2021/2022(1)

IF184301 Object Oriented Programming

Lecture #3b

Classes and Objects

Misbakhul Munir IRFAN SUBAKTI

司馬伊凡

Мисбакхул Мунир Ирфан Субакти

Recap

- Basic Java Class
- Strings

Basic Java Class: Hello World

```java
public class Hello {

    public static void main(String[] args) {
        System.out.println("Hello world!");
    }

}
```

- The simplest thing we can do is to print `Hello world!` from the program.
- In the main method, just write `System.out.println("Hello world!");`
- Note that the capital letter, the dots, the type of quotes and the semicolon are all important.
- We can write out, we need also to be able to read in, which is where the args variable comes in handy.
  - `String aLineOfText = args[0];`
  - `args` contains all the words passed in to Java, and you can access them in turn using `args[0]`, `args[1]`, `args[2]`, `args[3]`, … and so on.
  - `args[0]` contains the first word, `args[1]` contains the second word, `args[2]` is the third, … and so on.

Basic Java Class: Hello World (continued)

- These words are placed into args by typing them after your program name on the command line: `java Hello2 Welcome to our Webpro lecture!` and the program would print `Welcome` as the value of `args[0]`, `to` as `args[1]`, `our` as `args[2]`, `Webpro` as `args[3]`, and `lecture!` as `args[4]`.
- If we were to combine `args` for reading in words, and `System.out.println` for printing to the screen, we could then print the words given.
- `String` is what's known as an *object type*
  - For now, you don't need to worry about this
  - It's sufficient to know that in this example, we are assigning the text in `args[0]` to the `String` variable `aLineOfText`
  - In fact, we can print that text back out: `System.out.println("She said, " + aLineOfText + "! ");`
  - Note how we've used the + to "add" words together.

Hello.java | Hello2.java

```java
1  public class Hello2 {
2
3      public static void main(String[] args) {
4          String aLineOfText = "";
5          if (args.length > 0) {
6              aLineOfText = args[0];
7              System.out.println("She said, " + aLineOfText + "!");
8          } else {
9              System.out.println("Hello world!");
10         }
11     }
12
13 }
```

```
D:\ITS\2021 ITS\08 OOP\Program\hello\src>java Hello2 Welcome to our Webpro lecture!
She said, Welcome!
```

Numbers & arithmetic operations

- The interesting stuff really starts to happen when we work with numbers and other types.
- Take a look at the following code
  - `int i = 0; int j = 3;`
  - We're declaring two variables of type `int`, and giving them values of `0` and `3` respectively.
  - We can do a lot with those numbers; in fact, basic arithmetic works just like we'd expect in Java.
- If we write
  - `int result = 2 + i * j;`
  - The Java program will multiply `i` and `j` together, and then add `2`, just like our calculator would.
  - The value worked out is then placed in `result`, just like we were given `i` the value `0`, and `j` was given the value `3` before.
  - It follows mathematical precedence, so it will do the multiplication of `i` and `j` first, before adding `2`.

Numbers & arithmetic operations (cont’d)

- `+`, `-`, `\`, and `*` represent the usual suspects of addition, subtraction, division and multiplication.
- Brackets function as we expect too, but for powers of, we need to use a method from the Math package, specially `Math.pow()`.
- We put two values in the brackets, one for the value and one for the power.
  - For example: `Math.pow(2, 3);` will give you a value of 8, or 2^3.
- If we take `System.out.println` from the previous example, we can also print out the results:
  - `System.out.println(Math.pow(2, 3));` That's not all we can do with numbers in Java, but it's enough for now.

Strings

- A `String` is a type of object in Java.
- It essentially refers to text.
- It's not as simple as it looks - it's actually a sequence of a more primitive type, `char` (which represents a single character).
- Strings are represented within double quotes, but characters only use single quotes

```
String myString = "Hello, my name is Yarik";
char myCharacter = 'y';
```

  - Note that although `String` begins with a capital letter, `char` doesn't. This is an important difference.

String: For what?

- A lot!
- Check it out about the String class in the API at
  - `https://docs.oracle.com/javase/8/docs/api/`
- We can see all the things that can be done to Strings
  - Some of them can be seen in the following

`startsWith(String prefix)`

- We can use this method to tell us if a `String` starts with a certain letter or sequence of letters.
- It returns a `Boolean`

```
String myString = "the quick brown fox
jumps over the lazy dog";
if (myString.startsWith("the"))
  System.out.println("It does!");
else
  System.out.println("It doesn't!");
```

substring(int beginIndex) and substring(int beginIndex, int endIndex)

- These two allow us to get part of a string

```
String mySecondString = "HelloYarik";
String secondToFifthWords =
mySecondString.substring(5); // "Yarik"
String firstWord = mySecondString.substring(0, 5);
// FirstWord is "Hello"
```

- Note that numbering starts from 0 with just about everything in Computer Science!

charAt(int index)

- This returns the char that is the letter at position index - again we start at 0

```
String anotherString = "abcde";
char characterTwo = anotherString.charAt(3);
// CharacterTwo is 'd'
```

`length()`

- Simply returns the length of the string, an `int`.

`String`: Converting to integers/doubles

- Another thing we can do with Strings that isn't contained in their API is convert them into integers or doubles.
- We can do this by using `Integer.parseInt()`, or `Double.parseDouble()`, but remember that putting anything other than numbers through either of these methods will throw errors at you.
- They are often useful when working with input from `args[0]`, so make a note of them!

Conventions

- Variable names
  - There are a few things we'd like you to know about conventions for naming in Java. Class names generally start with a capital letter
    - `public class MyClass {`
  - As such, the class above would be in a file named `MyClass.java`
  - The class will still work if not named starting with a capital letter, but conventions should be adhered to!
  - Something else to consider is variable naming: variables, and methods (more on those soon) start with lower case letters
    - `int myIntValue = 4;`
    - `String myName = "Naya";`

Variable names

- Note also the way that we've named those variables, using something called *camel case*
  - Each new 'word' in the variable name starts with a capital letter.
  - Again, this is just convention.
- Some things, like types, must start with the correct case—we'll have seen this if we've tried to write
  - `system.out.println;` or
  - `string s = "...";`

Commenting

- It’s important to start commenting our work, so that our reader can see what we’re doing.
- It’s also helpful to ourselves — our comments will help us to identify what each part of our work does when we go back to it.
- Getting into the habit now will serve us well once we start doing more complicated works.
- We will lose marks for not commenting, so start now!

Line comments

- Sometimes, we may just wish to document the function of a single line:
  - `String shorterString = s.substring(0, 3) ; // Take the first three characters of s`
- While it’s important to use comments to make our code easier to understand, don’t overdo it, and in particular, make sure that our comments are meaningful and do not just repeat the code in another language
  - `int i; // Declare an int called i`
  - `i = 3; // Set i to 3`
  - `System.out.println("i is equal to " + i); // Print what i is equal to`
- This can get quite annoying to read. As a general rule, if someone else in our cohort could easily understand what a line does, don’t comment it.

Block comments

- If a comment is big enough that it has to span more than one line, use a block.
- These generally go before the code:

```
/* Set up a Scanner, read a line of text
 * and store in line,
 * print back to user.
 */
Scanner sc = new Scanner(System.in);
String line = sc.nextLine();
System.out.println(line);
```

- Again, be careful not to over comment and make sure that it doesn’t just paraphrase the code!

Classes and Objects

- Being an object-oriented programming language, classes and objects are the very bedrock on which Java is built.
- Think of a class like a blueprint for something which we're going to build upon later, and "objects of that class" being implementations of the blueprint that we can actually do something with (i.e., hold data).
- In the lecture, we saw the classes `Counter` and `Tester`.
  - These classes are simply templates, which represent what information will be stored about each instance Object of type `Counter` and `Tester`.

Object: What is that?

- If a class is a blueprint, then an object of that class is a design based upon it.
- Let’s take a couple of examples.
  - We have a class `Dog` that represents a generic `Dog`.
  - Of course, it’s not possible for someone to just have a “generic Dog”.
  - They have to have a specific breed of `Dog` — else all Dogs would be the same.
  - Every `Dog` is an instance object of the class `Dog`.
  - Let’s say that several Dogs live in a kennel. First, we need to set up our `Dog` class.

Class Dog

```
public class Dog {
        // Global fields
        private int age;
        private double weight;
        private String name;
        private String breed;
        /* We don’t need a main method, because we don’t want to run "Dog" on its own.
           We just need a Constructor
         */
        public Dog(String name, String breed, int age, double weight) {
                this.name = name;
                this.age = age;
                this.weight = weight;
                this.breed = breed;
        }
}
```

Class `Dog` (continued)

- Notice the keyword **`this`**.
- If we didn't have that, what would happen?
  - The line **`name = name`** wouldn't make sense!
  - Java wouldn't know which was the field, and which the parameter.
  - The keyword **`this`** means (in this case) "set the name field in this object to be equal to the parameter name".
- Another way to think of classes and objects is like a spreadsheet. The class is like the column headers — the objects are like rows!

Creating Objects

- It's all well and good writing Objects, but what do we then do with them? Well, first we need to create or instantiate the Object. For most Objects, the only way to do that is via the new keyword.
  - `Dog fido = new Dog("Fido" , "Jack Russell", 5, 12.5) ;`
- What's going on above? Well, the `new` keyword tells Java to make room in memory for a new Object, and then calls the constructor of the given class to create the new Object. The arguments we give must exactly match the ones specified in the constructor, or Java will throw errors at us.
- Now we've created an object, we can apply methods to it, such as getters or setters.

Getters, Setters & Testing

- Often, we'll need to get information from our objects—in the example above, we would need to find out what the `name`, `breed`, `weight` or `age` of each dog was.
- Note that the class fields for each of these are marked `private`.
- That means we can't access them directly, or change them directly—we need to use getter and setter methods.

Getters

- A getter method is used to access the information in an object.
- Often, it just returns the value of one field.

```
public String getName( ) {
    return name;
}
```

- Note the use of the `return` keyword. This returns information to from wherever the method was called. If the return type of a method is anything but `void`, then there must be a `return` statement.

Setters

- As we already covered, the fields in `Dog` are `private`.
- This means they can only be modified by the class itself—via setter methods.

```
public void setName(String name) {
    this.name = name;
}
```

- Note that the return type of these methods is generally `void`. This is because they don't need to return something, and so don't need a `return` statement.

Other methods and testing

- One of the most important methods in any class is the **`toString()`** method. This returns a `String` representation of the object, and will usually look like nonsense, unless you redefine it. In our case, we want the **`toString`** method in `Dog` to tell us everything about a particular dog.

```
public String toString( ) {
    return "Dog name: " + getName( ) + "; breed: " + getBreed( )
          + "; age: " + getAge( ) + "; weight: " + getWeight( )
          + "kg";
}
```

- Note that we're using the getter methods to construct the `String`. This is good practice, as we may want to do something to the values in the fields before they're returned.
- Now that the `Dog` class is constructed, we can think about testing it.
  - We can do this by creating several instances of `Dog` in some other class.
  - With each of those instances, we can perform a few operations, and then check that the values are what we expect.

Other methods and testing (continued)

```
public static void main(String [ ] args) {
    Dog firstDog = new Dog("Fred", "Yorkshire Terrier", 3, 7.6);
    Dog secondDog = new Dog("Bella", "Rhodesian Ridgeback", 4, 11.6);

    System.out.println(firstDog);
    System.out.println(secondDog.toString()); // Equivalent to the line
                                              // above's behaviour

    firstDog.setAge(6);
    secondDog.setWeight(14.0);
    System.out.println(secondDog.getName() + " is now " +
        secondDog.getWeight() + " kg"); // Should be 14.0
    . . .
}
```

`JUnit` test

- Now, we are going to create a simple `JUnit` test in order to test the `getAge()` method.
- Simply put, we want to verify that once we have created a `Dog` object and its getter is called (in this case the `getAge()`), the correct age is being returned.
- First, we start by defining a new class `DogTest`, with the necessary import statements.
- Note that we do not have to type the `import` statements explicitly, but these will be automatically generated in Eclipse once we specify the appropriate annotations and `JUnit` commands, and place our cursor over them.

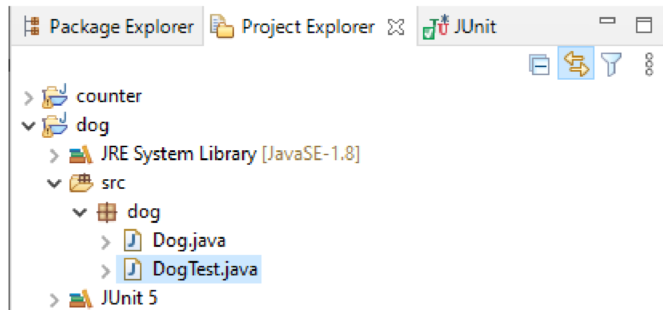
`JUnit` test (continued)

- `Dog.java`

```java
2
3 public class Dog {
4     // Global fields
5     private int age;
6     private double weight;
7     private String name;
8     private String breed;
9
10    public int getAge() {
11        return age;
12    }
13    public void setAge(int age) {
14        this.age = age;
15    }
16    public double getWeight() {
17        return weight;
18    }
19    public void setWeight(double weight) {
20        this.weight = weight;
21    }
22    public String getName() {
23        return name;
24    }
25    public void setName(String name) {
26        this.name = name;
27    }
28    public String getBreed() {
29        return breed;
30    }
31    public void setBreed(String breed) {
32        this.breed = breed;
33    }
34
35    /* We don't need a main method , because we don't want to run "Dog" on its own.
36       We just need a Constructor  */
37    public Dog(String name, String breed, int age, double weight) {
38        this.name = name;
39        this.age = age;
40        this.weight = weight;
41        this.breed = breed;
42    }
43    @Override
44    public String toString() {
45        return "Dog name: " + getName( ) + "; breed: " + getBreed( ) + "; age: "
46                + getAge( ) + "; weight: " + getWeight( ) + "kg";
47    }
48 }
```

JUnit test (continued)

- DogTest.java



Dog.java | DogTest.java

```java
package dog;

import static org.junit.Assert.*;
import org.junit.Test;

public class DogTest {

    /**
     * Testing the getAge() method
     */
    @Test
    public void test1() {
        Dog firstDog = new Dog("fred", "Yorkshire Terrirer", 3, 7.6);
        // 3 is what I expect if getAge() is called
        int expected = 3;
        // The actual value
        int firstDogAge = firstDog.getAge();
        // Note that it is a JUnit convention to
        // put the expected argument first
        assertEquals(expected, firstDogAge);
        // or you might want to add a message
        // that may help you to debug a test
        // in case an assertion fails
        assertEquals("Test1 of getAge() in DogTest", expected, firstDogAge);
    }

}
```

JUnit test (continued)

- Once the above JUnit test is run, we can note the green bar in Eclipse indicating that the test was successful.
- In contrast, a red bar indicates that a test has failed.