CS5010, Fall 2019

Lab 2: Equality, Exceptions and Testing in Java

Therapon Skoteiniotis[1] and Tamara Bonaci
skotthe@ccs.neu.edu, t.bonaci@northeastern.edu

1. Summary

In today's lab, we will:

- Configure our development environment to use Gradle software management tool
- Practice designing classes that inherit other classes ("is a" relationship)
- Practice designing classes that contain other classes ("has a" relationship)
- Practice writing and running unit tests for classes and methods that we write
- Explore methods `equals()` and `hashCode()`, their contract, and their testing
- Analyze exceptions in Java: throwing and properly handling (catching) exceptions, as well as writing our own exceptions, and testing methods that throw exceptions

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 your designated repo on the Khoury GitHub. Please push package named Lab2 to your individual repo, and once you are done working on the lab, tag it with **Lab2-final**.

The deadline to push your lab work for at least three of the lab problems is by **11:59pm Friday, October 4, 2019.**

2.Submission Requirements

2.1. Gradle Software Management Tool

In this lab assignment, we will again use **Gradle software management tool**. Gradle is a tool that allows us to manage our source code in order to:

- Compile that code
- Compile our test code

[1] Assignment modified from the original version prepared by Dr. Therapon Skoteiniotis.

- Run our test code
- Analyze our source code
- Generate reports about our source code
- Package our software
- Deploy/share our software

Please note that you are expected to use Gradle configuration from this assignment in all subsequent lab and homework assignments.

2.2.1. Gradle and IntelliJ

IntelliJ has a special project type for Gradle projects.

Please follow these steps to setup the project layout for Gradle projects.

1. Start IntelliJ and create a new project.
2. In the selection window, in the left-hand pane, **instead of selecting Java, select Gradle**.
3. Once you select Gradle in the left-hand pane, the right-hand pane populates with additional libraries and frameworks. Make sure to select "**Java**", and click "`Next`".
4. You will see a dialog box, asking you to enter "**GroupId**" and "**ArtifactId**".
    1. **ArtifactID**: enter the project name (such as "Lab2")
    2. Ignore the **GroupId** box.
5. On the next screen, make sure that the selected "Gradle JVM" is Java 1.8, and accept the defaults for everything else.
6. Accept the defaults on the next screen too, and click "`Finish`".

At this point, you should have a Gradle project created, but you are not quite done yet.

The last step is to configure your **`build.gradle`** file so that it does everything we want it to do in this course. The easiest way to do this is to open the "**`build.gradle`**" file in your project directory, and copy/paste the file below into that file (the text with the dark background) (this **`build.gradle`** file is also available on Piazza and the course website for your convenience).

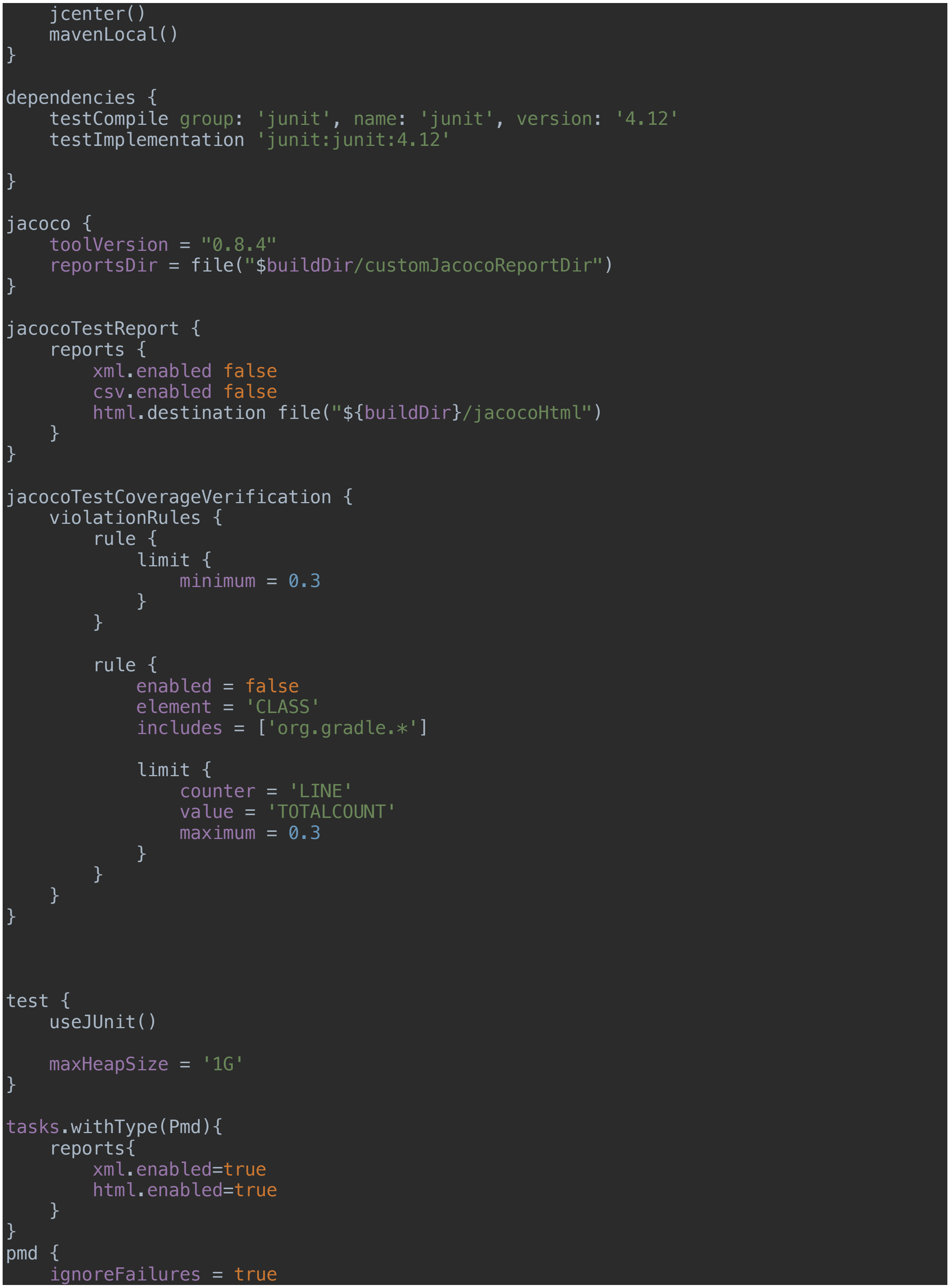
```
plugins {
    id 'java'
    id 'pmd' // PMD: source code analyzer to find common programming flaws
    id 'jacoco' // Code coverage
}

defaultTasks 'clean', 'build', 'javadoc', 'check', 'test'

apply plugin: 'java'

group = 'edu.neu.khoury.pdp'
version = '1.0-SNAPSHOT'
description = 'PDP Fall 2019 Seattle'
sourceCompatibility = '8'
sourceCompatibility = 1.8

repositories {
```



```
    jcenter()
    mavenLocal()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
    testImplementation 'junit:junit:4.12'

}

jacoco {
    toolVersion = "0.8.4"
    reportsDir = file("$buildDir/customJacocoReportDir")
}

jacocoTestReport {
    reports {
        xml.enabled false
        csv.enabled false
        html.destination file("${buildDir}/jacocoHtml")
    }
}

jacocoTestCoverageVerification {
    violationRules {
        rule {
            limit {
                minimum = 0.3
            }
        }

        rule {
            enabled = false
            element = 'CLASS'
            includes = ['org.gradle.*']

            limit {
                counter = 'LINE'
                value = 'TOTALCOUNT'
                maximum = 0.3
            }
        }
    }
}



test {
    useJUnit()

    maxHeapSize = '1G'
}

tasks.withType(Pmd){
    reports{
        xml.enabled=true
        html.enabled=true
    }
}
pmd {
    ignoreFailures = true
    pmdTest.enabled=true
    ruleSets = [
            "category/java/bestpractices.xml",
            "category/java/errorprone.xml",
            "category/java/codestyle.xml"
    ]
}

tasks.withType(JavaCompile) {
    options.encoding = 'UTF-8'
}

task docs(type: Javadoc) {
    source = sourceSets.main.allJava

}

check.dependsOn jacocoTestCoverageVerification
jacocoTestReport.mustRunAfter test

task doAll{
    dependsOn test
    dependsOn check
    dependsOn javadoc
    dependsOn build

    doLast {
        println 'all done!'
    }
}

jacocoTestReport {
    doLast {
        println "file://$buildDir/jacocoHtml/index.html"
    }
}

javadoc {
    doLast {
        println "file://$buildDir/docs/javadoc/index.html"
    }
}

test {
    finalizedBy jacocoTestReport
}
```

2.2. Code Criteria

- **Naming convention:** Your package name should follow this naming convention LabN, where you replace N with the assignment number, e.g., all your code for this assignment must be in a package named Lab2.
- **Gradle built:** Your project should successfully build using the provided **build.gradle** file, and it should generate all the default reports.
- **Javadoc generation:** Your Javadoc generation should complete with no errors or warnings.
- **Checkstyle report:** Your Checkstyle report must have no violations.

- **Code coverage report:** Your JaCoCo report must indicate 70% or more code coverage per package for "Branches" **and** "Instructions".

- **Methods hashCode(), equals(), toString():** all of your classes *have to* provide appropriate implementations for methods:

  - `boolean equals(Object o)`
  - `int hashCode()`
  - `String toString()`

(appropriate means that it is sufficient to autogenerate these methods, as long as autogenerated methods suffice for your specific implementation). Please don't forget to autogenerate your methods in an appropriate order - starting from the ancestor classes, towards the concrete classes.

- **Javadoc:** please include a short description of your class/method, as well as tags from `@params` and `@returns` in your Javadoc documentations (code comments). Additionally, if your method throws an exception, please also include a tag `@throws` to indicate that.

- **UML diagrams:** Please include UML diagrams for the final versions of your designs for every problem. In doing so, please note that you do not have to hand-draw your UML diagrams anymore. Auto-generating them from your code will be sufficient.

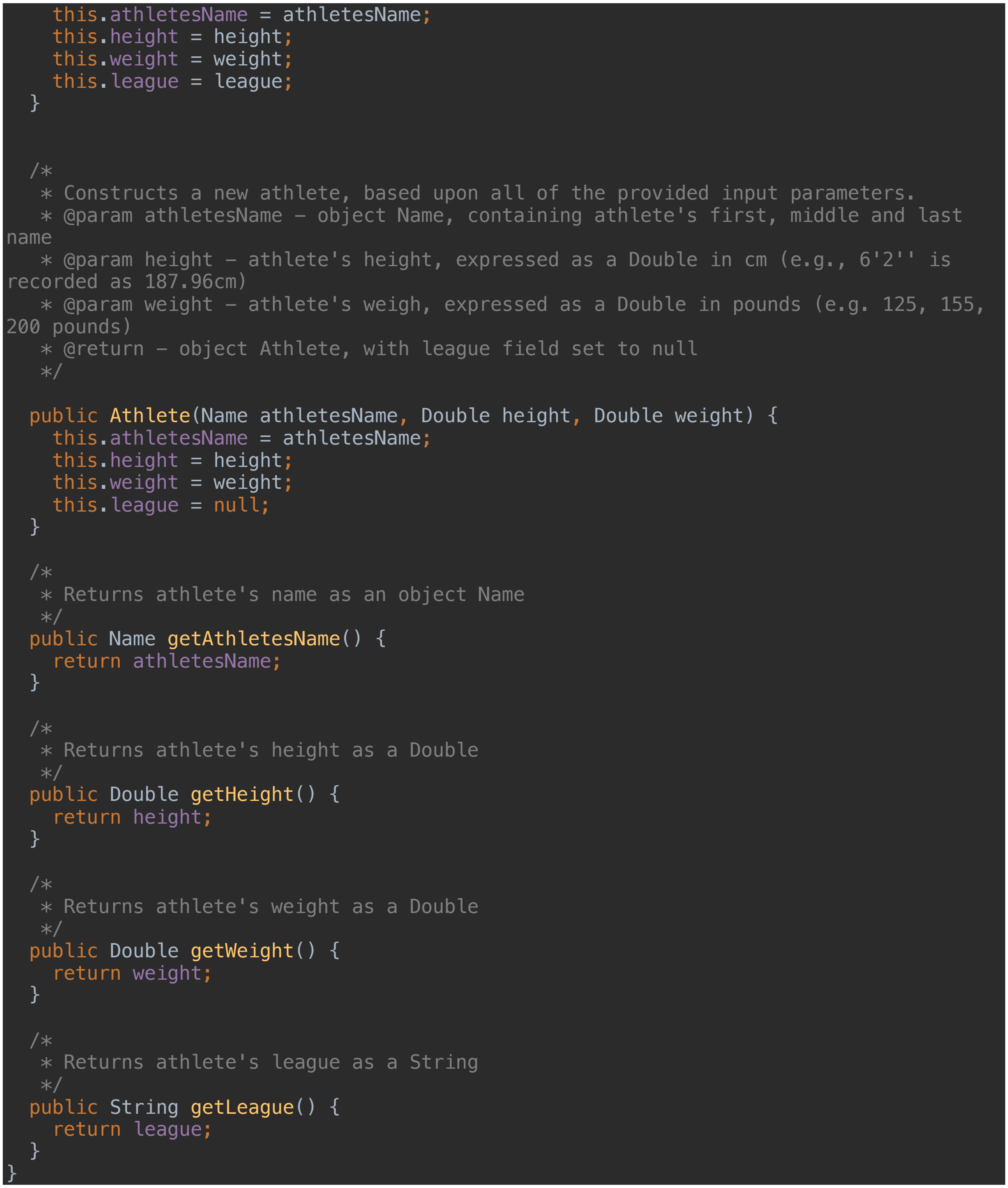
Assignment 1 ("is a" Relationship)

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;
  }
}
```

Your assignments:

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

2. 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
3. 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

4. Test classes Athlete, Runner and BaseballPlayer, by implementing the corresponding tests classes.

5. Generate UML class diagrams for classes Athlete, Runner and BaseballPlayer.

6. Generate Javadoc for classes Runner and BaseballPlayer.

Assignment 2 ("has a" Relationship)

Consider the following class Restaurant, that contains the following information:

- A String restaurant's name
- Class Address - address, where the class Address contains fields:
    - String street and number
    - String city
    - String ZIP code
    - String state
    - String country
- Class Menu - menu, where the class Menu contains fields:
    - A List<String> meals
    - A List<String> desserts
    - A List<String> beverages
    - A List<String> drinks
- A Boolean open/closed

Your assignments:

1. Implement classes Address, Menu and Restaurant, by defining their fields, and providing constructor(s), and getters and setters for it.

2. Test classes Address, Menu and Restaurant, by implementing the corresponding tests in the classes AddressTest, MenuTest and RestaurantTest.

Assignment 3 (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](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, every time when you decide to override method equals(),you **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](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 prefered 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.

Let's look at an example using some 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'.
     */                                                 1
    public Integer getY() {
        return this.y;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public boolean equals(Object o) { 1
        if (this == o) return true;     2
        if (o == null || getClass() != o.getClass()) return false; 3

        Posn posn = (Posn) o; 4
        if (this.x != null ? !this.x.equals(posn.x) : posn.x != null)
return false; 5
        return this.y != null ? this.y.equals(posn.y) : posn.y ==
null; 6

    }


 /**
     * {@inheritDoc}
     */
    @Override
    public int hashCode() {
        int result = this.x != null ? this.x.hashCode() : 0; 7
        result = 31 * result + (this.y != null ? this.y.hashCode() :
0); 8
        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`.

6 and then `y`. The `?:` is the Java if-expression; an if statement that returns a value. The test is 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 `:`.

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`.

8 Your assignments:

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()`.

Assignment 4 (Testing and Exceptions)

You were tasked to build a prototype of a video game. The game designers have an idea, and they would like you to build a program to test their idea out.

The game consists of `Pieces`. A `Piece` can be:

- `Civilian`
- `Soldier`

A `Civilian` is one of:

- `Farmer`
- `Engineer`

A `Soldier` is one of:

- `Sniper`
- `Marine`

The designers provided the following properties

1. All `Pieces` contain information about their:
   - `Name`, containing information about a `Piece’s` first and last name
   - `Age`, which is an Integer in the range [0, 128], containing information a Piece’s age
2. `Civilians` generate wealth. Each `Civilian` must keep track of its wealth, and wealth is a positive real number.
   - We should be able to increase a `Civilian's` wealth by passing a number to add to the current wealth of a `Civilian`.

    - We should be also able to decrease a `Civilian's` wealth by passing a number to remove from the current wealth of a `Civilian`.
3. `Soldiers` keep track of their stamina. Each `Soldier` must keep track of its stamina, and stamina is a real number in the range [0, 100].
    - We should be able to increase a `Soldier's` stamina by passing a number to add to the current stamina of a `Soldier`.
    - We should be able to decrease a `Soldier's` stamina by passing a number to remove from the current stamina of a `Soldier`.

Your assignments:

1. Design a Java program to capture the information and properties described by the game designers.
2. **Exceptions and testing a method that can throw an exception**
    a. Update your `Piece` so that the value provided for a `Piece's` age is never less than 0, or larger than 100. If the provided value for age is outside of the range [0, 128], you should throw a custom-built `IncorrectAgeRangeException` exception.
    b. Write tests to show that your implementation appropriately deals with cases where the values provided for the age work as expected.
3. **Exceptions and testing a method that calls a method that can throw an exception**
    a. Update your `Soldier` so that the value provided for a `Soldier's` strength is always in the range [0, 100]. If the provided strength value is outside of that range, you should throw a custom-built `IncorrectStrengthValueException` exception.
    b. Now also update your methods to increase and decrease a `Soldier's` stamina, in order to properly handle exceptions that can be thrown.
    c. Write tests for methods increasing and decreasing a `Soldier's` stamina, to show that your implementation appropriately deals all possible cases of stamina increase and decrease.

Assignment 5 (Design Problem)

You were tasked with building a prototype of a video game. The game designers have an idea, and they would like you to build a program to test their idea out.

The game consists of `Pieces`. A `Piece` can be:

- `Civilian`
- `Soldier`

A `Civilian` is one of:

- `Farmer`
- `Engineer`

A `Soldier` is one of:

- `Sniper`
- `Marine`

The designers provided the following properties:

4. All `Pieces` contain information about their:
   - o `Name`, containing information about a `Piece’s` first and last name
   - o `Age`, which is an Integer in the range [0, 128], containing information a Piece’s age
5. `Civilians` generate wealth. Each `Civilian` must keep track of their wealth, and wealth is a positive real number.
   - o We should be able to increase a `Civilian's` wealth by passing a number to add to the current wealth of a `Civilian`.
   - o We should be also able to decrease a `Civilian's` wealth by passing a number to remove from the current wealth of a `Civilian`.
6. `Soldiers` keep track of their stamina. Each `Soldier` must keep track of their stamina, and stamina is a real number in the range [0, 100].
   - o We should be able to increase a `Soldier's` stamina by passing a number to add to the current stamina of a `Soldier`.
   - o We should be able to decrease a `Soldier's` stamina by passing a number to remove from the current stamina of a `Soldier`.

Your assignments:

1. Design a Java program to capture the information and properties described by the game designers.
2. Generate the final UML Class Diagram of your solution using IntelliJ, and push it to the package Problem 5, along with your code.
3. Update your `Piece` so that the value provided for a `Piece’s` age is never less than 0, or larger than 100. If the provided value for age is outside of the range [0, 128], you should throw a custom-built `IncorrectAgeRangeException` exception.
4. Write tests to show that your implementation appropriately deals with cases where the values provided for the age work as expected.
5. Update your `Civilian` so that the value provided for a `Civilian’s` wealth is a positive real number. If the provided wealth value is negative, you should throw a custom-built `IncorrectWealthValueException` exception.
6. Write tests to show that your implementation appropriately deals with cases where the values provided for the wealth work as expected.