Mahidol University
Faculty of Information
and Communication Technology

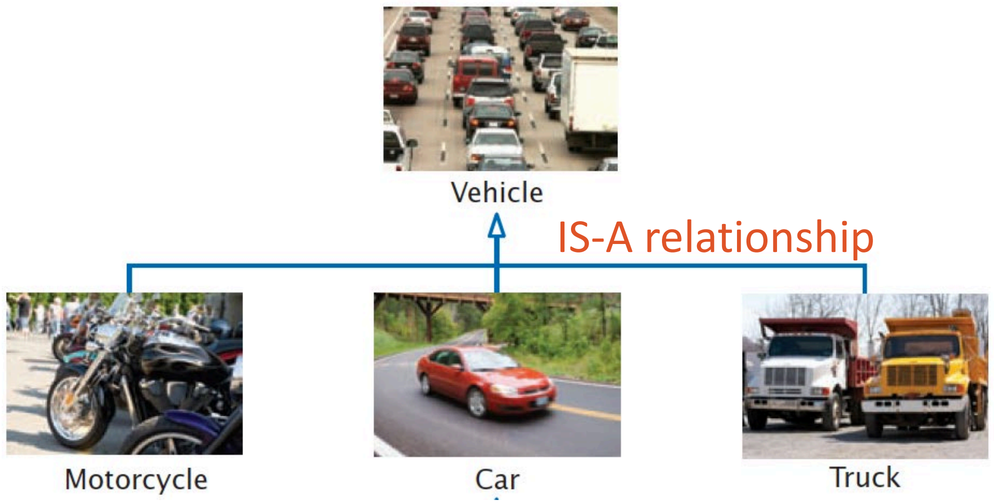
LECTURE 07 Polymorphism

ITCS123 ObjectOriented Programming

Dr. Siripen Pongpaichet
Dr. Petch Sajjacholapunt
Asst. Prof. Dr. Ananta Srisuphab

Semester 2/2023 22 Feb 2024

Recap – Lecture 06



Vehicle

IS-A relationship

Motorcycle

Car

Truck

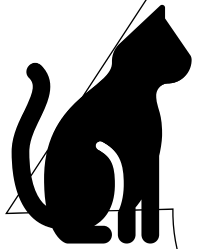
- **What** is inheritance?
  - IS-A relationship hierarchies
- **Why** do we need it?
  - Reuse code -> subclass inherits all variables & methods
- **How** to implement subclass
  - `extends` reserved word indicate that a class inherits from a superclass
  - Only include things that differ from superclass
  - A subclass and override a superclass method by providing a new implementation
- **Overriding Method**
  - This method can extend or replace the functionality of the superclass method
  - Use super reserved word to call a superclass method, use super(…) in the constructor to call superclass constructor
- **Special Topics**
  - Final, Protected Access, `instanceof` operator

e.g., `display()` method to print food information. This method does thing differently on Food class vs EnergyDrink class.

In the past week, how many **hours** you spent on coding and reviewing lecture outside the classroom?

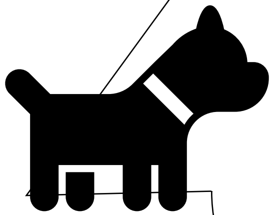
Can you apply inheritance on these two classes

```
name:
color:
hungry:
energy:
----------------
Cat(nanme,color,hungry,energy)
getHungry()
getEnergy()
greeting() -> Meow! I'm + [name]
isAlive()
play()
sleep()
feedFood(food)
displayCat()
```



Cat Class

```
name:
color:
hungry:
energy:
sense: // sense of smell
----------------
Dog(nanme,color,hungry,energy,sense)
getHungry()
getEnergy()
getSense()
greeting() -> Bark! I'm + [name]
isAlive()
play()
sleep()
feedFood(food)
displayDog()
```



Dog Class

common attributes: same variable name and type OR **common methods:** same method name and method body

similar methods: same method name but different method body

unique attributes/method: only available in a specific class

Cat Class

```
name:
color:
hungry:
energy:
---------------
Cat(nanme,color,hungry,energy)
getHungry()
getEnergy()
greeting() -> Meow! I'm + [name]
isAlive()
play()
sleep()
feedFood(food)
displayCat()
```

Dog Class

```
name:
color:
hungry:
energy:
sense: // sense of smell
---------------
Dog(nanme,color,hungry,energy,sense)
getHungry()
getEnergy()
getSense()
greeting() -> Bark! I'm + [name]
isAlive()
play()
sleep()
feedFood(food)
displayDog()
```

Superclass
(general class)

Animal Class

```
name:
color:
hungry:
energy:
---------------
Animal(naname,color,hungry,energy)
getHungry()
getEnergy()
greeting() -> I'm + [name] // same on both animal
isAlive()
play()
sleep()
feedFood(food)
```

Subclass only have things that differ from superclass!!!

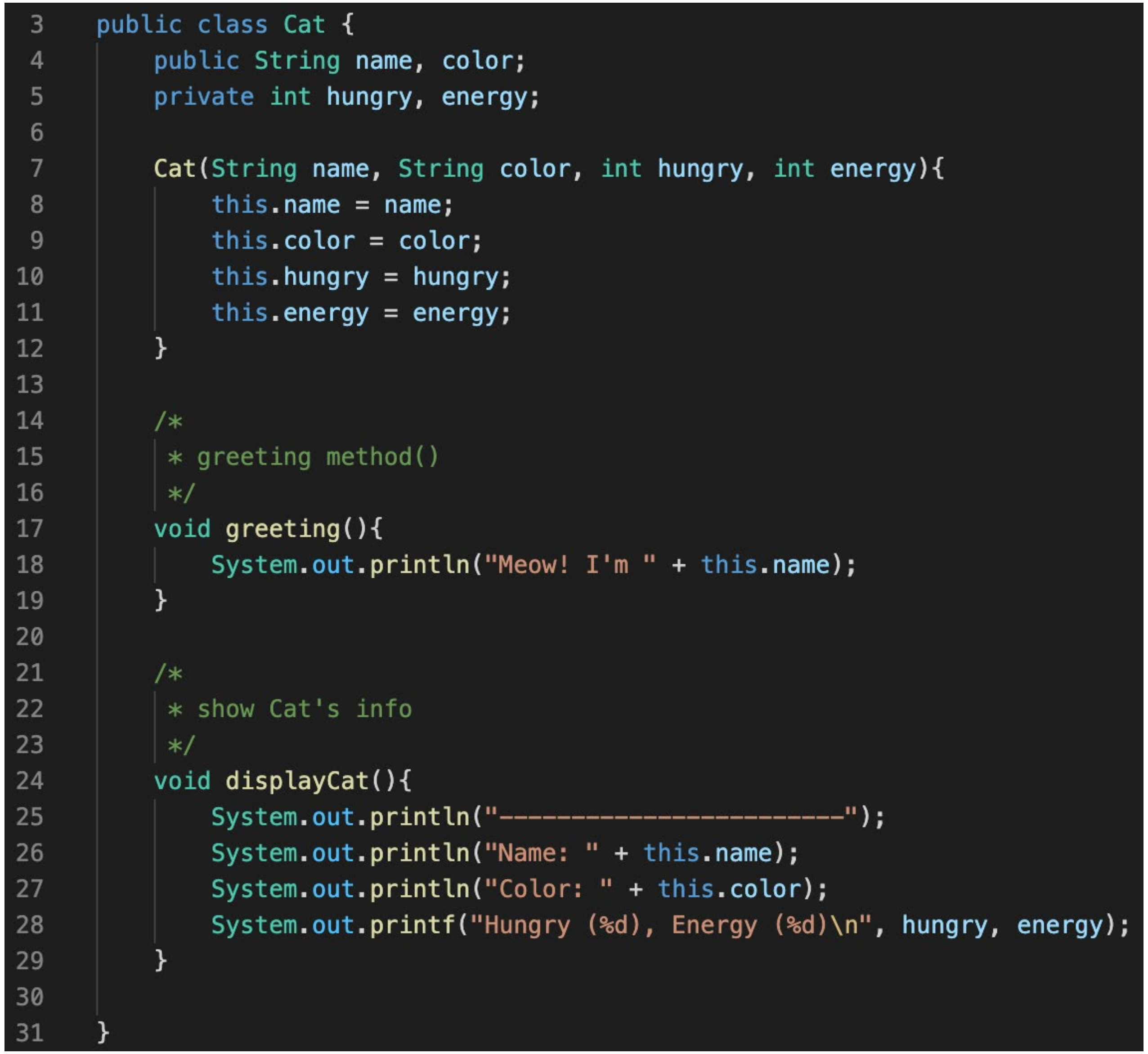
Cat Class

```
---------------
Cat(nanme,color,hungry,energy)
greeting() -> Meow! + super.greeting()
displayCat()
```

Dog Class

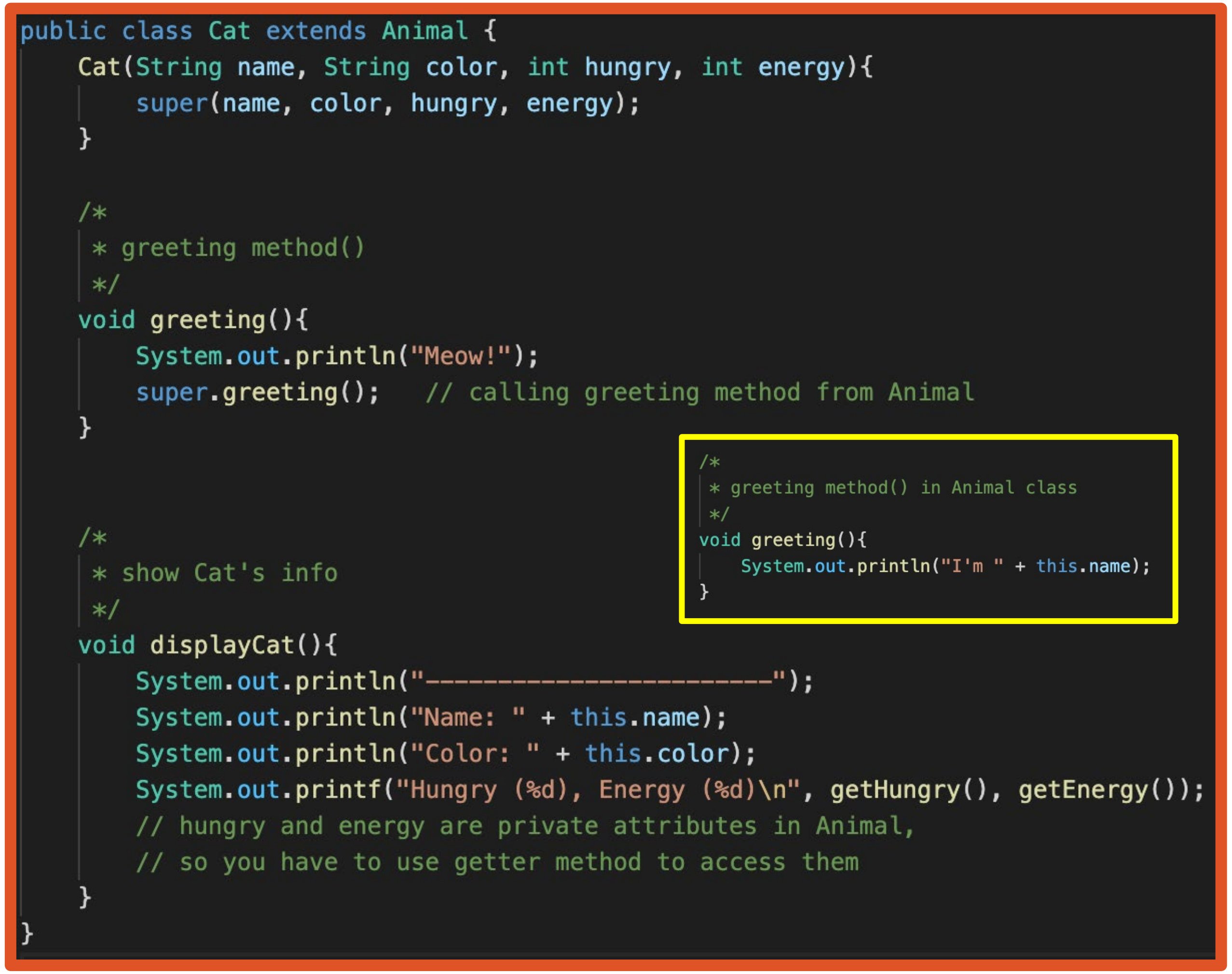
```
sense: // sense of smell
---------------
Dog(nanme,color,hungry,energy,sense)
getSense()
greeting() -> Bark! + super.greeting()
displayDog()
```

Do you see any different between these two Cat classes?



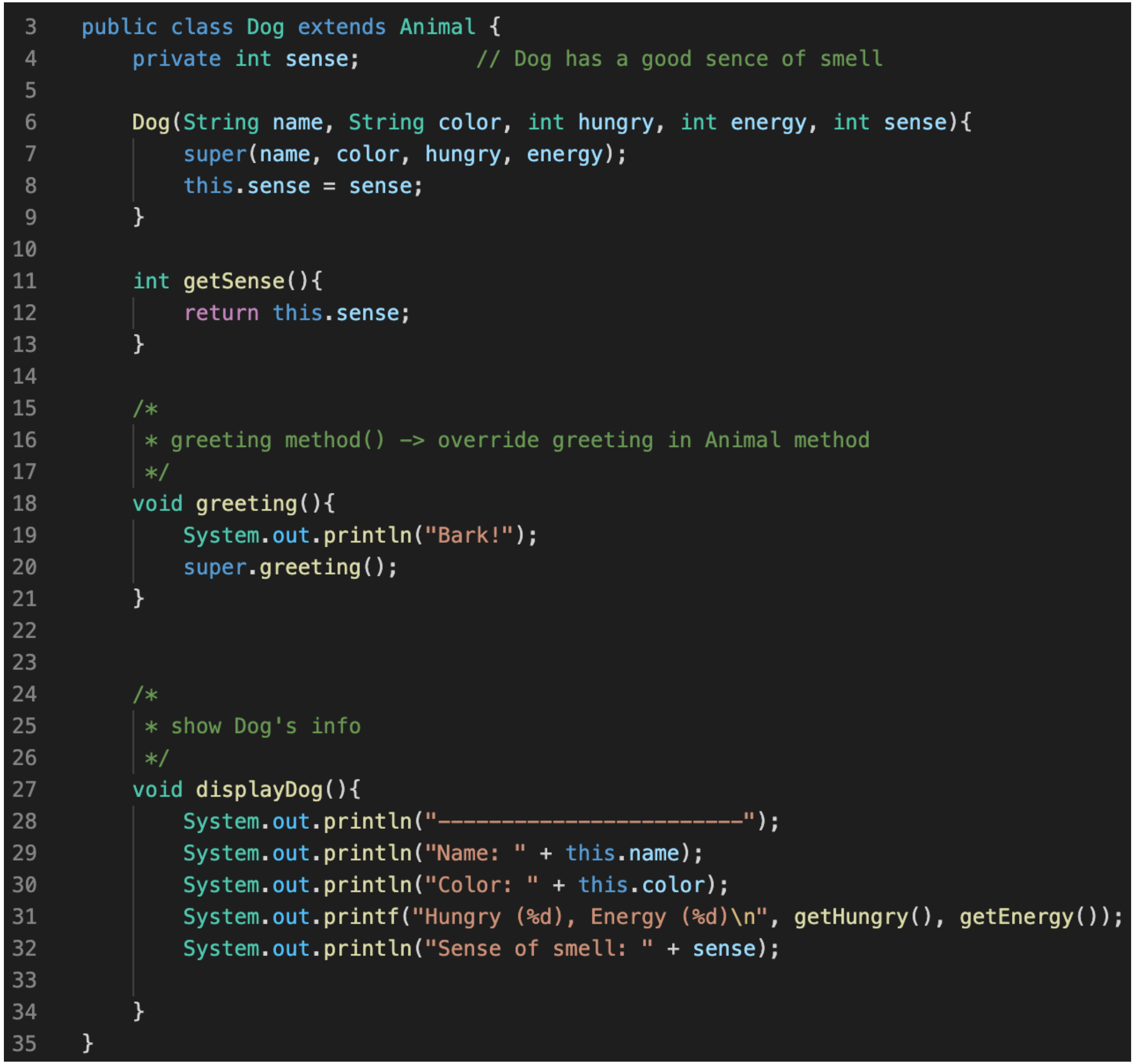
```java
public class Cat {
    public String name, color;
    private int hungry, energy;

    Cat(String name, String color, int hungry, int energy){
        this.name = name;
        this.color = color;
        this.hungry = hungry;
        this.energy = energy;
    }

    /*
     * greeting method()
     */
    void greeting(){
        System.out.println("Meow! I'm " + this.name);
    }

    /*
     * show Cat's info
     */
    void displayCat(){
        System.out.println("------------------------");
        System.out.println("Name: " + this.name);
        System.out.println("Color: " + this.color);
        System.out.printf("Hungry (%d), Energy (%d)\n", hungry, energy);
    }

}
```

Extends, Attribute, Constructor, body of greeting method, how to access common attribute which are private?

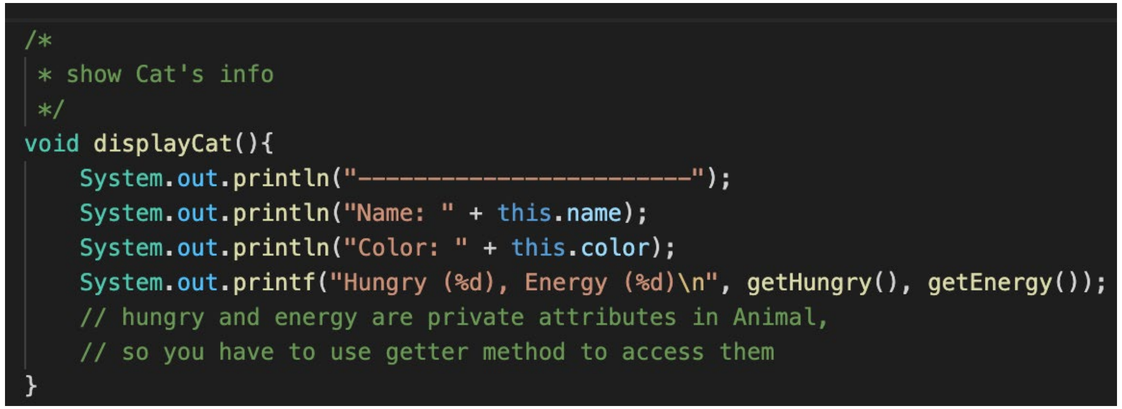


```java
public class Cat extends Animal {
    Cat(String name, String color, int hungry, int energy){
        super(name, color, hungry, energy);
    }

    /*
     * greeting method()
     */
    void greeting(){
        System.out.println("Meow!");
        super.greeting();   // calling greeting method from Animal
    }


    /*
     * show Cat's info
     */
    void displayCat(){
        System.out.println("------------------------");
        System.out.println("Name: " + this.name);
        System.out.println("Color: " + this.color);
        System.out.printf("Hungry (%d), Energy (%d)\n", getHungry(), getEnergy());
        // hungry and energy are private attributes in Animal,
        // so you have to use getter method to access them
    }
}
```

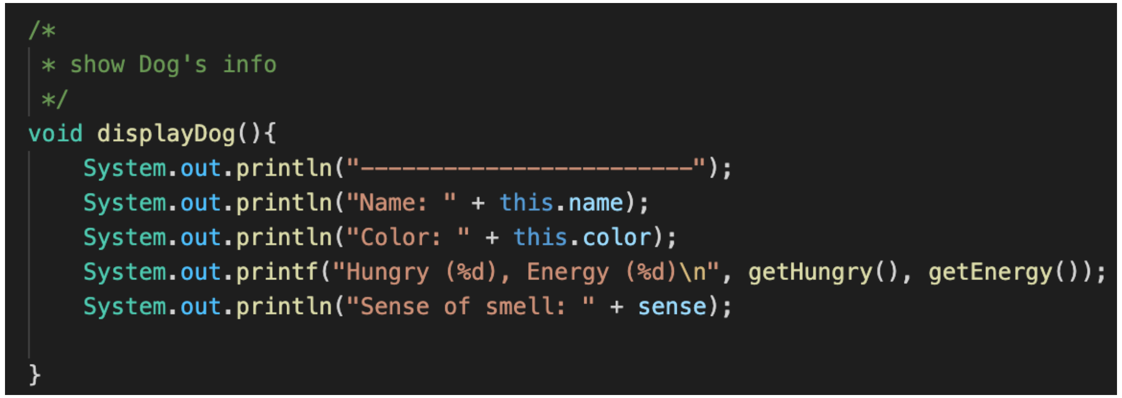
```java
/*
 * greeting method() in Animal class
 */
void greeting(){
    System.out.println("I'm " + this.name);
}
```



```java
public class Dog extends Animal {
    private int sense;         // Dog has a good sence of smell

    Dog(String name, String color, int hungry, int energy, int sense){
        super(name, color, hungry, energy);
        this.sense = sense;
    }

    int getSense(){
        return this.sense;
    }

    /*
     * greeting method() -> override greeting in Animal method
     */
    void greeting(){
        System.out.println("Bark!");
        super.greeting();
    }


    /*
     * show Dog's info
     */
    void displayDog(){
        System.out.println("------------------------");
        System.out.println("Name: " + this.name);
        System.out.println("Color: " + this.color);
        System.out.printf("Hungry (%d), Energy (%d)\n", getHungry(), getEnergy());
        System.out.println("Sense of smell: " + sense);

    }
}
```



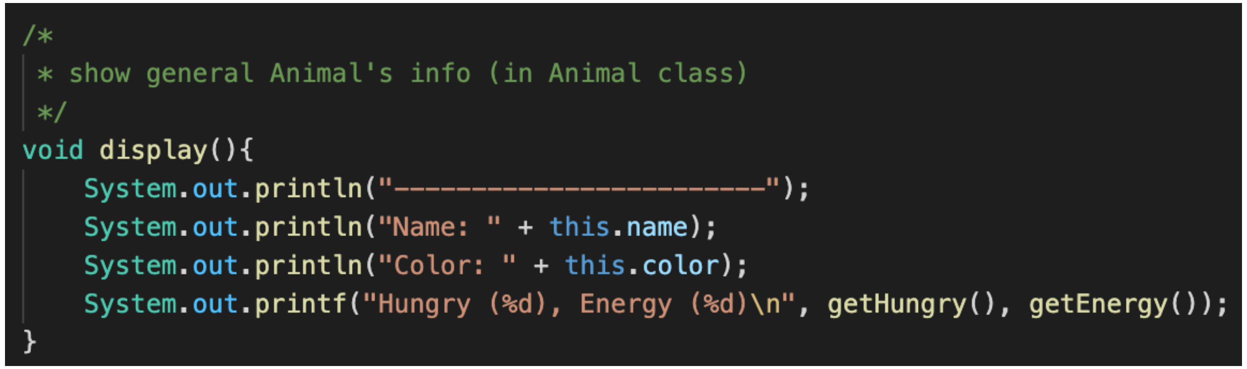
```
/*
 * show Cat's info
 */
void displayCat(){
    System.out.println("------------------------");
    System.out.println("Name: " + this.name);
    System.out.println("Color: " + this.color);
    System.out.printf("Hungry (%d), Energy (%d)\n", getHungry(), getEnergy());
    // hungry and energy are private attributes in Animal,
    // so you have to use getter method to access them
}
```



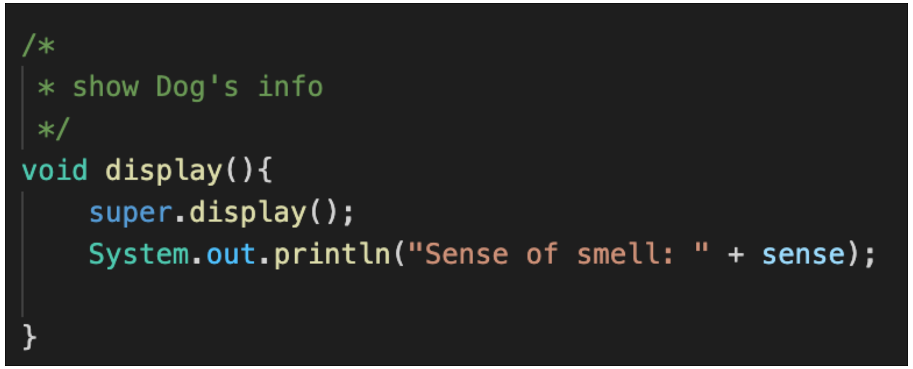
```
/*
 * show Dog's info
 */
void displayDog(){
    System.out.println("------------------------");
    System.out.println("Name: " + this.name);
    System.out.println("Color: " + this.color);
    System.out.printf("Hungry (%d), Energy (%d)\n", getHungry(), getEnergy());
    System.out.println("Sense of smell: " + sense);

}
```

These two methods have different name yet have very similar behavior (method's body).

So we can simplify them and put the common code in superclass (Animal)



```
/*
 * show general Animal's info (in Animal class)
 */
void display(){
    System.out.println("------------------------");
    System.out.println("Name: " + this.name);
    System.out.println("Color: " + this.color);
    System.out.printf("Hungry (%d), Energy (%d)\n", getHungry(), getEnergy());
}
```

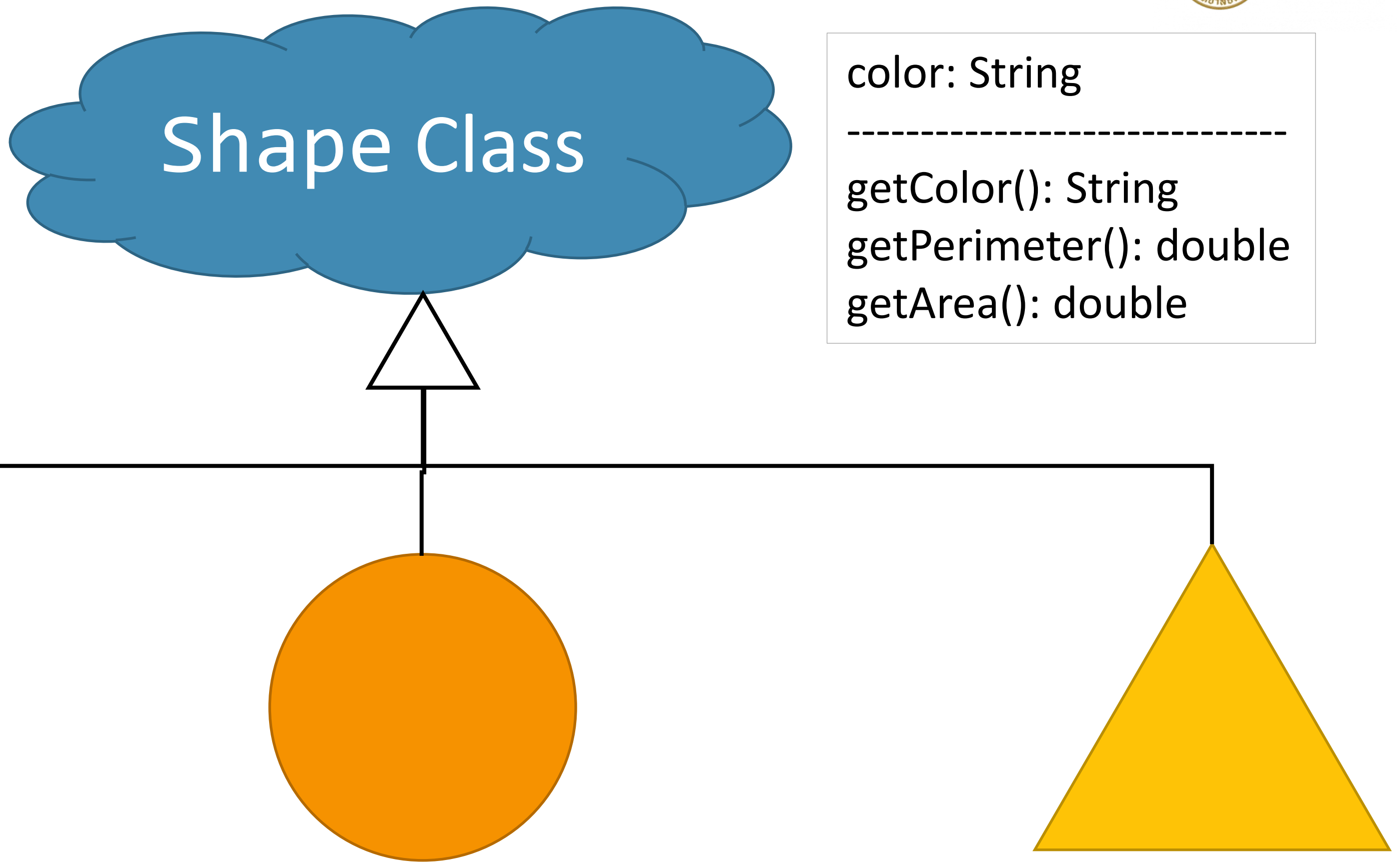


```
/*
 * show Dog's info
 */
void display(){
    super.display();
    System.out.println("Sense of smell: " + sense);

}
```

Let’s try some more examples

- `Student` Class
  - Data variables: student's id, name, gpa
  - Methods: getInfo(), getID(), getName(), enroll(Course x), getGrade(Coure x), is_gradudated()
- `Instructor` Class
  - Data variables: - instructor’s id, name, salary
  - Methods: getInfo(), getID(), getName(), teach(Course x), getSalary(), raiseSalary(), calculateBonus()
- Can you create `Person` superclasse?

Practice Again!



color: String

----

getColor(): String
getPerimeter(): double
getArea(): double

Class Learning Outcome Today

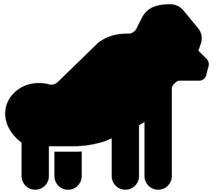
- To explain the difference between overriding and overloading methods
- To demonstrate the use of an appropriate type of object and type of variable

```
TypeVariable variableName = new TypeObject(list of params)
```

```
Food f = new Snack(“xxx”, 200, “bag”);
Food f = new Food(“yyy”, 300);
```

Part 1:
Dynamic Binding in Java

Superclass
(general class)



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

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

    public String getName(){
        return this.name;
    }

    public void greeting(){
        System.out.println("I'm " + name);
    }
}
```

What will be the output of this?

```java
Animal a = new Animal("Olaf");
a.greeting();

Cat c = new Cat("Elsa");
c.greeting();

Dog d = new Dog("Pika");
d.greeting();
```

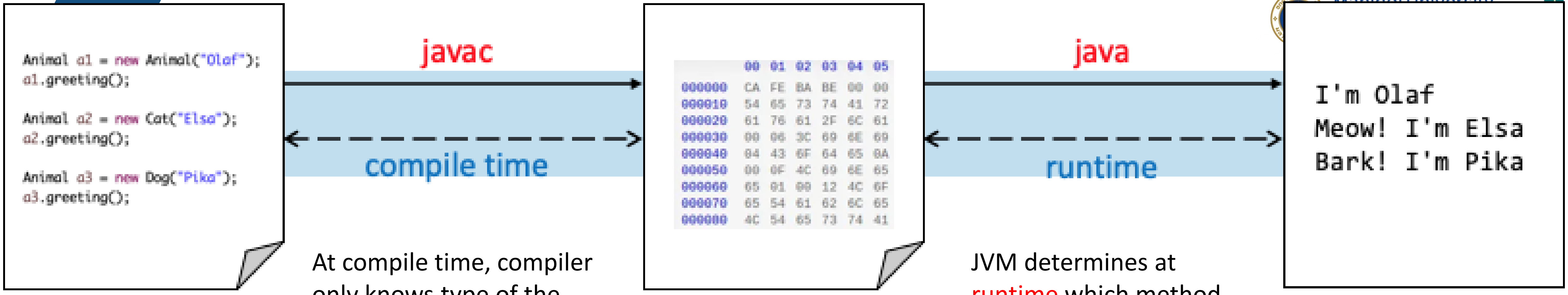
```java
Animal a1 = new Animal("Olaf");
a1.greeting();

Animal a2 = new Cat("Elsa");
a2.greeting();

Animal a3 = new Dog("Pika");
a3.greeting();
```

```java
Animal[] pets = new Animal[3];
pets[0] = new Animal("Olaf");
pets[1] = new Cat("Elsa");
pets[2] = new Dog("Pika");

for(Animal p: pets){
    p.greeting();
}
```

Subclass
(specialized class)

```java
public class Cat extends Animal{
    public Cat(String name){
        super(name);
    }

    @Override
    public void greeting(){
        System.out.print("Meow!  ");
        super.greeting();
    }

    public void chasing(){
        System.out.println("Chasing mouse...");
    }

}
```

```java
public class Dog extends Animal{

    public Dog(String name) {
        super(name);
    }

    @Override
    public void greeting(){
        System.out.print("Bark!  ");
        super.greeting();
    }

    public void catching(){
        System.out.println("Catching frisbee...");
    }

}
```



```
Animal a1 = new Animal("Olaf");
a1.greeting();

Animal a2 = new Cat("Elsa");
a2.greeting();

Animal a3 = new Dog("Pika");
a3.greeting();
```

```
        00 01 02 03 04 05
000000  CA FE BA BE 00 00
000010  54 65 73 74 41 72
000020  61 76 61 2F 6C 61
000030  00 06 3C 69 6E 69
000040  04 43 6F 64 65 0A
000050  00 0F 4C 69 6E 65
000060  65 01 00 12 4C 6F
000070  65 54 61 62 6C 65
000080  4C 54 65 73 74 41
```

```
I'm Olaf
Meow! I'm Elsa
Bark! I'm Pika
```

Tester.java

At compile time, compiler only knows type of the variable. So it can only check wheter the method is available for that type.

Tester.class

JVM determines at runtime which method to call depending on the **type of the object** (**NOT** the type of the variable)

Program output (Console)

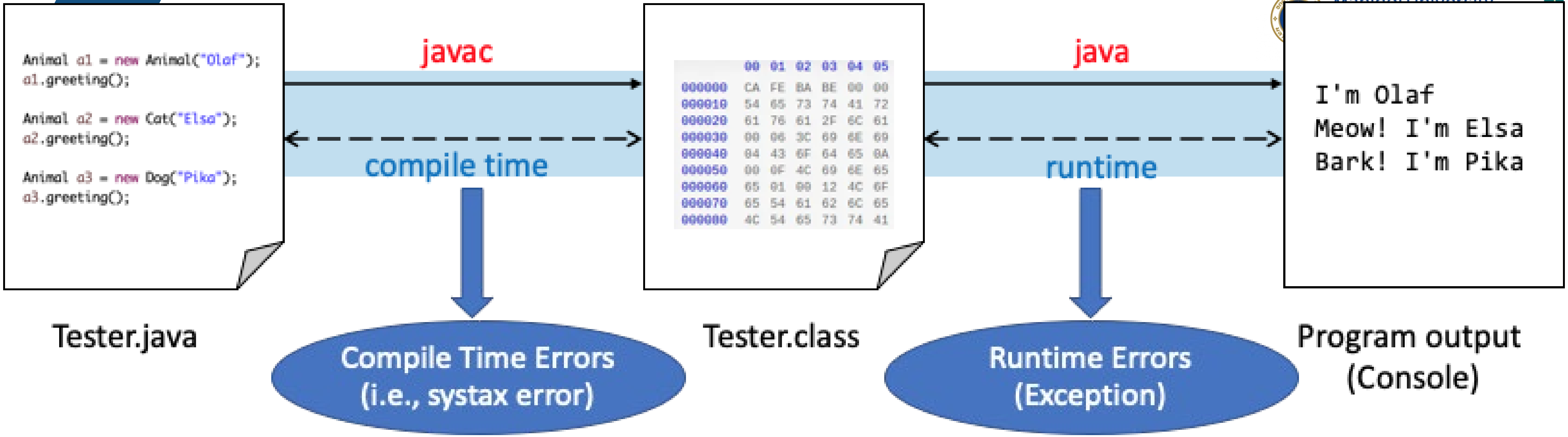
```
TypeVariable variableName = new TypeObject(list of params)
```

***** SAME OUTPUT *****

```
Animal a = new Animal("Olaf");
a.greeting();

Cat c = new Cat("Elsa");
c.greeting();

Dog d = new Dog("Pika");
d.greeting();
```

```
Animal a1 = new Animal("Olaf");
a1.greeting();

Animal a2 = new Cat("Elsa");
a2.greeting();

Animal a3 = new Dog("Pika");
a3.greeting();
```

```
Animal[] pets = new Animal[3];
pets[0] = new Animal("Olaf");
pets[1] = new Cat("Elsa");
pets[2] = new Dog("Pika");

for(Animal p: pets){
    p.greeting();
}
```



```
Animal a1 = new Animal("Olaf");
a1.greeting();

Animal a2 = new Cat("Elsa");
a2.greeting();

Animal a3 = new Dog("Pika");
a3.greeting();
```

```
        00 01 02 03 04 05
000000  CA FE BA BE 00 00
000010  54 65 73 74 41 72
000020  61 76 61 2F 6C 61
000030  00 06 3C 69 6E 69
000040  04 43 6F 64 65 0A
000050  00 0F 4C 69 6E 65
000060  65 01 00 12 4C 6F
000070  65 54 61 62 6C 65
000080  4C 54 65 73 74 41
```

```
I'm Olaf
Meow! I'm Elsa
Bark! I'm Pika
```

```
Animal a1 = new Animal("Olaf");
a1.greeting();

Animal a2 = new Cat("Elsa");
a2.chasing();

Animal a3 = new Dog("Pika");
a3.catching();
```

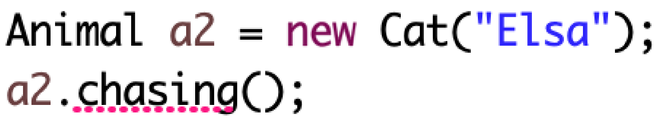
```
Animal a1 = new Animal("Olaf");
a1.greeting();

Animal a2 = new Cat("Elsa");
((Cat)a2).chasing();

Animal a3 = new Dog("Pika");
((Cat)a3).chasing();
```

```
Exception in thread "main" java.lang.ClassCastException:
```

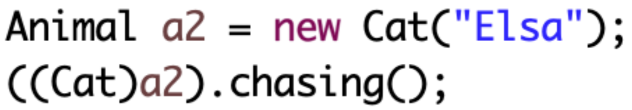
Part 2:
instanceof operator
and Casting

instanceof Operator

- As you know, superclass cannot see or access to the subclass variables and methods.
- If you declare variable with superclass type, you cannot call the method of subclass -> Compile error. SO, you will have to **cast** first.

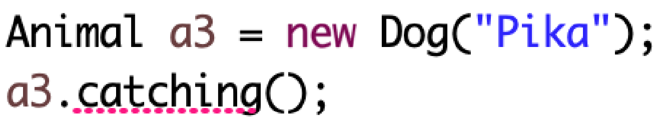


```
Animal a2 = new Cat("Elsa");
a2.chasing();
```

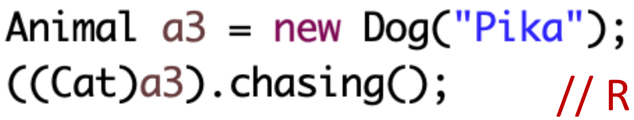


```
Animal a2 = new Cat("Elsa");
((Cat)a2).chasing();
```

- However, casting can be dangerous. What if the object doesn't belong that class.



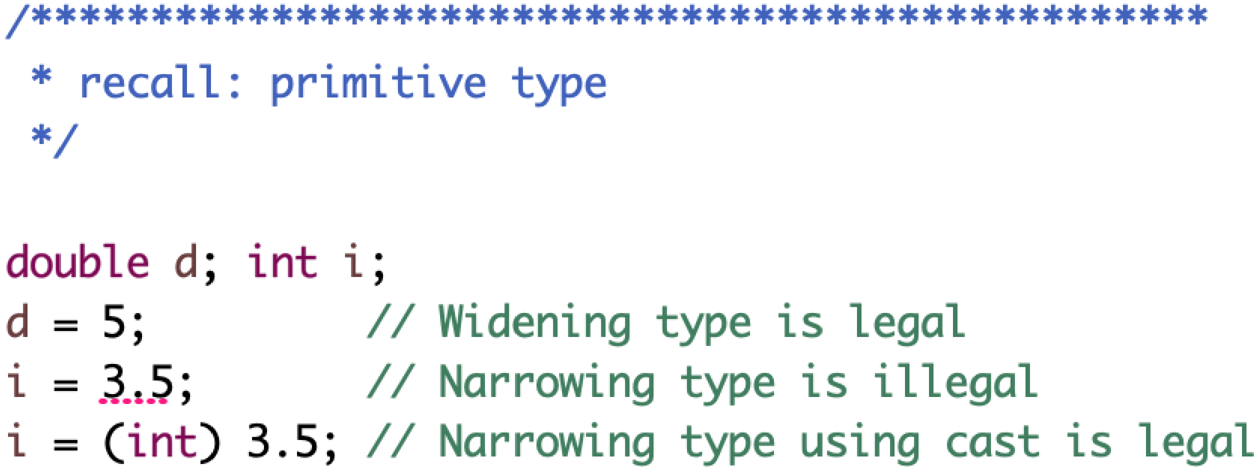
```
Animal a3 = new Dog("Pika");
a3.catching();
```



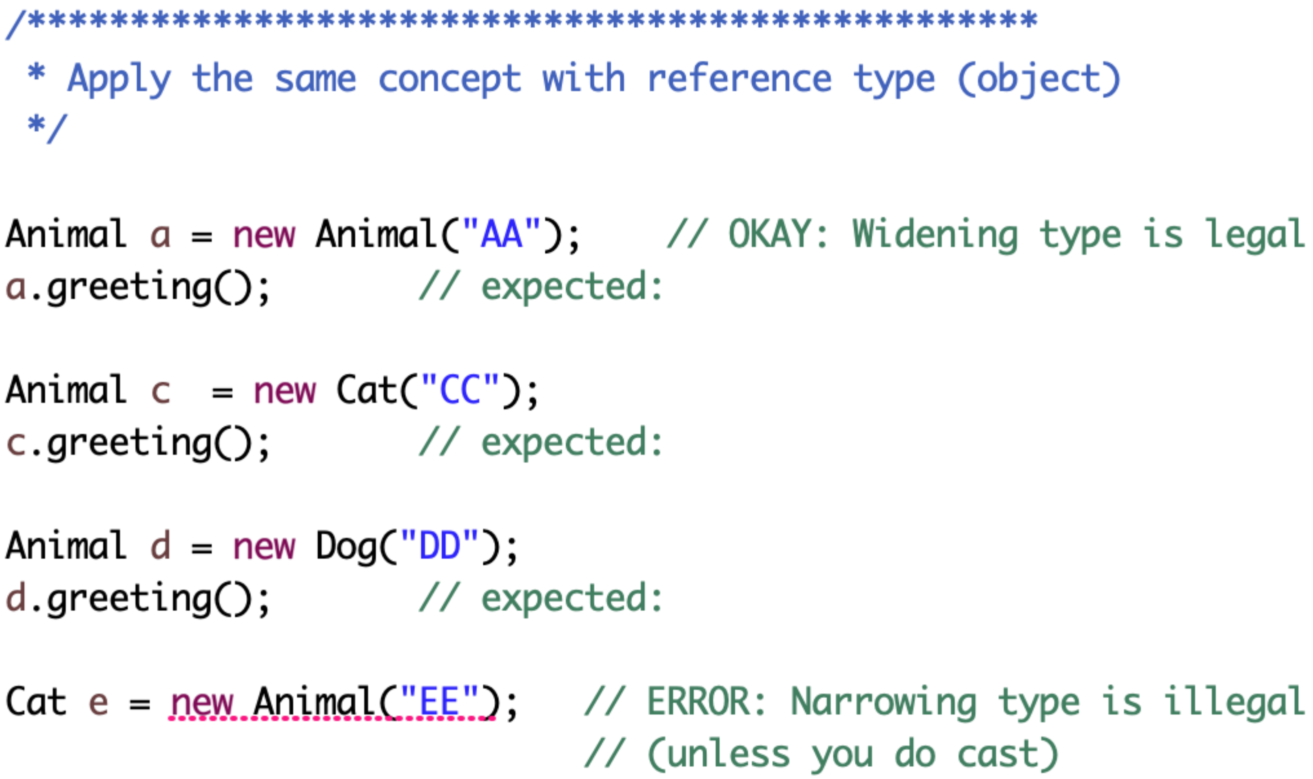
```
Animal a3 = new Dog("Pika");
((Cat)a3).chasing();
```

// RUNTIME ERROR

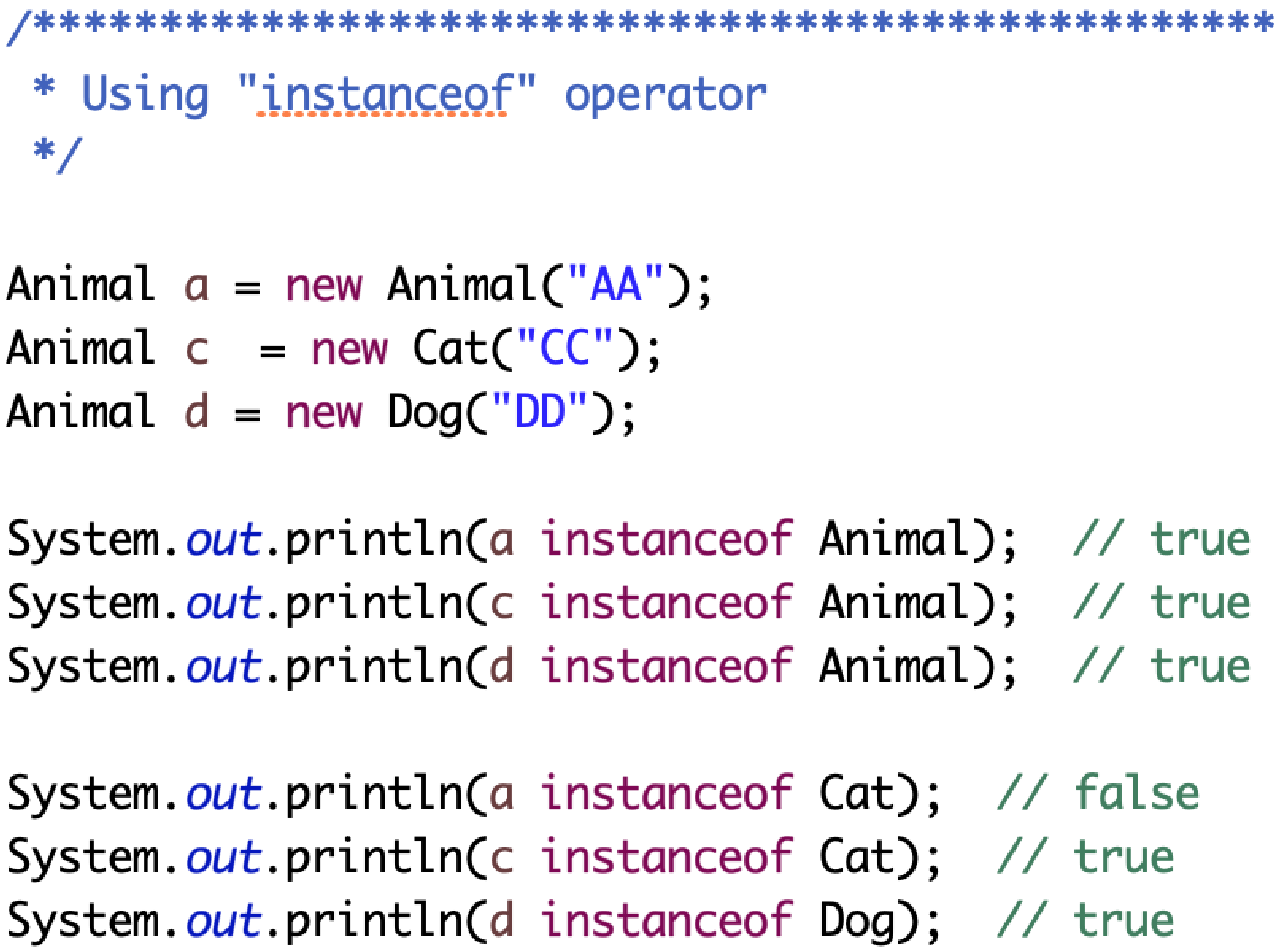
- SO, it is a good idea to check the class of an object before casting using `instanceof` operator



```java
/*************************************************
 * recall: primitive type
 */

double d; int i;
d = 5;          // Widening type is legal
i = 3.5;        // Narrowing type is illegal
i = (int) 3.5; // Narrowing type using cast is legal
```

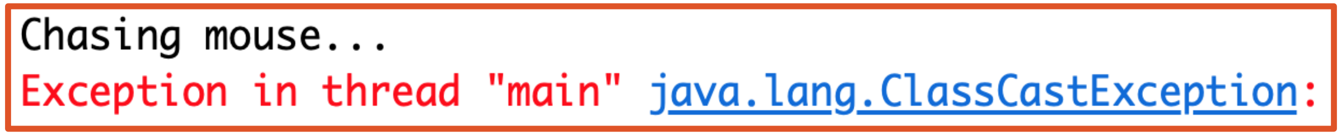


```java
/*************************************************
 * Apply the same concept with reference type (object)
 */

Animal a = new Animal("AA");    // OKAY: Widening type is legal
a.greeting();        // expected:

Animal c  = new Cat("CC");
c.greeting();        // expected:

Animal d = new Dog("DD");
d.greeting();        // expected:

Cat e = new Animal("EE");   // ERROR: Narrowing type is illegal
                            // (unless you do cast)
```



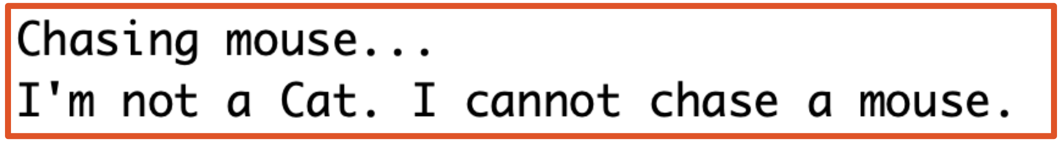
```
/******************************************************
 * Using "instanceof" operator
 */

Animal a = new Animal("AA");
Animal c  = new Cat("CC");
Animal d = new Dog("DD");

System.out.println(a instanceof Animal);  // true
System.out.println(c instanceof Animal);  // true
System.out.println(d instanceof Animal);  // true

System.out.println(a instanceof Cat);  // false
System.out.println(c instanceof Cat);  // true
System.out.println(d instanceof Dog);  // true
```

```
/*************************************************
 * Casting from a generic reference to more specific object.
 */
Animal c  = new Cat("CC");
Animal d = new Dog("DD");

Cat x = (Cat) c;
x.chasing();    // OK

Cat y = (Cat) d; // Run-time ERROR: ClassCastException since d is Dog
y.chasing();
```

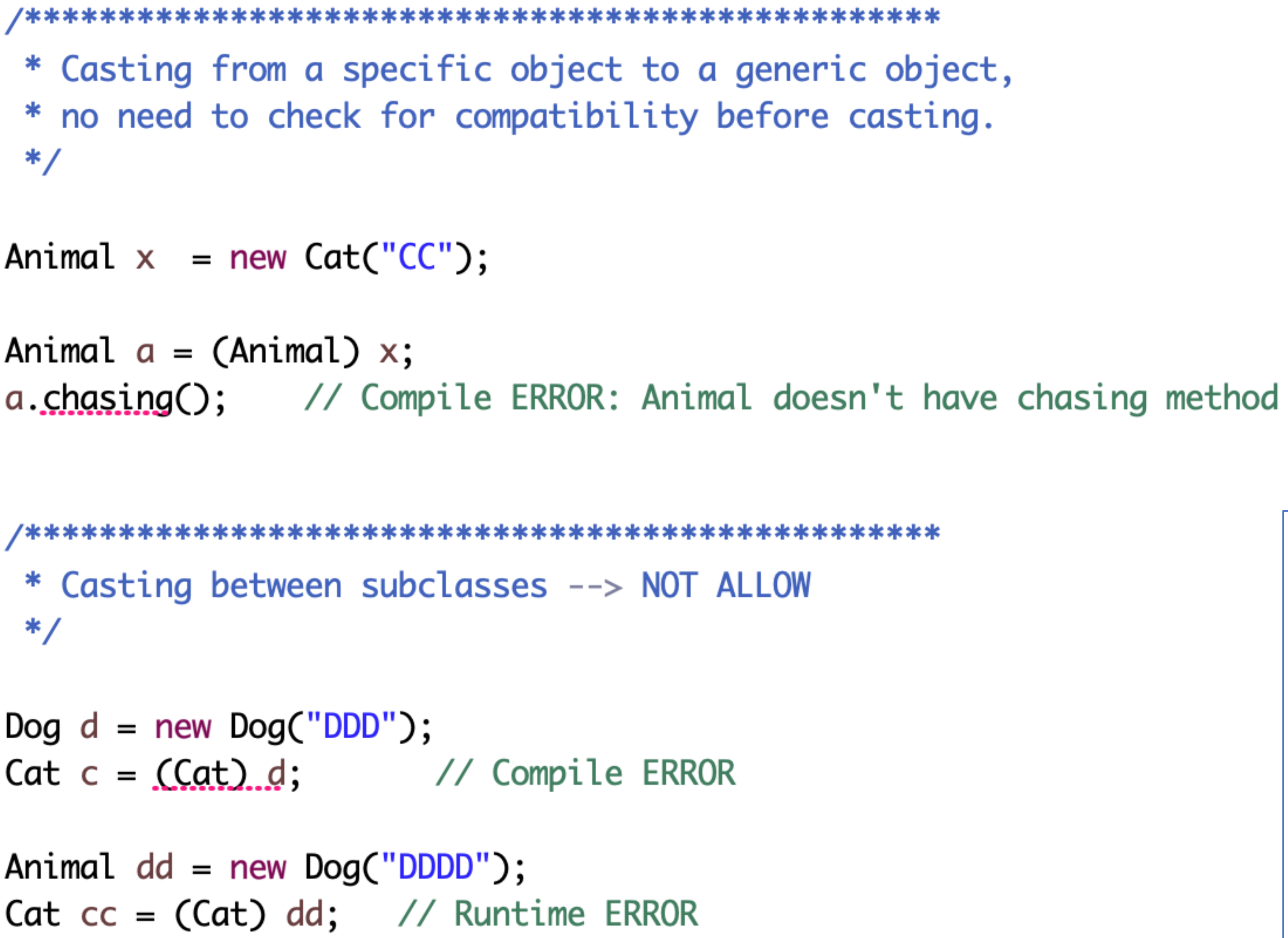


```
Chasing mouse...
Exception in thread "main" java.lang.ClassCastException:
```

```
/*************************************************
 * To avoid casting error,
 * one must check for compatibility (using instanceof)
 */

if(d instanceof Cat){
    Cat z = (Cat) d;
    z.chasing();
} else{
    System.out.println("I'm not a Cat. I cannot chase a mouse.");
}
```

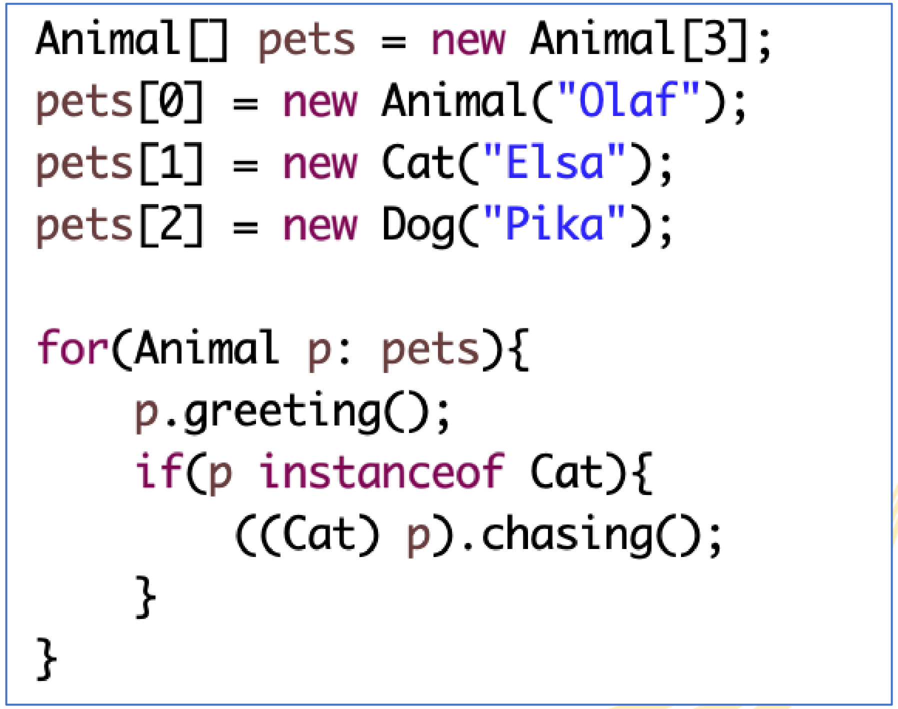


```
Chasing mouse...
I'm not a Cat. I cannot chase a mouse.
```



```
/************************************************
 * Casting from a specific object to a generic object,
 * no need to check for compatibility before casting.
 */

Animal x  = new Cat("CC");

Animal a = (Animal) x;
a.chasing();    // Compile ERROR: Animal doesn't have chasing method

/************************************************
 * Casting between subclasses --> NOT ALLOW
 */

Dog d = new Dog("DDD");
Cat c = (Cat) d;        // Compile ERROR

Animal dd = new Dog("DDDD");
Cat cc = (Cat) dd;    // Runtime ERROR
```

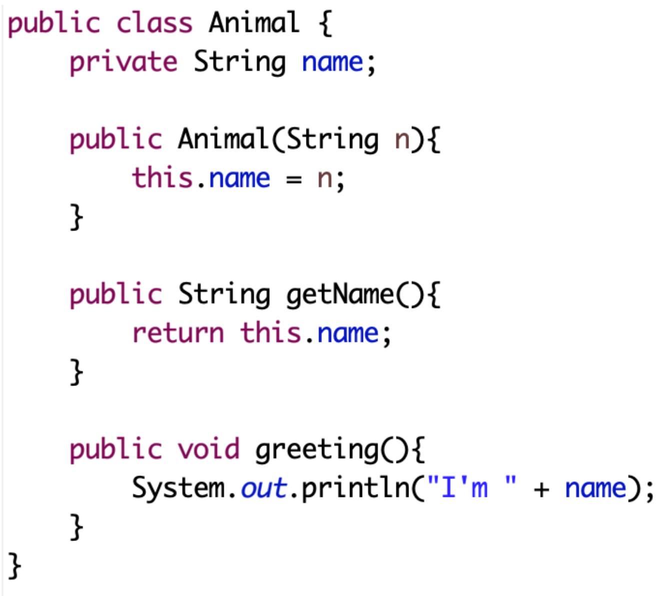
Another example of how to use **instanceof** operator in the Array of superclass



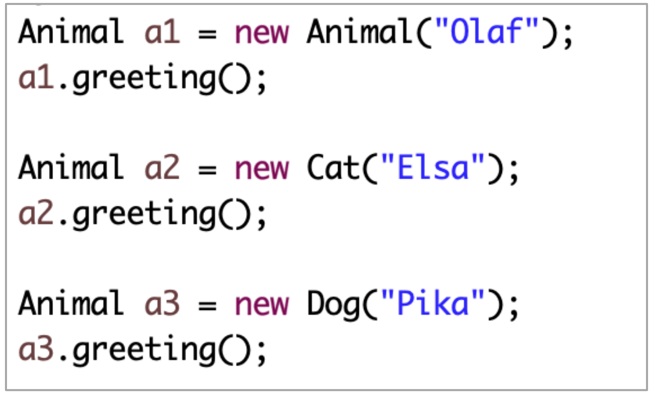
```
Animal[] pets = new Animal[3];
pets[0] = new Animal("Olaf");
pets[1] = new Cat("Elsa");
pets[2] = new Dog("Pika");

for(Animal p: pets){
    p.greeting();
    if(p instanceof Cat){
        ((Cat) p).chasing();
    }
}
```

Part 3:
Override and Overload

Can we do this?



```
public class Animal {
    private String name;

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

    public String getName(){
        return this.name;
    }

    public void greeting(){
        System.out.println("I'm " + name);
    }
}
```



```
Animal a1 = new Animal("Olaf");
a1.greeting();

Animal a2 = new Cat("Elsa");
a2.greeting();

Animal a3 = new Dog("Pika");
a3.greeting();
```

```
public static void main(String args[]) {
    System.out.println("Hi! I love java.");
    System.out.println(100);
    System.out.println(true);
    System.out.println(99.99999 + "%");

    myPrint(5);         // call myPrint(   ?   )
    myPrint(5.0);       // call myPrint(   ?   )
}

static void myPrint(int i) {
    System.out.println("int i = " + i);
}

static void myPrint(double d) {
    System.out.println("double d = " + d);
}
```

How about this?

```
public static void main(String args[]) {
    myPrint(5);        // call myPrint(int)
    myPrint(5.0);      // call myPrint(double)
}

static void myPrint(int i) {
    System.out.println("int i = " + i);
}

static void myPrint(double d) {
    System.out.println("double d = " + d);
}

static double myPrint(double d) {
    System.out.println("double d = " + d);
    return d
}
```

So.. What is Polymorphism?

- **‘poly’ = many,** and **‘morph” = forms**
- **Polymorphism = “having multiple forms”** allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.
  - In the real world, you can use universal remote control to control all kinds of digital TV. Even though each TV, each bran may execute differently
  - In oop, it describes the concept that you can access objects of different types through the same interface. This makes program *easily extensible*. (simply say – calling the same method but doing different ways depends on the objects)

I don’t care what brand or model you are, as long as you are a digital TV, I can control you.

- Java support two kinds of polymorphism -> **overload** and **override**

Polymorphism

Overloading

- Two or more methods with different signatures

Overriding

(Note that we already leared this last week!!)

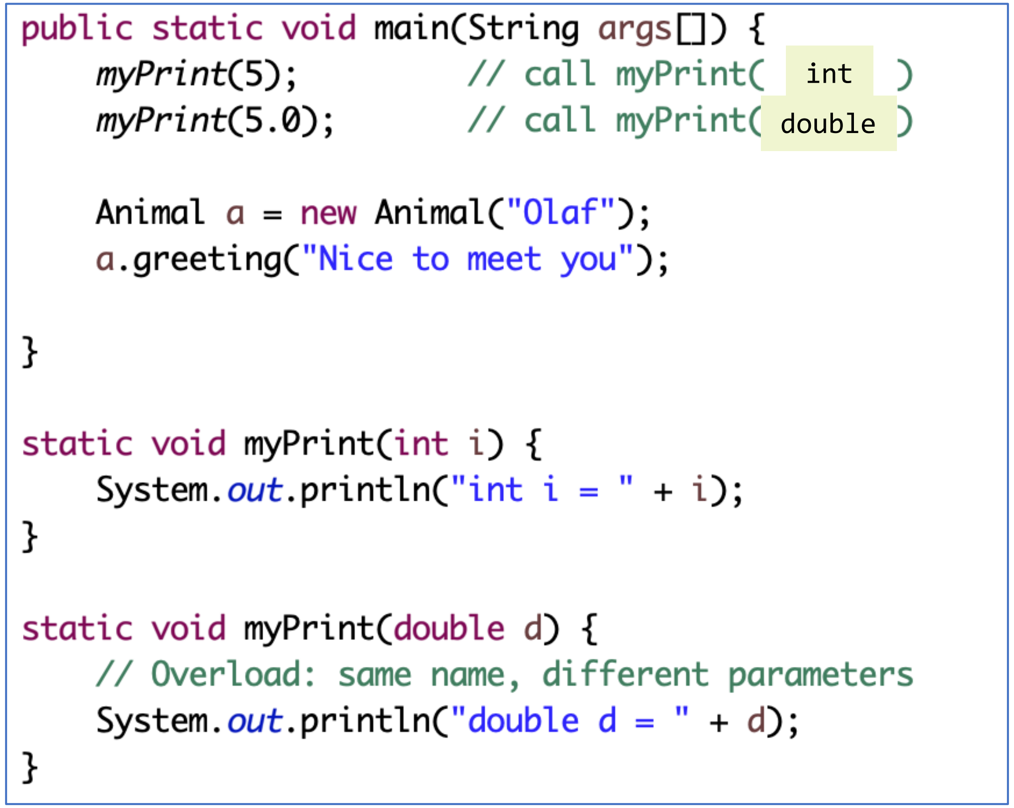
- Replacing an inherited method with another method having the same signature

Signature in Java

`foo(int i)` and `foo(int i, int j)` are different

`foo(int i)` and `foo(int k)` are the same

`foo(int i, double d)` and `foo(double d, int i)` are different



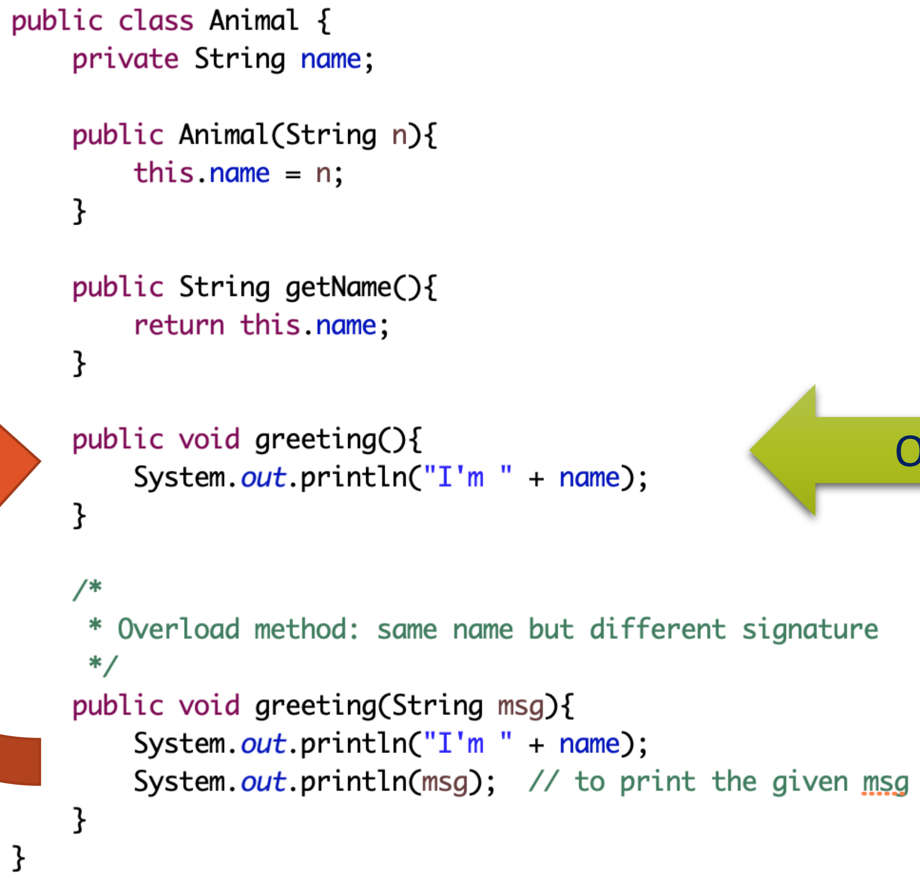
```
public static void main(String args[]) {
    myPrint(5);          // call myPrint( int )
    myPrint(5.0);        // call myPrint( double )

    Animal a = new Animal("Olaf");
    a.greeting("Nice to meet you");

}

static void myPrint(int i) {
    System.out.println("int i = " + i);
}

static void myPrint(double d) {
    // Overload: same name, different parameters
    System.out.println("double d = " + d);
}
```

Overloading

Overload: in the same class, two methods can have the same name, provided they differ in their parameter types. These are different methods, each with its own implementation. The Java compiler considers them to be completely unrelated.

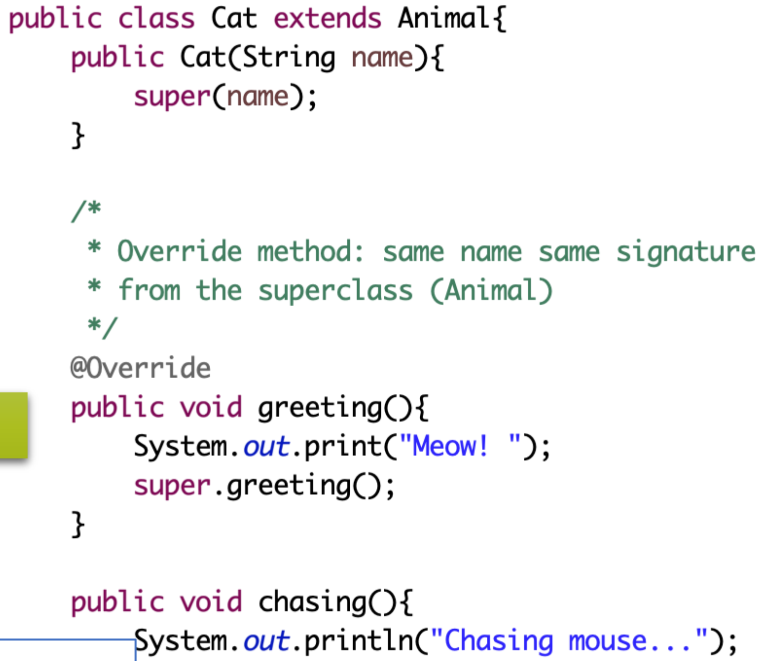
Overriding: where a subclass method provides an implementation of a method whose parameter variables have the same types.

[Note! If you mean to override a method but use a parameter variable with a different type, then you accidentally introduce an overloaded method.]



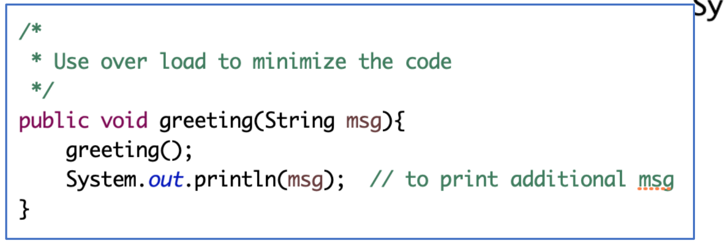
```java
public class Animal {
    private String name;

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

    public String getName(){
        return this.name;
    }

    public void greeting(){
        System.out.println("I'm " + name);
    }

    /*
     * Overload method: same name but different signature
     */
    public void greeting(String msg){
        System.out.println("I'm " + name);
        System.out.println(msg);   // to print the given msg
    }
}
```

Override greeting() method



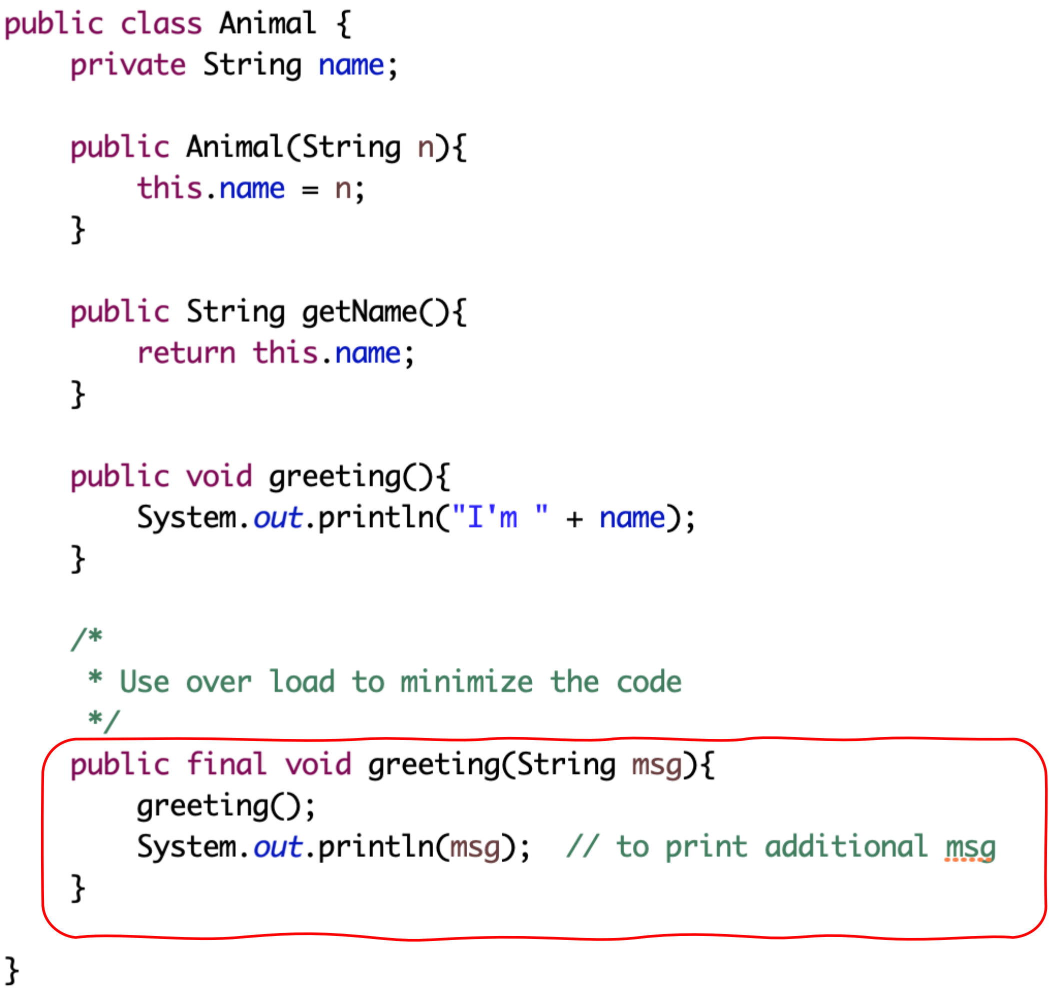
```java
public class Cat extends Animal{
    public Cat(String name){
        super(name);
    }

    /*
     * Override method: same name same signature
     * from the superclass (Animal)
     */
    @Override
    public void greeting(){
        System.out.print("Meow! ");
        super.greeting();
    }

    public void chasing(){
        System.out.println("Chasing mouse...");
```

Example overloading usage

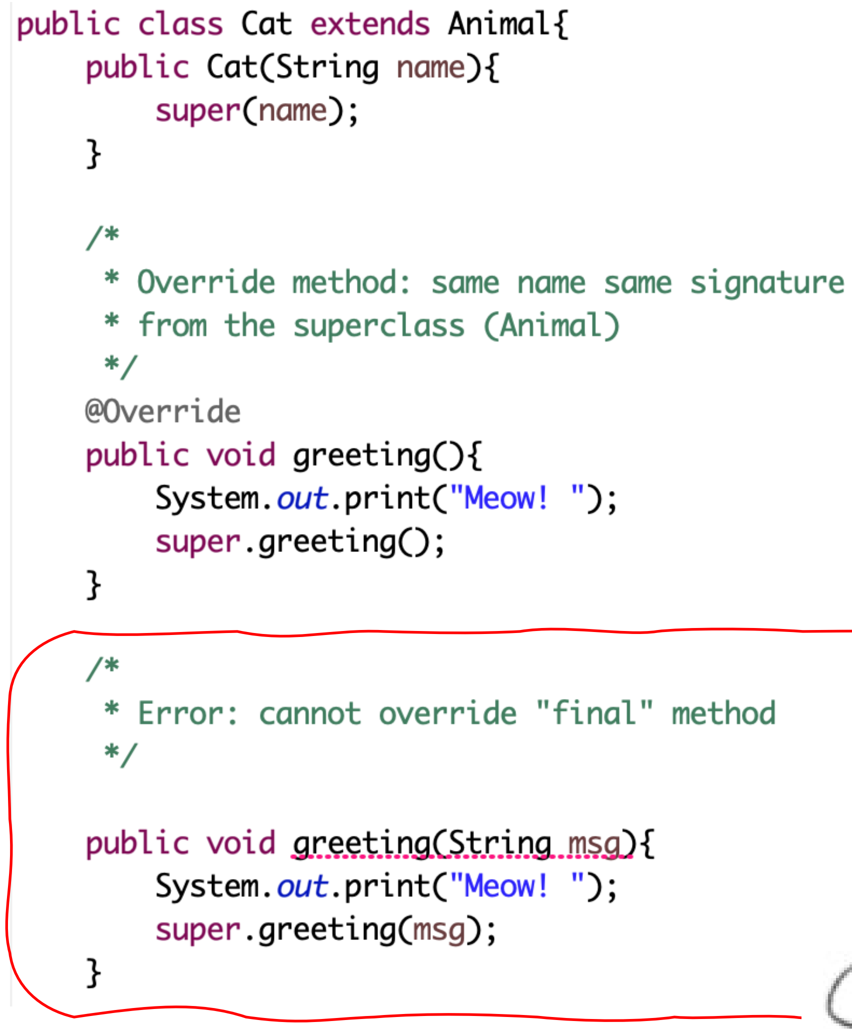


```java
/*
 * Use over load to minimize the code
 */
public void greeting(String msg){
    greeting();
    System.out.println(msg);  // to print additional msg
}
```

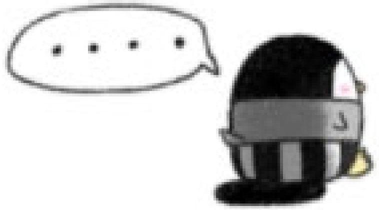
final method



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

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

    public String getName(){
        return this.name;
    }

    public void greeting(){
        System.out.println("I'm " + name);
    }

    /*
     * Use over load to minimize the code
     */
    public final void greeting(String msg){
        greeting();
        System.out.println(msg);   // to print additional msg
    }

}
```



```java
public class Cat extends Animal{
    public Cat(String name){
        super(name);
    }

    /*
     * Override method: same name same signature
     * from the superclass (Animal)
     */
    @Override
    public void greeting(){
        System.out.print("Meow!  ");
        super.greeting();
    }

    /*
     * Error: cannot override "final" method
     */

    public void greeting(String msg){
        System.out.print("Meow!  ");
        super.greeting(msg);
    }
```



Tip: Avoid type tests and use Polymorphims instead

Some programmers use specific type tests in order to implement behavior that varies with each class. For eample to find area of different kinds of shape.

```
if(a instanceof Circle){
  // find area with circle formula
} else if (a instanceof Square) {
  // find area with square formula
}
```

Then, if there is a new class “Triangle”, you will have to change your main program to suppport new class.

```
else if (a instance of Triangle) {
    // find area with triangle formula
}
```

Actually, when you add a new class Triangle extends from Shape, you can most likely run the existing program without any error.

Whenever you find yourself trying to use **type tests** in a hierarchy of classes, reconsider and use **polymorphism** instead. Declare a method doTheTask in the superclass, override it in the subclasses, and call

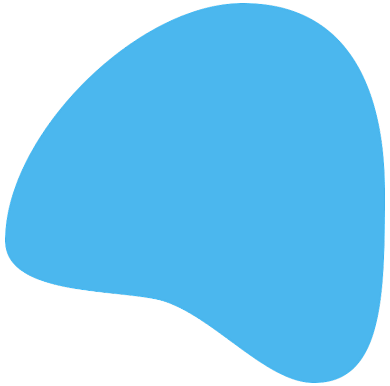
```
a.getArea();
```

The program automatically checks object type and calls a method based on that object type for you =D

Abstract Classes

- When you extend an existing class, you have the *choice* whether or not to override the methods of the superclass. Sometimes, it is desirable to **force programmers to override a method**.
- That happens when there is no good default for the superclass, and only the subclass programmer can know how to implement the method properly.

For eample, we might not know how to calculate the area of an uknown shape yet. Only the subclass (e.g., circle, sqaure, triangle) knows how to find its area.



This is too abstract!!!

```
public class Shape {

    public  double getArea() {
        // what should we do???
    }
}
```

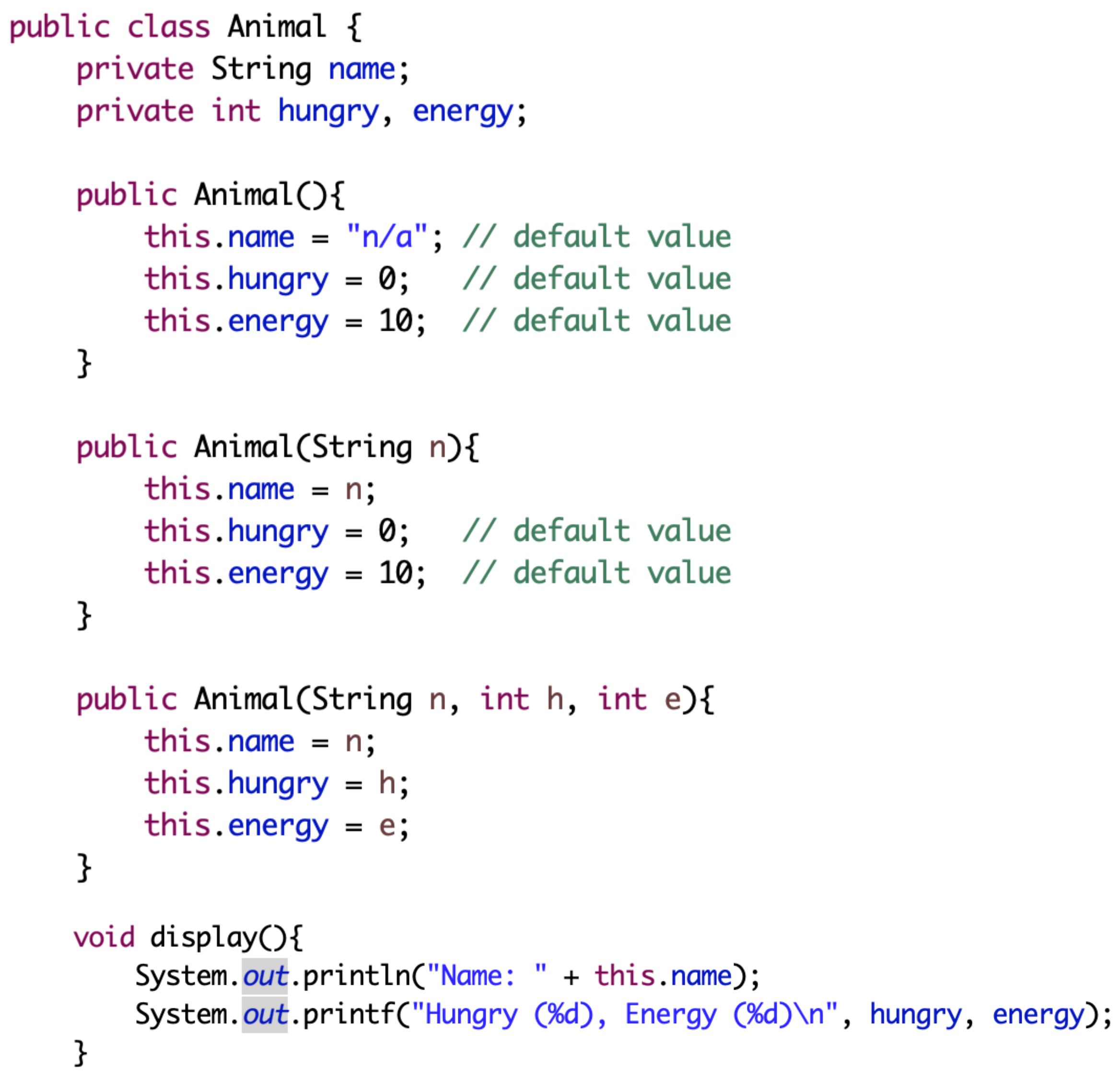
```
public abstract class Shape {

    public  abstract double getArea();
    // no implementation

}
```

Don’t forget semicolon (;)

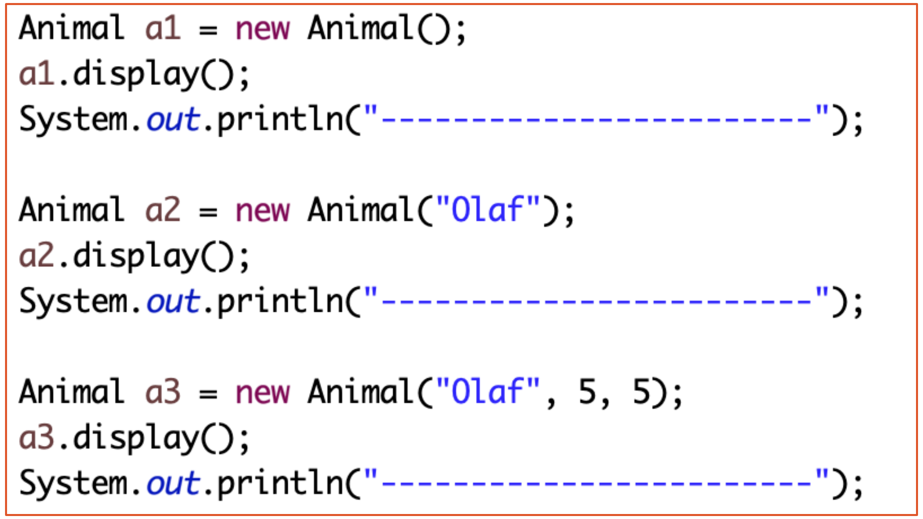
Part 4:
Overload Constructor

Sometimes, you might want to constructor your object in **different way** too! You can “**overload**” constructors as well as methods:

A common reason for overloading constructors is (as above) to provide default values for missing parameters as shown below

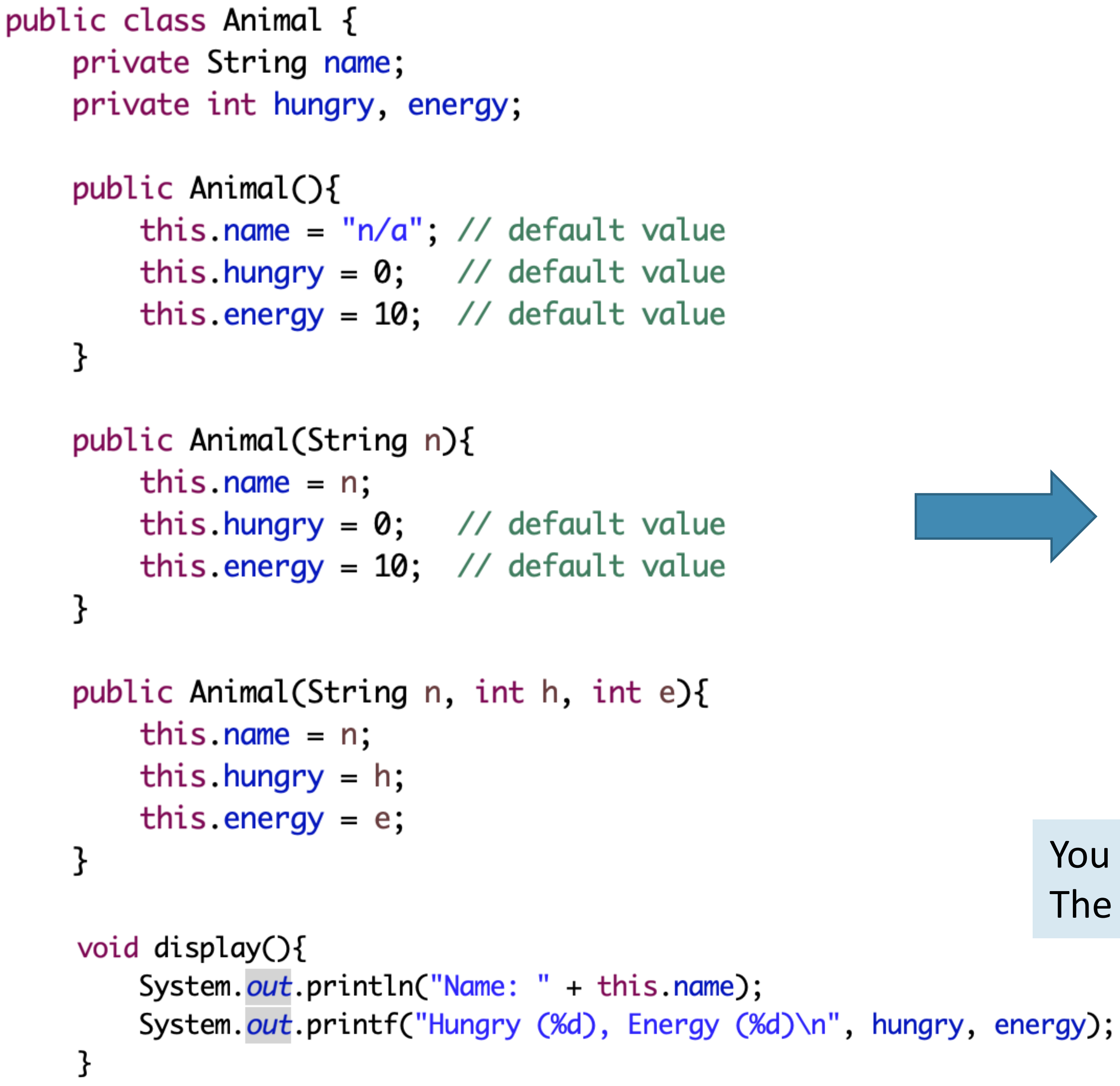


```java
public class Animal {
    private String name;
    private int hungry, energy;

    public Animal(){
        this.name = "n/a"; // default value
        this.hungry = 0;    // default value
        this.energy = 10;  // default value
    }

    public Animal(String n){
        this.name = n;
        this.hungry = 0;    // default value
        this.energy = 10;  // default value
    }

    public Animal(String n, int h, int e){
        this.name = n;
        this.hungry = h;
        this.energy = e;
    }

    void display(){
        System.out.println("Name: " + this.name);
        System.out.printf("Hungry (%d), Energy (%d)\n", hungry, energy);
    }
```



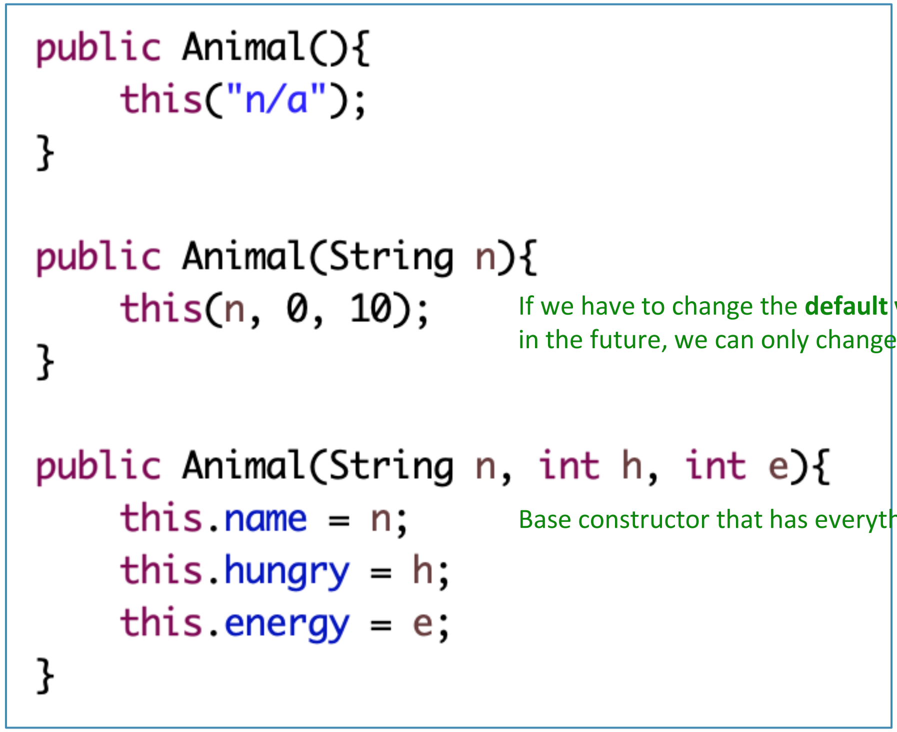
```java
Animal a1 = new Animal();
a1.display();
System.out.println("------------------------");

Animal a2 = new Animal("Olaf");
a2.display();
System.out.println("------------------------");

Animal a3 = new Animal("Olaf", 5, 5);
a3.display();
System.out.println("------------------------");
```

```
Name: n/a
Hungry (0), Energy (10)
------------------------
Name: Olaf
Hungry (0), Energy (10)
------------------------
Name: Olaf
Hungry (5), Energy (5)
------------------------
```



```
public class Animal {
    private String name;
    private int hungry, energy;

    public Animal(){
        this.name = "n/a"; // default value
        this.hungry = 0;   // default value
        this.energy = 10;  // default value
    }

    public Animal(String n){
        this.name = n;
        this.hungry = 0;   // default value
        this.energy = 10;  // default value
    }

    public Animal(String n, int h, int e){
        this.name = n;
        this.hungry = h;
        this.energy = e;
    }

    void display(){
        System.out.println("Name: " + this.name);
        System.out.printf("Hungry (%d), Energy (%d)\n", hungry, energy);
    }
```

Call other constructor method to reduce the code



```
public Animal(){
    this("n/a");
}

public Animal(String n){
    this(n, 0, 10);
}

public Animal(String n, int h, int e){
    this.name = n;
    this.hungry = h;
    this.energy = e;
}
```

If we have to change the **default value** in the future, we can only change at one place.

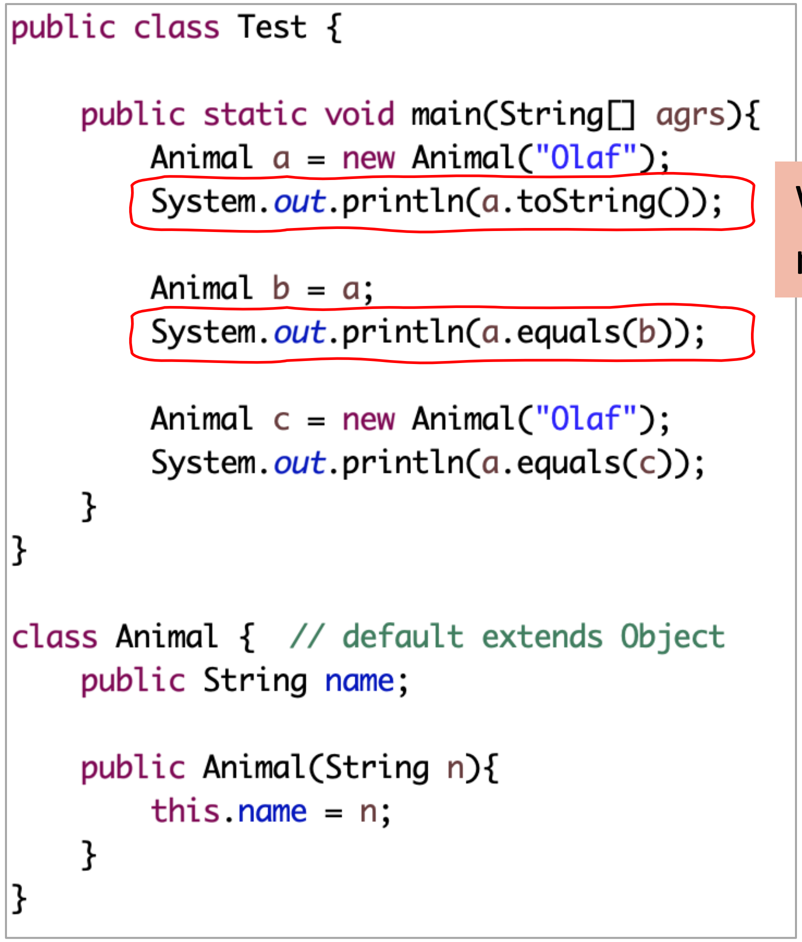
Base constructor that has everything

You call the other constructor with the keyword `this`
The call must be the very first thing the constructor does

Part 5:
Object Superclass

toString()

equals()

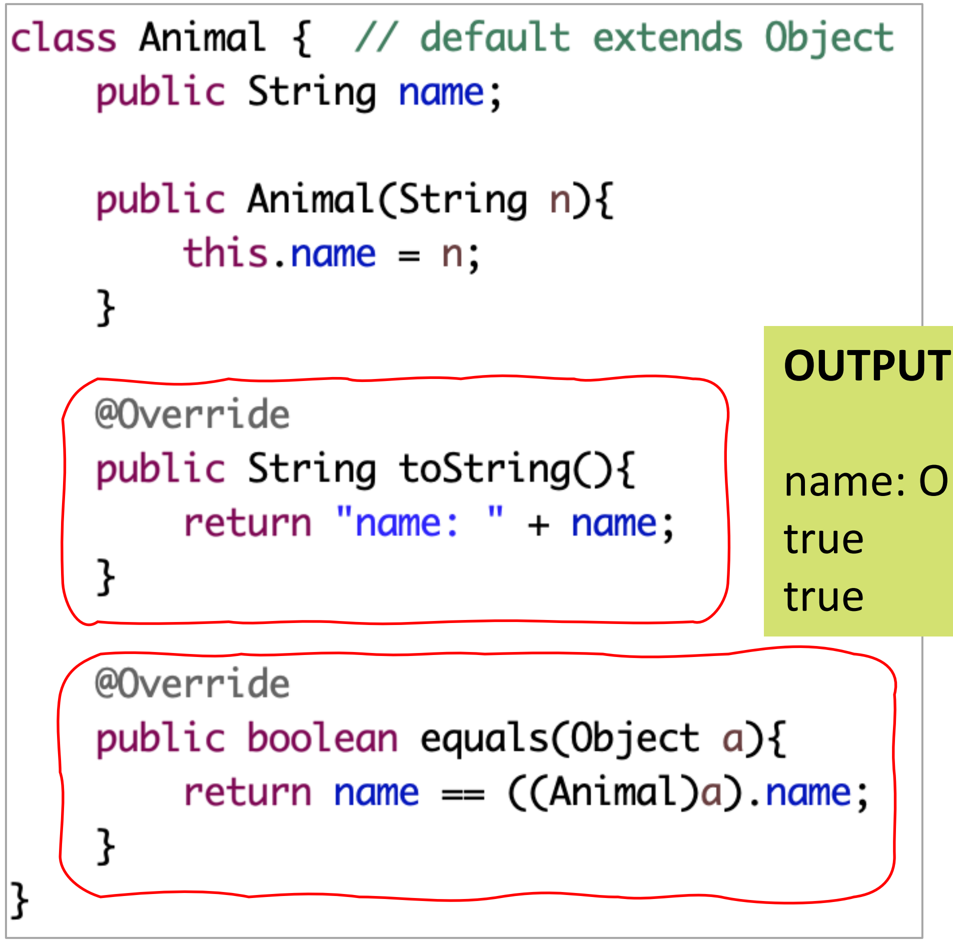


```java
public class Test {

    public static void main(String[] agrs){
        Animal a = new Animal("Olaf");
        System.out.println(a.toString());

        Animal b = a;
        System.out.println(a.equals(b));

        Animal c = new Animal("Olaf");
        System.out.println(a.equals(c));
    }
}

class Animal {  // default extends Object
    public String name;

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

OUTPUT – without override

```
Animal@7852e922
true
false
```

Where is **toString() and equals()** methods in class Animal?



```java
class Animal {  // default extends Object
    public String name;

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

    @Override
    public String toString(){
        return "name: " + name;
    }

    @Override
    public boolean equals(Object a){
        return name == ((Animal)a).name;
    }
}
```

OUTPUT – with override

```
name: Olaf
true
true
```

More on toString()

- It is almost always a good idea to override **toString()** to return something “meaningful” about the object
  - When debugging, it helps to be able to print objects

- When you print objects with `System.out.print` or `System.out.println`, they automatically call the objects `toString()` method

```
Animal var = new Animal("Olaf");
System.out.println(var);

// same as
// System.out.println(var.toString());
```

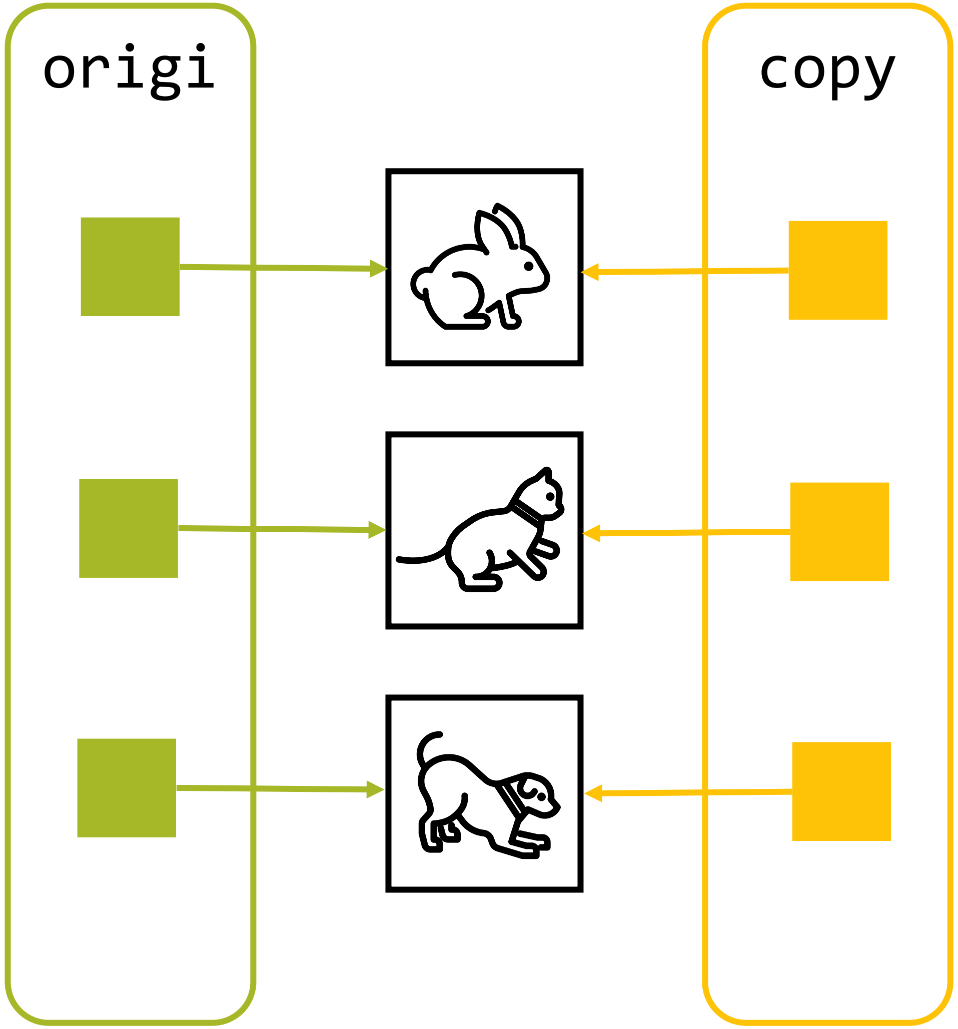
More on equals()

- **Primitives** can always be tested for equality with **==**
- For **objects**, == tests whether the two are the same object (same address or not)
  - BUT two **strings** "abc" and "abc" may or may not be ==
- Objects can be tested with the method **`equals(Object o)`**
  - Unless overridden, this method just uses **==**
  - It is overridden in the class String
  - It is not overridden for arrays; == tests if its operands are the same array (same address)
- *Rule of thumb:*
  - Never use == to test equality of Strings or arrays or other objects ; USE **`equals()`** instead
    - Strings, → `"hello".equals("Hello"); // False or  "hello".equalsIgnoreCase("Hello"); // True`
    - Two arrays are equal if they contain the same elements in the same order. → java. util.Arrays.equals(a1, a2)
  - If you test your own objects for equality (by their values/data not address), **override equals**

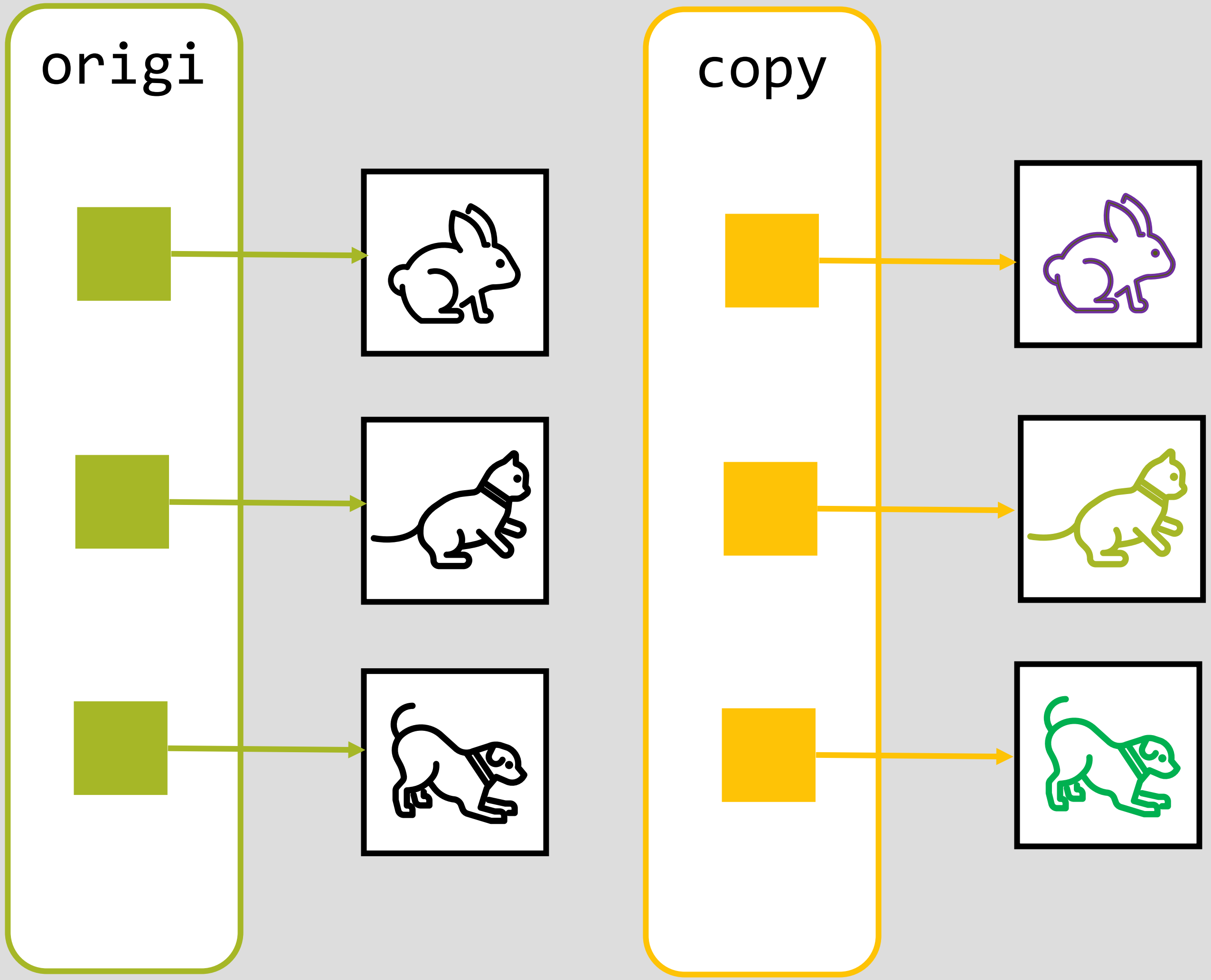
Additional Topic
(if time permits)

Shallow Clone vs Deep Clone

Shallow Clone

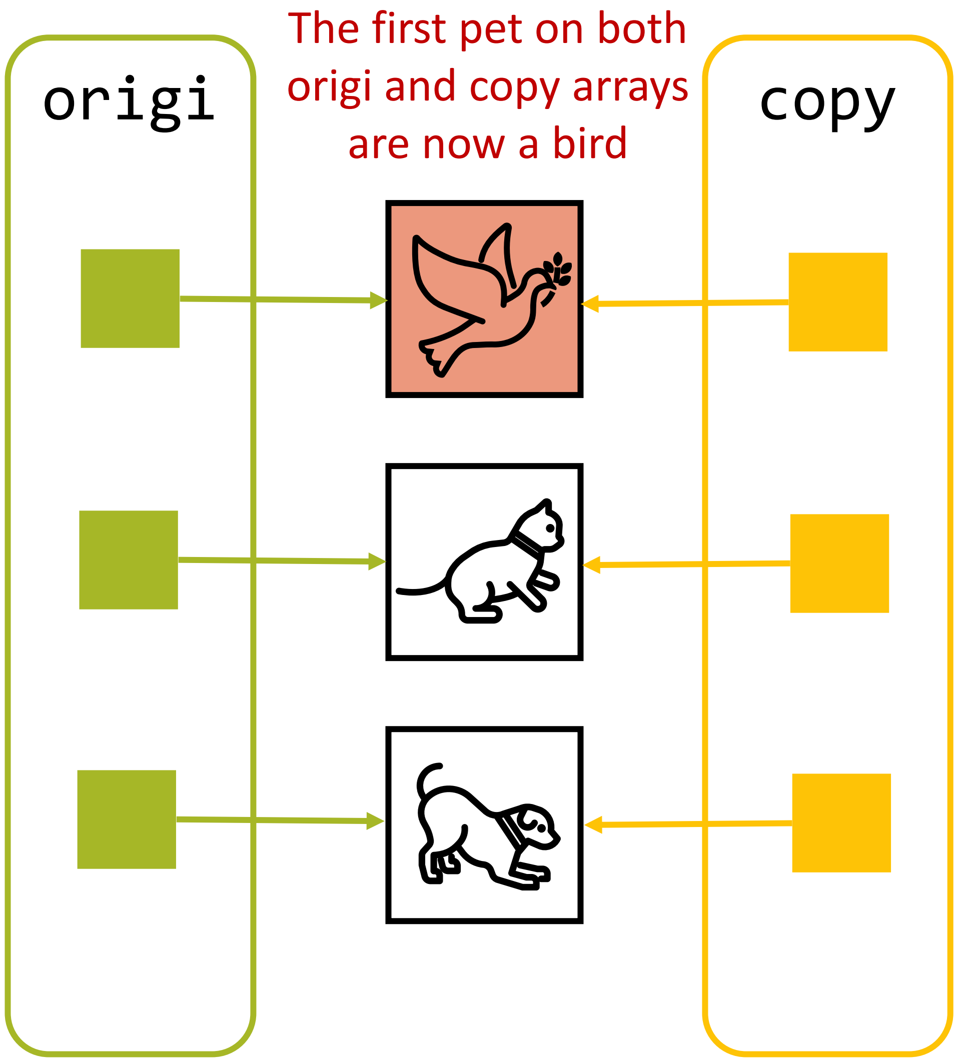


Deep Clone



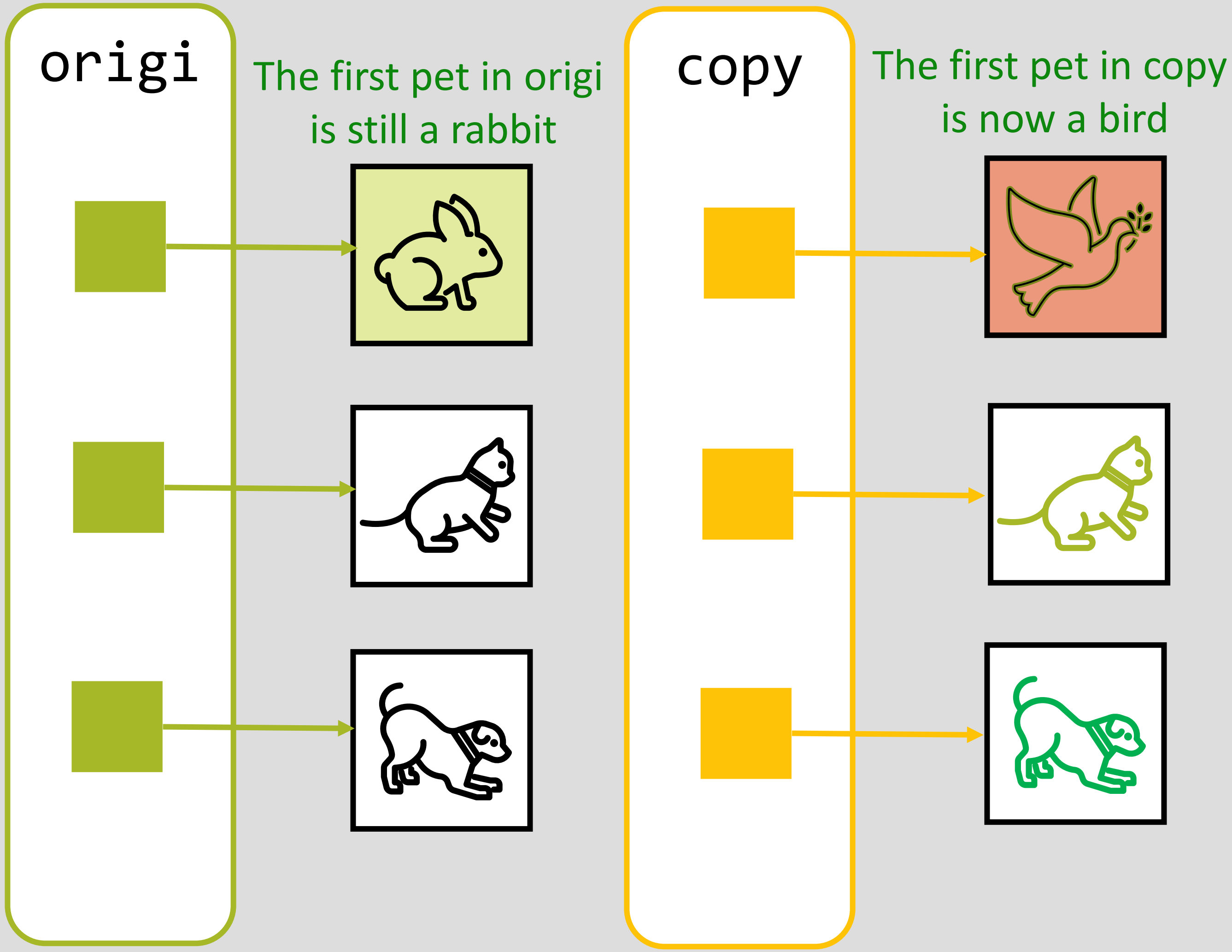
After we update the first pet in "copy" array to be a bird, what happened?

Shallow Clone



The first pet on both origi and copy arrays are now a bird

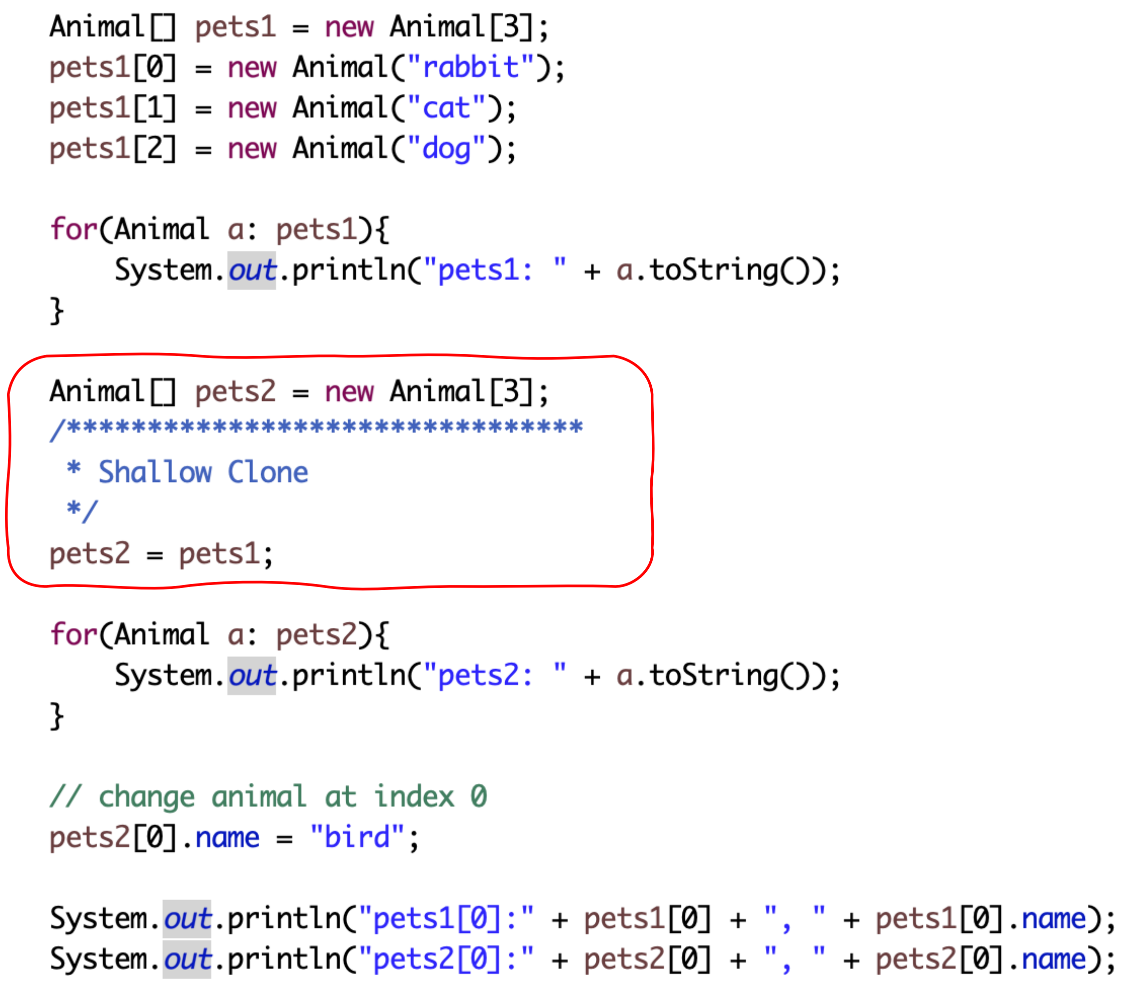
Deep Clone



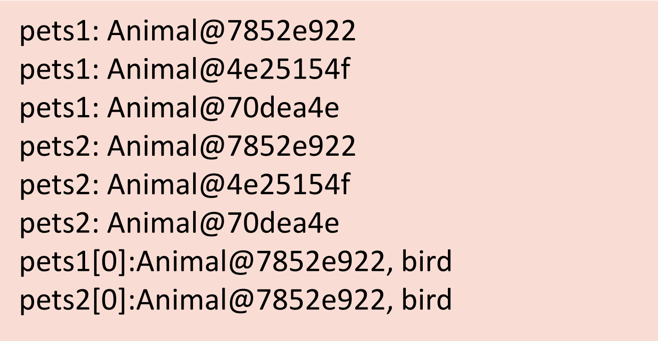
The first pet in origi is still a rabbit

The first pet in copy is now a bird

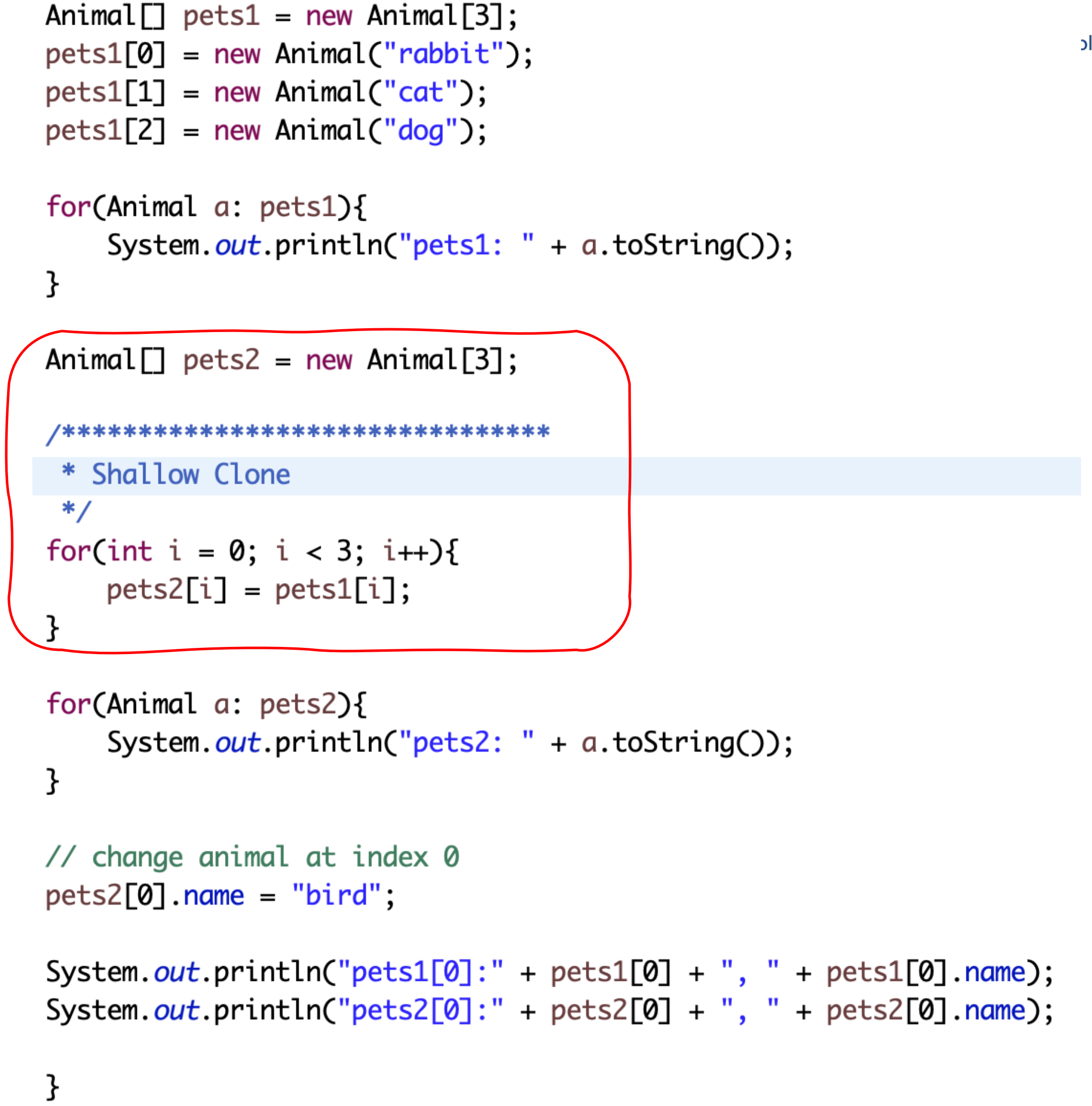
Shallow Clone



```java
Animal[] pets1 = new Animal[3];
pets1[0] = new Animal("rabbit");
pets1[1] = new Animal("cat");
pets1[2] = new Animal("dog");

for(Animal a: pets1){
    System.out.println("pets1: " + a.toString());
}

Animal[] pets2 = new Animal[3];
/********************************
 * Shallow Clone
 */
pets2 = pets1;

for(Animal a: pets2){
    System.out.println("pets2: " + a.toString());
}

// change animal at index 0
pets2[0].name = "bird";

System.out.println("pets1[0]:" + pets1[0] + ", " + pets1[0].name);
System.out.println("pets2[0]:" + pets2[0] + ", " + pets2[0].name);
```

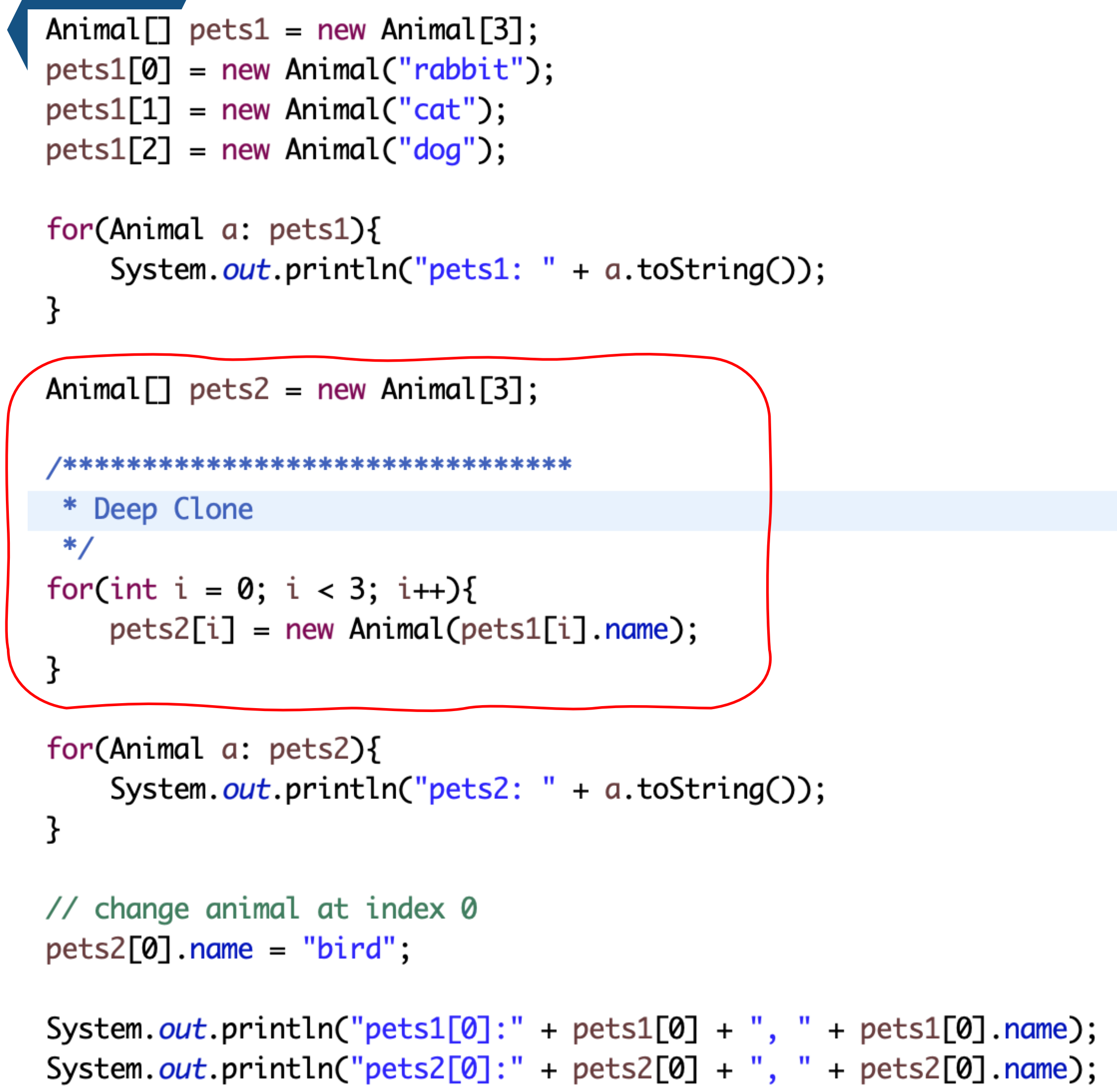


```
pets1: Animal@7852e922
pets1: Animal@4e25154f
pets1: Animal@70dea4e
pets2: Animal@7852e922
pets2: Animal@4e25154f
pets2: Animal@70dea4e
pets1[0]:Animal@7852e922, bird
pets2[0]:Animal@7852e922, bird
```



```java
Animal[] pets1 = new Animal[3];
pets1[0] = new Animal("rabbit");
pets1[1] = new Animal("cat");
pets1[2] = new Animal("dog");

for(Animal a: pets1){
    System.out.println("pets1: " + a.toString());
}

Animal[] pets2 = new Animal[3];

/********************************
 * Shallow Clone
 */
for(int i = 0; i < 3; i++){
    pets2[i] = pets1[i];
}

for(Animal a: pets2){
    System.out.println("pets2: " + a.toString());
}

// change animal at index 0
pets2[0].name = "bird";

System.out.println("pets1[0]:" + pets1[0] + ", " + pets1[0].name);
System.out.println("pets2[0]:" + pets2[0] + ", " + pets2[0].name);

}
```

Deep Clone



```
Animal[] pets1 = new Animal[3];
pets1[0] = new Animal("rabbit");
pets1[1] = new Animal("cat");
pets1[2] = new Animal("dog");

for(Animal a: pets1){
    System.out.println("pets1: " + a.toString());
}

Animal[] pets2 = new Animal[3];

/*******************************
 * Deep Clone
 */
for(int i = 0; i < 3; i++){
    pets2[i] = new Animal(pets1[i].name);
}

for(Animal a: pets2){
    System.out.println("pets2: " + a.toString());
}

// change animal at index 0
pets2[0].name = "bird";

System.out.println("pets1[0]:" + pets1[0] + ", " + pets1[0].name);
System.out.println("pets2[0]:" + pets2[0] + ", " + pets2[0].name);
```

```
pets1: Animal@7852e922
pets1: Animal@4e25154f
pets1: Animal@70dea4e
pets2: Animal@5c647e05
pets2: Animal@33909752
pets2: Animal@55f96302
pets1[0]:Animal@7852e922, rabbit
pets2[0]:Animal@5c647e05, bird
```