2021/2022(1)

IF184301 Object Oriented Programming

Lecture #5a

Javadoc, Testing & Objects

Misbakhul Munir IRFAN SUBAKTI

司馬伊凡

Мисбакхул Мунир Ирфан Субакти

Class: Get back!

- First we will consider classes and objects, and how to write and test them. Let's go back to consider the structure of a class.
- At the top of the class there's a sensible place to put *fields*. It referred to as *global variables*, *class variables*, and other things besides. They are *global* because they have *scope* throughout the class—any method can access or change them. They are *private* so that only the class itself can access or change them.
- Underneath that is the *constructor*, which is what gets called to *set* the fields' values whenever some other class creates a `new Dog(...)`. Often there will only be one constructor (if we don't write one, Java puts one there for us, but it won't do much). Sometimes, though, we'll need more than one.
- Under the constructor go the methods! One convention is to put *getter* methods (which return the fields, or something to do with them) first, and then *setter* methods (which modify fields) next, followed by the `toString()` method at the bottom. Often we also write an `equals()` method to test whether two objects should be considered as the same. However, it's important to note that methods, constructors and fields can really go in whatever order we want (just don't go putting methods inside methods).

Constructor: Multiple existence

- Let's think about a class representing bank accounts. When a user sets up a bank account, they can either open it with some money (in which case, they get a 50 overdraft), or they can open it empty (in which case, they only get a 10 overdraft). How can we represent this in Java? Well, we can give the `BankAccount` class *multiple* constructors—depending on the way in which a `BankAccount` object is instantiated, different actions are performed:

```
public class BankAccount {
        private double balance;
        private int overdraft;
        public BankAccount(double openingAmount) {
                balance = openingAmount;
                overdraft = 50;
        }
        public BankAccount( ) { // No parameters supplied, so default
                balance = 0;
                overdraft = 10;
        }
        // Getters , setters
}
```

- Now, to create an instance of BankAccount with a zero opening balance, we can write `BankAccount ba1 = new BankAccount();`. If we want to open an account with £25, we can write `BankAccount ba2 = new BankAccount(25);`, and then the overdraft is extended to £50. Having multiple constructors is useful for situations when things need to have different values depending on which initial parameters are supplied to the constructor.

Javadoc

- As programmers, we'll often need to know how other programmers' methods can work with our own. For this reason, we've already shown that *commenting* is very useful. However, although comments *within* methods are important to people who will view our source code, what about people who just want to know what inputs a method requires, what outputs are given, and what the function of our method is? We need to be able to comment on the *function of a method* without going into its implementation detail, and for that we use *Javadoc*.
- In the following diagram, we see some of this:

```
/**
 * Get dog name
 * @return dog's name
 */
public String getName( ) {
    return name;
}
```

Javadoc (continued)

- This is quite a simple method, so documenting it is easy too. A Javadoc comment begins with a slash, two asterisks "`/**`"and a return, and ends with `*/`. The first line is a *brief* description of what the method does. Then, we describe the *parameters of the method*, each in the form

  `@param` `<`**`parameterName`**`> <`**`description`**`>`

- and lastly (if applicable), we say *what* the method returns, using `@return <what is returned>`. For a more complex method, we might need both of these components. Back to the `Bank Account`:

```
/**
 * Calculate interest after a certain number of years
 * @param initial the initial balance
 * @param years the number of years
 * @param rate the interest rate
 * @return the compound interest on the balance
/
public double interestOn(double initial, int years, double rate) {
      double totalInterest;
      // Calculate interest
      return totalInterest;
}
```

- It's important that our Javadoc comment *all* of your methods, *including* the constructor. Then try out producing an API for your code: in the terminal (in the directory of our work), type `javadoc MyFile.java`. The tool will produce HTML for the class (note we can generate Javadoc for all of our classes, if we've written it, with `javadoc *.java`). Open `index.html` in a browser, and we should find it looks rather like the normal Java API!

Using API Classes: `Random`

- Talking of the API, there are a number of useful classes in there for us to use! We need to be able to generate *random numbers*. We can do this either using `Math.random()`, or by using the `java.util.Random` class. `java.util.Random` has the advantage that we can create random `int`s, or even `boolean`s, instead of simply `double`s. To use it, we need to import `java.util.Random` at the top of our class, and then create a new *instance* of `Random` with the line (for example)

  ```
  Random rand = new Random();
  ```

- Now, anywhere we want a random `double` (between 0 and 1), we can simply ask `rand` for it:

  ```
  double randomNumber = rand.nextDouble();
  ```

- Note that this method gets only double values. What if we want a random integer? Well, there are methods for that:

  ```
  int randomInt = rand.nextInt();
  // An int between 0 (inclusive) and 10 (exclusive)
  int upperBoundedInt = rand.nextInt(10);
  ```

- There are other methods provided by the `Random` class to give other forms of randomness. See the API page at https://docs.oracle.com/javase/8/docs/api/java/util/Random.html for more details.

JUnit: Recap

- In order to ensure that our code works, we are using JUnit to build test cases. This is not only to ensure it works now, but it also allows us to check again after making any changes.
- A JUnit test is defined like a normal Java class, and each test case is a regular Java method. However there are some differences between JUnit classes and methods, and regular Java classes and methods.
- First Java classes may have a `public static void main(String args[])` method, as the way to start the Java application. JUnit tests however do not have a main method.
- JUnit test cases, also start with `@Test`. This is because they are a normal method, and so this marker allows JUnit to tell if the method is a test case or not. JUnit will only know that the method is a test case, if it starts with `@Test`.
- Finally Java applications are started on the command line, in the style:

```
java MyApplication
```

- Where MyApplication is a class with a main method. JUnit tests however are run slightly differently:

```
java org.junit.runner.JUnitCore MyTestClass
```

- Here the main application is `JUnitCore`, which in turns runs our tests in `MyTestClass`. The key point to take away, is that JUnit is both a library for writing tests, and an application for running tests. It does both!

JUnit Test: Example

- Almost anything can be tested with a JUnit test. Imagine the following class from an earlier lecture:

```
public class Dog {
    private String name;
    private int age;
    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public int getAge() {
        return this.age;
    }
    public void incrementAge() {
        this.age++;
    }
}
```

JUnit Test: Example (continued)

- In this class, there are two methods that require testing: getAge() and incrementAge(). We would then test this using the following JUnit test:

```
import static org.junit.Assert.*;
import org.junit.Test;
public class DogTest {
    @Test
    public void testGetAge() {
        Dog rover = new Dog("Rover", 5);
        // Test age is equal to 5
        assertEquals("Age is 5", 5, rover.getAge());
        // Increment age by 1
        rover.incrementAge();
        // Test value has been updated correctly
        assertEquals("Age is now 6", 6, rover.getAge());
    }
}
```

JUnit Test: Example (continued)

- In this example, we are testing that the two methods work correctly using the `assertEquals()` method.
- This method takes an string explanation of the test, the expected result and the method call.
- If the result of the method call matches the expected result, the test passes, otherwise it fails.
- The `@Test` notation is important to identify the method as a test method.
- We can have multiple test methods within a single test class, and it is important we should use multiple test cases to test each method.
- Test the relevant parts of our program. When we test an array it is important to test the array bounds. For a conditional, are all cases covered by at least one test? Are boundary cases covered? There are other methods that can be used for this, including `assertTrue(test)`, which is for testing methods that return boolean (along with the matching `assertFalse()`).
- Note that because of rounding errors a test for equality on the type `double` should be done using `assertEquals(String testName, double expected, double actual, double epsilon)` to test that the two values differ by at most epsilon.
- A full list of the available assert methods can be found here: https://github.com/junit-team/junit/wiki/Assertions
- There is also a very useful guide on using JUnit tests with Eclipse here: https://www.vogella.com/tutorials/JUnit/article.html#installation_junit