Java

Inheritance

Hannes Ueck, Jakob Krude

2. Dezember 2020

Java-Kurs

Overview

Inheritance

A special Delivery

Our class *Letter* is a kind of *Delivery* denoted by the keyword **extends**.

- *Letter* is a **subclass** of the class *Delivery*
- *Delivery* is the **superclass** of the class *Letter*

```java
    public class Letter extends Delivery {

    }

```

As mentioned implicitly above a class can has multiple subclasses. But a class can only inherit directly from one superclass.

Example

We have the classes: *PostOffice*, *Delivery* and *Letter*. They will be used for every example in this section and they will grow over time.

```
public class Delivery {

    private String address;
    private String sender;

    public void setAddress(String addr) {
        address = addr;
    }

    public void setSender(String snd) {
        sender = snd;
    }

    public void printAddress() {
        System.out.println(this.address);
    }
}

```

Inherited Methods

The class *Letter* also inherits all methods from the superclass *Delivery*.

```java
    public class PostOffice {

        public static void main(String[] args) {

            Letter letter = new Letter();

            letter.setAddress("cafe ascii, Dresden");

            letter.printAddress();
            // prints: cafe ascii, Dresden
        }
    }

```

Override Methods

The method printAddress() is now additional definded in *Letter*.

```java
    public class Letter extends Delivery {

        @Override
        public void printAddress() {
            System.out.println("a letter for " + this.address);
        }
    }

```

`@Override` is an annotation. It helps the programer to identify overwritten methods. It is not neccessary for running the code but improves readability. What annotations else can do we discuss in a future lesson.

Override Methods

Now the method `printAddress()` defined in *Letter* will be used instead of the method defined in the superclass *Delivery*.

```java
public class PostOffice {

    public static void main(String[] args) {

        Letter letter = new Letter();

        letter.setAddress("cafe ascii, Dresden");

        letter.printAddress();
        // prints: a letter for cafe ascii, Dresden
    }
}

```

Super()

If we define a **constructor with arguments** in *Delivery* we have to define a constructor with the same list of arguments in every subclass.

```java
    public class Delivery {

        private String address;
        private String sender;

        public Delivery(String address, String sender) {
            this.address = address;
            this.sender = sender;
        }

        public void printAddress() {
            System.out.println(address);
        }
    }

```

Super()

For the constructor in the subclass *Letter* we can use `super()` to call the constructor from the superclass.

```
    public class Letter extends Delivery {

        public Letter(String address, String sender) {
            super(address, sender);
        }

        @Override
        public void printAddress() {
            System.out.println("a letter for " + this.
    address);
        }
    }

```

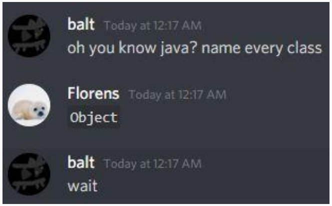
Super() - Test

```
public class PostOffice {

    public static void main(String[] args) {
        Letter letter =
            new Letter("cafe ascii, Dresden", "");

        letter.printAddress();
        // prints: a letter for cafe ascii, Dresden
    }
}

```

Object

Every class is a subclass from the class *Object*. Therefore every class inherits methods from *Object*.

See `http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html` for a full reference of the class *Object*.



balt Today at 12:17 AM
oh you know java? name every class

Florens Today at 12:17 AM
`Object`

balt Today at 12:17 AM
wait

toString()

Letter is a subclass of *Object*. Therefore *Letter* inherits the method `toString()` from *Object*.
`System.out.println(argument)` will call `argument.toString()` to receive a printable String.

```java
    public class PostOffice {

        public static void main(String[] args) {
            Letter letter =
                new Letter("cafe ascii, Dresden", "");

            System.out.println(letter);
            // prints: Letter@_some_HEX-value_
            // for example: Letter@4536ad4d
        }
    }

```

Override toString()

```
public class Letter extends Delivery {

    public Letter(String address, String sender) {
        super(address, sender);
    }

    @Override
    public String toString() {
        return "a letter for " + this.address;
    }
}

```

Override toString() - Test

```java
public class PostOffice {

    public static void main(String[] args) {
        Letter letter =
            new Letter("cafe ascii, Dresden", "");

        System.out.println(letter);
        // a letter for cafe ascii, Dresden
    }
}

```

Comparing objects

== vs .equals()

==

- Used to compare primitive datatypes
  (int, float, string, ...)

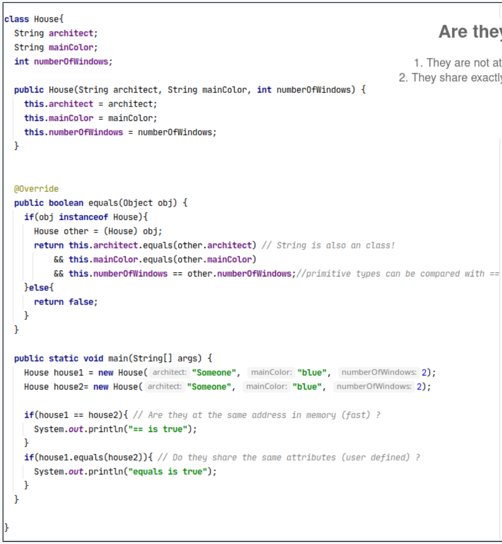
BUT:

- (object1 == object2)
  → memoryaddress of the objects is compared

.equals()

- Used to compare objects
- Returns true if all of the attributes are the same
- Method is inherited from Object, but defaults to ==
- Can be customized by overwriting .equals() in class

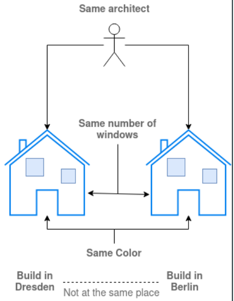
== vs .equals(): At a glance



```java
class House{
  String architect;
  String mainColor;
  int numberOfWindows;

  public House(String architect, String mainColor, int numberOfWindows) {
    this.architect = architect;
    this.mainColor = mainColor;
    this.numberOfWindows = numberOfWindows;
  }


  @Override
  public boolean equals(Object obj) {
    if(obj instanceof House){
      House other = (House) obj;
      return this.architect.equals(other.architect) // String is also an class!
          && this.mainColor.equals(other.mainColor)
          && this.numberOfWindows == other.numberOfWindows;//primitive types can be compared with ==
    }else{
      return false;
    }
  }

  public static void main(String[] args) {
    House house1 = new House( architect: "Someone",  mainColor: "blue",  numberOfWindows: 2);
    House house2= new House( architect: "Someone",  mainColor: "blue",  numberOfWindows: 2);

    if(house1 == house2){ // Are they at the same address in memory (fast) ?
      System.out.println("== is true");
    }
    if(house1.equals(house2)){ // Do they share the same attributes (user defined) ?
      System.out.println("equals is true");
    }
  }

}
```

Are they the same?

1. They are not at the *same address* → no
2. They share exactly the *same attributes* → yes



Same architect
Same number of windows
Same Color
Build in Dresden
Not at the same place
Build in Berlin