Chapter 4.

State and Behavior of an Object

2019-2020

COMP2396 Object-Oriented Programming and Java

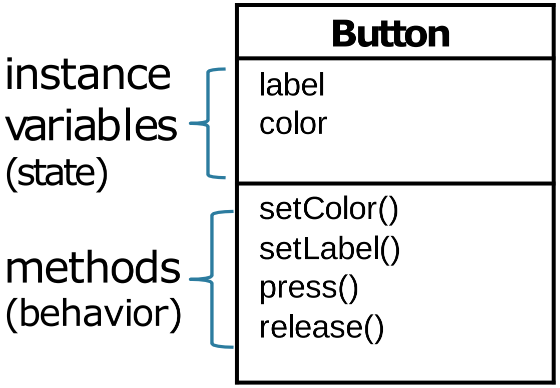
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)

Department of Computer Science, The University of Hong Kong

Instances of the Same Class

- Recall that a class is a blueprint for an object, it describes what an object knows and does

- Instances of the same class (i.e., objects made from the same class) therefore have
  - Same set of instance variables
  - Same set of methods



instance variables (state)

methods (behavior)

Button

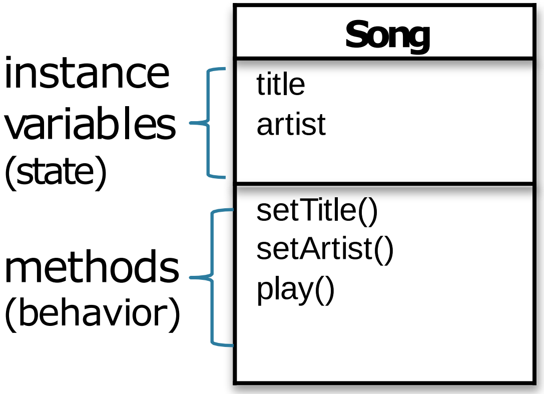
label
color

setColor()
setLabel()
press()
release()

- Each instance is, however, said to be unique in the sense that it can have different values stored in its instance variables (e.g., buttons with different labels and colors)

Behavior Depends on State

- Can each instance have different method behavior?
  - Every instance of the same class has the same set of methods
  - Methods can, however, behave differently depending on the values of the instance variables



instance variables (state)

methods (behavior)

Song

title
artist

setTitle()
setArtist()
play()

```
void play() {
   SoundPlayer player = new SoundPlayer();
   player.play(title);
}
```

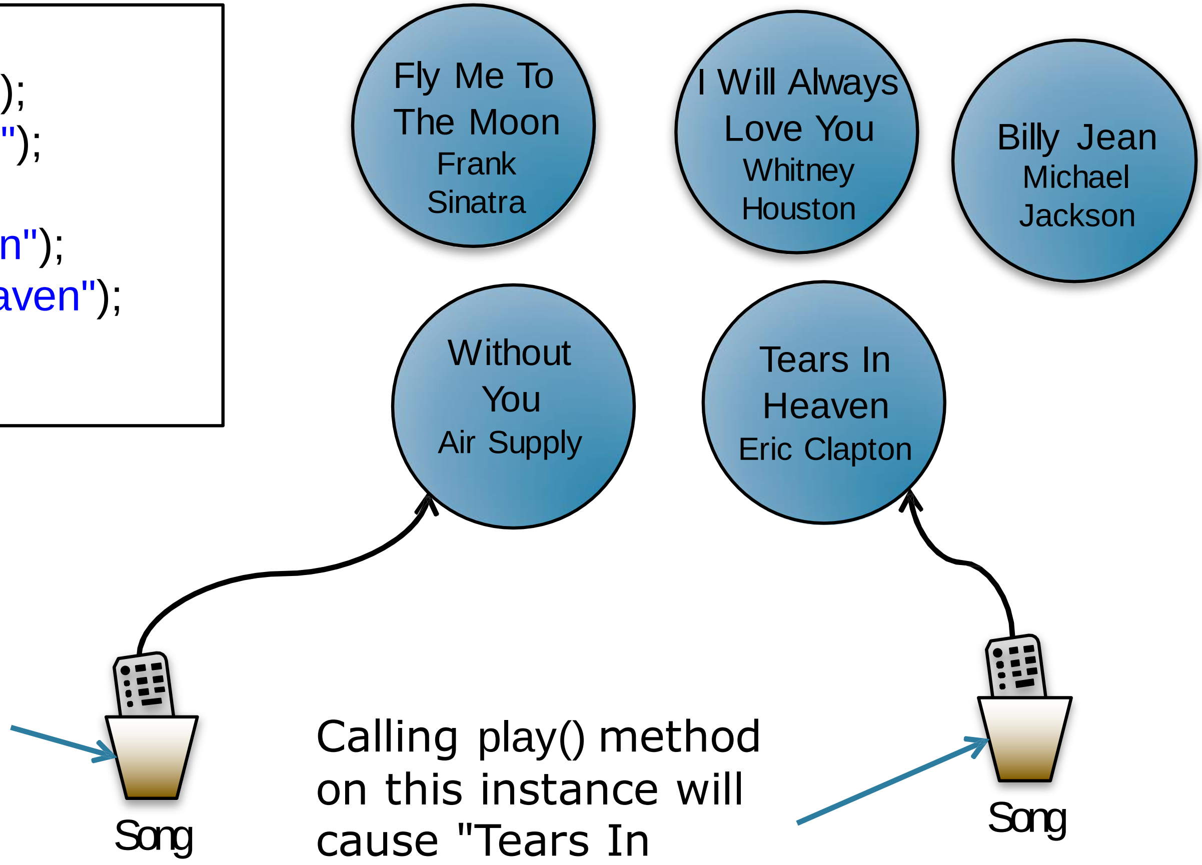
The play() method will play a song represented by the value of the instance variable title

Behavior Depends on State

- Example

```
Song s4 = new Song();
s4.setArtist("Air Supply");
s4.setTitle("Without You");
Song s5 = new Song();
s5.setArtist("Eric Clapton");
s5.setTitle("Tears In Heaven");
s4.play();
s5.play();
```

5 instances of the Song class



Calling play() method on this instance will cause "Without You" to play

Calling play() method on this instance will cause "Tears In Heaven" to play

Example: Dogs Bark Differently

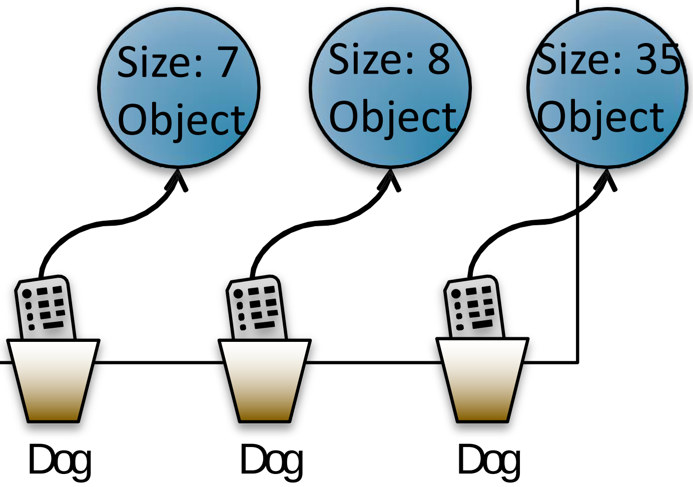
- Example

```
class Dog {
   int size;

   void makeNoise()
      {  if (size > 60) {
         System.out.println("Woof! Woof!");
      } else if (size > 14) {
         System.out.println("Ruff! Ruff!");
      } else {
         System.out.println("Yip! Yip!");
      }
   }
}
```

Example: Dogs Bark Differently

Example

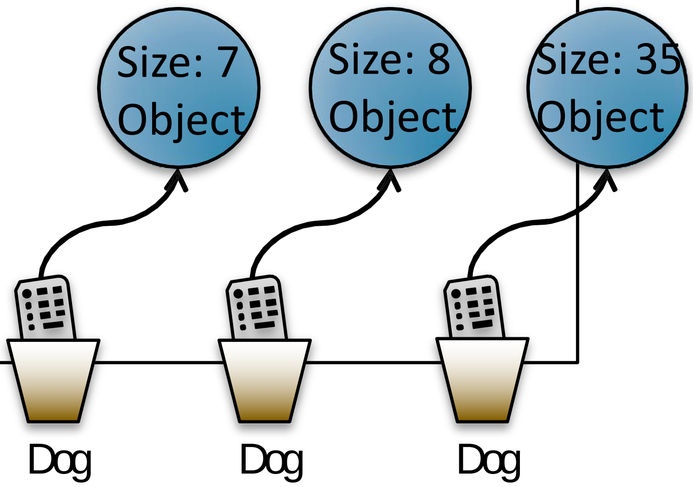
```
class DogTestDrive {
  public static void main(String[] args) {
    Dogone = newDog();
    one.size = 70;
    Dogtwo = newDog();
    two.size = 8;
    Dogthree = newDog();
    three.size = 35;

    one.makeNoise();
    two.makeNoise();
    three.makeNoise();
  }
}
```



Example: Dogs Bark Differently

- Sample output

```
Woof! Woof!
Yip!  Yip!
Ruff! Ruff!
```

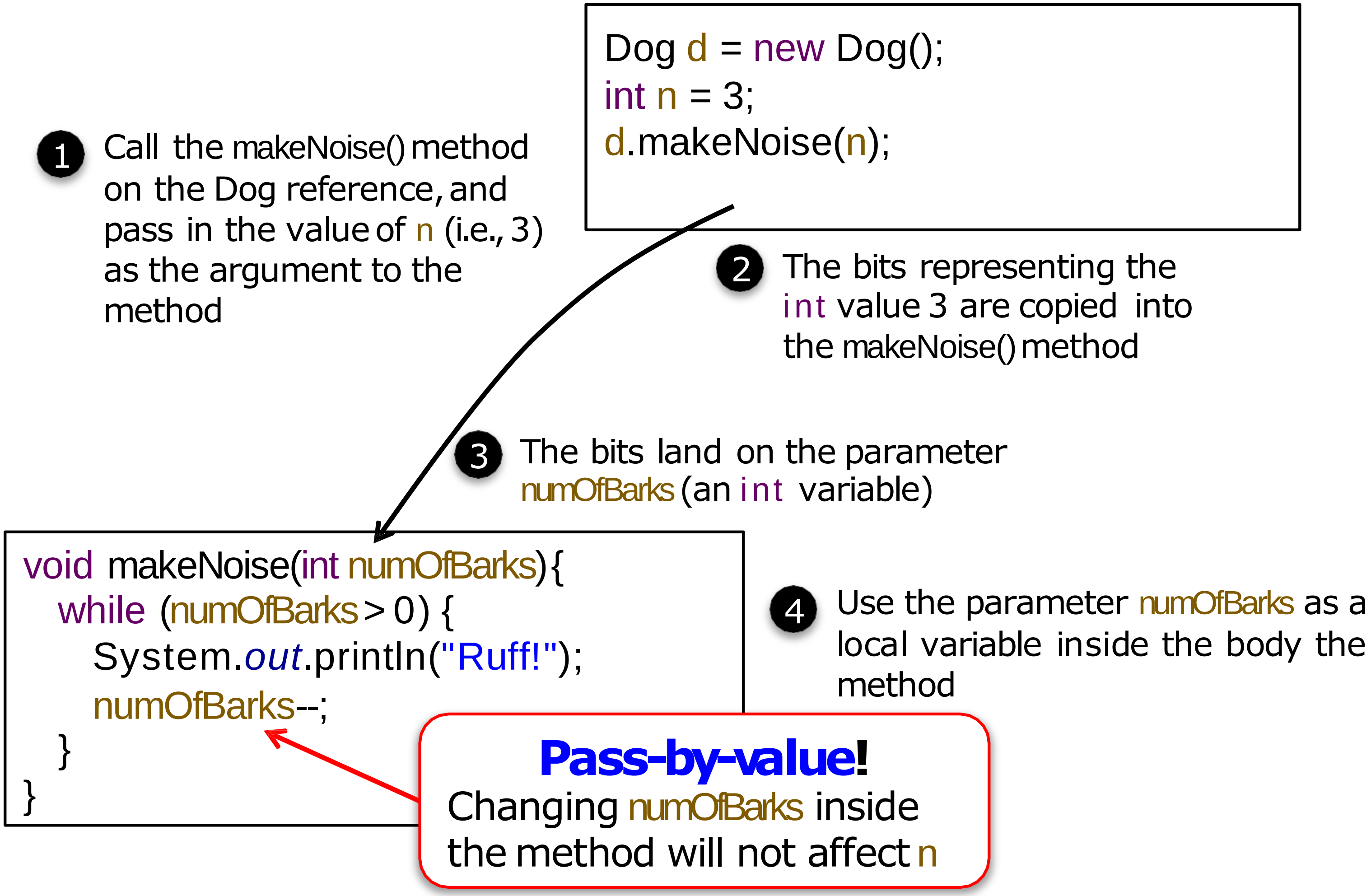


Methods: Parameters

- Methods can have (multiple) parameters
- Parameters are simply placeholders in the methods
- They are nothing more than local variables that can be used inside the body of the method
- Arguments are the actual values passed into the methods (i.e., actual values used in calling a method)
- The number and type of the arguments that are passed into a method must match the type and order of the parameters declared by the method
- Java uses pass-by-value mechanism for parameter–passing (i.e., argument values are copied to the parameters)

Methods: Parameters

Example



```
Dog d = new Dog();
int n = 3;
d.makeNoise(n);
```

1 Call the makeNoise() method on the Dog reference, and pass in the value of n (i.e., 3) as the argument to the method

2 The bits representing the int value 3 are copied into the makeNoise() method

3 The bits land on the parameter numOfBarks (an int variable)

```
void makeNoise(int numOfBarks){
   while (numOfBarks > 0) {
      System.out.println("Ruff!");
      numOfBarks--;
   }
}
```

4 Use the parameter numOfBarks as a local variable inside the body the method

Pass-by-value!

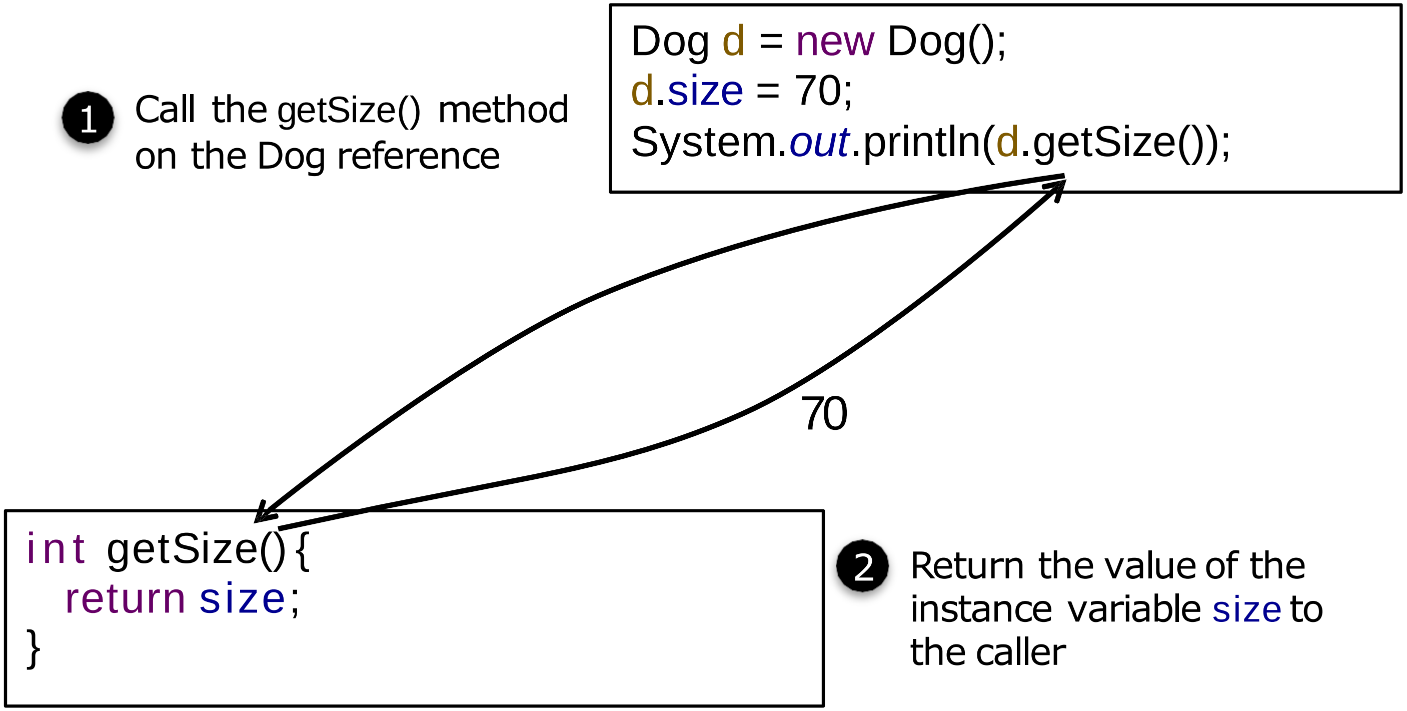
Changing numOfBarks inside the method will not affect n

Methods: Return Value

- Methods can return values
- Every method must declare a return type
- A void return type means the method does not return anything
- If a method declares a non-void return type, it must return a value compatible with the declared return type

Methods: Return Value

- Example



```
Dog d = new Dog();
d.size = 70;
System.out.println(d.getSize());
```

❶ Call the getSize() method on the Dog reference

```
int getSize(){
  return size;
}
```

❷ Return the value of the instance variable size to the caller

Methods: Getters and Setters

- A getter is a method for getting the value of an instance variable
- A setter is a method for setting the value of an instance variable

```
class ElectricGuitar {
   String brand;
   int numOfPickups;
   boolean rockStarUsesIt;

   String getBrand() { return brand; }
   int getNumOfPickups() { return numOfPickups; }
   boolean getRockStarUsesIt() { return rockStarUsesIt; }

   void setBrand(String brand) { brand = brand; }  void
   setNumOfPickups(int num) { numOfPickups = num; }
   void setRockStarUsesIt(boolean yesOrNo) { rockStarUsesIt = yesOrNo; }
}
```

getters

setters

This won't assign the parameter to the instance variable!

Methods: Getters and Setters

- A getter is a method for getting the value of an instance variable
- A setter is a method for setting the value of an instance variable

```
class ElectricGuitar {
   String brand;
   int numOfPickups;
   boolean rockStarUsesIt;

   String getBrand() { return brand; }
   int getNumOfPickups() { return numOfPickups; }
   boolean getRockStarUsesIt() { return rockStarUsesIt; }

   void setBrand(String brand) { this.brand = brand; }  void
   setNumOfPickups(int num) { numOfPickups = num; }
   void setRockStarUsesIt(boolean yesOrNo) { rockStarUsesIt = yesOrNo; }
}
```

getters

setters

this is a reference to a current object whose method is being called

Encapsulation

- Without encapsulation, instance variables of an object are said to be exposed

- They might be changed to unacceptable values from outside the object, e.g.,

```
Dog myDog = new Dog();
myDog.size = -30; // a dog with a negative size?
```

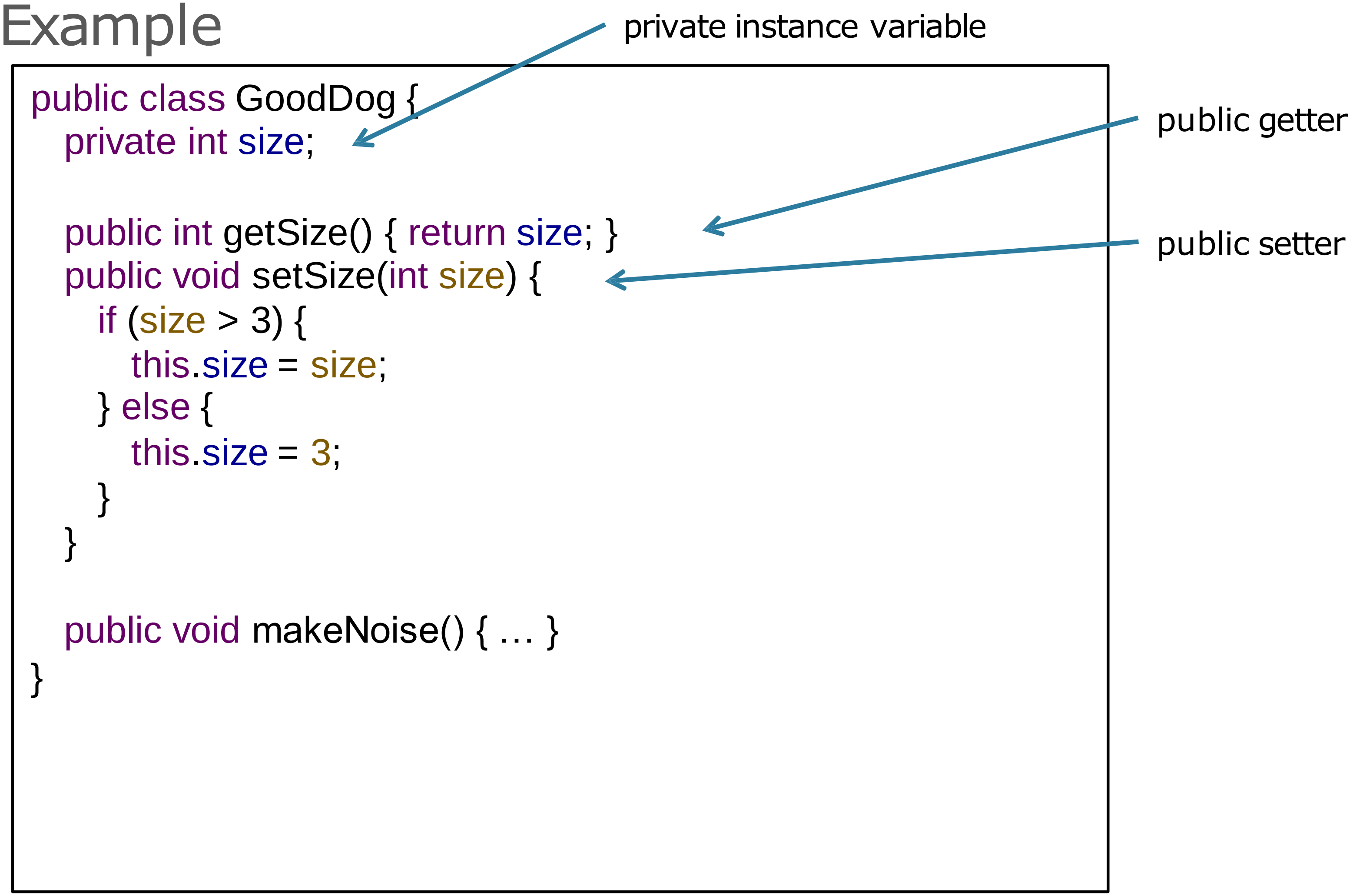
- The access modifier private can be used to protect instance variables of an object from external access

- Private instance variables (as well as private methods) of an object can only be accessed from within the object

- Public getters and setters (declared with the access modifier public) are provided for accessing the private instance variables from outside the object

Example: Dogs with Encapsulation

- Example



```
public class GoodDog {
  private int size;

  public int getSize() { return size; }
  public void setSize(int size) {
    if (size > 3) {
      this.size = size;
    } else {
      this.size = 3;
    }
  }

  public void makeNoise() { … }
}
```

private instance variable

public getter

public setter

Example: Dogs with Encapsulation

Example

```
public class GoodDogTestDrive {
   public static void main(String[] args) {  GoodDog
      one = new GoodDog();
      one.setSize(70);
      GoodDog two = new GoodDog();
      two.setSize(-30);
      System.out.println("Dog one: " + one.getSize());
      System.out.println("Dog  two: " + two.getSize());
   }
}
```

Sample output

```
Dog one: 70
Dog two: 3
```

Not –30!

Constructors

- The constructor of a class
  - A special method that is called automatically when an object is created, and before the object can be assigned to a reference
  - Must have the same name as the class
  - Must not have any return type, not even void
  - Cannot be called explicitly using the dot operator
  - Often used to initialize instance variables

- If no constructor is defined for a class, the compiler assumes a default constructor which is simply an empty method with no arguments

```
public Dog() {  }  // default constructor
```

Constructors

Example

```
    public class Duck {
       private int size = 30;
       private String name = "Donald";

1      public Duck(String name, int size) {
          this.name = name;
          this.size = size;
       }
2      public Duck(int size) { this.size = size; }
3      public Duck(String name) { this.name = name; }
4      public Duck() { }

       // ...
    }
```

A class may have more than 1 constructor as long as each constructor has a different argument list. This is an example of overloaded methods

The compiler will not assume the default constructor if 1 or more constructors have been defined. In this case, a constructor that takes no argument, if needed, has to be defined explicitly

Constructors

- Example

```
Duck duck1 = new Duck("Happy", 25);
Duck duck2 = new Duck(32);
Duck duck3 = new Duck("Sunday");
Duck duck4 = new Duck();
```

The actual constructor being called is determined by the type and order of the arguments supplied in creating the object

Constructors: Inheritance

- Constructors are not inherited
- When an object of a subclass is created, its constructor will immediately call its superclass's no-argument constructor
- The superclass's constructor will in turn call its superclass's no-argument constructor
- This goes all the way up the hierarchy until the Object class's no-argument constructor is reached (Object class is the ultimate superclass of all classes)
- This process is known as constructor chaining
- Always provide a no-argument constructor if possible!

Constructors: Inheritance

- Example

```
public class Animal {
   public Animal(int n) { /* constructor code... */ }
}
```

```
public class Pig extends Animal {
   private int size = 90;
   private String name = "Peppa";
}
```

```
Pig pig = new Pig();
```

This won’t compile!

When a Pig object is created, its default constructor will call the no-argument constructor of its superclass (i.e., Animal class). However, there is no no-argument constructor defined in the Animal class!

Constructors: Inheritance

- Example

```
public class Animal {
   public Animal(int n) { /* constructor code... */ }
   public Animal() { /* constructor code... */ }
}
```

```
public class Pig extends Animal {
   private int size = 90;
   private String name = "Peppa";
}
```

```
Pig pig = new Pig();
```

The problem can be solved by providing a no-argument constructor for the Animal class

Constructors: Inheritance

Example

```
public class Animal {
   public Animal(int n) { /* constructor code... */ }
}
```

```
public class Pig extends Animal {
   private int size = 90;
   private String name = "Peppa";
   public Pig() {
      super(3);
        // more constructor code...
   }
}
```

Another way to solve this is by making an explicit call to the constructor of its superclass using super() with appropriate arguments in the first statement of the constructor

```
Pig pig = new Pig();
```

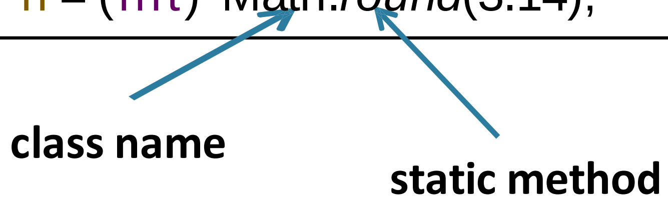
Static Methods

- Static methods are those whose behavior does not depend on the value of any instance variable (e.g., the round() method in the Math class)
- Static methods can run without any instance of the class
- The keyword `static` is used to define static methods

```
public static long round(double a) { … }
```

- A static method is called using the class name

```
int n = (int) Math.round(3.14);
```



class name

static method

Static Methods

- Static methods cannot use instance variables
  - Static methods are called using the class name
  - Static methods run without knowing about any particular instance of the class
  - There may not even be any instance of that class
- Example

```
public class Duck {
    private int size;

    public static void main (String[] args) {
        System.out.println("Size of duck is " + size);
    }

    public void setSize(int size) { this.size = size; }
    public int getSize() { return size; }
}
```

Which Duck?
Whose size?

This won’t compile!
Even if there are 10 Duck objects on the heap, the static method does not know about any of them!

Static Methods

- Static methods cannot use non-static methods
  - Non-static methods behave differently depending on the values of the instance variables
- Example

```
public class Duck {
    private int size;

    public static void main (String[] args) {
        System.out.println("Size of duck is " + getSize());
    }

    public void setSize(int size) { this.size = size; }
    public int getSize() { return size; }
}
```

Which Duck?
Whose getSize()?

This won't compile!
Even if there are 10 Duck objects on the heap, the static method does not know about any of them!

Static Variables

- A static variable is shared by all instances of a class (i.e., one copy per class)
- A static variable is initialized only when the class is first loaded, but not each time an instance is created
  - Initialized before any object of that class can be created
  - Initialized before any static method of the class runs
- The keyword `static` is used to define static variables
- A static variable can be accessed using the class name
- A static variable can be used inside static methods

Static Variables

- Example

static variable
(one copy per class)

instance variable
(one copy per instance)

```
public class Player {
    public static int playerCount = 0;
    private String name = "Default";
    public Player(String name) {
        this.name = name;
        playerCount++;
    }
    public static void main(String[] args) {
        System.out.println("#players: " + Player.playerCount);
        Player player = new Player("Peter Parker");
        System.out.println("#players: " + Player.playerCount);
    }
}
```

- Sample output

```
#players: 0
#players: 1
```

Instance Variable Initialization

- Instance variable (and also static variable)
  - Declared within a class but not within a method
  - Always gets a default value if no value has been explicitly assigned to it
    - Number primitives (including char) get 0 (or 0.0)
    - Booleans get false
    - Object references get null

Instance Variable Initialization

- Example

```
public class PoorDog {
   private int size;
   private String name;

   public int getSize() { return size;}
   public String getName() { return name; }
}
```

```
public class PoorDogTestDrive {
   public static void main(String[] args) {
      PoorDog dog = new PoorDog();
      System.out.println(dog.getName() + " has a size of " + dog.getSize());
   }
}
```

- Sample output

```
null has a size of 0
```

Local Variable Initialization

- Local variable (including method parameter)
  - Declared within a method
  - Do not get a default value
  - Must be initialized before use!
- Example

```
public class Foo {
   private int a;
   private int b;
   public void go() {
      int x;
      int y = a + b;
      int z = x + 3;
   }
}
```

This won't compile!

The local variable x has not been initialized!

Comparing Variables

- To compare primitives, use the == operator
  - Can be used to compare 2 primitive variables of any kind (save for booleans which can only be compared against booleans)
  - Simply compares the bits, and returns `true` if the bit patterns are the same
  - Doesn't care about the size of the variables, all the extra zeros on the left end don't matter
  - Example

```
int a = 3; // 32 bits
byte b = 3; // 8 bits
if (a == b) { /* true */ }
```

Comparing Variables

- To compare objects, use the == operator
  - Can be used to compare 2 reference variables declared for objects of the same class
  - Simply compares the bits, and returns `true` if the bit patterns are the same
  - Check whether 2 references are the same (i.e., referencing the same object on the heap)
  - Example

```
Dog dog1 = new Dog();
Dog dog2 = new Dog();
Dog dog3 = dog1;
if (dog1 == dog2) { /* false */ }
if (dog1 == dog3) { /* true */ }
if (dog2 == dog3) { /* false */ }
```

Keyword final

- The keyword final can be used to modify variables, including static variables, instance variables, local variables and even method parameters

- A final variable means, once initialized, its value cannot be changed

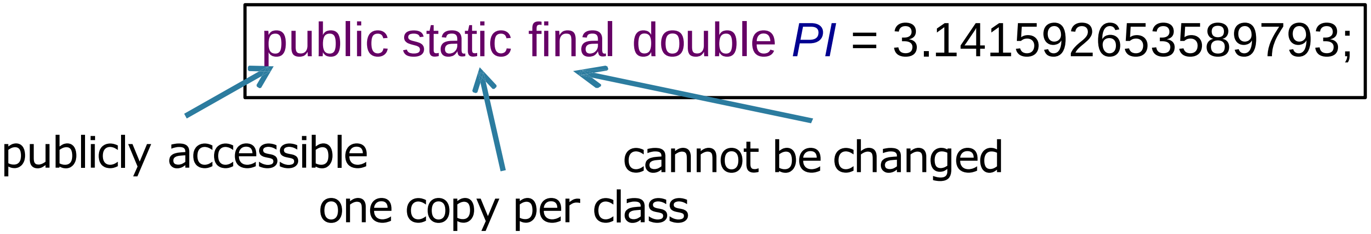
- A final instance variable must be initialized either at the time it is declared or in the constructor (a final instance variable does NOT get a default value!)

```
public class Bottle {
   public final int volume = 330;
}
```

```
public class Bottle2 {
   public final int volume;

   public Bottle2(int volume) {
      this.volume = volume;
   }
}
```

Keyword final

- To define a constant, mark a variable both as `static` and `final` (e.g., PI defined in the Math class)



```
public static final double PI = 3.141592653589793;
```

- A static final variable must be initialized either at the time it is declared or in a static initializer

```
public class Foo {
   public static final int FOO_X = 25;
}
```

```
public class Foo2 {
   public static final int FOO_X;
   static {
      FOO_X = 25;
   }
}
```

The static initializer runs as soon as the class is loaded, before any static method can be called and before any static variables can be used

Keyword final

- The keyword final can also be used to modify methods and classes
  - A final method means it cannot be overridden in a subclass

```
public final double circumference() { … }
```

  - A final class means it cannot be extended

```
public final class UltimateCircle {  …}
```

If a class is marked **final**, it is not necessary to mark any of its methods **final** as it is not possible to have any subclass

Chapter 4.

End

2019-2020
COMP2396 Object-Oriented Programming and Java
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong