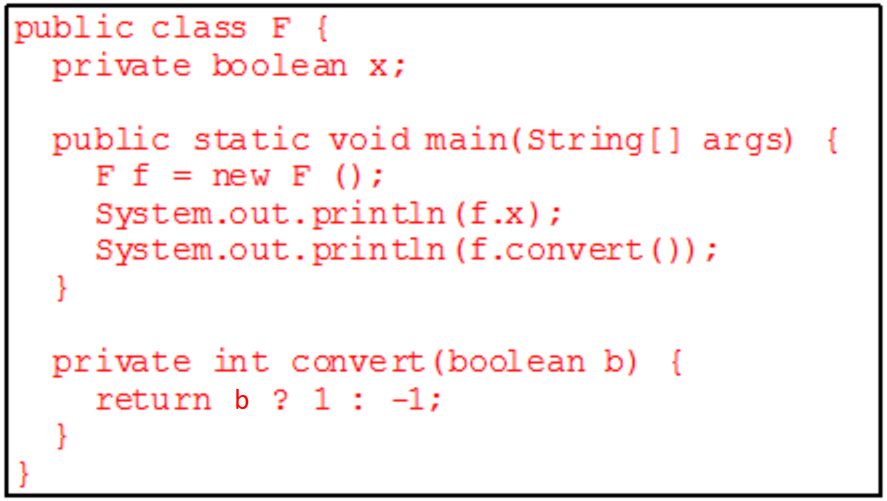


```
package p1;

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

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

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();
  }
}
```

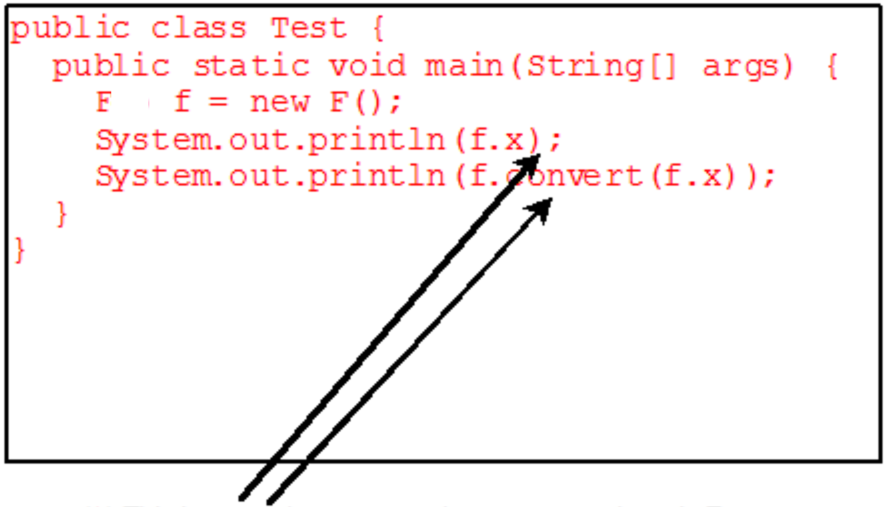
The private modifier restricts access to within a class, the default modifier restricts access to within a package, and the public modifier enables unrestricted access.

Visibility modifiers: public, private



```
public class F {
  private boolean x;

  public static void main(String[] args) {
    F f = new F ();
    System.out.println(f.x);
    System.out.println(f.convert());
  }

  private int convert(boolean b) {
    return b ? 1 : -1;
  }
}
```

(a) This is OK because object f is used inside the F class



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

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

Private members can be used within their own classes

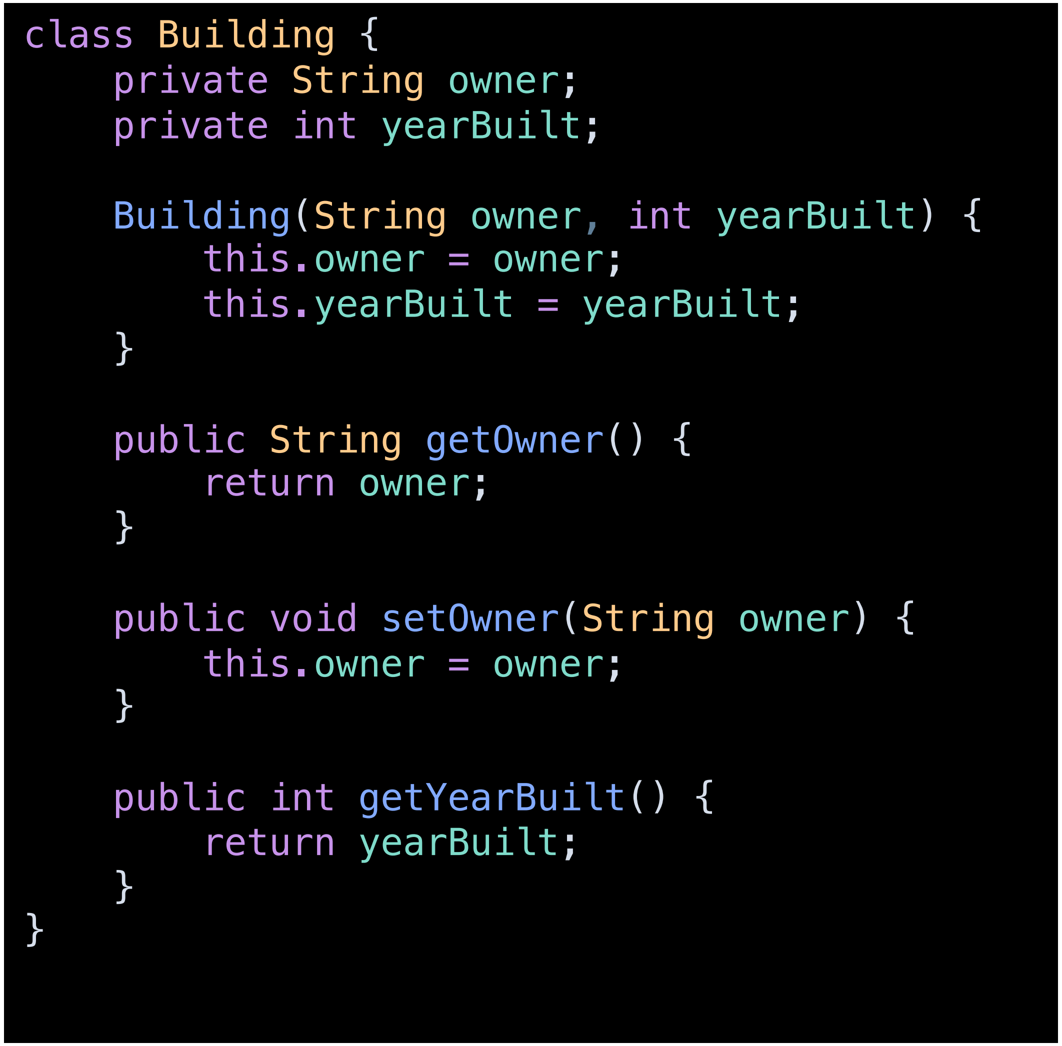
Why put data fields private

- To protect data (Encapsulation)
  - Preventing other programmers tamper the attribute values directly
- Code readability
- **Having a control** on how data field is accessed and mutated
- To make class easy to maintain (related to inheritance)
  - impose class constraints
  - having invariant components

Data Field Encapsulation

- Keep attributes private, if possible
- Use get method (also called **getter / accessor**) to **return** the values of attributes, e.g., double getRadius()
- Use set method (also called **setter / mutator**) to **update** attributes, e.g., void setRadius(double radius)()
- **Design principles:**
  - **Minimize the accessibility of attributes or methods (always private, unless needed to be accessed outside of class definition)**
  - **Use getter and setter when interacting with objects' attributes from different class**

Encapsulation Demo



```
class Building {
    private String owner;
    private int yearBuilt;

    Building(String owner, int yearBuilt) {
        this.owner = owner;
        this.yearBuilt = yearBuilt;
    }

    public String getOwner() {
        return owner;
    }

    public void setOwner(String owner) {
        this.owner = owner;
    }

    public int getYearBuilt() {
        return yearBuilt;
    }
}
```

Not every attribute must have its setter and getter

Having yearBuilt setter is not right in the design perspective. Why?

Passing objects to methods

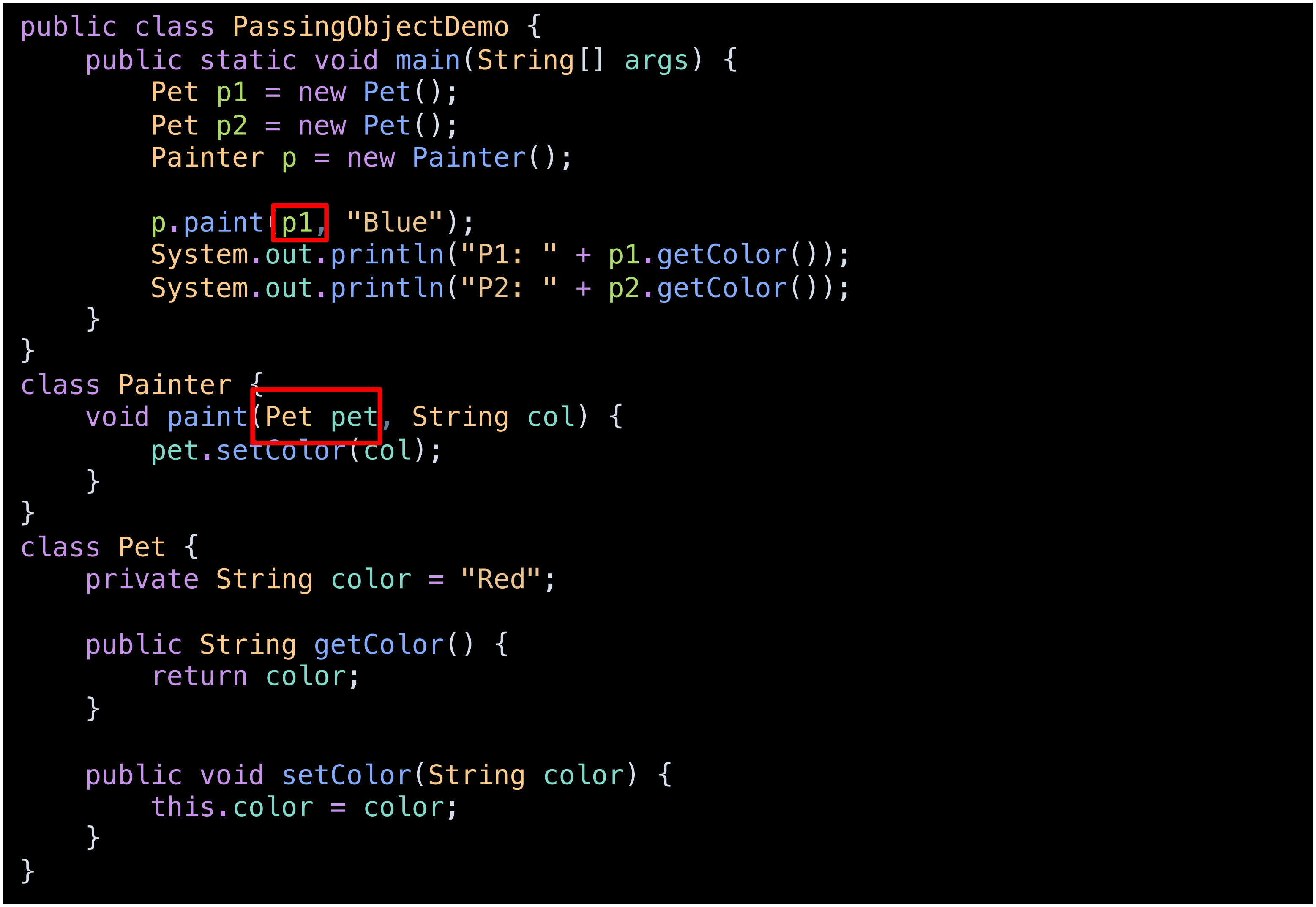
- Remember that a variable contains value (primitive) or reference to an object

- int a → a contains integer value
- int[] b → b contains reference to array object

- Primitive type: value is passed as an argument

- Reference type: value (reference to an object) is passed as an argument

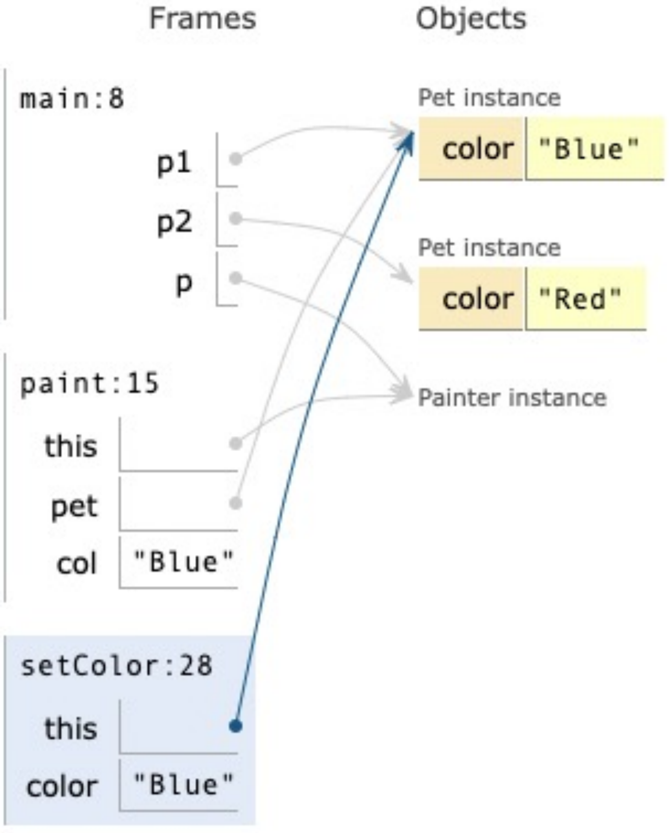
Passing objects to methods



```java
public class PassingObjectDemo {
    public static void main(String[] args) {
        Pet p1 = new Pet();
        Pet p2 = new Pet();
        Painter p = new Painter();

        p.paint(p1, "Blue");
        System.out.println("P1: " + p1.getColor());
        System.out.println("P2: " + p2.getColor());
    }
}
class Painter {
    void paint(Pet pet, String col) {
        pet.setColor(col);
    }
}
class Pet {
    private String color = "Red";

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

Visualize Your Code in pythontutor

Yes, you heard it right. Our beloved pythontutor can visualize java code


Frames
Objects
main:8
p1
p2
p
Pet instance
color "Blue"
Pet instance
color "Red"
Painter instance
paint:15
this
pet
col "Blue"
setColor:28
this
color "Blue"