

# CS5004, Object-Oriented Design and Analysis

## Lab 3: Javadoc and UML Diagrams. Inheritance, Equality and Exceptions in Java

Therapon Skoteiniotis<sup>1</sup>, Tamara Bonaci, and Abi Evans  
[skotthe@ccs.neu.edu](mailto:skotthe@ccs.neu.edu), [t.bonaci@northeastern.edu](mailto:t.bonaci@northeastern.edu), [ab.evans@northeastern.edu](mailto:ab.evans@northeastern.edu)

**Lab Deadline: Friday February 11<sup>th</sup> 11:59pm.**

### 1. Summary

In today's lab, we will:

- Configure our IntelliJ to use a proper code style
- Practice writing documentation, and generating Javadoc
- Practice deriving UML diagrams for our implemented code
- Practice designing simple classes, and classes that leverage inheritance
- Talk about equality of objects in Java, and methods `equals()` and `hashCode()`

**Note 1:** Labs are intended to help you get started, and give you some practice while the course staff is present and able to provide assistance. You are not required to finish all the questions during the lab, but you are expected to push your lab work to a designated repo on the Khoury GitHub at the end of the lab. .

### 2. Setting up Code Style

In this course, we will be using the Google Java Style guide, defined here:  
(<https://google.github.io/styleguide/javaguide.html>) .

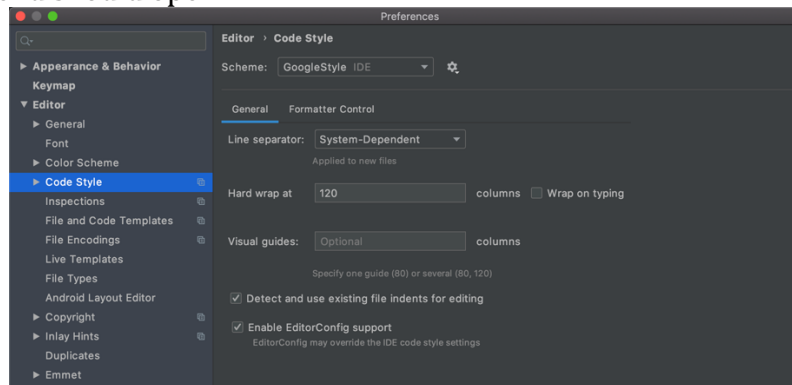
From now on, all code that you write for this course **must** follow this Google Java Style guide.

---

<sup>1</sup> Assignment modified from the original version prepared by Dr. Therapon Skoteiniotis.

1. Please download the saved configuration file for IntelliJ [intellij-java-google-style.xml](https://raw.githubusercontent.com/google/styleguide/gh-pages/intellij-java-google-style.xml) (<https://raw.githubusercontent.com/google/styleguide/gh-pages/intellij-java-google-style.xml>)
  - a. **Open the link in the preceding line in your browser, and save the .xml file on your local machine. Be sure to keep the extension as .xml**
2. In IntelliJ, open `Preferences`
  - a. On Windows window to `File → Settings` on the main menu.
  - b. On Mac go to `IntelliJ → Preferences` on the main menu.

The following menu should open:



3. Select `Editor → Code Style` but **do not expand the item**. In the right-hand side pane at the top you will see a Settings (gear) icon next to the `Scheme` dropdown menu. Click on the icon, select `Import scheme` then `IntelliJIDEA code style XML` to open a pop up file explorer window that allows you to select the file you wish to import.
4. Select the file you just downloaded named `intellij-java-google-style.xml`. This is the file from step 1.

IntelliJ will **try**, and format your code while you type. However, when editing a file at different locations in the file IntelliJ might not indent properly.

To force IntelliJ to re-indent all your code select `Code → Reformat Code` from the main IntelliJ menu. Read the [IntelliJ documentation on code formatting for more options](https://www.jetbrains.com/help/idea/2016.3/reformatting-source-code.html) (<https://www.jetbrains.com/help/idea/2016.3/reformatting-source-code.html>)

## 3. Javadoc

In the last lab, we considered the class `Author`, provided here again for your convenience.

```

public class Author {

    private String name;
    private String email;
    private String address;

    /**
     * Creates a new author given the author's name, email and address as strings. *
     * @param name the author's name
     * @param email the author's email address
     * @param address the authors physical address
     */
    public Author(String name, String email, String address) { this.name = name;
        this.email = email;
        this.address = address;
    }

    /**
     * @return the name */
    public String getName() { return this.name;
    }

    /**
     * @return the email */
    public String getEmail() { return this.email;
    }

    /**
     * @return the address */
    public String getAddress() {
        return this.address; }
    }
}



```

For all the code that we write, we should have also written Javadoc documentation.

**Note 3:** You may want to check the course Canvas how-to page on how to generate Javadoc templates for your code.

Let's now see a the HTML version of your code's documentation.

1. Select your project in the Package Explorer tab.
2. On IntelliJ's main menu select `Tools → Generate Javadoc`. A new pop-up window will appear
3. In the new pop-up window
  - a. Select **"Whole Project"**
  - b. Unselect **"Include test sources"**

- c. Find the line that starts with the text **"Output Directory"**. At the end of that line click the  and specify the output folder that you would like IntelliJ to place all the HTML files that will be generated.
- d. Click . IntelliJ will run Javadoc, and show its progress along with any warnings or errors in the bottom tab of the main IntelliJ window. If the Javadoc generation had no errors then IntelliJ will open your browser and point it to the newly generated documentation.

## 4. Inheritance and “Is a” Relationship

Consider the following class `Athlete`, with code provided below.

```
/*
 * Class Athlete contains information about an athlete, including athlete's name,
 * their height, weight and league.
 */
public class Athlete {

    private Name athletesName;
    private Double height;
    private Double weight;
    private String league;

    /**
     * Constructs a new athlete, based upon all of the provided input parameters.
     * @param athletesName - object Name, containing athlete's first, middle and last
     * name
     * @param height - athlete's height, expressed as a Double in cm (e.g., 6'2'' is
     * recorded as 187.96cm)
     * @param weight - athlete's weigh, expressed as a Double in pounds (e.g. 125, 155,
     * 200 pounds)
     * @param league - athelete's league, expressed as String
     * @return - object Athlete
     */
    public Athlete(Name athletesName, Double height, Double weight, String league) {
        this.athletesName = athletesName;
        this.height = height;
        this.weight = weight;
        this.league = league;
    }

    /**
     * Constructs a new athlete, based upon all of the provided input parameters.
     * @param athletesName - object Name, containing athlete's first, middle and last
     * name
     * @param height - athlete's height, expressed as a Double in cm (e.g., 6'2'' is
     * recorded as 187.96cm)
     * @param weight - athlete's weigh, expressed as a Double in pounds (e.g. 125, 155,
     * 200 pounds)
     * @return - object Athlete, with league field set to null
     */
}
```

```

public Athlete(Name athletesName, Double height, Double weight) {
    this.athletesName = athletesName;
    this.height = height;
    this.weight = weight;
    this.league = null;
}

/*
 * Returns athlete's name as an object Name
 */
public Name getAthletesName() {
    return athletesName;
}

/*
 * Returns athlete's height as a Double
 */
public Double getHeight() {
    return height;
}

/*
 * Returns athlete's weight as a Double
 */
public Double getWeight() {
    return weight;
}

/*
 * Returns athlete's league as a String
 */
public String getLeague() {
    return league;
}
}

```

## Lab Problem 1

1. Create two new classes, `Runner` and `BaseballPlayer`, that inherit states and behavior of the class `Athlete`.

Class `Runner` has the following additional states:

- The best 5K time, expressed as a `Double`
- The best half-marathon time, expressed as a `Double`
- Favorite running event, expressed as a `String`

Class `BaseballPlayer` has the following additional states:

- Team, expressed as a `String`
- Average batting, expressed as a `Double`
- Season home runs, expressed as an `Integer`

2. Test classes `Athlete`, `Runner` and `BaseballPlayer`, by implementing the corresponding tests classes.
3. Generate UML class diagrams for classes `Athlete`, `Runner` and `BaseballPlayer`.
4. Generate Javadoc for classes `Runner` and `BaseballPlayer`.

**Note 4:** You may want to check the course Canvas how-to page on how to generate UML diagrams for your Java projects:

## 5. Equality in Java

Java provides two mechanisms for checking equality between values.

1. `==`, the double equality check is used to check
  - a. **equality between primitive types**
  - b. **"memory equality" check**, i.e. a check whether or not the two references point to the same object in memory
2. `equals()` is a method defined in the class `Object` that is inherited to all classes. The JVM expects developers to override the `equals()` method in order to define the notion of equality between objects of classes that they define.

Overriding the `equals()` method however imposes extra restrictions. These restrictions are spelled out in the `equals()` [method documentation \(link: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-\)](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-), and repeated here for your convenience.

Method `equals()` should be

1. **Reflexive** - for a non-null reference value `x`, `x.equals(x)` returns `true`
2. **Symmetric** - for non-null reference values `x` and `y`, `x.equals(y)` returns `true` if and only if `y.equals(x)` returns `true`
3. **Transitive** - for non-null reference values `x`, `y`, and `z`,
  - a. if `x.equals(y)` returns `true` **and**
  - b. `y.equals(z)` returns `true`, **then**
  - c. `x.equals(z)` **must** return `true`
4. **Consistent** - for non-null references `x`, `y`, multiple invocations of `x.equals(y)` should return the same result provided the data inside `x` and `y` has **not** been altered.
5. For any non-null reference value `x` `x.equals(null)` returns `false`

In in your code, when you decide to override method `equals()`, you **absolutely must** override method `hashCode()` as well, in order to uphold the `hashCode()` method's contract. The contract for `hashCode()` is spelled out in `hashCode()`'s [documentation \(link: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--\)](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--), and repeated here for your convenience.

Here are the conditions for the `hashCode()` method's contract:

1. For a non-null reference value `x`, multiple invocations of `x.hashCode()` must return the same value provided the data inside `x` has not been altered.
2. for any two non-null reference values `x`, `y`
  - a. if `x.equals(y)` returns `true` **then**
  - b. `x.hashCode()` and `y.hashCode()` **must** return the same result
3. for any two non-null reference values `x`, `y`
  - a. if `x.equals(y)` returns `false` **then**
  - b. it is preferred but not required that `x.hashCode()` and `y.hashCode()` return **different/distinct** results.

As you know, your IDE has the ability to automatically generate default implementations for `equals()` and `hashCode()`. The default implementations generated by your IDE are **typically** what you need. However, sometimes we will have to amend/write our own.

**JUnit4** relies on `equals()` and `hashCode()` in your reference types for `Assert.assertEquals()`. The implementation of `Assert.assertEquals()` essentially calls `equals()` on your objects.

## Lab Problem 2

Let's look at an example using class `Posn`:

*Posn.java*

```
/**
 * Represents a Cartesian coordinate.
 *
 */
public class Posn {

    private Integer x;
    private Integer y;

    public Posn(Integer x, Integer y) {
        this.x = x;
        this.y = y;
    }
}
```

```

}

/**
 * Getter for property 'x'.
 *
 * @return Value for property 'x'.
 */
public Integer getX() {
    return this.x;
}

/**
 * Getter for property 'y'.
 *
 * @return Value for property 'y'.
 */
public Integer getY() {
    return this.y;
}

/**
 * {@inheritDoc}
 */
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Posn posn = (Posn) o;
    if (this.x != null ? !this.x.equals(posn.x) : posn.x != null)
        return false;
    return this.y != null ? this.y.equals(posn.y) : posn.y ==
null;

}

/**
 * {@inheritDoc}
 */
@Override
public int hashCode() {
    int result = this.x != null ? this.x.hashCode() : 0;
    result = 31 * result + (this.y != null ? this.y.hashCode() :
0);
    return result;
}

/**
 * {@inheritDoc}
 */

```



```


@Override
public String toString() {
    return "Posn{" +
        "x=" + x +
        ", y=" + y +
        '}';
}
}

```

The strategy used to check for equality is recursive; we check each field in turn, and since a field can be a reference type, we need to call its `equals()` method. There are many different implementations of `equals()`, we will go over this specific one here, so that we have one example of how we could write an `equals()` method.

The strategy used for the implementation of `hashCode()` is also recursive; we use each field and get its hash code, we then compose all field hash codes together along with a prime number to get this object's hash code.

- 1 Observe that the compile-time type for the argument `o` is `Object` not `Posn`.
- 2 We first check whether the argument passed is in fact the same exact object in memory as `this` object using `==`.
- 3 We then check that whether or not the argument `o` is `null`, or that the runtime-type of `o` is **not** the same as the runtime-type of `this`. If either of these conditions is true, then the two values cannot be equal.
- 4 If the runtime-types are the same then we **cast**--force the compile-time type of a variable to change to a new compile-time type-- `o` to be a `Posn`, and give it a new name `posn`.
- 5 Now we check each field for equality, first `x`.  
and then `y`. The `? :` is the Java if-expression; an if statement that returns a value. The test is
- 6 the code before `?`, if that expression returns `true` then we evaluate the expression found between `?` and `:`. If the test expression returns `false` then we evaluate the expression found after the `:`.
- 7 If a field, `x` in this case, is `null` then return `0` else grab its hash code by calling `hashCode()` on that object.

 Repeat to get the hash code for `y`, add the field hash code's together and multiply by 31.

### Your task:

- 1) Create a new test class, `PosnTest`, and implement unit tests for methods `equals()` and `hashCode()`. In doing so, make sure that these tests validate all of the conditions set forth by the contract for `equals()` and `hashCode()`.

**Note 5:** You may want to check the following link to get some ideas no how to test methods `equals()` and `hashCode()` : <https://www.springfuse.com/2009/10/11/testing-various-equals-and-hashcode-strategies.html>