ÖZYEĞİN UNIVERSITY

CS105
Introduction to Object-Oriented Programming

Prof. Dr. Nizamettin AYDIN

naydin@itu.edu.tr

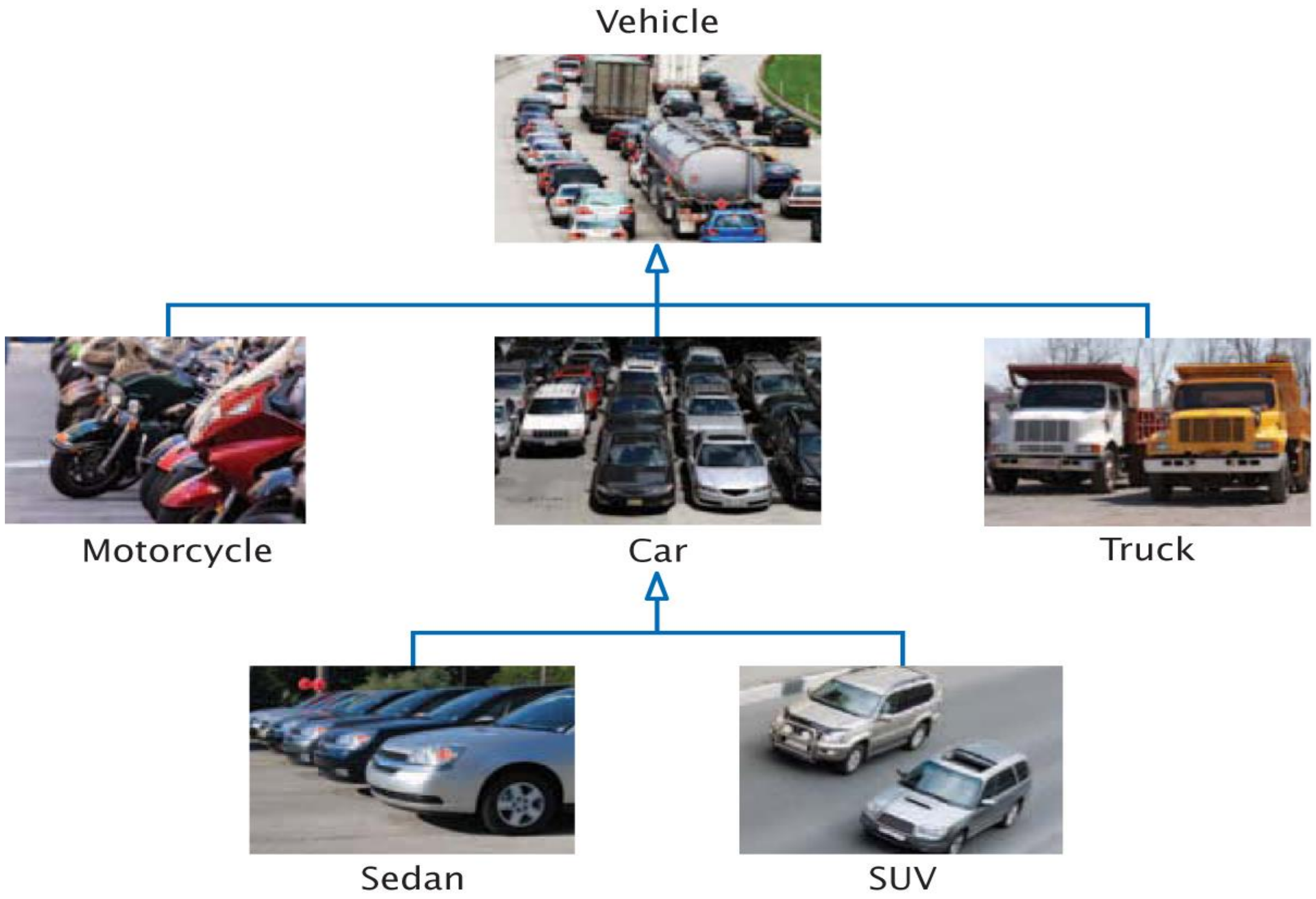
nizamettin.aydin@ozyegin.edu.tr

Inheritance

Outline

- Inheritance Hierarchies
- Inheritance
- protected Members
- Class Hierarchy
- Extending from Object Class
- Derived Classes
- Constructors
- super();
- Constructor Call
- Overriding (Overwriting)
- Overloading vs. Overriding

Inheritance Hierarchies



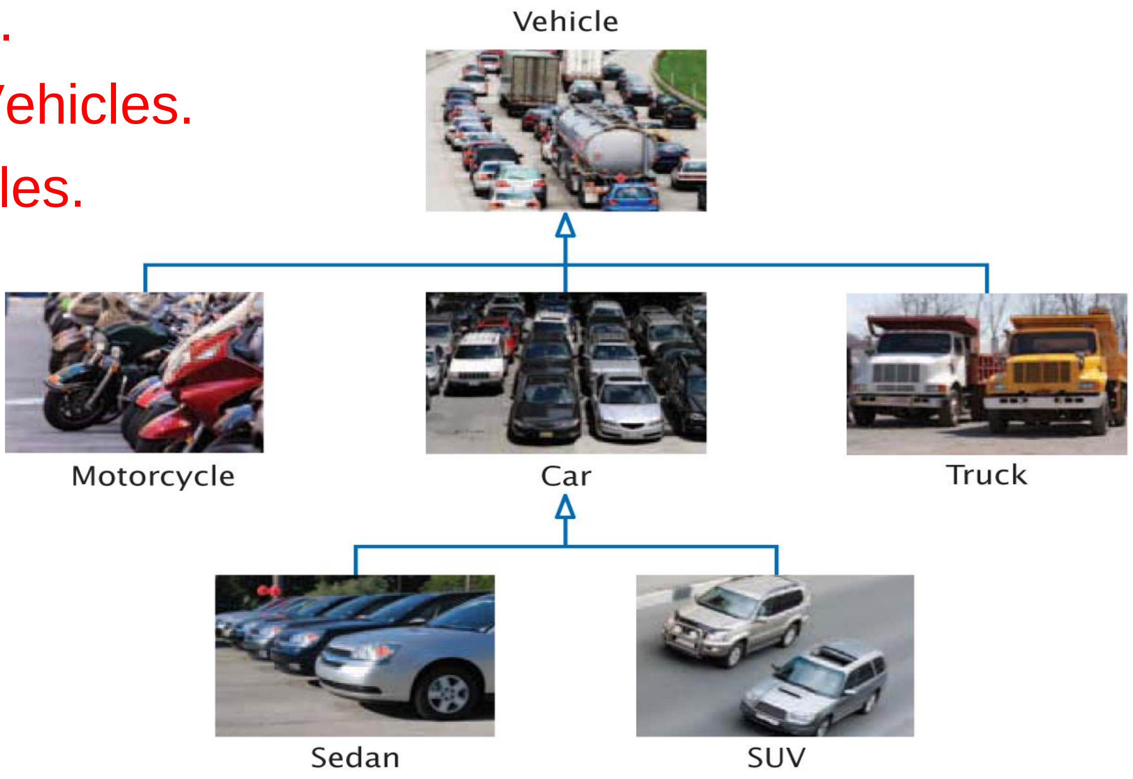
Inheritance

- In object-oriented design, inheritance is a relationship between
  - a more general class (called the base class)

  and
  - a more specialized class (called the derived class).
- Classes are created from existing ones
  - Absorbing attributes and behaviors
    - The derived class inherits data and behavior from the base class.
    - Adding new capabilities
- Software reusability
- Every car (in previous slide) is a vehicle.
- **IS-A**
  - denotes inheritance.

Inheritance

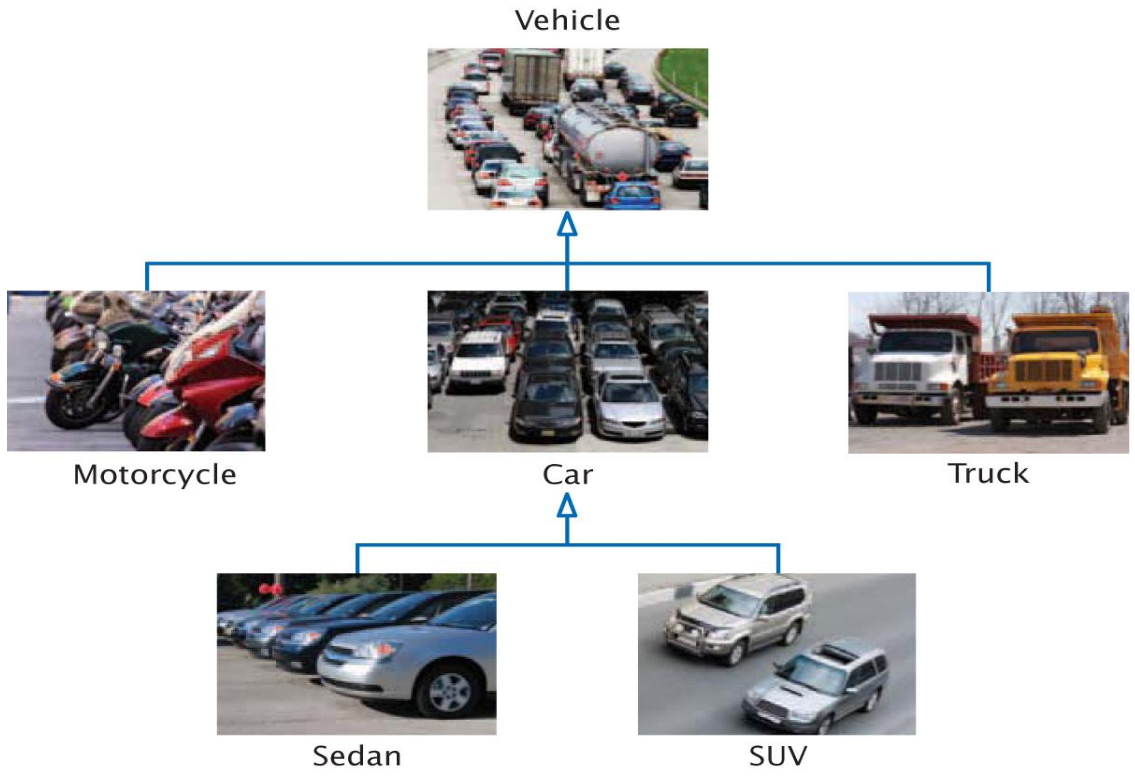
- **The IS-A Relationship**
  - –All Cars are Vehicles.
  - –All Motorcycles are Vehicles.
  - –All Sedans are Vehicles.



- `Vehicles` is the *base* class.
- `Car` is a *derived* class.
- `Truck` *derives* from `Vehicle`

Inheritance

- Everything about being a Vehicle is inherited by Cars and Trucks and SUVs.



- Those things specific to Cars are only inherited by Sedans and SUVs.

Inheritance

- **The Substitution Principle:**
- The substitution principle states that you can always use a derived-class object when a base-class object is expected.
- Suppose we have an algorithm or function that manipulates a Vehicle object.
- Since a car IS-A vehicle, we can supply a Car object to such an algorithm or function, and it will work correctly.

Inheritance

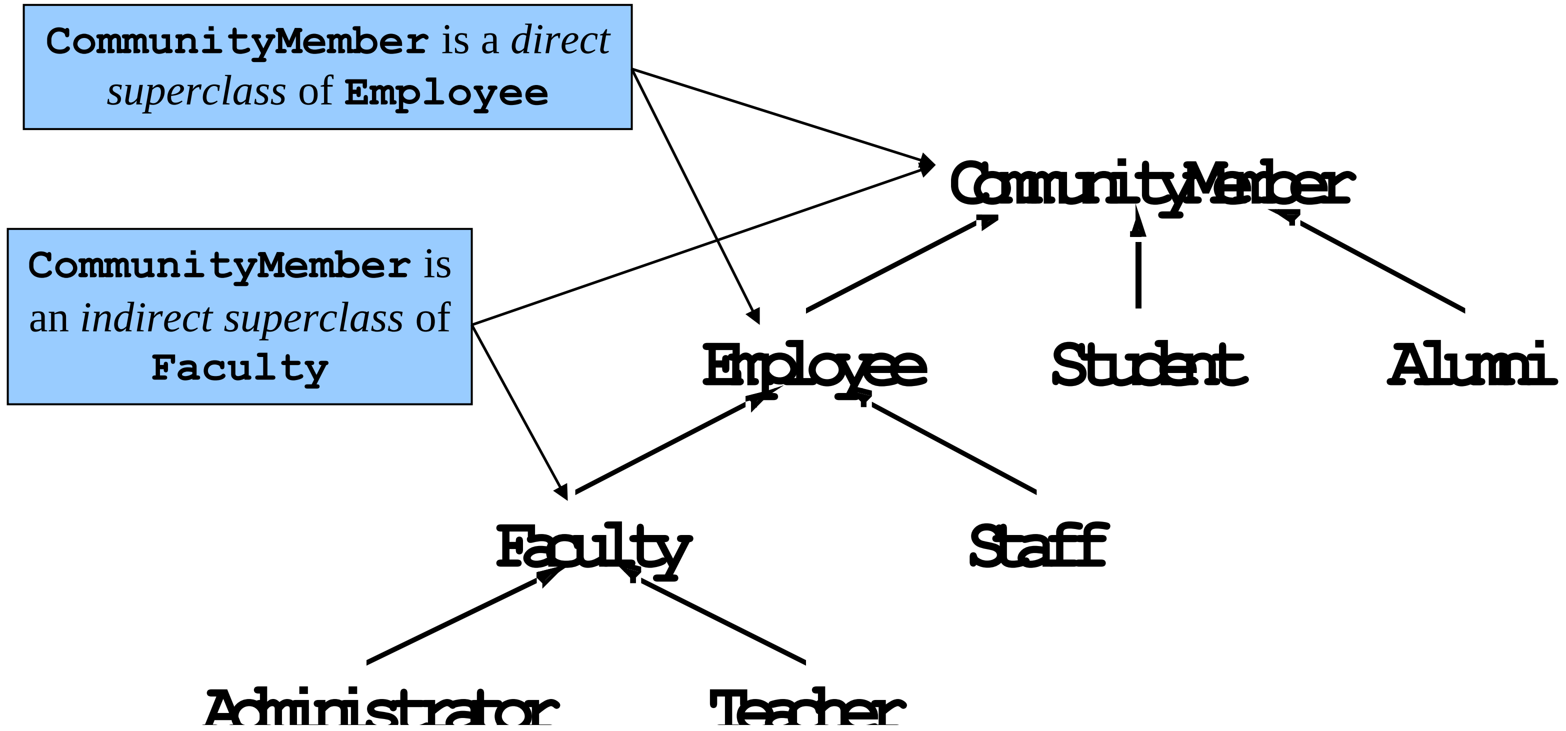
- **Superclasses and Subclasses**
- "Is a" Relationship
  - Object "is an" object of another class
    - Rectangle "is a" quadrilateral
      - Class Rectangle inherits from class Quadrilateral
  - Form tree-like hierarchical structures

Superclass	Subclasses
Student	**GraduateStudent** **UndergraduateStudent**
Shape	**Circle** **Triangle** **Rectangle**
Loan	**CarLoan** **HomeImprovementLoan** **MortgageLoan**
Employee	**FacultyMember** **StaffMember**
Account	**CheckingAccount** **SavingsAccount**

Some simple inheritance examples in which the subclass "is a" superclass.

Inheritance

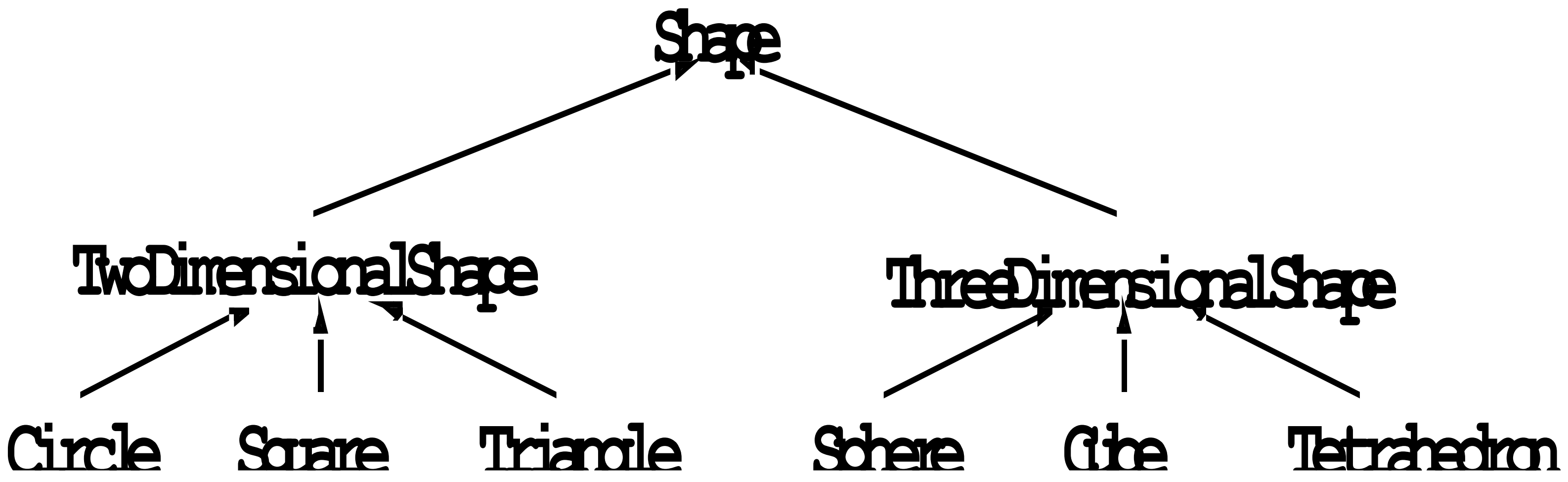
- An inheritance hierarchy for university `CommunityMembers`.



CommunityMember is a *direct superclass* of **Employee**
CommunityMember is an *indirect superclass* of **Faculty**
CommunityMember
Employee
Student
Alumni
Faculty
Staff
Administrator
Teacher

Inheritance

- A portion of a Shape class hierarchy.



protected Members

- **protected** access members
  - Between `public` and `private` in protection
  - Accessed only by
    - Superclass methods
    - Subclass methods
    - Methods of classes in same package
      - package access
- Relationship between Superclass and Subclass Objects:
- Subclass object
  - can be treated as superclass object
    - Reverse is not true
      - Shape is not always a Circle
  - Every class implicitly extends `java.lang.Object`
    - Unless specified otherwise in class definition’s first line

```
1     // Fig. 9.4: Point.java
2     // Definition of class Point
3
4     public class Point {
5        protected int x, y; // coordinates
6
7        // No-argument constructor
8        public Point()
9        {
10          // implicit call to superclass constructor occurs here
11          setPoint( 0, 0 );
12       }
13
14       // constructor
15       public Point( int xCoordinate, int yCoordinate )
16       {
17          // implicit call to superclass constructor occurs here
18          setPoint( xCoordinate, yCoordinate );
19       }
20
21       // set x and y coordinates of Point
22       public void setPoint( int xCoordinate, int yCoordinate )
23       {
24          x = xCoordinate;
25          y = yCoordinate;
26       }
27
28       // get x coordinate
29       public int getX()
30       {
31          return x;
32       }
33
```

`protected` members prevent clients from direct access (unless clients are `Point` subclasses or are in same package)

Point.java

Line 5
protected members prevent clients from direct access (unless clients are **Point** subclasses or are in same package)

```java
   // get y coordinate
   public int getY()
   {
      return y;
   }

   // convert into a String representation
   public String toString()
   {
      return "[" + x + ", " + y + "]";
   }

} // end class Point
```

Point.java

```java
1     // Fig. 9.5: Circle.java
2     // Definition of class Circle
3
4     public class Circle extends Point {   // inherits from Point
5        protected double radius;
6
7        // no-argument constructor
8        public Circle()
9        {
10          // implicit call to superclass constructor occurs here
11          setRadius( 0 );
12       }
13
14       // constructor
15       public Circle( double circleRadius, int xCoordinate,
16          int yCoordinate )
17       {
18          // call superclass constructor to set coordinates
19          super( xCoordinate, yCoordinate );
20
21          // set radius
22          setRadius( circleRadius );
23       }
24
25       // set radius of Circle
26       public void setRadius( double circleRadius )
27       {
28          radius = ( circleRadius >= 0.0 ? circleRadius : 0.0 );
29       }
30
```

`Circle` is a **`Point`** subclass

`Circle` inherits **`Point`**’s **`protected`** variables and **`public`** methods (except for constuctor)

Implicit call to **`Point`** constructor

Explicit call to **`Point`** constructor using **`super`**

Circle.java

Line 4
Circle is a **Point**
ubclass

ine 4
ircle inherits
Point’s
protected
and
ethods
(except for
constuctor)

Line 10
Implicit call to
Point
constructor

Line 19
Explicit call to
Point
constructor using
super

```
31      // get radius of Circle
32      public double getRadius()
33      {
34         return radius;
35      }
36
37      // calculate area of Circle
38      public double area()
39      {
40         return Math.PI * radius * radius;
41      }
42
43      // convert the Circle to a String
44      public String toString()
45      {
46         return "Center = " + "[" + x + ", " + y + "]" +
47                "; Radius = " + radius;
48      }
49
50   }  // end class Circle
```

Override method **`toString`** of class **`Point`** by using same signature

Circle.java

Lines 44-48
Override method
toString of class
nt by using
ne signature

```java
1    // Fig. 9.6: InheritanceTest.java
2    // Demonstrating the "is a" relationship
3
4    // Java core packages
5    import java.text.DecimalFormat;
6
7    // Java extension packages
8    import javax.swing.JOptionPane;
9
10   public class InheritanceTest {
11
12      // test classes Point and Circle
13      public static void main( String args[] )
14      {
15         Point point1, point2;
16         Circle circle1, circle2;
17
18         point1 = new Point( 30, 50 );
19         circle1 = new Circle( 2.7, 120, 89 );
20
21         String output = "Point point1: " + point1.toString() +
22            "\nCircle circle1: " + circle1.toString();
23
24         // use "is a" relationship to refer to a Circ
25         // with a Point reference
26         point2 = circle1;  // assigns Circle to a Point reference
27
28         output += "\n\nCircle circle1 (via point2 r
29            point2.toString();
30
31         // use downcasting (casting a superclass reference to a
32         // subclass data type) to assign point2 to circle2
33         circle2 = ( Circle ) point2;
34
```

Instantiate **Point** and **Circle** objects

Circle invokes its overridden **toString** method

Superclass object can reference subclass object

Point still invokes **Circle**'s overridden **toString** method

Downcast **Point** to **Circle**

InheritanceTest.java

Lines 18-19
Instantiate
objects

Line 22
Circle invokes
method **toString**

ss
object
references
ss

9
Point invokes

method

Line 33
Downcast **Point**

```
35          output += "\n\nCircle circle1 (vi
36             circle2.toString();
37
38          DecimalFormat precision2 = new DecimalFormat( "0.00" );
39          output += "\nArea of c (via circle2): " +
40             precision2.format( circle2.area() );
41
42          // attempt to refer to Point object with Circle reference
43          if ( point1 instanceof Circle ) {
44             circle2 = ( Circle ) point1;
45             output += "\n\ncast successful";
46          }
47          else
48             output += "\n\npoint1 does not refer to a Circle";
49
50          JOptionPane.showMessageDialog( null, output,
51             "Demonstrating the \"is a\" relationship",
52             JOptionPane.INFORMATION_MESSAGE );
53
54          System.exit( 0 );
55       }
56
57    }  // end class InheritanceTest
```

Circle invokes its overridden **toString** method

Circle invokes method **area**

Use **instanceof** to determine if **Point** refers to **Circle**

If **Point** refers to **Circle**, cast **Point** as **Circle**

InheritanceTest.java

Line 36

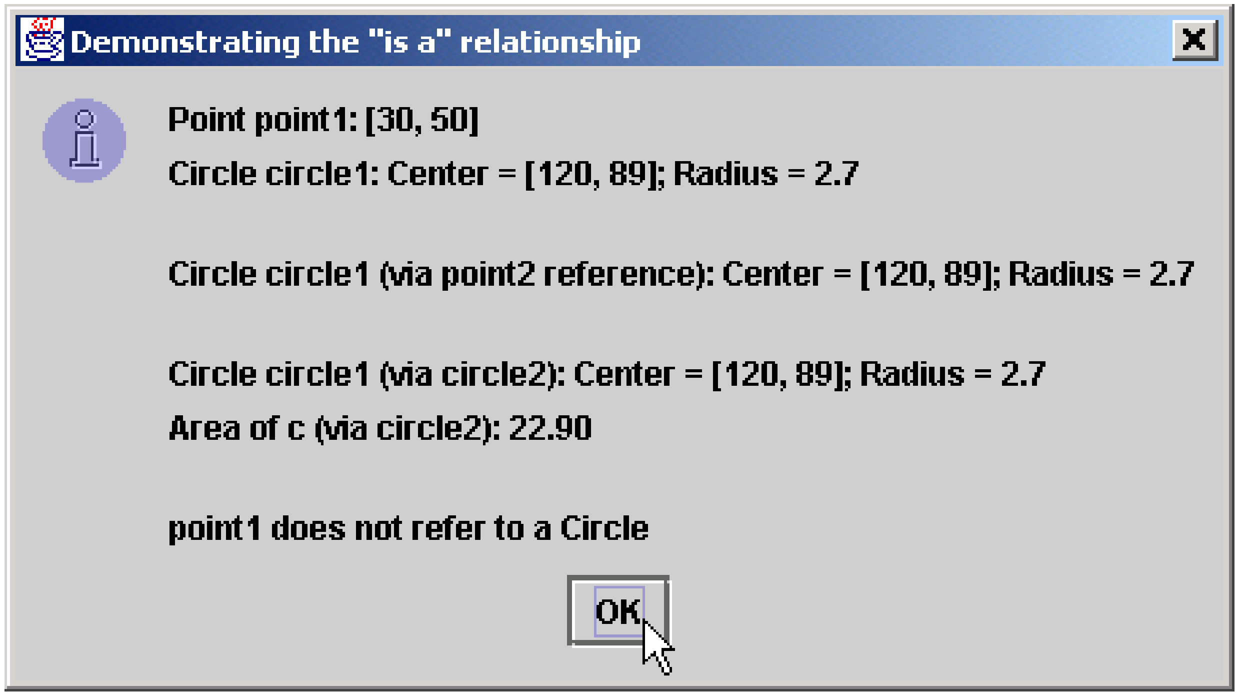
kes
its overridden
toString method

kes
method **area**

eof
f
Point refers to
Circle

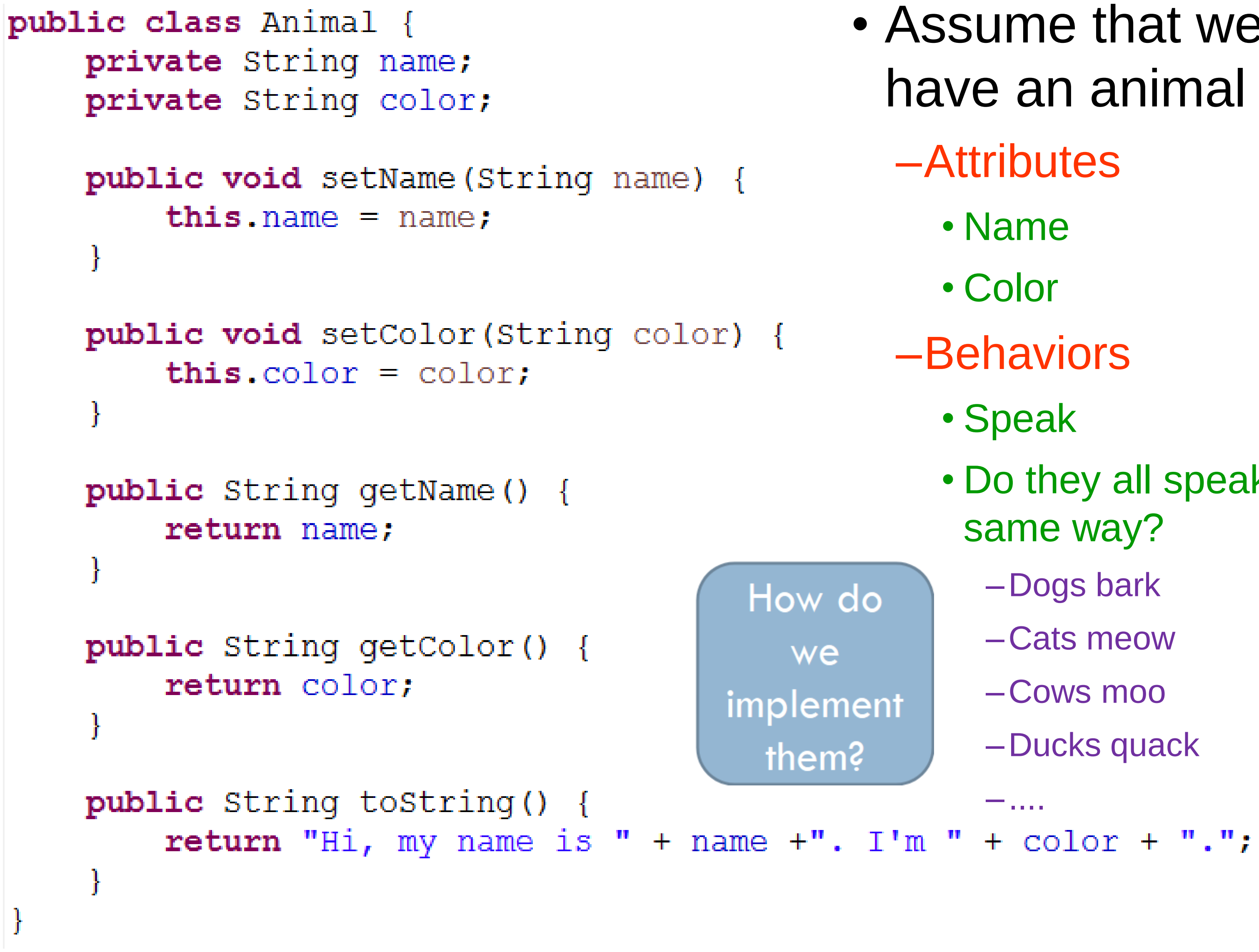
Line 44
If **Point** refers to
Circle, cast
Point as **Circle**

- Assigning subclass references to superclass references



```
Point point1: [30, 50]
Circle circle1: Center = [120, 89]; Radius = 2.7

Circle circle1 (via point2 reference): Center = [120, 89]; Radius = 2.7

Circle circle1 (via circle2): Center = [120, 89]; Radius = 2.7
Area of c (via circle2): 22.90

point1 does not refer to a Circle
```

Animal Class



```java
public class Animal {
    private String name;
    private String color;

    public void setName(String name) {
        this.name = name;
    }

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

    public String getName() {
        return name;
    }

    public String getColor() {
        return color;
    }

    public String toString() {
        return "Hi, my name is " + name +". I'm " + color + ".";
    }
}
```

- Assume that we have an animal class
  - Attributes
    - Name
    - Color
  - Behaviors
    - Speak
    - Do they all speak the same way?
      - Dogs bark
      - Cats meow
      - Cows moo
      - Ducks quack
      - ....

How do we implement them?

Different type of animals

- We will talk about two bad solutions:
  - –First way:
    - Inside the same class
  - –Second way:
    - Writing a different class

First Way – Inside the same class

- Need to hold an additional attribute:
  - –Type of the animal

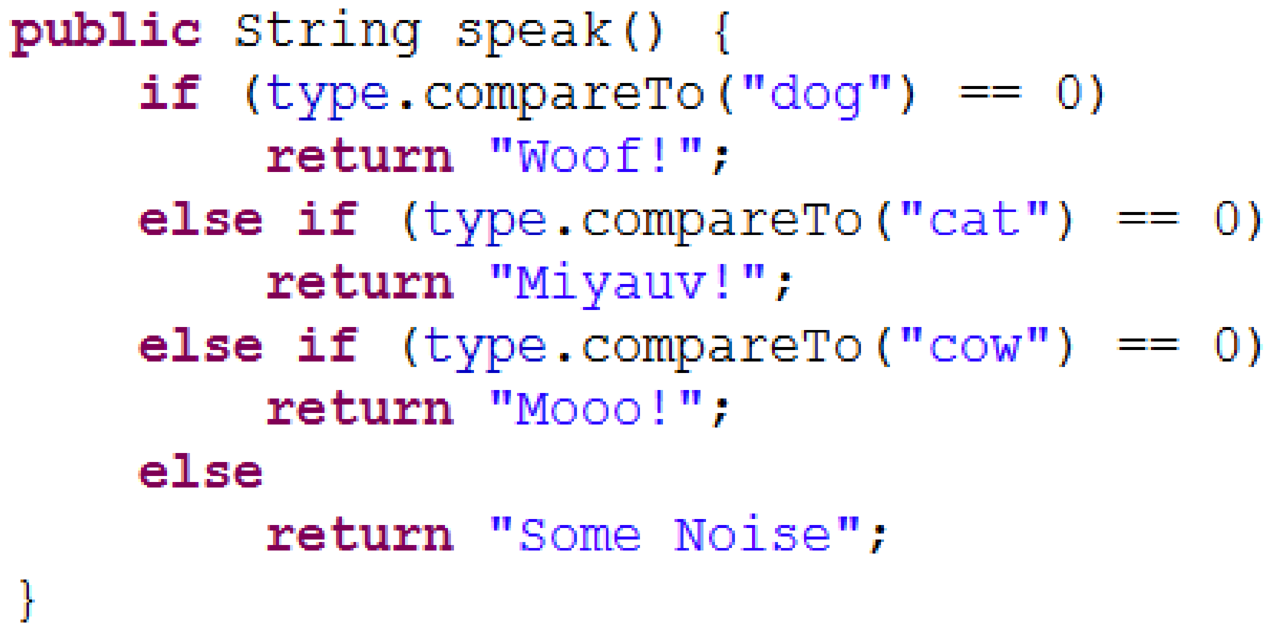
```
public class Animal {
    private String name;
    private String color;
    private String type;

    public Animal (String name, String type) {
        this.name = name;
        this.type = type;
    }
```

- Main:

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

    Animal cat = new Animal("Serafettin", "cat");
    Animal dog = new Animal("Scooby", "dog");
    Animal cow = new Animal("Sarı Kız", "cow");
```

First Way – Inside the same class

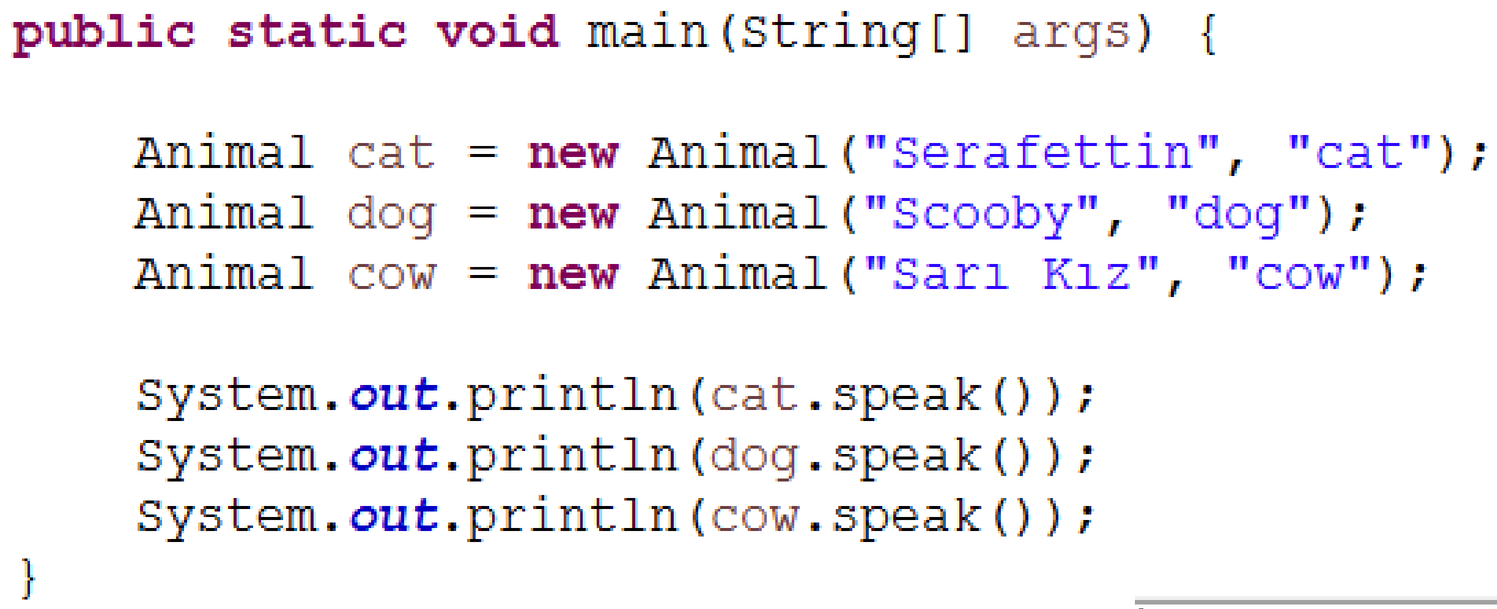
- Speak function
  - –Check the type of the animal and speak accordingly



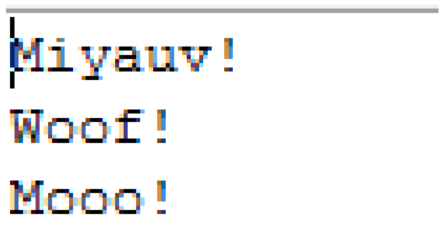
```
public String speak() {
    if (type.compareTo("dog") == 0)
        return "Woof!";
    else if (type.compareTo("cat") == 0)
        return "Miyauv!";
    else if (type.compareTo("cow") == 0)
        return "Mooo!";
    else
        return "Some Noise";
}
```

First Way – Inside the same class

- Main:



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

    Animal cat = new Animal("Serafettin", "cat");
    Animal dog = new Animal("Scooby", "dog");
    Animal cow = new Animal("Sarı Kız", "cow");

    System.out.println(cat.speak());
    System.out.println(dog.speak());
    System.out.println(cow.speak());
}
```



```
Miyauv!
Woof!
Mooo!
```

First Way – Problem 1

- Many dog types, all bark differently
  - –Loudness
  - –Pace
  - –...

- Do we need to define another dog type variable?

Source: https://www.popchartlab.com/products/the-diagram-of-dogs

```
private String dogType;
```

First Way – Problem 1

- Many dog types, all bark differently
  - –Loudness
  - –Pace
  - –...

```
private String dogType;
```

Ugly Code ☹

```
public String speak() {
    if (type.compareTo("dog") == 0) {
        if (dogType.compareTo("kangal dog") == 0)
            return "Loud Woof!";
        if (dogType.compareTo("chow dog") == 0)
            return "Cute Woof!";
    }
    else if (type.compareTo("cat") == 0)
        return "Miyauv!";
    else if (type.compareTo("cow") == 0)
        return "Mooo!";
    else
        return "Some Noise";
}
```

First Way – Problem 2

- Jumping is behavior of some animals

```java
public void jump() {
    System.out.println(this.name + "jumped!");
}
```

- cat, dog can jump but not the fish...
- In main, we should not call jump for fish, but right now we can as follows:

```java
Animal fish = new Animal("Nemo", "fish");
fish.jump();
```

❌

Different type of animals

- We will talk about two bad solutions:
  - First way:
    - Inside the same class ✗
  - Second way:
    - Writing a different class
- **Second way: Class for each type**
  - We will have individual classes for each animal.

Second way: Class for each type

Cat

```java
public class Cat {
    private String name;
    private String color;

    public Cat (String name) {
        this.name = name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String getName() {
        return name;
    }
    public String getColor() {
        return color;
    }
    public String speak() {
        return "Miyauv";
    }
}
```

Dog

```java
public class Dog {
    private String name;
    private String color;

    public Dog (String name) {
        this.name = name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String getName() {
        return name;
    }
    public String getColor() {
        return color;
    }
    public String speak() {
        return "Woof";
```

Code repetition ☹.

It is hard to keep the common code consistent

Different type of animals

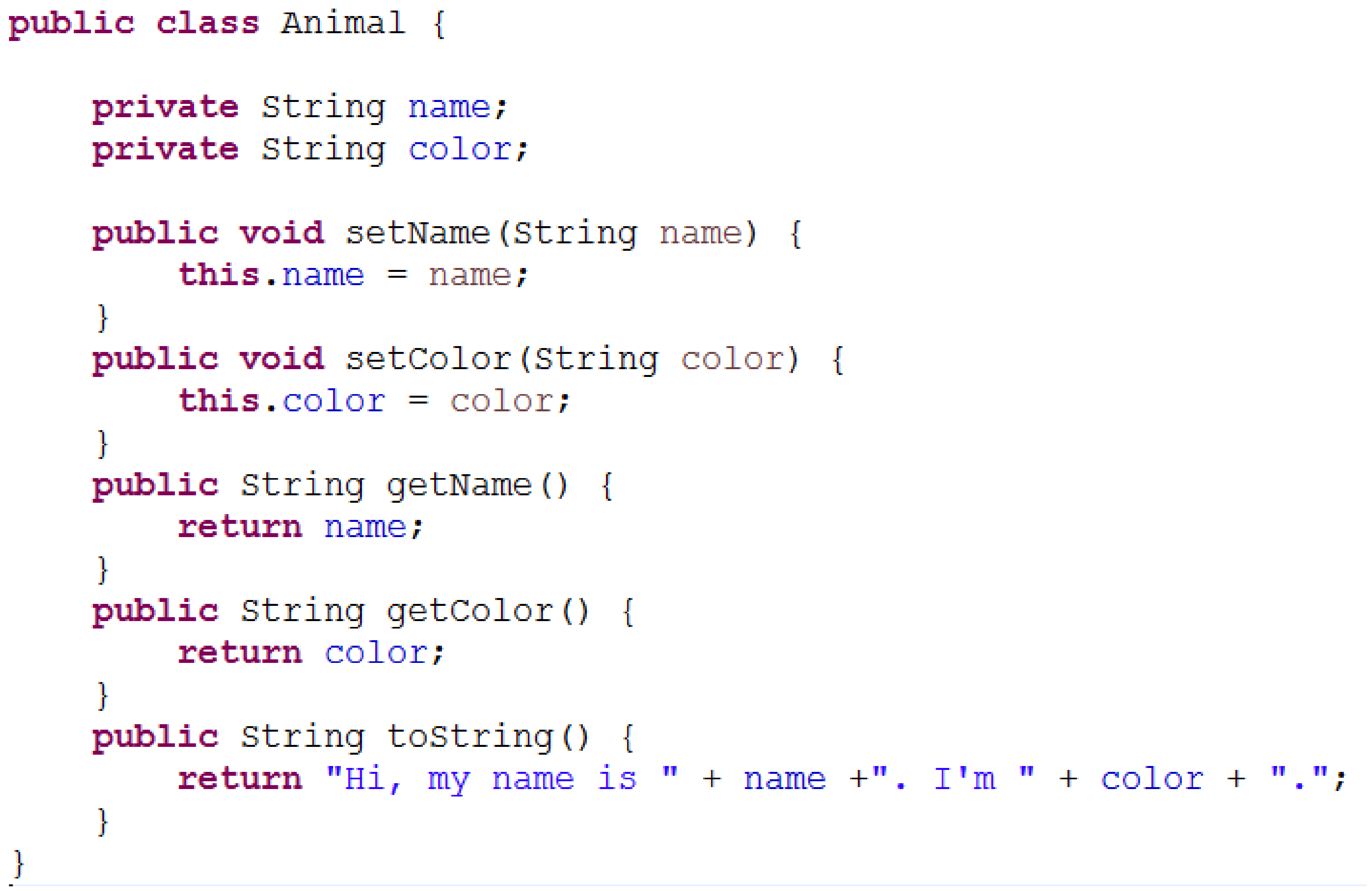
- We will talk about two bad solutions:
  - First way: ✗
    - Inside the same class
  - Second way: ✗
    - Writing a different class

- A good solution is to use inheritance ✓
  - Keep the common attributes and functionalities in one class
  - Split only the different attributes and functionalities in different classes.

Inheritance

- A class can inherit some of its attibutes and behaviors from another class.
- A derived class inherits from the base class.
- A sub class inherits from/extends the super class.
- Keep the common attributes and functionalities in one class
  - –Animal class
    - name and color
    - setter and getter functions
- Split only the different attributes and functionalities in different classes.
  - –Cat, dog, cow ... classes
    - speak, jump function

Animal Class



```java
public class Animal {

    private String name;
    private String color;

    public void setName(String name) {
        this.name = name;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String getName() {
        return name;
    }
    public String getColor() {
        return color;
    }
    public String toString() {
        return "Hi, my name is " + name +". I'm " + color + ".";
    }
}
```

Cat and Dog Classes

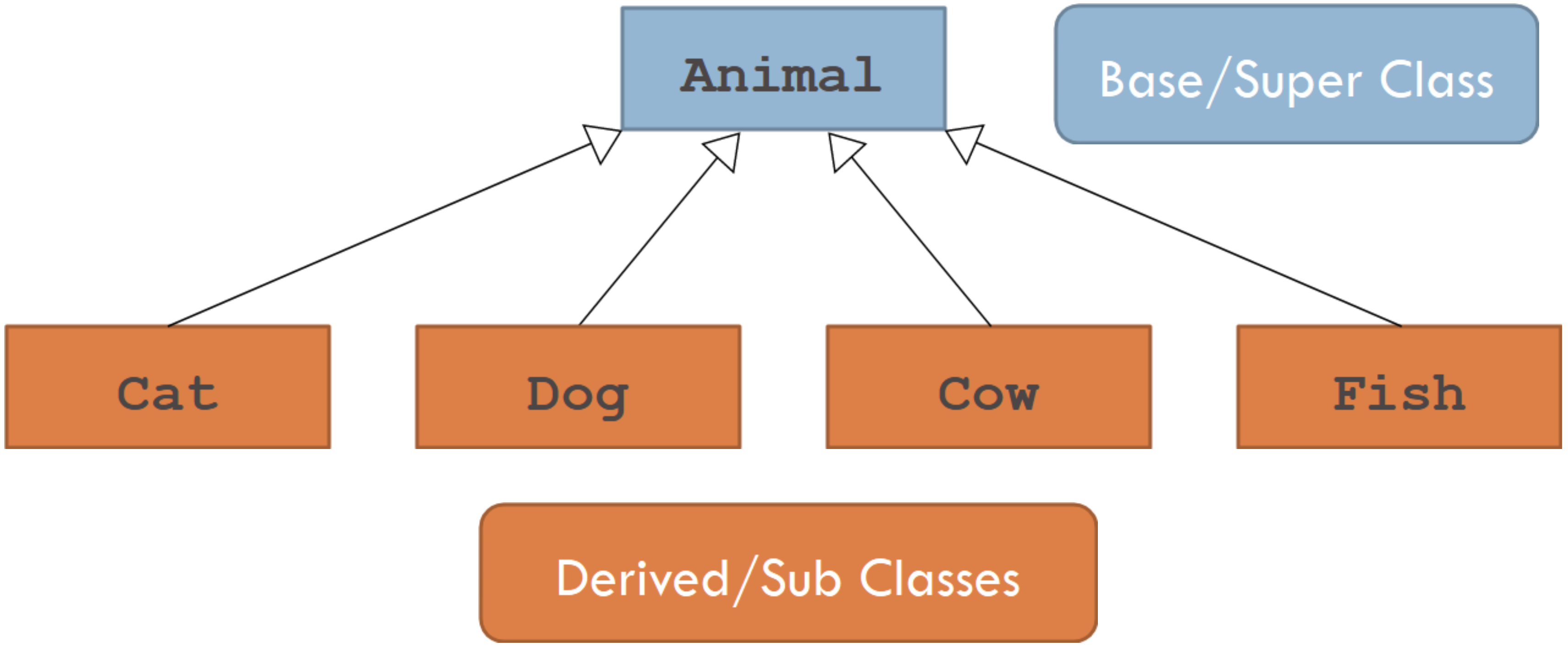
```
public class Cat extends Animal {
    public Cat(String name) {
        setName(name);
        setColor("gray");
    }
}
```

inherits from

```
public class Dog extends Animal {
    public Dog(String name) {
        setName(name);
        setColor("gray");
    }
}
```

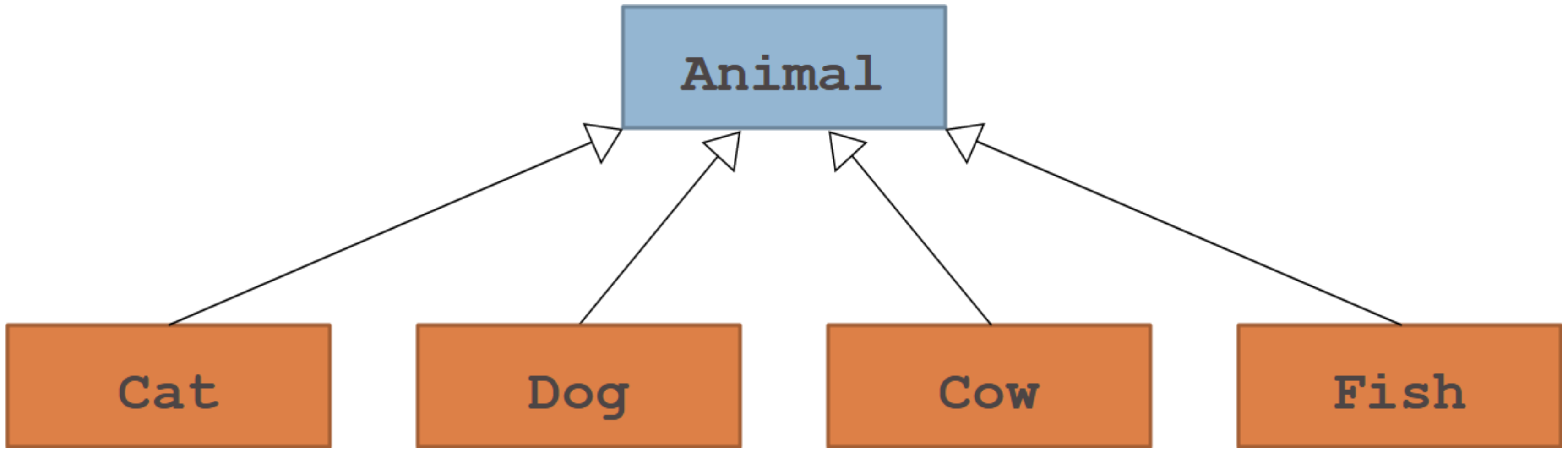
Class Hierarchy

- Classes in Java form **hierarchies**.
- Animal class represent all animal objects.



- The four subclasses correspond to particular animal type object.

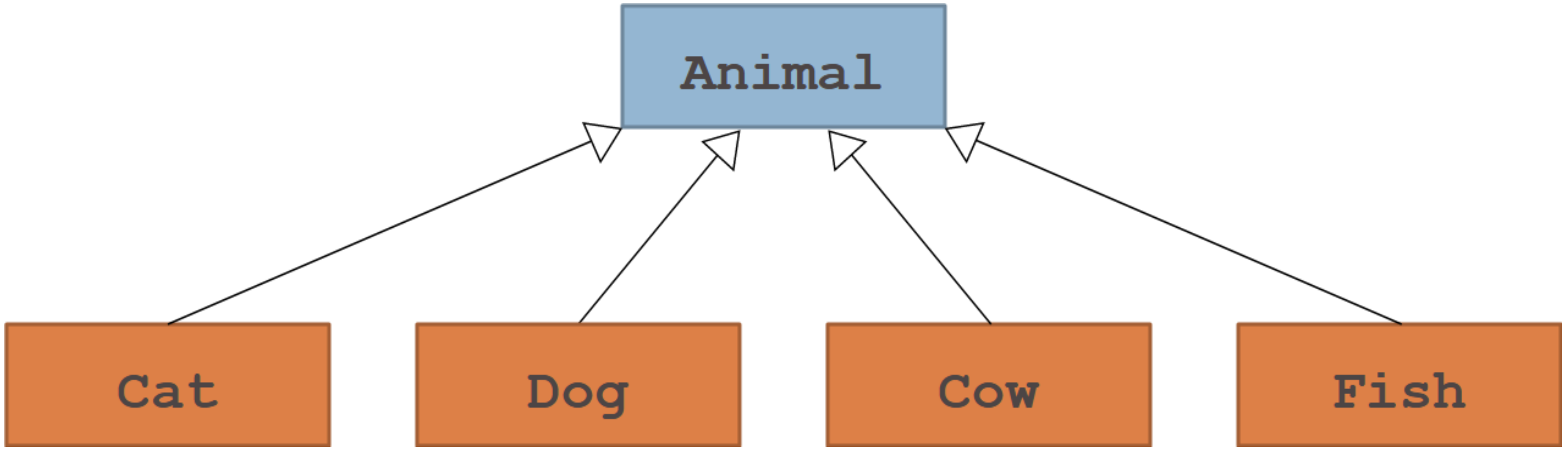
Class Hierarchy



- This class diagram shows that Cat, Dog, Cow and Fish is also an Animal but the inverse is **NOT TRUE**.
- Any animal is not a Cat, any Animal is not a Dog, etc.

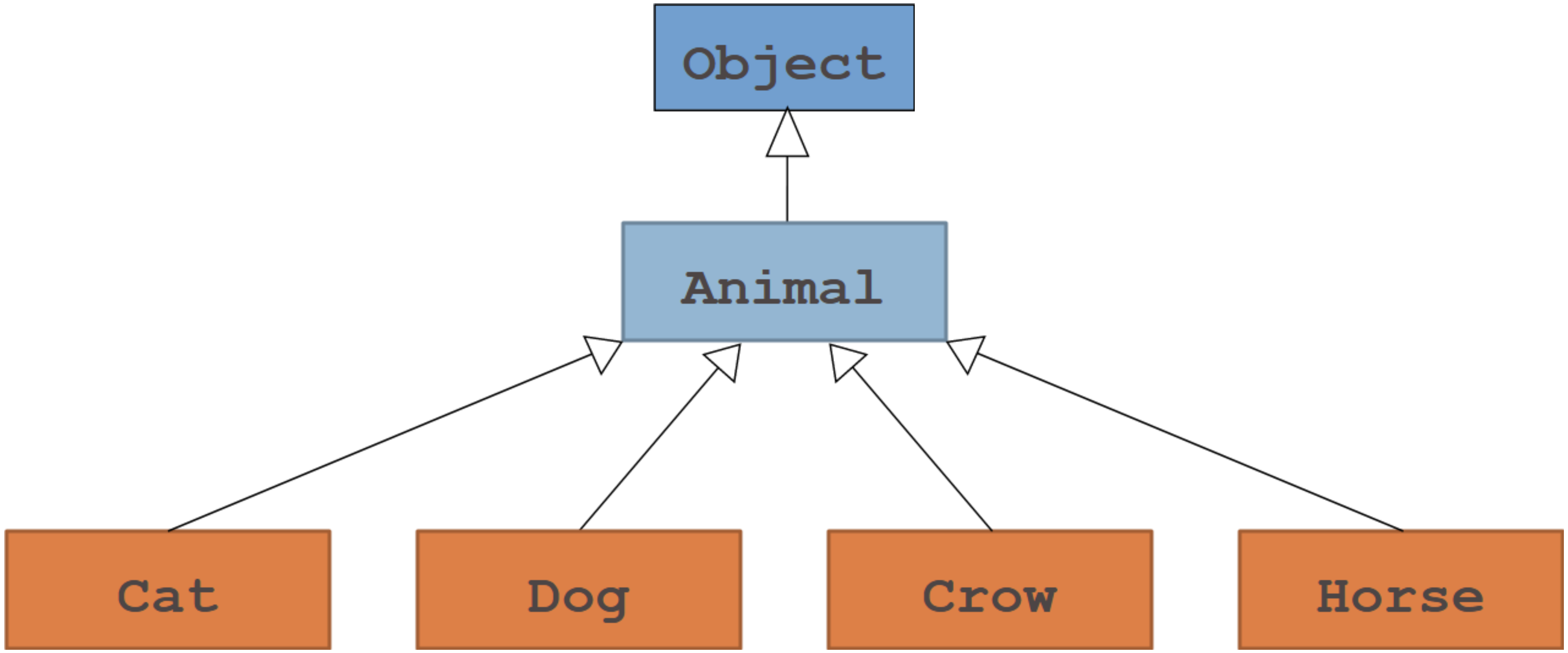
Class Hierarchy

- Can the Animal class be also a derived class?



- When you define a new class in Java, that class automatically **inherits** the behavior of its superclass.
- If no superclass is defined, by default, the class will inherit from the **Object** class.

Class Hierarchy



Extending from Object Class

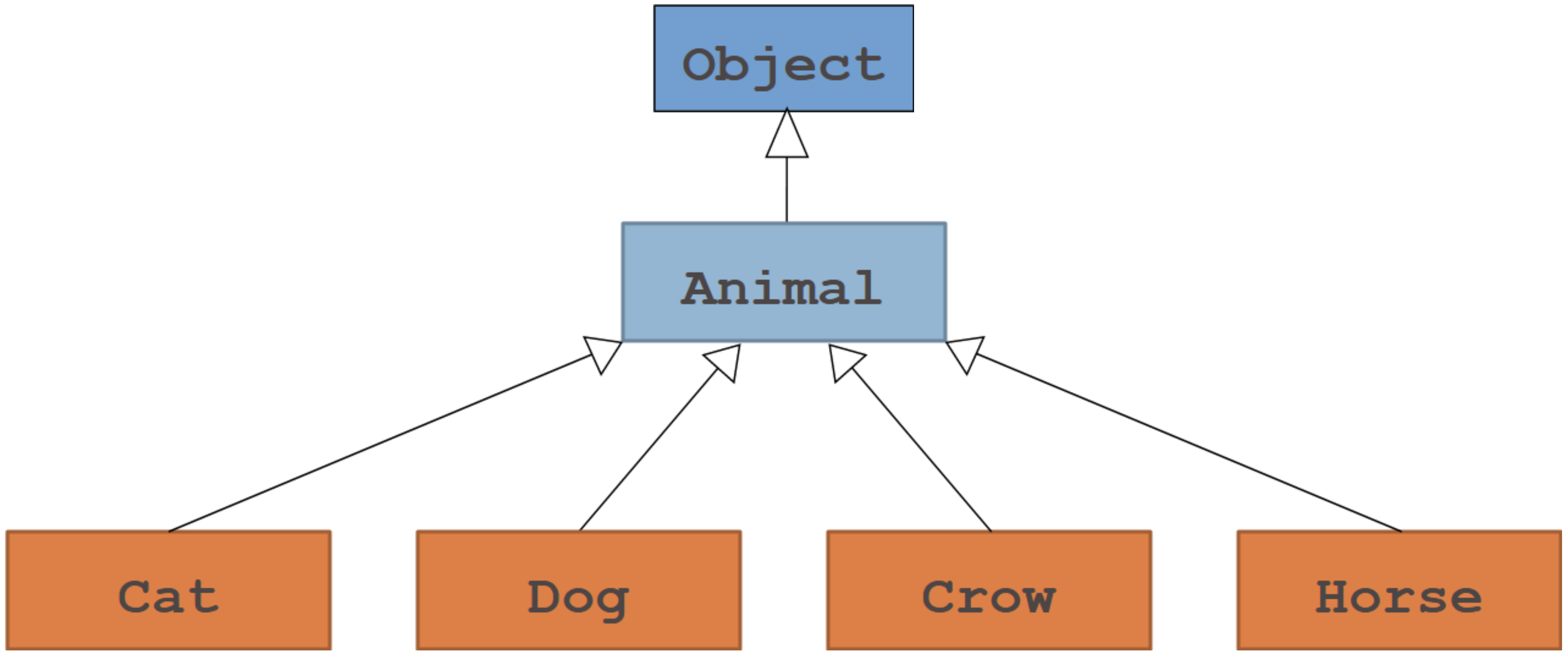
```
public class Animal { ...
```

- is the same as

```
public class Animal extends Object { ...
```

- The **extends** clause on the header line specifies the name of the superclass.
- If the **extends** clause is missing, the new class becomes a direct subclass of **Object**, which is the root of Java’s class hierarchy.

Class Hierarchy



- Except for the class named **Object** that stands at the top of the hierarchy, every class in Java is a **subclass** of some other class.

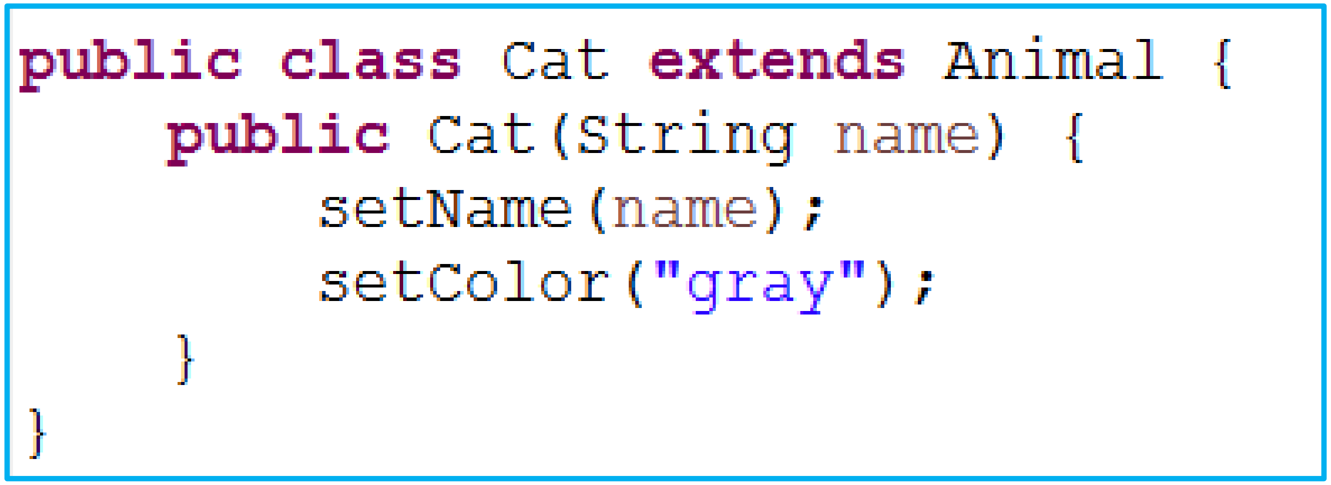
Lets get back to our Animal class

```
public class Animal {

    private String name;
    private String color;

    public void setName(String name) {
        this.name = name;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String getName() {
        return name;
    }
    public String getColor() {
        return color;
    }
    public String toString() {
        return "Hi, my name is " + name +". I'm " + color + ".";
    }
}
```

```
public class Cat extends Animal {
    public Cat(String name) {
        setName(name);
        setColor("gray");
    }
}
```

```
public class Dog extends Animal {
    public Dog(String name) {
        setName(name);
        setColor("gray");
    }
}
```

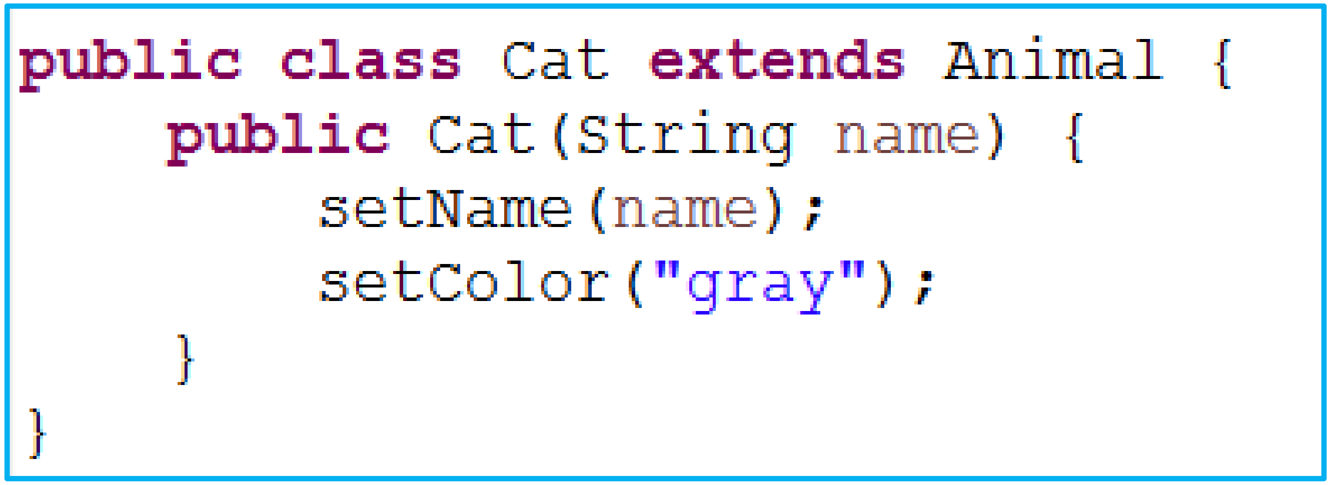
Derived Classes



```java
public class Cat extends Animal {
    public Cat(String name) {
        setName(name);
        setColor("gray");
    }
}
```

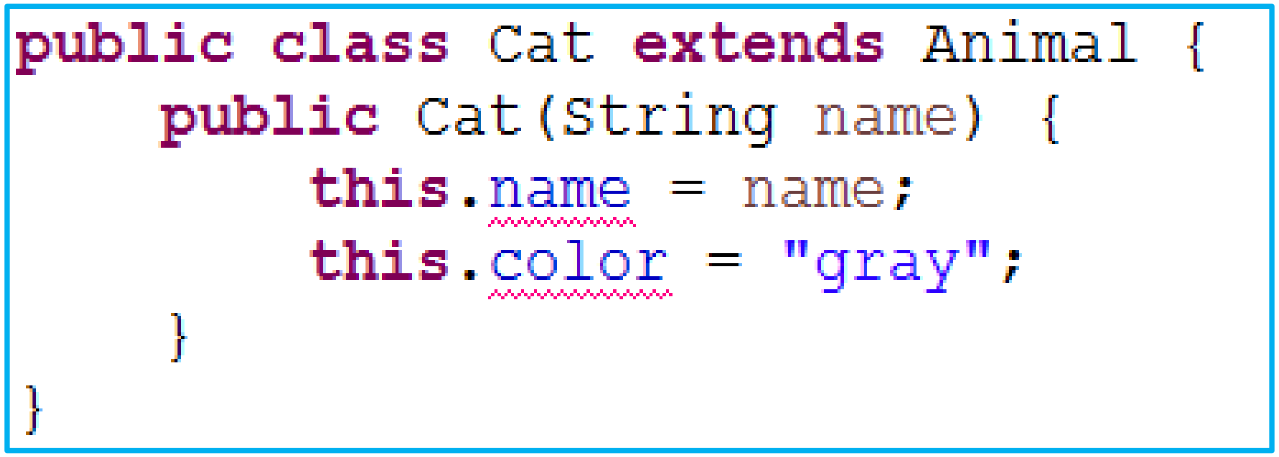
- Instead of calling the set methods can we just modify the name and color directly?

Derived Classes



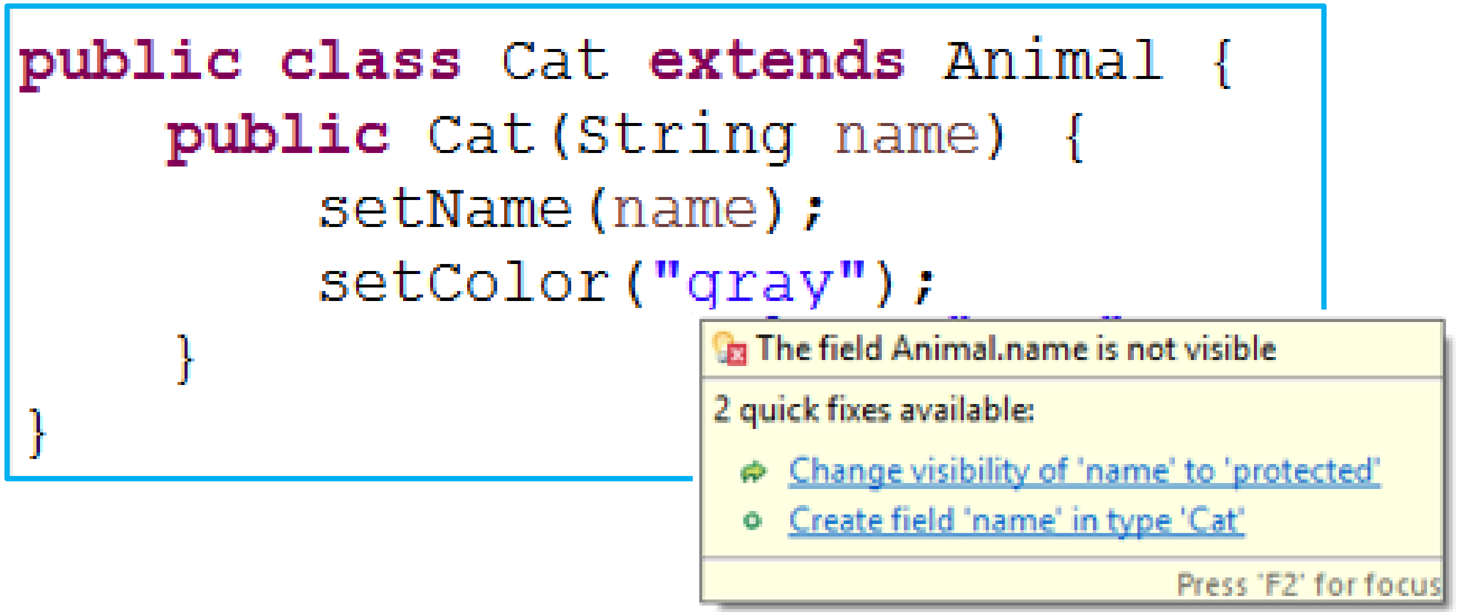
```java
public class Cat extends Animal {
    public Cat(String name) {
        setName(name);
        setColor("gray");
    }
}
```

- What is the problem?



```java
public class Cat extends Animal {
    public Cat(String name) {
        this.name = name;
        this.color = "gray";
    }
}
```

Derived Classes



```
public class Cat extends Animal {
    public Cat(String name) {
        setName(name);
        setColor("gray");
    }
}
```

The field Animal.name is not visible

2 quick fixes available:

- Change visibility of 'name' to 'protected'
- Create field 'name' in type 'Cat'

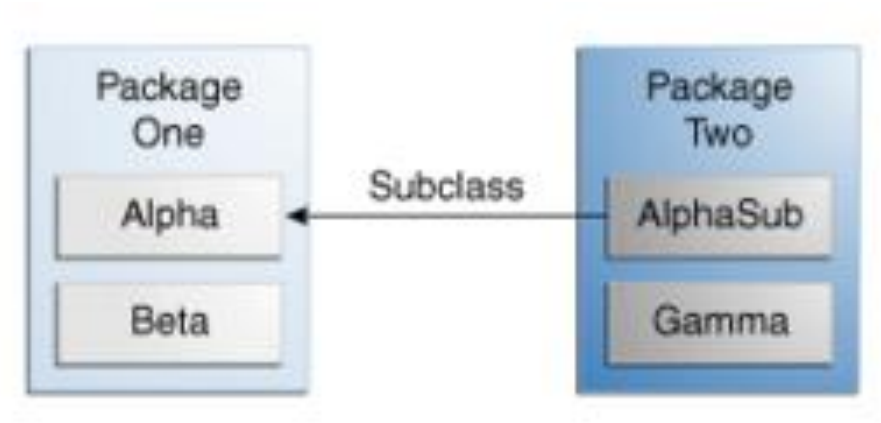
Press 'F2' for focus

- The name and color is private therefore cannot be accessed from the derived/sub class Cat.
- Therefore we need to use the getters and setters method to reach these private class instances.

What is inherited?

- All class instances and functions of the base/super class are inherited.
- But not all of them are visible from the sub class
  - Public and protected ones are visible
  - Default and private ones are NOT visible
    - Please note that default ones are visible if they are in the same package.
  - These can be accessed only through getter and setter functions.

Remember Visibility



Alpha	Beta	AlphaSub	Gamma
public	Y	Y	Y
protected	Y	Y	N
default	Y	N	N
private	N	N	N

Lets get back to our Animal class

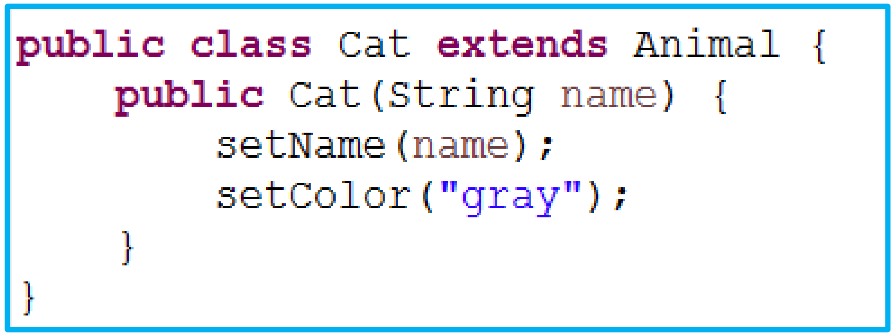
```
public class Animal {

    private String name;
    private String color;

    public void setName(String name) {
        this.name = name;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String getName() {
        return name;
    }
    public String getColor() {
        return color;
    }
    public String toString() {
        return "Hi, my name is " + name +". I'm " + color + ".";
    }
}
```

```
public class Cat extends Animal {
    public Cat(String name) {
        setName(name);
        setColor("gray");
    }
}
```

```
public class Dog extends Animal {
    public Dog(String name) {
        setName(name);
        setColor("gray");
    }
}
```

Constructors

- What happens inside this constructor?



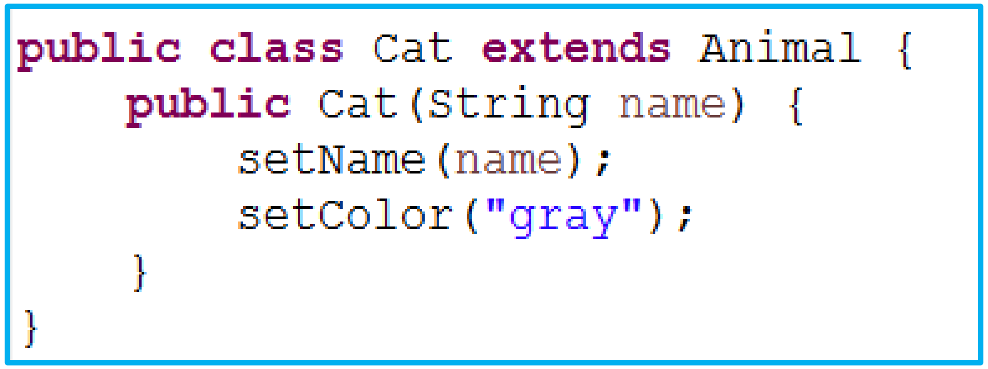
```java
public class Cat extends Animal {
    public Cat(String name) {
        setName(name);
        setColor("gray");
    }
}
```

- Initially the default constructor of the super class is called by compiler.

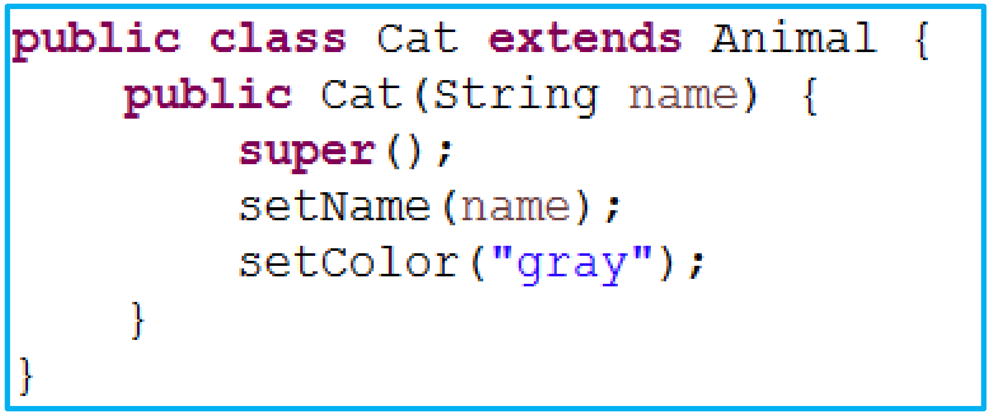
- Whenever you create an object of an extended class, Java must call some constructor for the super class object to ensure that its structure is correctly initialized.

Constructors

- Below two are the same!



```
public class Cat extends Animal {
    public Cat(String name) {
        setName(name);
        setColor("gray");
    }
}
```

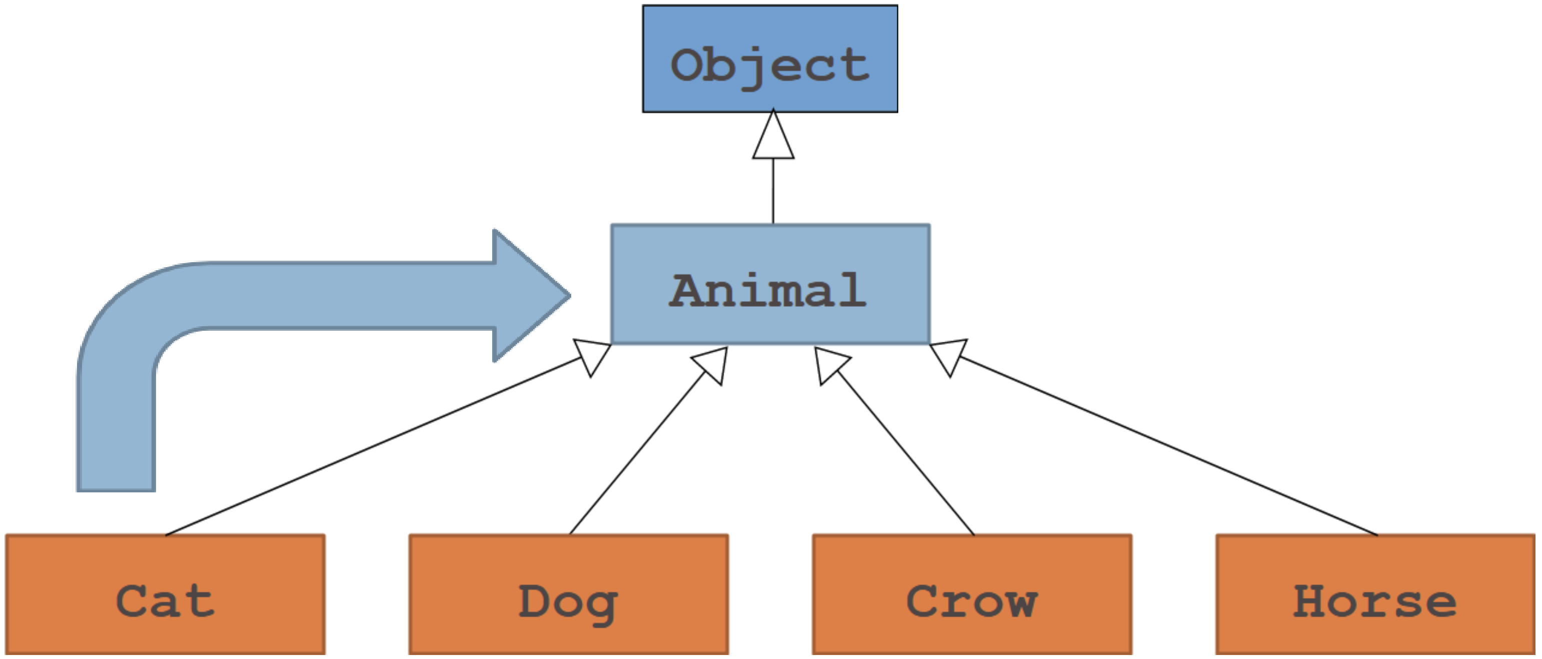


```
public class Cat extends Animal {
    public Cat(String name) {
        super();
        setName(name);
        setColor("gray");
    }
}
```

`super();`

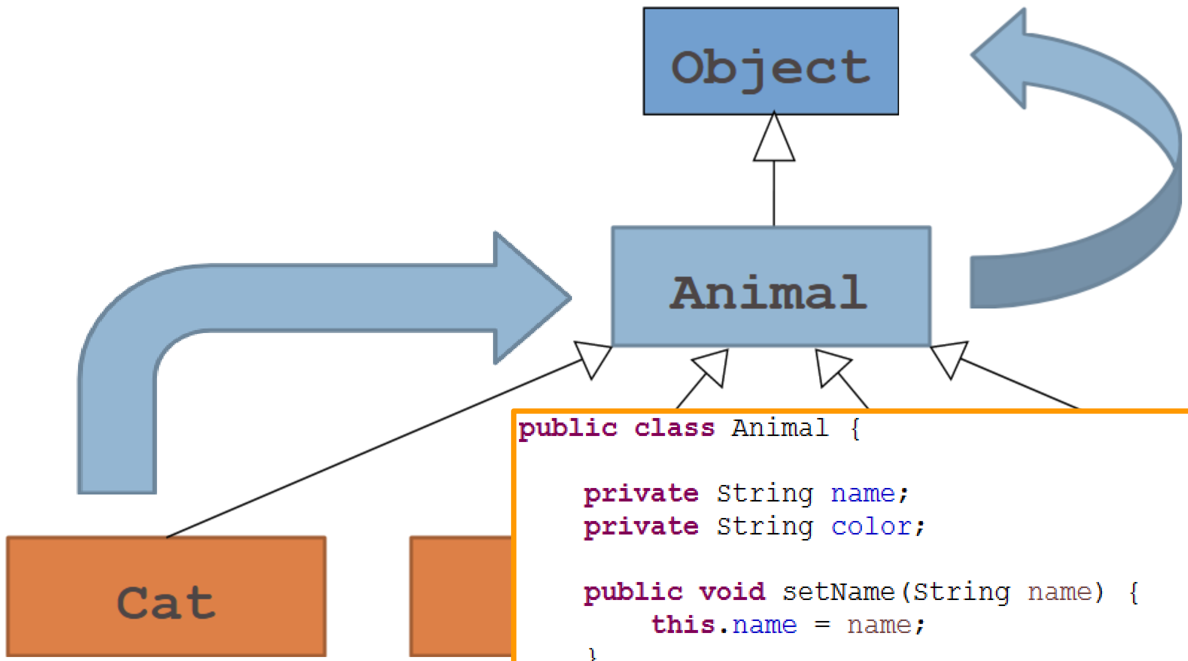
- Similar to `this();`
- `this(arg);`   // same class constructor call
- `super(arg);`   // super class constructor call
- Both of these need to be used in the first line of the constructor.
- If not, then default `super()` will be included by compiler

Constructor Call



```java
public class Cat extends Animal {
    public Cat(String name) {
        super();
        setName(name);
        setColor("gray");
    }
}
```

Constructor Call

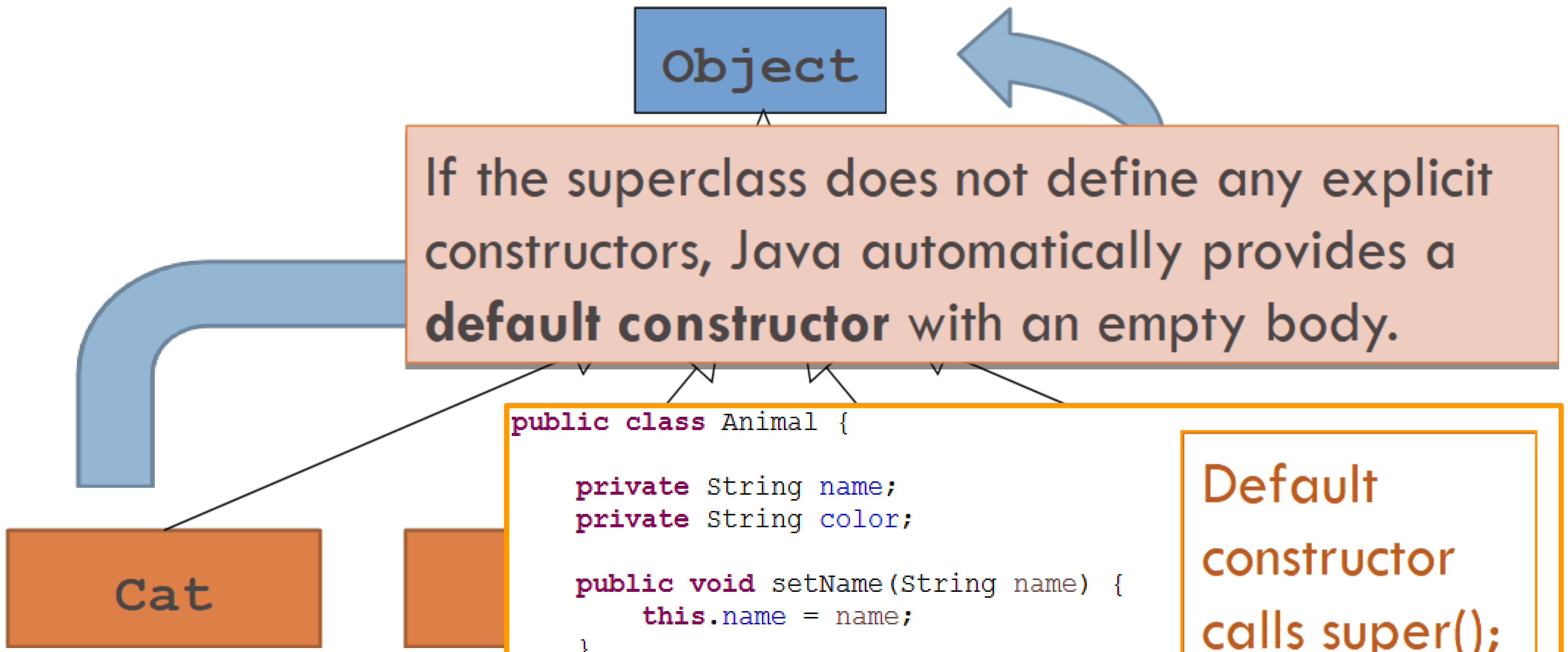


```
public class Cat extends
    public Cat(String na
        super();
        setName(name);
        setColor("gray")
    }
}
```

```
public class Animal {

    private String name;
    private String color;

    public void setName(String name) {
        this.name = name;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String getName() {
        return name;
    }
    public String getColor() {
        return color;
    }
    public String toString() {
        return "Hi, my name is " + name +". I'm " + color + ".";
    }
}
```

Default constructor calls super();

Constructor Call



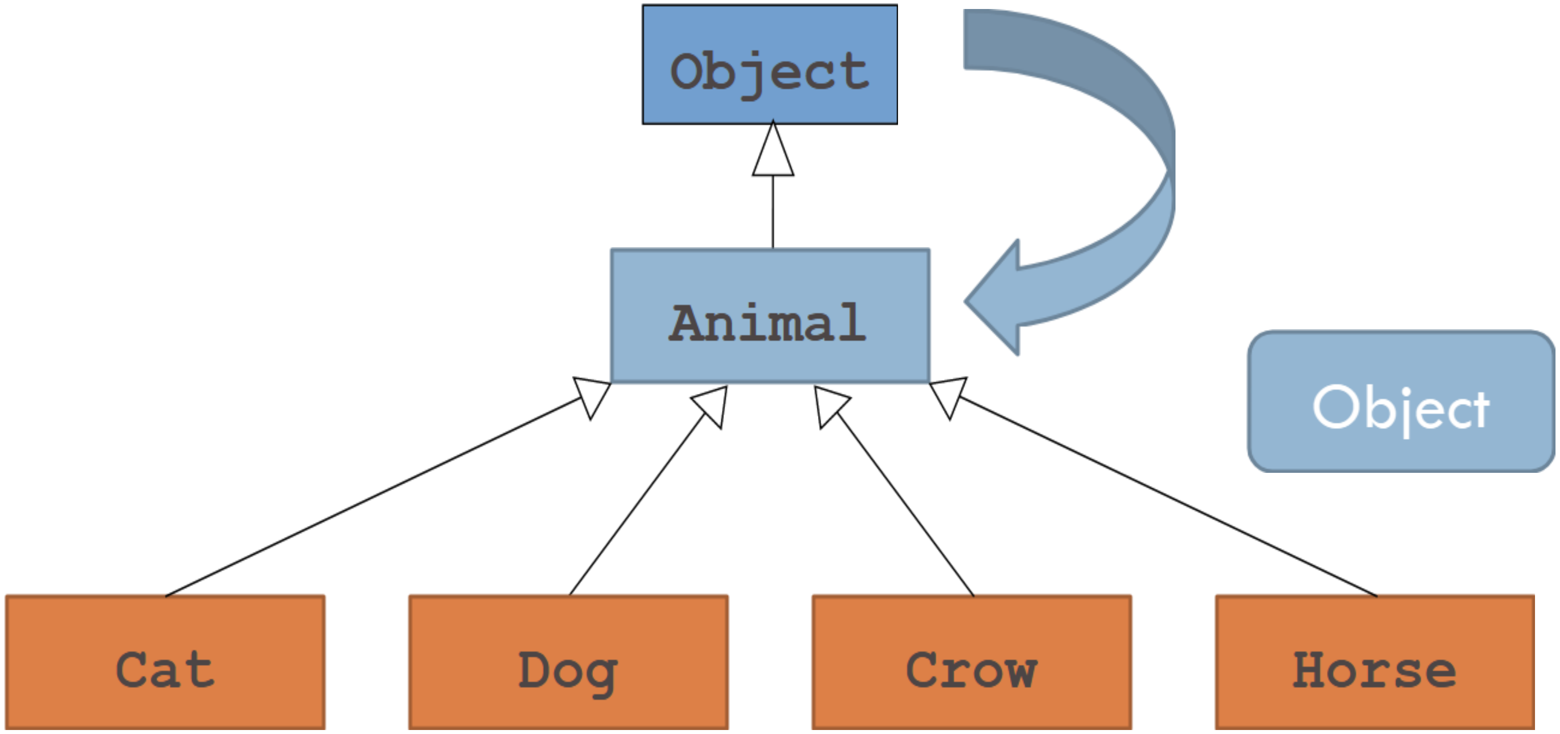
If the superclass does not define any explicit constructors, Java automatically provides a **default constructor** with an empty body.

```
public class Cat extends
    public Cat(String na
        super();
        setName(name);
        setColor("gray")
    }
}
```

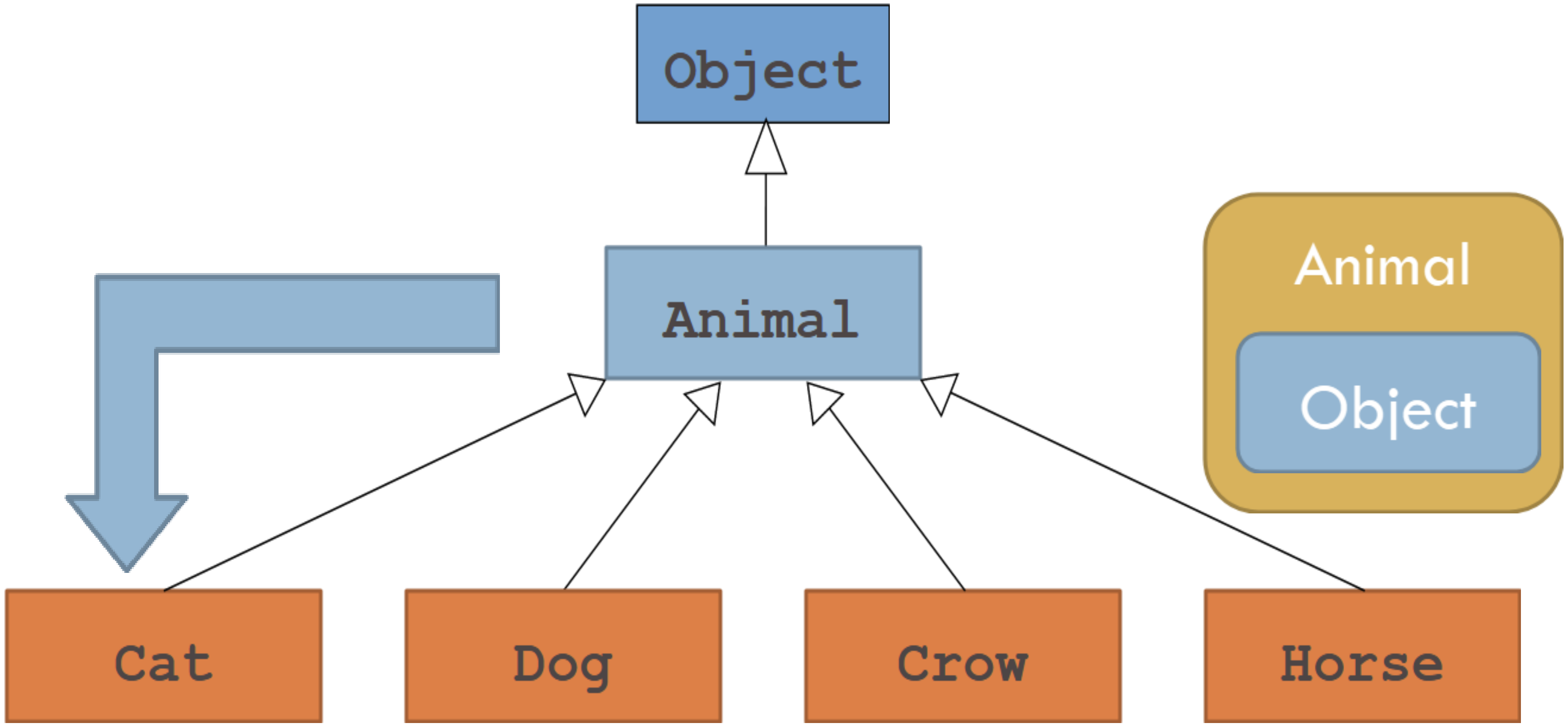
```
public class Animal {

    private String name;
    private String color;

    public void setName(String name) {
        this.name = name;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String getName() {
        return name;
    }
    public String getColor() {
        return color;
    }
    public String toString() {
        return "Hi, my name is " + name +". I'm " + color + ".";
    }
}
```

Default constructor calls super();

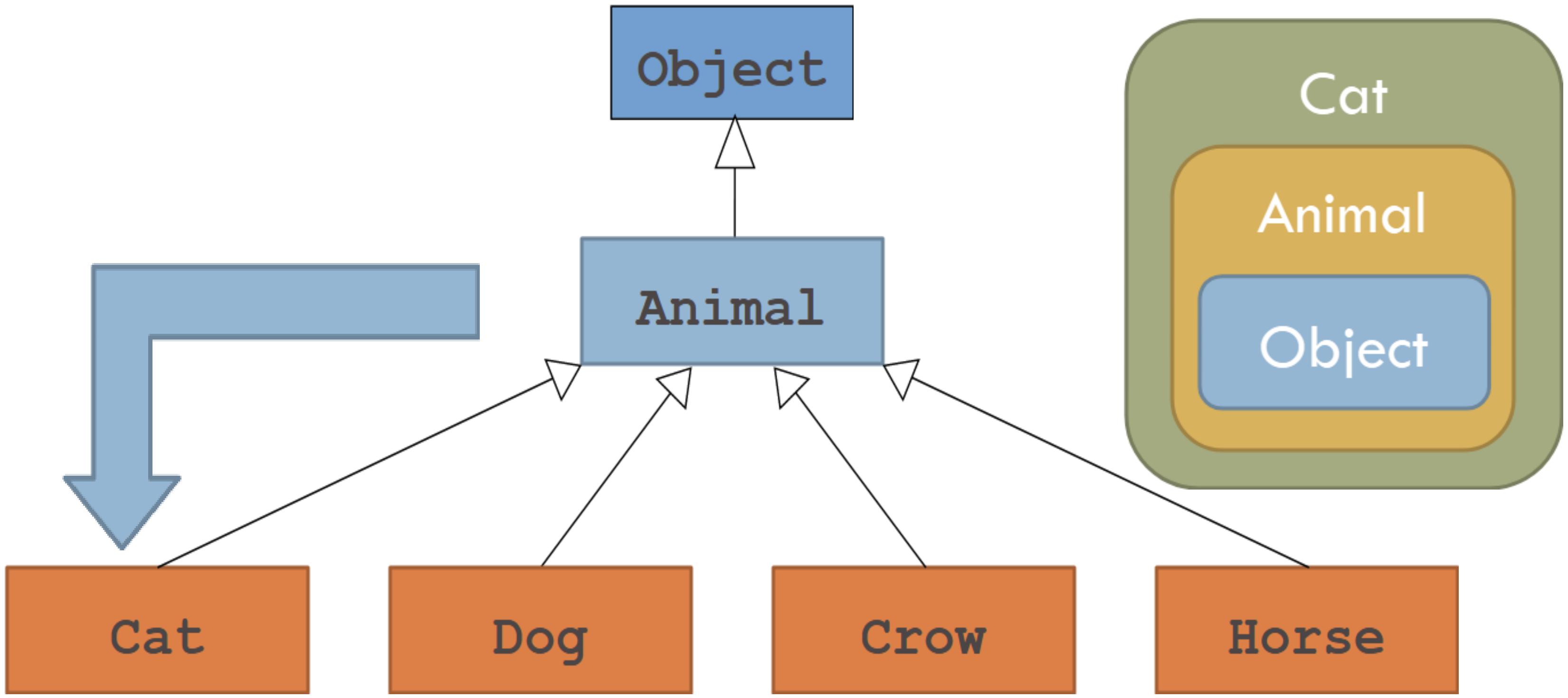
Constructor Call



Constructor Call

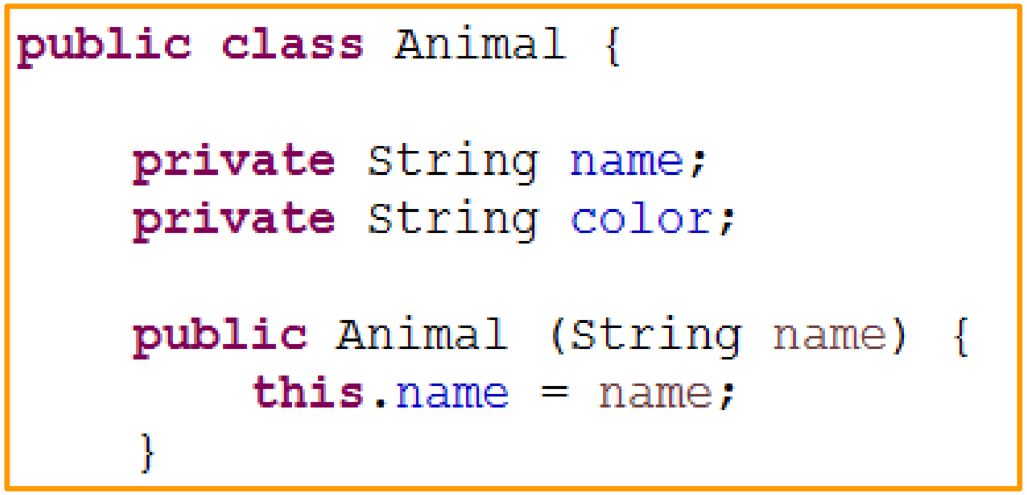


Constructor Call

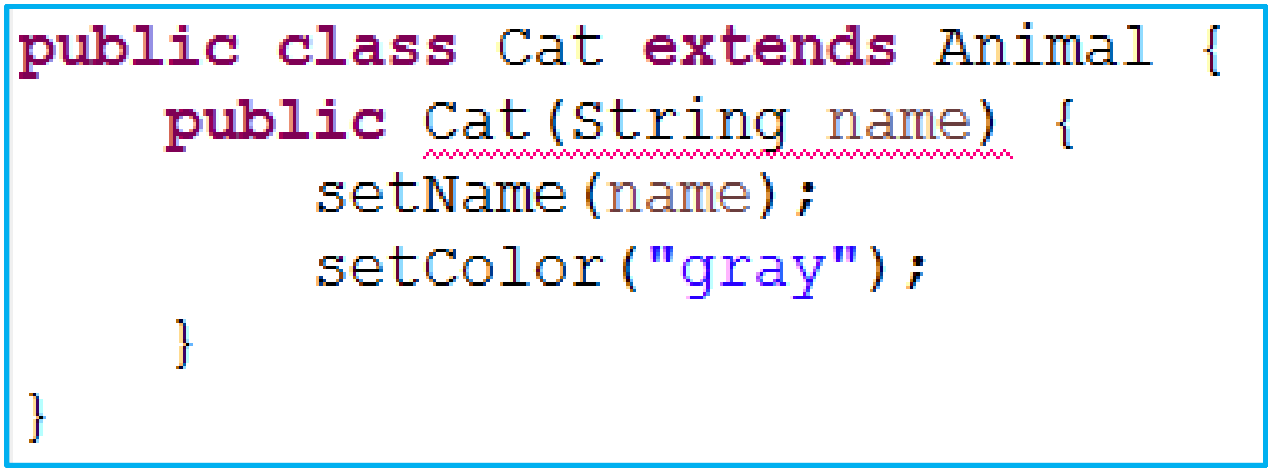


Explicit Animal Constructor

- Lets have an explicit Animal constructor



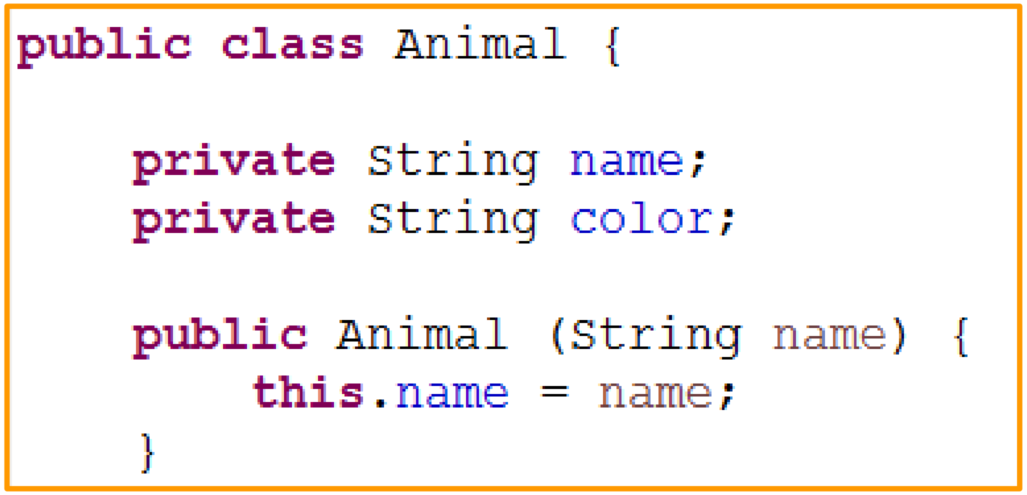
```
public class Animal {

    private String name;
    private String color;

    public Animal (String name) {
        this.name = name;
    }
```



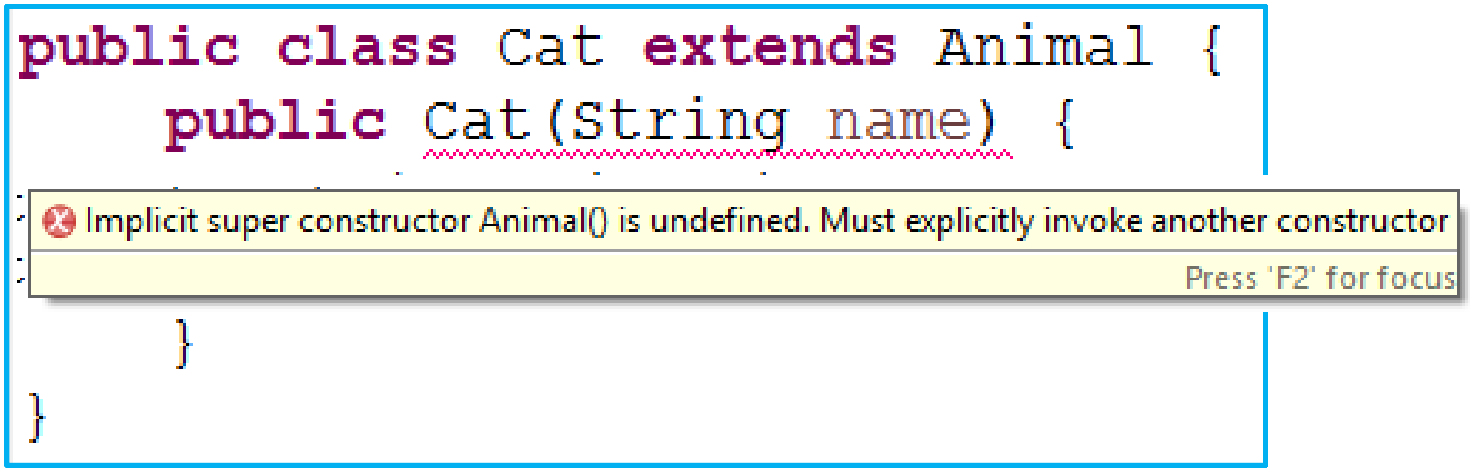
```
public class Cat extends Animal {
    public Cat(String name) {
        setName(name);
        setColor("gray");
    }
}
```

Explicit Animal Constructor

- Lets have an explicit Animal constructor



```
public class Animal {

    private String name;
    private String color;

    public Animal (String name) {
        this.name = name;
    }
```



```
public class Cat extends Animal {
    public Cat(String name) {
```

Implicit super constructor Animal() is undefined. Must explicitly invoke another constructor

Press 'F2' for focus

```
    }
}
```

```
public class Animal {

    private String name;
    private String color;

    public Animal (String name) {
        this.name = name;
    }
```

```
public class Cat extends Animal {
    public Cat(String name) {
        setName(name);
        setColor("gray");
    }
}
```

✗

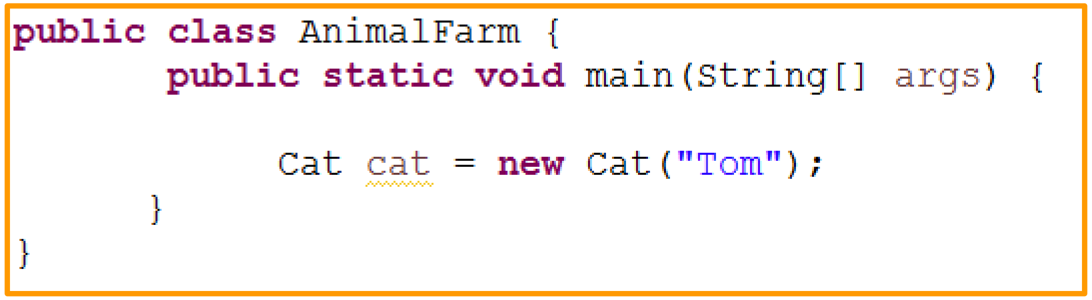
```
public class Cat extends Animal {
    public Cat(String name) {
        super(name);
        setColor("gray");
    }
}
```

✓

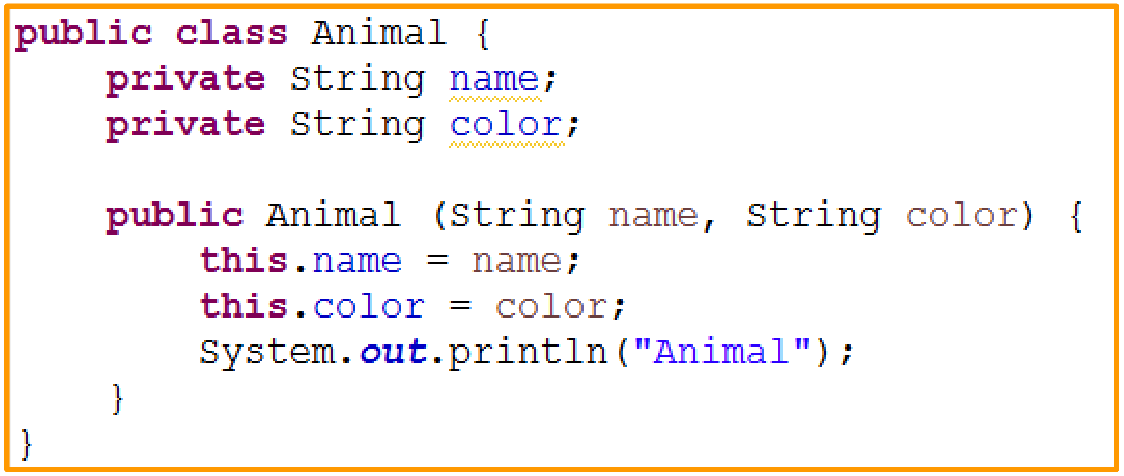
Constructor Calls

- Java therefore invokes the superclass constructor in one of the following ways:
  - Classes that begin with an explicit call to **`this`** invoke one of the other constructors for this class, delegating responsibility to that constructor for making sure that the superclass constructor gets called.
  - Classes that begin with a call to **`super`** invoke the constructor in the super class that matches the argument list provided.
  - Classes that begin with no call to either **`super`** or **`this`** invoke the default super class constructor with no arguments.

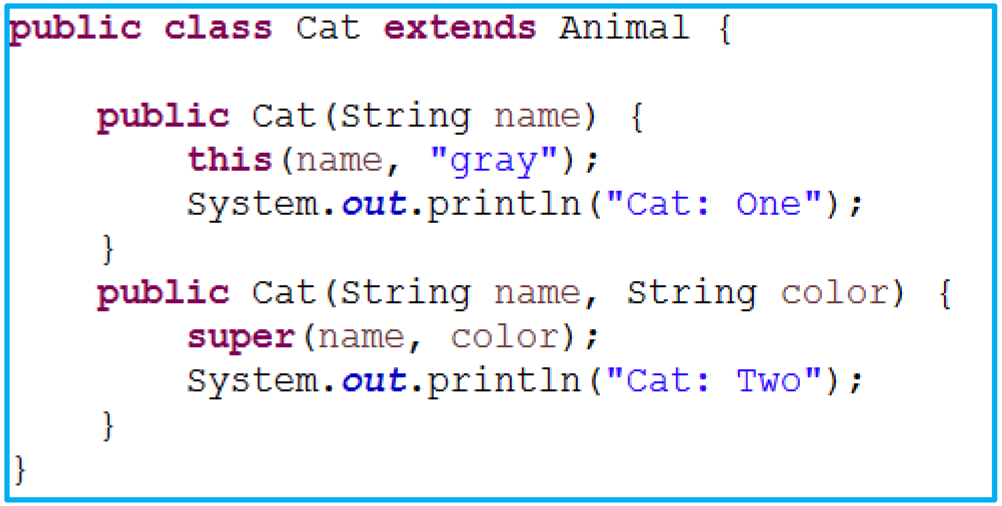
Example



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

		Cat cat = new Cat("Tom");
	}
}
```

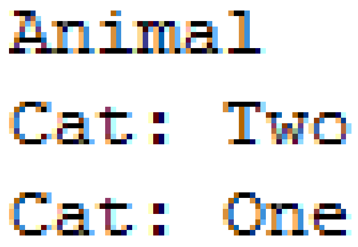


```java
public class Animal {
    private String name;
    private String color;

    public Animal (String name, String color) {
        this.name = name;
        this.color = color;
        System.out.println("Animal");
    }
}
```



```java
public class Cat extends Animal {

    public Cat(String name) {
        this(name, "gray");
        System.out.println("Cat: One");
    }
    public Cat(String name, String color) {
        super(name, color);
        System.out.println("Cat: Two");
    }
}
```

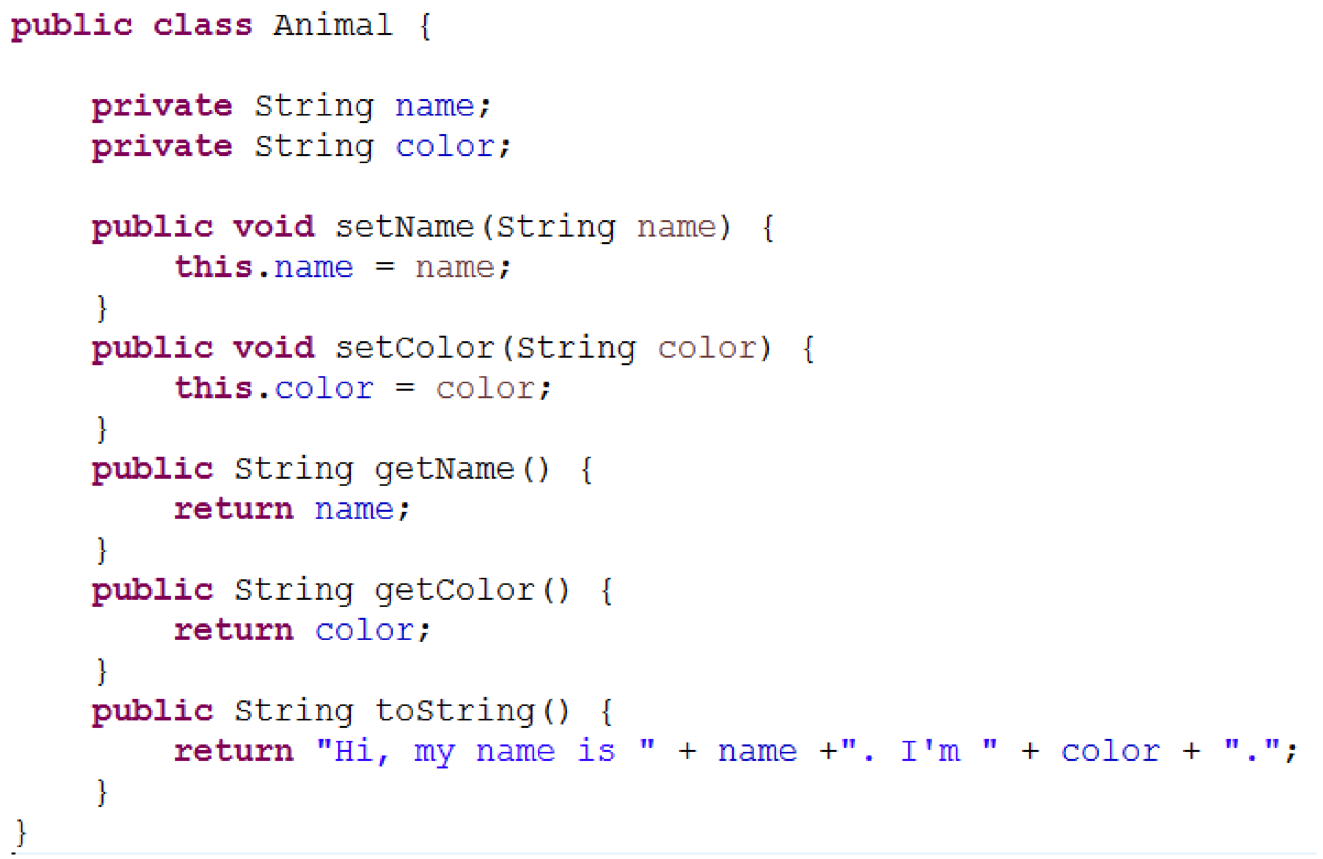
- What is the output?



```
Animal
Cat: Two
Cat: One
```

Lets implement some functions

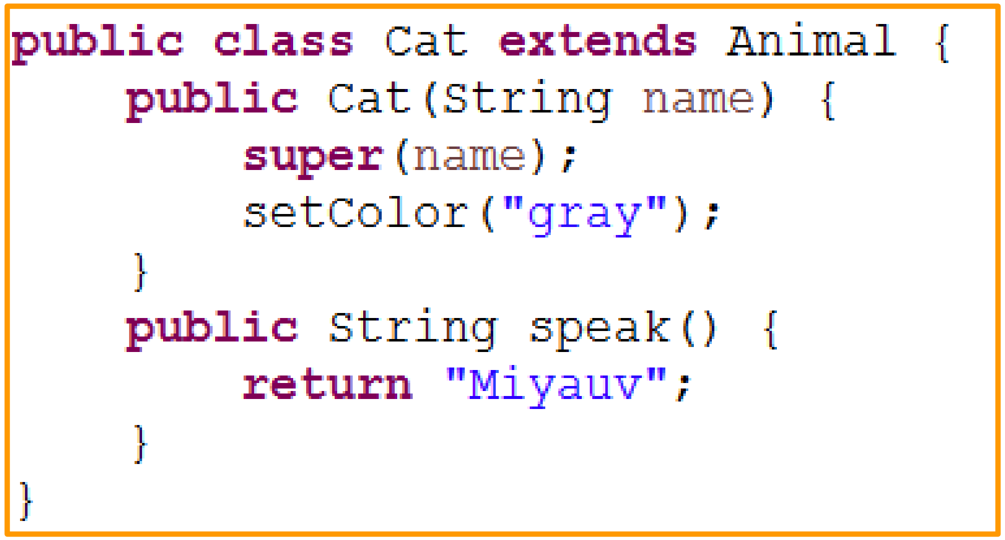
- Animals speak differently, so **speak** function needs to be implemented differently.
- In animal class we don’t have a `speak()` function



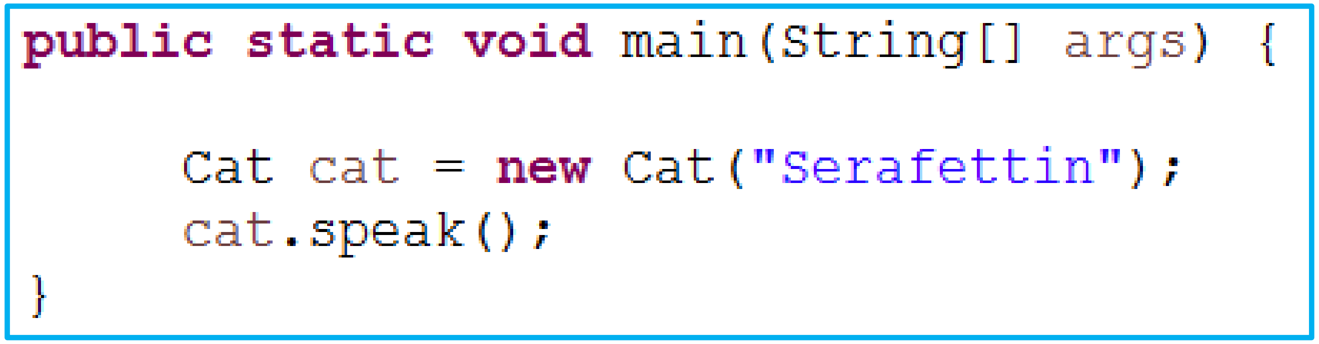
```java
public class Animal {

    private String name;
    private String color;

    public void setName(String name) {
        this.name = name;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String getName() {
        return name;
    }
    public String getColor() {
        return color;
    }
    public String toString() {
        return "Hi, my name is " + name +". I'm " + color + ".";
    }
}
```

Cat Class

- Speak function implemented within the Cat class.



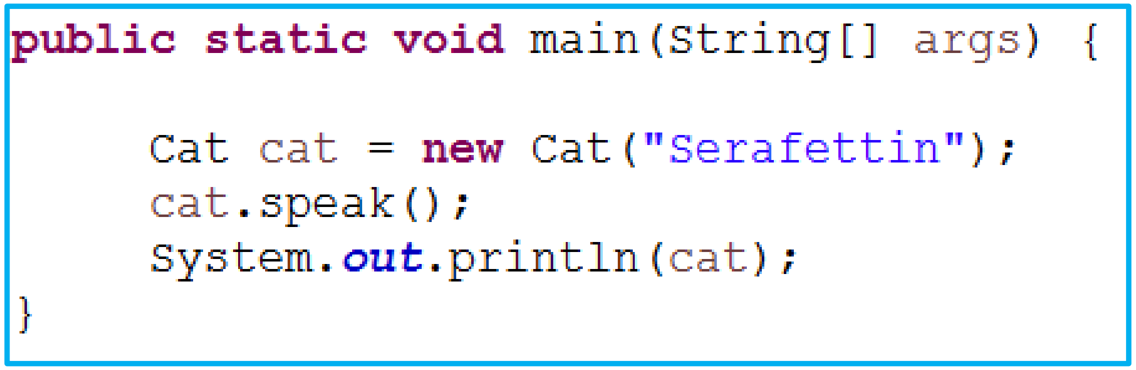
```java
public class Cat extends Animal {
    public Cat(String name) {
        super(name);
        setColor("gray");
    }
    public String speak() {
        return "Miyauv";
    }
}
```



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

    Cat cat = new Cat("Serafettin");
    cat.speak();
}
```

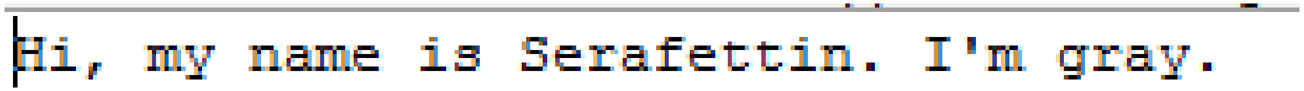
toString method

- Can we call the toString method from a cat object?



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

        Cat cat = new Cat("Serafettin");
        cat.speak();
        System.out.println(cat);
}
```

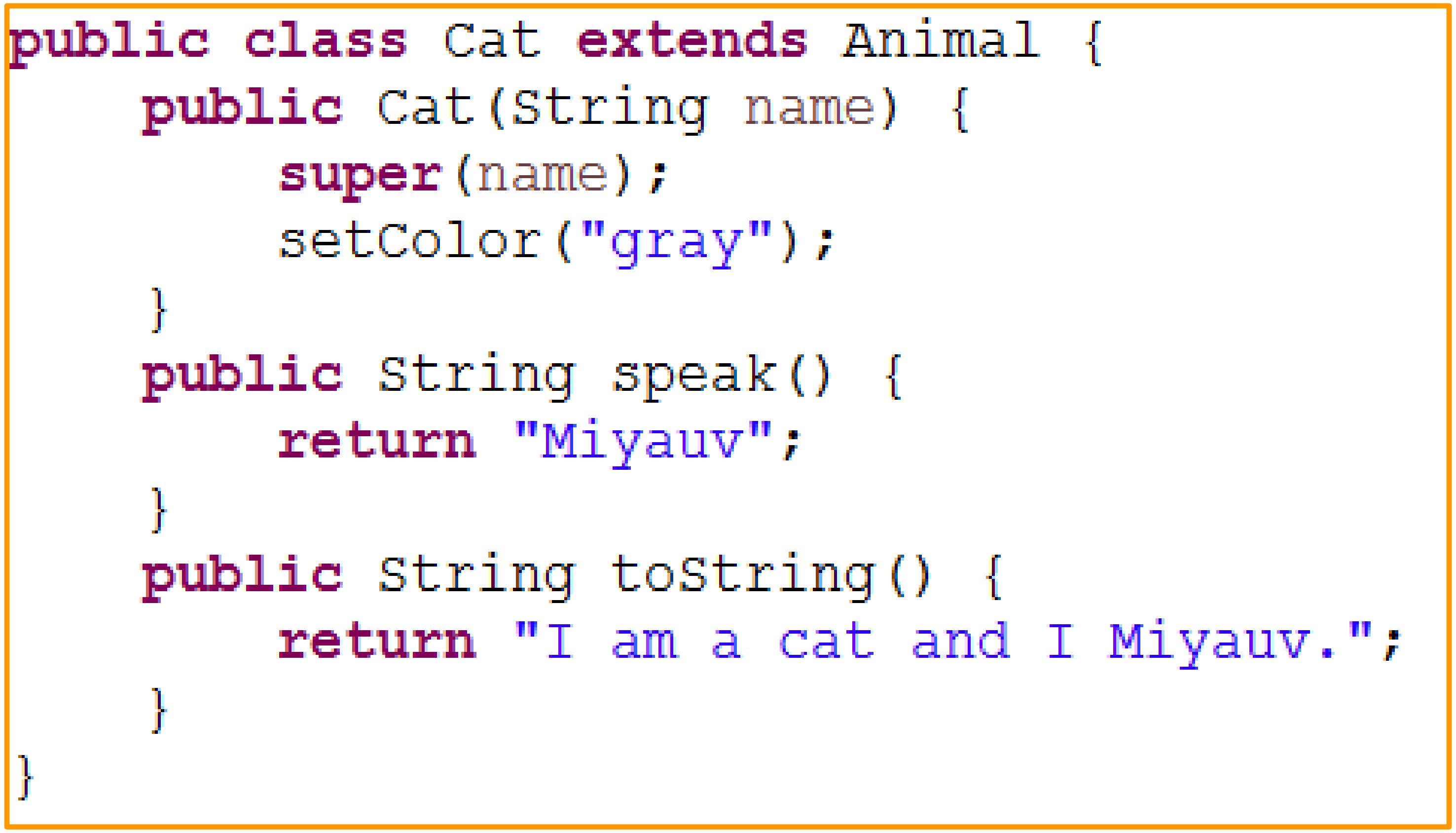
- Yes, we can.
- What will be the output?



```
Hi, my name is Serafettin. I'm gray.
```

toString method

- Can the Cat class has its own **toString** function?



```java
public class Cat extends Animal {
    public Cat(String name) {
        super(name);
        setColor("gray");
    }
    public String speak() {
        return "Miyauv";
    }
    public String toString() {
        return "I am a cat and I Miyauv.";
    }
}
```

Overriding (Overwriting)

- Yes, it can.
  - It is called function **overriding**.
- A **subclass** may redefine a method that is defined by a **superclass**.
  - In this case, it is said that the subclass **overrides** the method.
- When one class extends another, the subclass is allowed to **override** method definitions in its superclass.
- Whenever you invoke that method on an instance of the extended class, Java chooses the new version of the method provided by that class and not the original version provided by the superclass.
- The decision about which version of a method to use is always made on the basis of what the type of object in fact is (**run-time**) and not on what it happens to be declared as at that point in the code (**compile-time**).

What will be the output?

```
public class Cat extends Animal {
    public Cat(String name) {
        super(name);
        setColor("gray");
    }
    public String speak() {
        return "Miyauv";
    }
    public String toString() {
        return "I am a cat and I Miyauv.";
    }
}
```

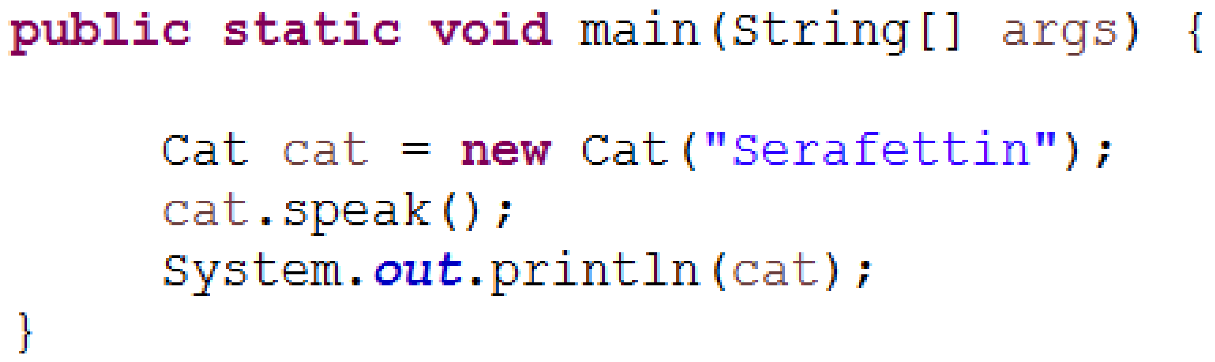
```
I am a cat and I Miyauv.
```

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

    Cat cat = new Cat("Serafettin");
    cat.speak();
    System.out.println(cat);
}
```

toString method

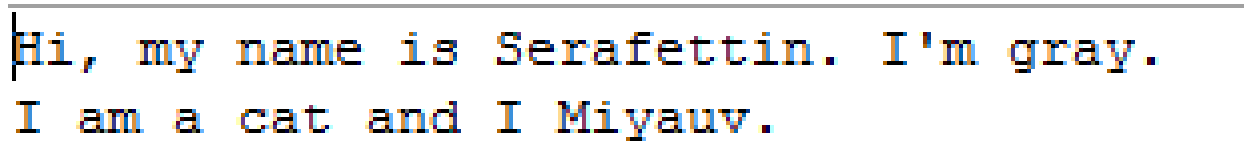
- Cat is still an animal.

- How can I print animal representation as well as the cat representation?



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

    Cat cat = new Cat("Serafettin");
    cat.speak();
    System.out.println(cat);
}
```

- How can I print out the following from the above main?



```
Hi, my name is Serafettin. I'm gray.
I am a cat and I Miyauv.
```

- If you need to invoke the original version of a method, you can do so by using the keyword **super** as a receiver.

- For example, if you needed to call the original version of an **init** method as specified by the superclass, you could call

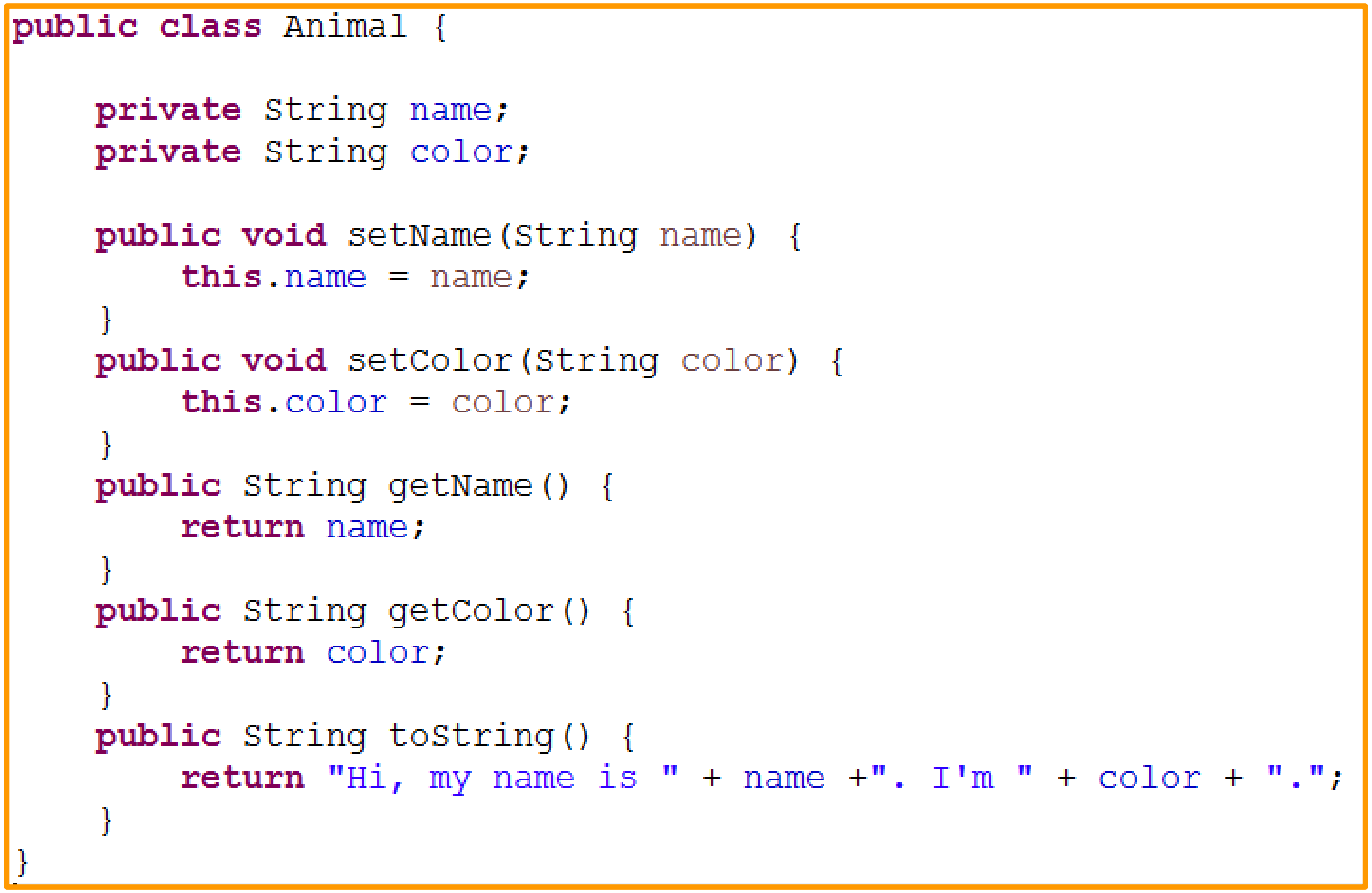
```
super.init();
```

Overloading vs. Overriding

- What is the difference between these two?
- Overloading
  - Same class has the same function name but with different parameters.
- Overriding
  - Subclass has the same function signature (name and parameters) with the superclass

toString method in Animal Class

- Is **toString** an overriding function or not?
  - Yes, it overrides the **toString** method of the Object class



```java
public class Animal {

    private String name;
    private String color;

    public void setName(String name) {
        this.name = name;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String getName() {
        return name;
    }
    public String getColor() {
        return color;
    }
    public String toString() {
        return "Hi, my name is " + name +". I'm " + color + ".";
    }
}
```

Any Questions?