

# CS105

## Introduction to Object-Oriented Programming

**Prof. Dr. Nizamettin AYDIN**

**naydin@itu.edu.tr**

**nizamettin.aydin@ozyegin.edu.tr**

# Inheritance and Polymorphism

# Outline

- more benefits of inheritance
- Reference and Object
- Inheritance
- Polymorphism
- Compile Time vs. Runtime
- Casting
- Final method
- Static vs. Dynamic Binding

# more benefits of inheritance

- Assume that we have an animal farm with different types of animals and we don't know inheritance

```
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";  
    }  
}
```

Cat

```
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";  
    }  
}
```

Dog

- Assume that we have an animal farm with different types of animals and we don't know inheritance.
- Can we store all these different animal types in one data structure, like an array?

- Dog
- Cat
- Cow
- Mouse

Serafettin
Scooby
Sarı Kız
Rin Tin Tin
Tom
Jerry

- Assume that we have an animal farm with different types of animals and we don't know inheritance.
- Can we store all these different animal types in one data structure, like an array?

- Dog
- Cat
- Cow
- Mouse

Serafettin
Scooby
Sarı Kız
Rin Tin Tin
Tom
Jerry

- An array needs to hold objects of same type!

- Assume that we have an animal farm with different types of animals and we don't know inheritance.
- Can we store all these different animal types in one data structure, like an array?

- Dog
- Cat
- Cow
- Mouse

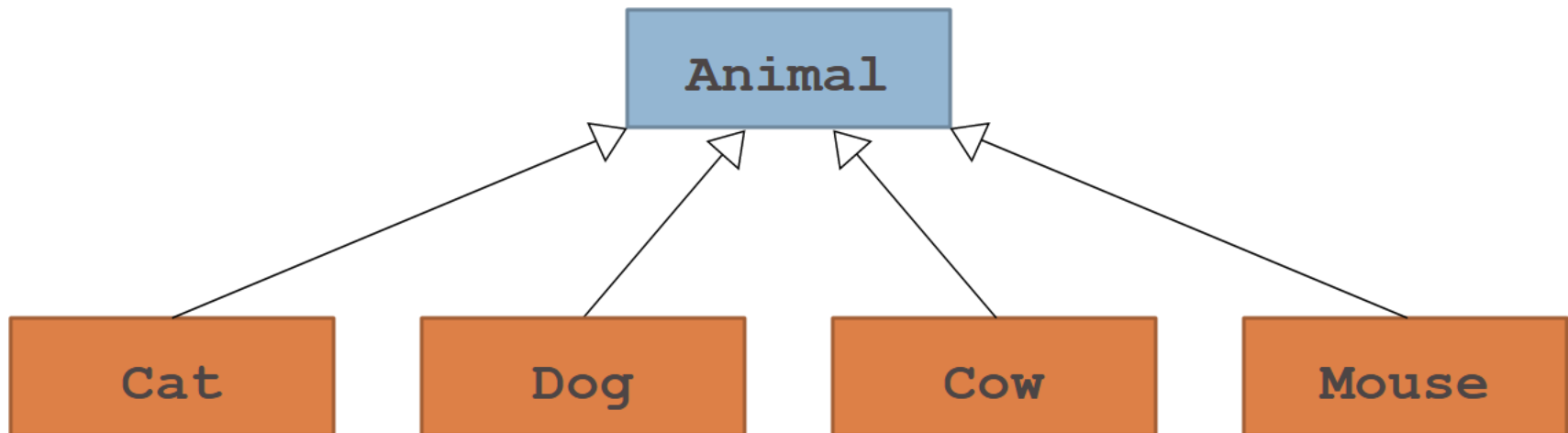
Serafettin
Scooby
Sarı Kız
Rin Tin Tin
Tom
Jerry

–An array needs to hold objects of same type!

Serafettin
Tom
Jerry

Sarı Kız
Scooby
Rin Tin Tin

- Inheritance gives us the ability to store all animals in one data structure.

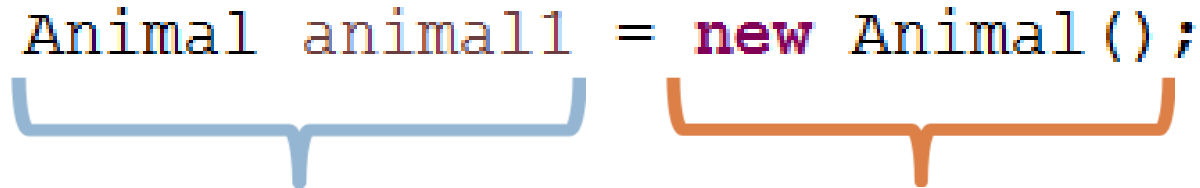


- A cat/dog/cow/mouse is an Animal



# Reference and Object

```
Animal animal1 = new Animal();
```



Reference

Object

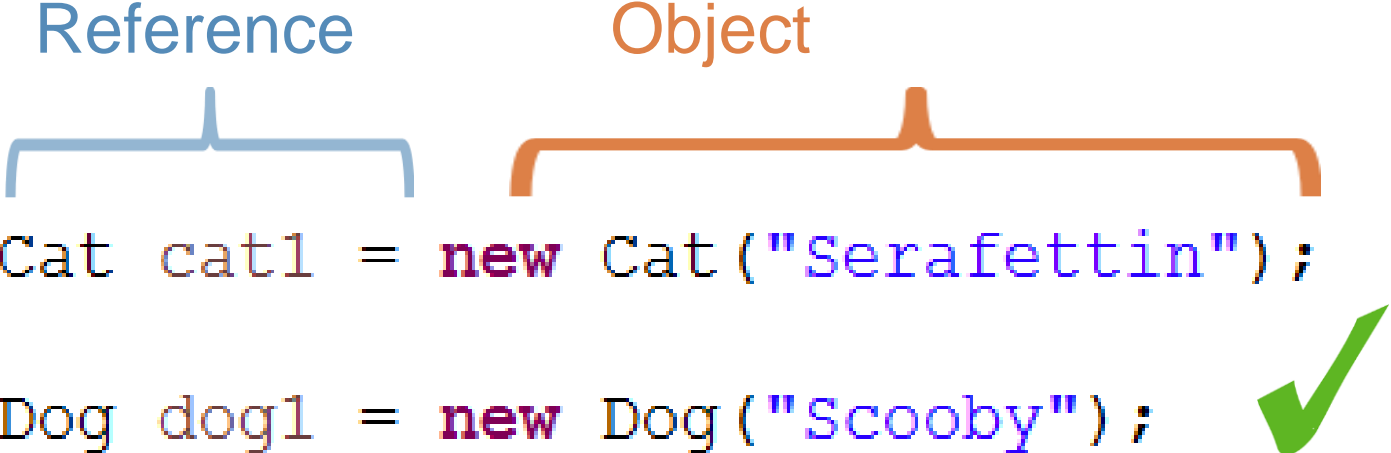


- **Animal is an animal**

# Reference and Object

Reference                      Object

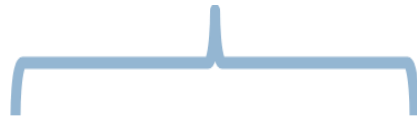
```
Cat cat1 = new Cat("Serafettin");  
Dog dog1 = new Dog("Scooby");
```



- Are these statements legal?
  - Cat is a cat
  - Dog is a dog
- Both of them are legal statements.

# Reference and Object

Reference



Object

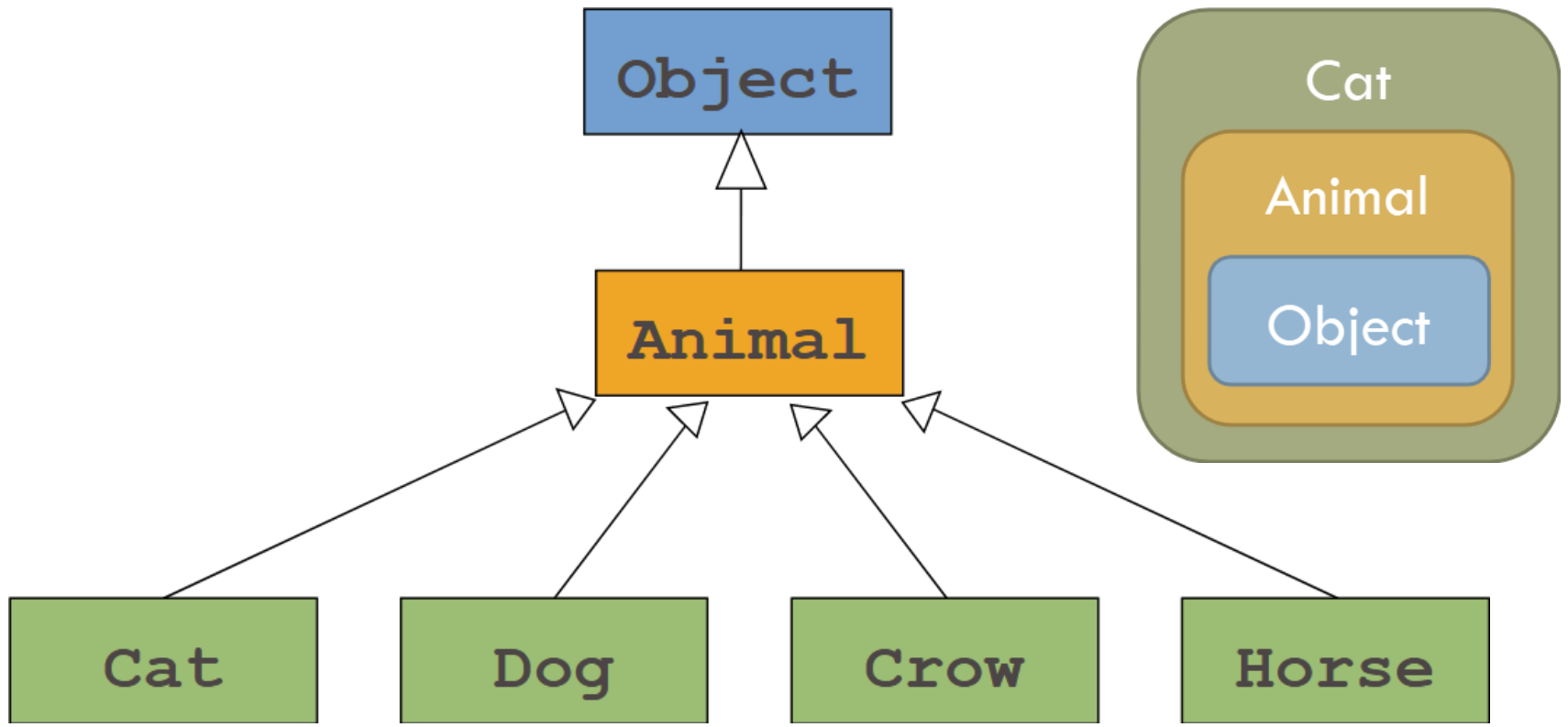


```
Animal animal2 = new Dog("Rin Tin Tin");
```

```
Animal animal3 = new Cat("Tom");
```



- Are these statements legal?
- Can an animal reference point to a cat/dog object?
  - Cat is an animal
  - Dog is an animal
- Both of them are legal statements.



# Inheritance

- Therefore we can keep all animal types in one single data structure.
- An animal array can store an animal object and other animal types (cat, dog etc.).

```
Animal[] animals = new Animal[3];  
animals[0] = new Animal();  
animals[1] = new Cat("Tom");  
animals[2] = new Dog("Rin Tin Tin");
```

# Reference and Object

Reference                      Object

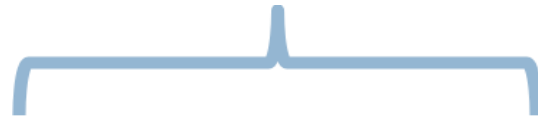
```
Cat cat2 = new Animal(); X
```

- Is this a legal statement?
- Not a legal statement
  - Not all animals is a cat.
  - An animal may not have all the capabilities of a cat.
  - Cats can jump but not all animals can

# Reference and Object

Reference

Object



```
Object object1 = new Cat ("Tom");
```



- Is this a legal statement?
- Cat is an object.
- This is a legal statement.

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

```
Animal animall = new Animal();  
Cat cat1 = new Cat("Serafettin");  
  
String color = cat1.getColor();  
animall = cat1;
```

- Are these statements valid?
  - An animal reference can refer to a cat object.
  - All cats are animal.



```

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

```

```

Animal animal1 = new Animal();
Cat cat1 = new Cat("Serafettin");

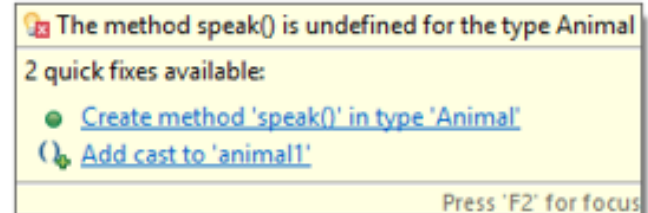
String color = cat1.getColor();
animal1 = cat1;
animal1.speak();

```

- Are these statements valid?

- animal1 is an animal reference

- Compiler knows the reference type but not the object type



```

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 toString() {
        return "Animal";
    }
}

```

```

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

```

```

public class Dog extends Animal {
    public Dog(String name) {
        setName(name);
        setColor("black");
    }
    public String toString() {
        return "Dog";
    }
}

```

```

Animal[] animals = new Animal[3];
animals[0] = new Animal();
animals[1] = new Cat("Tom");
animals[2] = new Dog("Rin Tin Tin");

for (int i = 0; i < animals.length; i++) {
    System.out.println(animals[i]);
}

```

```

Animal
Cat
Dog

```

- What is the output?

# Polymorphism

- Polymorphism
  - Helps build extensible systems
  - Programs generically process objects as superclass objects
    - Can add classes to systems easily
      - Classes must be part of generically processed hierarchy
- Polymorphism gives us the capability to call the right method.



# Compile Time vs. Runtime

- What does compiler do?
  - Compiler interprets our code
- Then what happens in runtime?
  - At runtime, the environment executes the interpreted code
- There are two steps of our programs:
  - Compiler time
  - Runtime
- **Compile time** decisions are based on **reference type**
- **Run time** decisions are based on **object type**


# Compiler

- Only knows about the reference type.
- When a method is called, it looks for that method inside that particular reference type 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 class Cat extends Animal {  
    public Cat(String name) {  
        setName(name);  
        setColor("gray");  
    }  
    public String speak() {  
        return "Miyauv";  
    }  
}
```

```
Animal animal1 = new Animal();  
Cat cat1 = new Cat("Serafettin");  
  
String color = cat1.getColor();  
animal1 = cat1;  
animal1.speak();
```

 The method speak() is undefined for the type Animal

2 quick fixes available:

- [Create method 'speak\(\)' in type 'Animal'](#)
- [Add cast to 'animal1'](#)

Press 'F2' for focus

# Run time

- At run time, the exact run time object is used to find where a method belongs to.
- The method used needs to match with the signature of the actual method.
- Example in the next slide...

```

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 toString() {
        return "Animal";
    }
}

```

```

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

```

```

public class Dog extends Animal {
    public Dog(String name) {
        setName(name);
        setColor("black");
    }
    public String toString() {
        return "Dog";
    }
}

```

- **toString()** method
- The method signatures match

```

Animal[] animals = new Animal[3];
animals[0] = new Animal();
animals[1] = new Cat("Tom");
animals[2] = new Dog("Rin Tin Tin");

for (int i = 0; i < animals.length; i++) {
    System.out.println(animals[i]);
}

```

```

Animal
Cat
Dog

```



```

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

```

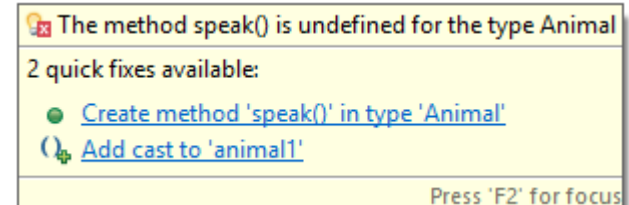
```

Animal animal1 = new Animal();
Cat cat1 = new Cat("Serafettin");

String color = cat1.getColor();
animal1 = cat1;
animal1.speak();

```

- Is there a way to fix this?
  - Lets assume that animal1 will always refer to a cat object.



```

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

```

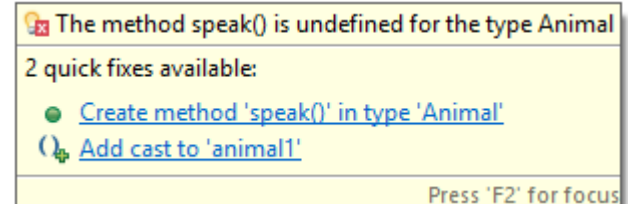
```

Animal animal1 = new Animal();
Cat cat1 = new Cat("Serafettin");

String color = cat1.getColor();
animal1 = cat1;
animal1.speak();

```

- It is possible with explicit casting



# Casting

- Widening

- Automatic type promotion (from int to double)

```
int a = 5;  
double b = a;  
System.out.println(b);
```

- Super-class reference = sub-class object;

```
Animal animal2 = new Cat("Tom");
```

- Narrowing

- Explicit casting (from double to int)

```
double c = 9.99;  
int d = ((int) c);  
System.out.println(d);
```

- Sub-class reference = (subclass) super-class reference;

```
((Cat) animal1).speak();
```

# Casting

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

```
Animal animal1 = new Animal();  
Cat cat1 = new Cat("Serafettin");  
  
animal1 = cat1;  
//animal1.speak(); // does not work  
((Cat) animal1).speak();
```

- Compiler will search for speak method inside the cat class.

# Casting

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

```
Animal animal1 = new Animal();  
Cat cat1 = new Cat("Serafettin");  
  
//animal1 = cat1;  
//animal1.speak(); // does not work  
((Cat) animal1).speak();
```

- What do you think will happen at this point?

# Casting

- We won't get a compiler error, relax

```
Animal animall = new Animal();  
Cat cat1 = new Cat("Serafettin");  
  
//animall = cat1;  
//animall.speak(); // does not work  
((Cat) animall).speak();
```

- It will be worse, we will get a run time error

# Casting

- We need to make sure that we don't cast wrong.
- How?
  - By doing run time type check
  - **instanceof** operator
    - Checks whether there is an **is a** relationship

```
Animal animall = new Animal();  
Cat cat1 = new Cat("Serafettin");  
  
//animall = cat1;  
//animall.speak(); // does not work  
if (animall instanceof Cat) {  
    ((Cat) animall).speak();  
}
```

# Final method

- A method in the super class that cannot be overridden in a subclass.
- Any idea which methods can be final?
- Methods that are declared private are implicitly final, because it's not possible to override them in a subclass.
- Methods that are declared static are implicitly final.



# Static vs. Dynamic Binding

- A final method's declaration can never change,
  - so all sub classes use the same method implementation, and calls to final methods are resolved at compile time—
  - this is known as **static (early) binding**.
- **Dynamic (late) binding**: methods to be executed are determined in run time,
  - depending on the object type.

# What is the output?

```
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 speak() {  
        return "Some Noise";  
    }  
    public String toString() {  
        return "Animal " + this.getName() +  
            " is in color " + this.getColor() +  
            " and speaks " + this.speak();  
    }  
}
```

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

```
public static void main(String[] args) {  
  
    Animal animal = new Cat("Tom");  
    System.out.println(animal);  
}
```

```
Animal Tom is in color gray and speaks Miyauv
```

# What is the output?

```
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 speak() {  
        return "Some Noise";  
    }  
    public String toString() {  
        return "Animal " + this.getName() +  
            " is in color " + this.getColor() +  
            " and speaks " + this.speak();  
    }  
}
```

# What is the output?

```
public class Cat extends Animal {  
    public Cat(String name) {  
        setName(name);  
        setColor("gray");  
    }  
    public String speak() {  
        return "Miyauv";  
    }  
    public String toString() {  
        return "Cat " + this.getName() +  
            " is in color " + this.getColor() +  
            " and speaks " + super.speak();  
    }  
}
```

```
public static void main(String[] args) {  
  
    Animal animal = new vanCat("Tom");  
    System.out.println(animal);  
}
```

```
public class vanCat extends Cat{  
  
    public vanCat(String name) {  
        super(name);  
        setColor("white");  
    }  
}
```

```
Cat Tom is in color white and speaks Some Noise
```

- Call to the `super.someMethod()` get bound at compile time.
- Call to the `this.someMethod()` get bound at run time.

**Any Questions?**