OBJECT ORIENTED PROGRAMMING IN JAVA

MIKHAIL MARKOV

DEPARTMENT OF COMPUTING AND SOFTWARE
MCMASTER UNIVERSITY

SFWRENG/COMPSCI 2S03 (FALL 2015)

TUTORIAL 5

Contacts

- **Graduate Teaching Assistants**:

Akhil Krishnan: akhil4490@gmail.com

Michael Liut: liutm@mcmaster.ca

Mikhail Markov: markoma@mcmaster.ca

- **Undergraduate Teaching Assistants**:

Jemar Jones: jonesjk@mcmaster.ca

Wenqiang Chen: chenw25@mcmaster.ca

Outline

- Class and Object
- Class and Object: example
- Class bird example: class definition
- Class bird example: objects
- Class bird example: class definition with constructors
- Class bird example: objects with constructors
- Encapsulation
- Encapsulation: example
- Immutability
- Immutability: example of changing the values
- Immutability: the same example with the “final” keyword
- Method Overloading
- Method Overloading: example

Class and Object

- **Class -** the "**blue-print**" that describes the possible behaviours and states that object of its type support.
- **Object -** an **instance** of the class (the **houses** built from the blue-print). Have specific values of the states and behaviours.

Class and Object: example

Example

- **Class**: bird
- **State**: size, colour, family, etc.
- **Behavior**: flying, swimming, eating, etc.
- **Objects**: sparrow, duck, goose, etc.
- Object_Sparrow States: size = small, colour = brown-grey, family = Passeridae; Sparrow Behavior: flying

Class bird example: class definition

```
public class bird {

// States
        public String size, colour, family;
// Behaviors
        void bird_behavior_flying() {
            System.out.println("The bird is flying now");
            }

        void bird_behavior_swimming() {
            System.out.println("The bird is swimming now");
            }

        void bird_behavior_eating() {
            System.out.println("The bird is eating now");
            }
}
```

Class bird example: objects

```
public class animals extends bird {

    public static void main(String[] args) {

        bird sparrow = new bird();  //create an object sparrow from the class bird
        bird pigeon = new bird();   //create an object pigeon from the class bird
// Assign values to every state of the object sparrow
        sparrow.colour = "brown-gray";
        sparrow.size = "small";
        sparrow.family = "Passeridae";
// Assign values to every state of the object pigeon
        pigeon.colour = "white";
        pigeon.size = "medium";
        pigeon.family = "Columbidae";

        System.out.println("\nThe bird sparrow is "+ sparrow.size + " in size and has "
        + sparrow.colour + " colour. The family: " + sparrow.family);
// Invoke method bird_behavior_eating() on the object
        sparrow.bird_behavior_eating();

        System.out.println("\nThe bird pigeon is "+ pigeon.size + " in size and has "
        + pigeon.colour + " colour. The family: " + pigeon.family);
// Invoke method bird_behavior_flying() on the object
        pigeon.bird_behavior_flying();
    }
}
```

The result:

```
The bird sparrow is small in size and has brown-gray colour. The family: Passeridae
The bird is eating now

The bird pigeon is medium in size and has white colour. The family: Columbidae
The bird is flying now
```

Class bird example: objects

We can **manually** assign the values of the states to every object.

```
// Assign values to every state of the object sparrow
        sparrow.colour = "brown-gray";
        sparrow.size = "small";
        sparrow.family = "Passeridae";
// Assign values to every state of the object pigeon
        pigeon.colour = "white";
        pigeon.size = "medium";
        pigeon.family = "Columbidae";
```

But what if we have hundreds of the similar objects?? ***Can we do better?***

Class bird example: objects

Yes we can!

We need to use **constructors**!

With multiple constructors we can create the objects with all or just several same states.

Class bird example: class definition with constructors

```
public class bird {

// States
        public String size, colour, family;

// Constructors
        bird(){size = "big"; colour = "gray-brown-black"; family = "Anatidae";};
        bird(String i){size = "big"; colour = i; family = "Anatidae";};
        bird(String i, String j, String k){size = i; colour = j; family=k;};

// Behaviors
        void bird_behavior_flying() {
             System.out.println("The bird is flying now");
             }

        void bird_behavior_swimming() {
             System.out.println("The bird is swimming now");
             }

        void bird_behavior_eating() {
             System.out.println("The bird is eating now");
             }
}
```

Class bird example: objects with constructors

```
public class animals extends bird {

    public static void main(String[] args) {

// create an object gooseCanada from the class bird with the 1st (default) constructor
        bird gooseCanada = new bird();
// create an object swan from the class bird with the 2nd constructor
        bird swan = new bird("white");
// create an object sparrow from the class bird with the 3rd constructor
        bird sparrow = new bird("small","brown-gray","Passeridae");

        System.out.println("\nThe bird sparrow is "+ sparrow.size + " in size and has "
        + sparrow.colour + " colour. The family: " + sparrow.family);

// Invoke method bird_behavior_eating() on the object
        sparrow.bird_behavior_eating();

        System.out.println("\nThe bird swan is "+ swan.size + " in size and has "
        + swan.colour + " colour. The family: " + swan.family);

// Invoke method bird_behavior_flying() on   the object

        swan.bird_behavior_flying();

        System.out.println("\nThe bird Canada goose is "+ gooseCanada.size + " in size and has "
        + gooseCanada.colour + " colour. The family: " + gooseCanada.family);

// Invoke sequentially multiple methods on the object
        gooseCanada.bird_behavior_swimming();
        gooseCanada.bird_behavior_flying();
        gooseCanada.bird_behavior_eating();
        gooseCanada.bird_behavior_swimming();
    }
}
```

Class bird example: objects with constructors

```java
public class animals extends bird {

    public static void main(String[] args) {

// create an object gooseCanada from the class bird with the 1st (default) constructor
        bird gooseCanada = new bird();
// create an object swan from the class bird with the 2nd constructor
        bird swan = new bird("white");
// create an object sparrow from the class bird with the 3rd constructor
        bird sparrow = new bird("small","brown-gray","Passeridae");

        System.out.println("\nThe bird sparrow is "+ sparrow.size + " in size and has "
        + sparrow.colour + " colour. The family: " + sparrow.family);

// Invoke method bird_behavior_eating() on the object
        sparrow.bird_behavior_eating();

        System.out.println("\nThe bird swan is "+ swan.size + " in size and has "
        + swan.colour + " colour. The family: " + swan.family);

// Invoke method bird_behavior_flying() on   the object

        swan.bird_behavior_flying();

        System.out.println("\nThe bird Canada goose is "+ gooseCanada.size + " in size and has "
        + gooseCanada.colour + " colour. The family: " + gooseCanada.family);

// Invoke sequentially multiple methods on the object
        gooseCanada.bird_behavior_swimming();
        gooseCanada.bird_behavior_flying();
        gooseCanada.bird_behavior_eating();
        gooseCanada.bird_behavior_swimming();
    }
}
```

The result:

```
The bird sparrow is small in size and has brown-gray colour. The family: Passeridae
The bird is eating now

The bird swan is big in size and has white colour. The family: Anatidae
The bird is flying now

The bird Canada goose is big in size and has gray-brown-black colour. The family: Anatidae
The bird is swimming now
The bird is flying now
The bird is eating now
The bird is swimming now
```

Encapsulation

*What will happen if we'll change the **modifier** of the class states from "**public**" **to** "**private**" and we'll try to assign the values of the states from the main function?*

```
public class bird {

// States
        private String size, colour, family;
// Behaviors
        void bird_behavior_flying() {
             System.out.println("The bird is flying now");
             }
}
```

Encapsulation

```
public class bird {

// States
        private String size, colour, family;
// Behaviors
        void bird_behavior_flying() {
             System.out.println("The bird is flying now");
             }
}
```

The result is …

```
public class animals extends bird {

    public static void main(String[] args) {

        bird sparrow = new bird();  //create an object sparrow from the class bird
// Assign values to every state of the object sparrow
        sparrow.colour = "brown-gray";
        sparrow.size = "small";
        sparrow.family = "Passeridae";

// Invoke method bird_behavior_eating() on the object
        sparrow.bird_behavior_eating();
    }
}
```

Encapsulation

*The **result** is an **exception** (doesn't compile) as the states of the class with the **modifier** "**private**" becomes **not visible**. So it can't be changed somehow from another class.*

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
    The field bird.colour is not visible
    The field bird.size is not visible
    The field bird.family is not visible
```

Encapsulation

*The **benefit of encapsulation** is that we can, for instance, define the possible values of the states. This values can't be changed to another (not in our list) like it's possible with the "public" modifier.*

Let's look at the example.

Encapsulation: example

```
public class bird {

// States
        private String size, colour, family;

// The method to change the state of the size
// according to the 3 possible values
        void changeStateSize(String isize){
            if (isize == "big") {
                size = "big";
                System.out.println("\nThe size was changed to big");
            }

            if (isize == "medium"){
                size = "medium";
                System.out.println("\nThe size was changed to medium");
            }

            if (isize == "small"){
                size = "small";
                System.out.println("\nThe size was changed to small");
            }
        }
// Print the current value of the size
        void birdSize() {
             System.out.println("The bird size is:" + size);
             }
}
```

Encapsulation: example

```
public class animals extends bird {

    public static void main(String[] args) {

        bird sparrow = new bird();  //create an object sparrow from the class bird
// Assign value to the state size of the object sparrow with the class method
// Let's try to change the size with the value from the list
        sparrow.changeStateSize("small");
        sparrow.birdSize();
// Let's try to change the size with the value NOT from the list
        sparrow.changeStateSize("huge");
        sparrow.birdSize();
    }
}
```

The result:

```
The size was changed to small
The bird size is:small
The bird size is:small
```

Immutability

In general we can change the values of the states multiple times.

Let's look at the example.

Immutability: example of changing the values

```
public class bird {

// States
        public String size;
        public String colour, family;

// Constructors
        bird(){size = "big"; colour = "gray-brown-black"; family = "Anatidae";};
        bird(String i){size = "big"; colour = i; family = "Anatidae";};
        bird(String i, String j, String k){size = i; colour = j; family=k;};

// Print the current value of the size
        void birdSize() {
         System.out.println("The bird size is:" + size);
        }
}
```

Immutability: example of changing the values

```
public class animals extends bird {

    public static void main(String[] args) {

// create an object swan from the class bird with the 2nd constructor
        bird swan = new bird("big","white", "Anatidae");
// create an object sparrow from the class bird with the 3rd constructor
        swan.birdSize();
// Let's change the value of the size
        swan.size = "small";
        swan.birdSize();
    }
}
```

The result:

```
The bird size is:big
The bird size is:small
```

Immutability: the same example with the “final” keyword

*Let’s add “**final**” keyword to add **immutability** and run the same code.*

```
public class bird {

// States
        public final String size;
        public String colour, family;
```

The result is an exception:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    The final field bird.size cannot be assigned
```

But if we’ll comment the change:

```
        swan.birdSize();
// Let's change the value of the size
//      swan.size = "small";
        swan.birdSize();
```

The code works:

```
The bird size is:big
The bird size is:big
```

Method Overloading

Method ***overloading*** in used when several *methods* of the same class have the *same name* but *different number or type of the parameters.*

Method Overloading: example

```
public class valuesXY {

// States of different types
    private int x;
    private float y;
    private double z;
// Default constructor. The values of the states should be 0 or 0.0
    valuesXY(){}
// Method writeNumber() with the parameter of type int
    public void writeNumber(int temp){x = temp;}
// Method writeNumber() with the parameter of type float
    public void writeNumber(float temp){y = temp;}
// Method writeNumber() with the parameter of type double
    public void writeNumber(double temp){z = temp;}
// Print the current values of the states
    public void printValues(){
        System.out.println("The value of x is: " + x + " and y is: " + y+ " and z is: " + z);
    }
}
```

Method Overloading: example

```
public class Overloading extends valuesXY{

    public static void main(String[] args) {
// New object x1 from the class valuesXY
        valuesXY x1 = new valuesXY();
// Assign different values to the states with the same-named method
        x1.printValues();
        x1.writeNumber(99);
        x1.printValues();
        x1.writeNumber(3.5f);
        x1.printValues();
        x1.writeNumber(44.4);
        x1.printValues();
    }
}
```

The result:

```
The value of x is: 0 and y is: 0.0 and z is: 0.0
The value of x is: 99 and y is: 0.0 and z is: 0.0
The value of x is: 99 and y is: 3.5 and z is: 0.0
The value of x is: 99 and y is: 3.5 and z is: 44.4
```