

# Section 5

## Object Behaviour

### Learning Outcomes

- After this lecture you should be able to:
  - Describe how methods use parameters
  - Describe how variables are used as parameters
  - Outline how methods return values
  - Describe the purpose of getter & setter methods
  - Describe Encapsulation (revisited)
  - Create arrays of Objects (revisited)
  - Discuss how Instance variables can be initialised
  - Outline the difference between instance & local variables
  - Describe the comparing primitive & reference variables
  - **Reading and studying** recommended text is essential to improve your understanding of the above

# Behaviour

- States affects behaviour & behaviour affects states
- Objects have states & behaviours represented by **instance variables** & **methods**
- What is the relationship between states & behaviours?
  - Each instance of a class (object of a type) can have its own unique values for its instance variables
  - Each object has behaviours that act on it's state
  - OR **methods use instance variable values**

# Dog Class

```
class Dog {  
    int size; // instance variable represents state  
    // method represents behaviour  
    void bark() {  
        if (size > 60) {  
            System.out.println("Woof! Woof!");  
        }  
        else if (size > 14) {  
            System.out.println("Ruff! Ruff!");  
        }  
        else {  
            System.out.println("Yip! Yip!");  
        }  
    }  
}
```

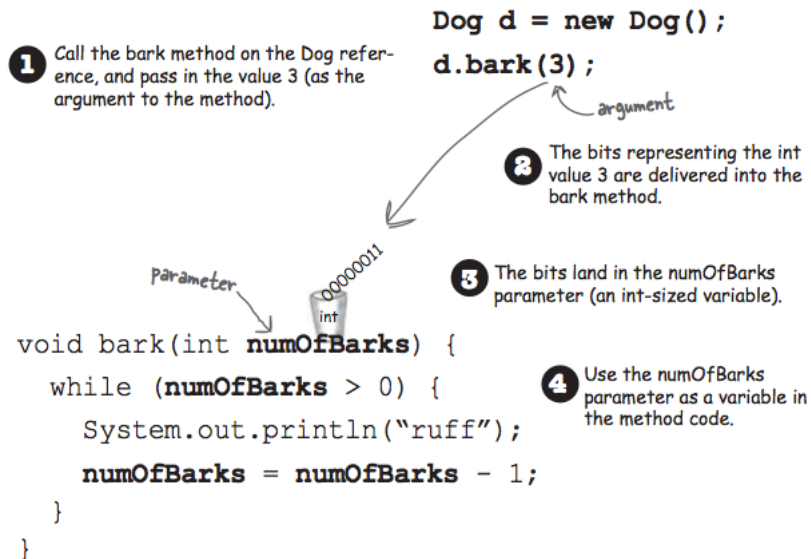
## Dog Tester Class

```
class DogTester {  
    public static void main(String[] args) {  
        Dog rex = new Dog();  
        rex.size=70;  
  
        Dog fido = new Dog();  
        fido.size=8;  
  
        Dog spot = new Dog();  
        spot.size=35;  
  
        rex.bark();  
        fido.bark();  
        spot.bark();  
    }  
}
```

## Method Parameters

- Passing parameters to methods
  - **Methods use parameters**, a **caller passes arguments**
  - Arguments are passed to methods by caller - `rex.bark(4);`
  - Arguments can be primitive types or reference types
  - The argument becomes the parameter
  - A parameter is a **local variable** with a type and a name to be used inside the body of a method
  - If a method takes a parameter you **must** pass it an argument of the appropriate type

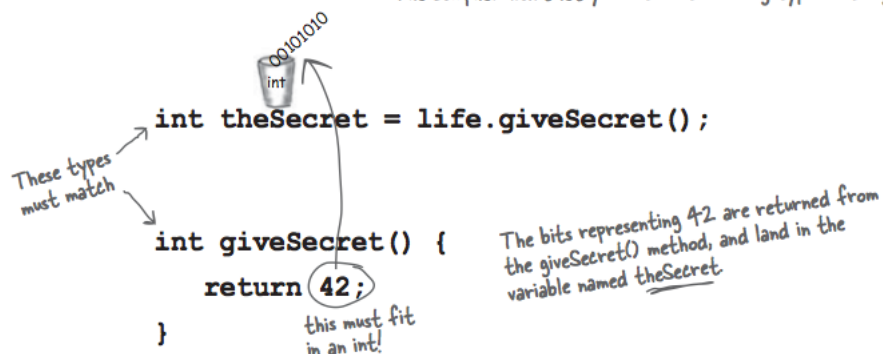
# Method Parameters



# Return values

- Methods can return values similar to the return type contained in the method signature

The compiler won't let you return the wrong type of thing.



# Multiple Arguments

- Methods can have multiple parameters
  - Must be comma separated
  - Passed arguments must be of the right type and in the correct order

Calling a two-parameter method, and sending it two arguments.

```
void go() {  
    TestStuff t = new TestStuff();  
    t.takeTwo(12, 34);  
}  
  
void takeTwo(int x, int y) {  
    int z = x + y;  
    System.out.println("Total is " + z);  
}
```

The arguments you pass land in the same order you passed them. First argument lands in the first parameter, second argument in the second parameter, and so on.

# Variables & Parameters

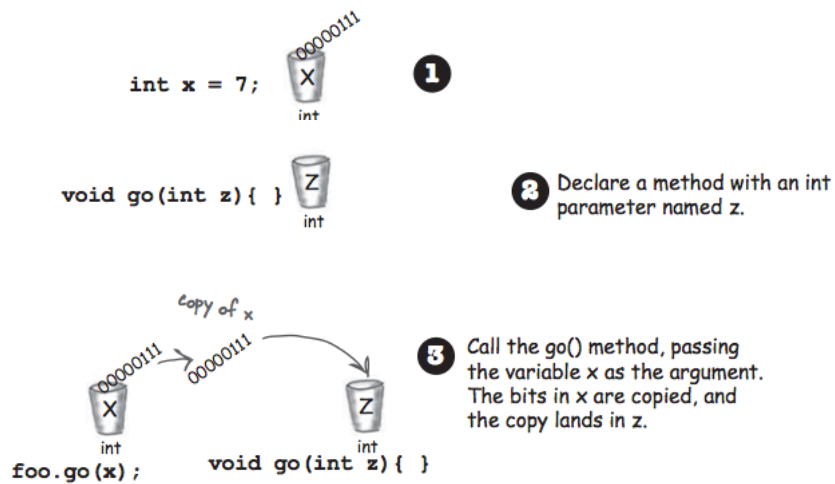
You can pass variables into a method, as long as the variable type matches the parameter type.

```
void go() {  
    int foo = 7;  
    int bar = 3;  
    t.takeTwo(foo, bar);  
}  
  
void takeTwo(int x, int y) {  
    int z = x + y;  
    System.out.println("Total is " + z);  
}
```

The values of foo and bar land in the x and y parameters. So now the bits in x are identical to the bits in foo (the bit pattern for the integer 7) and the bits in y are identical to the bits in bar.

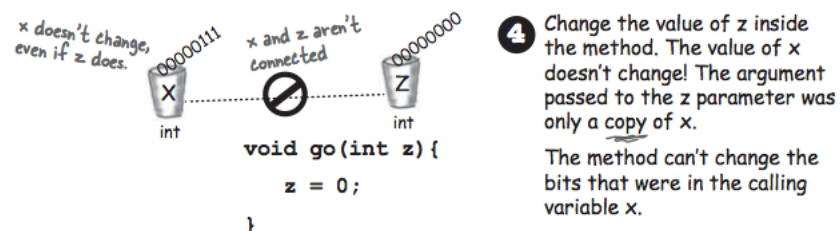
What's the value of z? It's the same result you'd get if you added foo + bar at the time you passed them into the takeTwo method

## Pass-by-value



## Pass-by-value

- z is a **local variable** and as such it's **scope** (lifetime) is that of the method within which it lives.



# Pass By Value

```
public class PassByValue {  
    public static void main(String[] args) {  
        int x=10, y=20;  
  
        System.out.println("main method before method call x = " + x + " y = " + y);  
        myMethod(x,y); // method call  
  
        System.out.println("main method after method call x = " + x + " y = " + y);  
    } // end main method  
  
    public static void myMethod(int x, double y ) {  
        System.out.println("myMethod before assignment x = " + x + " y = " + y);  
  
        x = 25;  
        y = 35.4;  
  
        System.out.println("myMethod after assignment x = " + x + " y = " + y);  
    } // end myMethod  
} // end class
```

## So Far

- Classes define what an object knows & does
  - Knows -> instance variable (state)
  - Does -> methods (behaviour)
- Methods use instance variables so objects of the same type can behave differently
- Methods can have parameters -> values can be passed
- Number & type must match the order & type of parameters declared by the parameter
- Values passed to and from methods can be implicitly **promoted** to a larger type OR explicitly **cast** to a smaller type

## So far (contd.)

- Values passed can be **literal** or a **variable** of the declared parameter type (other possibilities to follow)
- Methods must declare a return type
  - void return type means nothing is returned
  - `void myMethod(int x, double y) { }`
- If a method declares a non-void return type, it must return a value compatible with the declared return type
  - `int myMethod(int x) {return value;}`

## getters & setters

- Get & Set instance variables (normally)
- getters (Accessors)
  - Send back (return) a value associated with the getter

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

- setters (Mutators)
  - Takes an argument value and uses it to set the value of an instance variable

```
public void setSize(int s) {  
    size = s;  
}
```



# Encapsulation

- Data Hiding (not exposing our data)
- Exposing means using the dot (.) operator

```
rex.size = 0;
```

- Changing the size to zero!!!
- Better to allow access to instance variables **ONLY** through getters & setters
- Need to know about **public** & **private** access modifiers
  - Mark **your instance variables PRIVATE**
  - **Provide PUBLIC getters and setters** for access control

## getter & setter

```
class GoodDog {  
    private int size;  
  
    public int getSize() {  
        return size;  
    }  
  
    public void setSize(int s) {  
        size = s;  
    }  
  
    void bark() {  
        if (size > 60) {  
            System.out.println("Woof! Woof!");  
        } else if (size > 14) {  
            System.out.println("Ruff! Ruff!");  
        } else {  
            System.out.println("Yip! Yip!");  
        }  
    }  
}
```

## tester

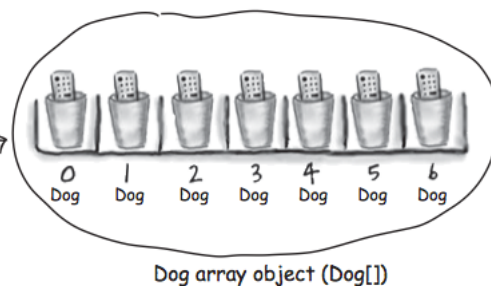
```
public class GoodDogTestDrive {  
    public static void main(String[] args) {  
        GoodDog one = new GoodDog();  
        one.setSize(70);  
  
        GoodDog two = new GoodDog();  
        two.setSize(8);  
  
        System.out.println("Dog one: " + one.getSize());  
        System.out.println("Dog two: " + two.getSize());  
  
        one.bark();  
        two.bark();  
    }  
}
```

## Arrays of Objects

- Accessing objects in an array of objects

- 1 Declare and create a Dog array, to hold 7 Dog references.

```
Dog[] pets;  
pets = new Dog[7];
```



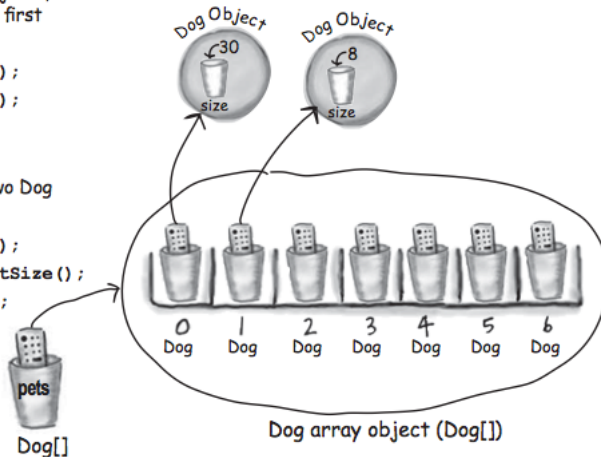
## Arrays of Objects

- 2 Create two new Dog objects, and assign them to the first two array elements.

```
pets[0] = new Dog();  
pets[1] = new Dog();
```

- 3 Call methods on the two Dog objects.

```
pets[0].setSize(30);  
int x = pets[0].getSize();  
pets[1].setSize(8);
```



## Arrays of Objects

```
public class GoodDogArray {  
    public static void main(String[] args) {  
        // declare & create Dog array  
        GoodDog[] pets = new GoodDog[7];  
  
        // Create two new objects and assign them to the first two array elements  
        pets[0] = new GoodDog();  
        pets[1] = new GoodDog();  
  
        // Call the methods on the two GoodDog objects  
        pets[0].setSize(30);  
        System.out.println("Size of pet[0] = " + pets[0].getSize());  
  
        pets[1].setSize(10);  
        System.out.println("Size of pet[1] = " + pets[1].getSize());  
    }  
}
```

# Declaring & Initialising instance Variables

- Variable declaration need a name and a type

```
int size;  
String name;
```

- Can initialise at the same time

```
int size = 20;  
String name = "Harry";
```

- When you don't initialise an instance variable...

**Instance variables always get a default value. If you don't explicitly assign a value to an instance variable, or you don't call a setter method, the instance variable still has a value!**

integers	0
floating points	0.0
booleans	false
references	null

# Declare & Initialise Instance Variables

```
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 one = new PoorDog();  
        System.out.println("Dog size is " + one.getSize());  
        System.out.println("Dog name is " + one.getName());  
    }  
}
```

```
File Edit Window Help CalVet  
% java PoorDogTestDrive  
Dog size is 0  
Dog name is null
```

What will these return??

What do you think? Will this even compile?

You don't have to initialize instance variables, because they always have a default value. Number primitives (including char) get 0, booleans get false, and object reference variables get null.

(Remember, null just means a remote control that isn't controlling / programmed to anything. A reference, but no actual object.)

# Instance Variables

- Instance variables always get a default value (even when you don't explicitly assign one or use the setter method)

```
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 one = new PoorDog();
        System.out.println("Dog size is " + one.getSize());
        System.out.println("Dog name is " + one.getName());
    }
}
```

# Instance VS Local

- 1 Instance** variables are declared inside a class but not within a method.

```
class Horse {
    private double height = 15.2;
    private String breed;
    // more code...
}
```

**Local variables do NOT get a default value! The compiler complains if you try to use a local variable before the variable is initialized.**

- 2 Local** variables are declared within a method.

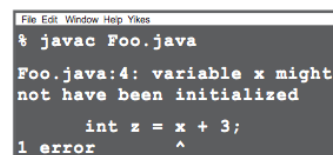
```
class AddThing {
    int a;
    int b = 12;

    public int add() {
        int total = a + b;
        return total;
    }
}
```

- 3 Local** variables MUST be initialized before use!

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

Won't compile!! You can declare x without a value, but as soon as you try to USE it, the compiler freaks out.



```
File Edit Window Help Yikes
% javac Foo.java
Foo.java:4: variable x might
not have been initialized
    int z = x + 3;
    ^
1 error
```

## Comparing Variables (primitives & reference)

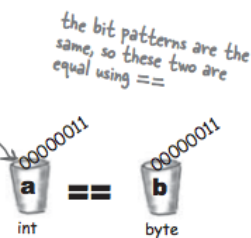
- To know if primitives are the same use the `==` operator
- To know if two objects (references) are the same use the `.equals()` method

The `==` operator can be used to compare two variables of any kind, and it simply compares the bits.

if `(a == b)` {...} looks at the bits in `a` and `b` and returns true if the bit pattern is the same (although it doesn't care about the size of the variable, so all the extra zeroes on the left end don't matter).

```
int a = 3;  
byte b = 3;  
if (a == b) { // true }
```

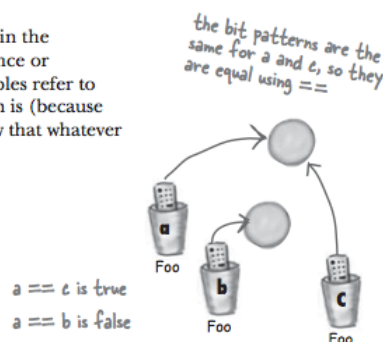
(there are more zeroes on the left side of the int, but we don't care about that here)



## Comparing Variables `==` & Reference

Remember, the `==` operator cares only about the pattern of bits in the variable. The rules are the same whether the variable is a reference or primitive. So the `==` operator returns true if two reference variables refer to the same object! In that case, we don't know what the bit pattern is (because it's dependent on the JVM, and hidden from us) but we *do* know that whatever it looks like, *it will be the same for two references to a single object*.

```
Foo a = new Foo();  
Foo b = new Foo();  
Foo c = a;  
if (a == b) { // false }  
if (a == c) { // true }  
if (b == c) { // false }
```



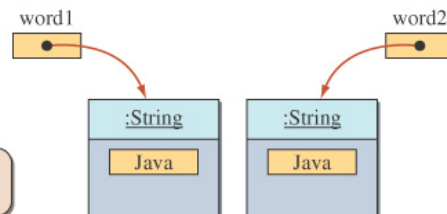
## Effect of Immutability (String Pool)

```
String word1, word2;
```

```
word1 = new String("Java");
```

```
word2 = new String("Java");
```

Whenever the **new** operator is used, there will be a new object.



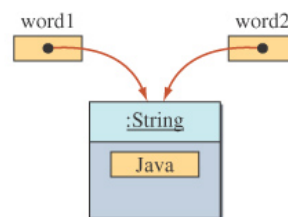
```
String word1, word2;
```

```
word1 = "Java";
```

```
word2 = "Java";
```

Literal string constant such as "Java" will always refer to the one object.

We can do this because String objects are immutable.



## == VS equals()

```
public class StringExample {  
    public static void main(String[] args) {  
        String s1 = "Java";  
        String s2 = new String("Java");  
  
        if (s1 == s2)  
            System.out.println("s1 = s2");  
        else  
            System.out.println("s1 != s2");  
  
        if (s1.equals(s2))  
            System.out.println("s1 = s2");  
        else  
            System.out.println("s1 != s2");  
    }  
}
```

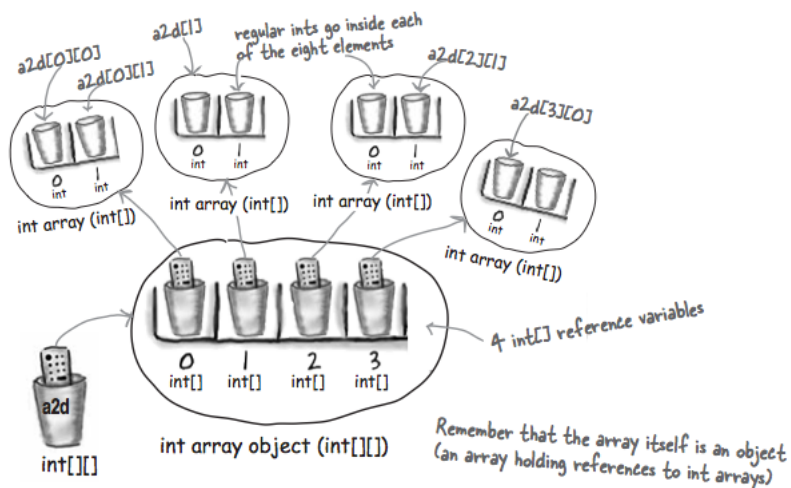
# Multidimensional Arrays

- In Java a multidimensional array can be viewed as an array of arrays.
- The JVM creates an array with 4 elements, each of which is a reference variable referring to a newly created int array with 2 elements

```
int [][] a2d = new int[4][2];
```

Row Column

# Multidimensional Arrays





## 2D Array

```
public class TwoDimensionalArray {  
    public static void main(String[] args) {  
        int[][] a2d = new int[4][2];  
        for(int i=0; i < a2d.length; i++){  
            for(int j=0; j < a2d[i].length; j++){  
                a2d[i][j]=(int)(Math.random()*10);  
            }  
        }  
        for(int i=0; i < a2d.length; i++){  
            for(int j=0; j < a2d[i].length; j++){  
                System.out.print(a2d[i][j] + " ");  
            }  
            System.out.println();  
        }  
    }  
}
```

## Methods & Instance Variables

- **Remember**
  - Methods use instance variables, keep your variables *private*



## Summary

- Method parameters
- variables & parameters
- Pass-by-value
- Method parameters, return values
- getter & setter methods
- Encapsulation (revisited)
- Arrays of Objects (revisited)
- Instance variables, declaration & initialising
- Comparing primitive & reference variables
- Multidimensional Arrays