CS 111, Programming Fundamentals II
Lab 5: Inheritance and javadoc

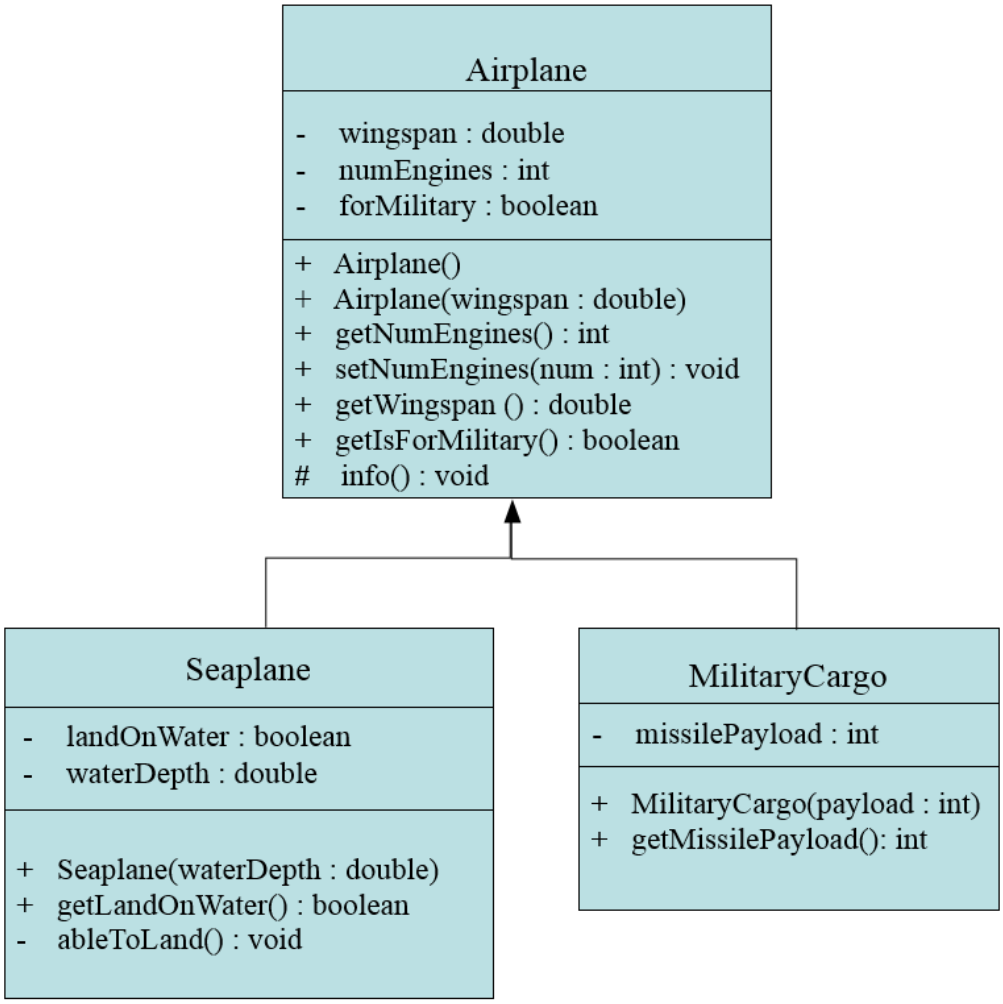
Computer Science

This lab is meant to give you hands-on experience with inheritance and javadoc, a very handy tool available in the Java SDK which generates API style documentation from the code and comments in your Java files. This lab will help you to better understand Project 1.

Introduction

In lecture you were introduced to the concept of inheritance, which will make your life MUCH easier, if you go on to become a developer and work as member of a large development team. Inheritance allows you to "reuse" code, and to rely on a parent or a superclass to provide templates. With inheritance, you don't have to reinvent the wheel each time that you write code.

Assume that you are a developer, and you've been given the following UML diagrams, for a parent class (**Airplane**), and two subclasses (**MilitaryCargo** and **Seaplane**) that both extend the **Airplane** class. The composite UML diagram can be seen below. It indicates that **MilitaryCargo** and **Seaplane** are derived from **Airplane** class.



I. javadoc

You accessed the API in the last lab as you searched for the ***reverse()*** method of the **StringBuilder** class. Java's online API is very nicely structured and organized into a section that clearly designates the fields, a section that designates the constructor, and a section that designates the methods for a class. Using the javadoc tool, you can make similarly nicely formatted API style documentation for your own Java files. In this part of the lab, you'll generate the documentation for the **Airplane** and **MilitaryCargo** classes that you've been given, and add comments to **Seaplane.java** and build its documentation.

Javadocs relies on properly formatted comments that accompany fields and methods to generate an HTML documentation page for a java file (**See Figure 2**). Thus, in order to have javadocs generate documentation, you need to adhere to the following commenting conventions (which are an industry standard):

- For a method or a field, include in comments before the method or field, a short description (first single sentence) and a long description (which is optional).

- For a method, after the comments, include one of several available tags to provide javadoc a description of the parameters and return value of the method. The most commonly used tags:

    1. **@param**, which indicates the parameter(s) of the method
    2. **@return**, which indicates what the method returns

A sample method, property documented using the javadoc standard, is shown in Figure 1. **Notice that the beginning multiple line comment must begin with /** alone on a line**. The "**/****" is specific for javadocs.

```
/**
*The constructor.
*
*Takes as input a single argument, of type double.
*@param wingspan to set the wingspan of the airplane
*/
public Airplane(double wingspan){
   this.wingspan = wingspan;
}


/**
*This getter method retrieves the value of the numEngines
*instance field.
*@return number of engines on the airplane
*/


public int getNumEngines(){
   return numEngines;
}
```

Figure 1 : A sample, properly documented methods, with a short description (single, first sentence), a longer description (the remaining sentences after the short description), and which uses the @param and @return tags to describe the parameters and return type of the method.

To complete this part of the lab:

1. Download the **Airplane.java, MilitaryCargo.java**, and **Seaplane.java** files from Canvas.

2. The **Airplane.java** and **MilitaryCargo.java** files are already property documented. To generate the javadoc documentation for them, open **Airplane.java** in jGRASP, and select **File → GenerateDocumentation**. JGRASP will generate the HTML documentation page, which when viewed, should like that in **Figure 2**.

3. Take a look at the generated API documentation page for **MilitaryCargo**. Notice how javadoc has used the comments in the **MilitaryCargo.java** file to make the well-formatted documentation page. Not only does javadoc inform you the details about the fields and methods, but it also indicates that **MilitaryCargo** inherits from the **Airplane** class.

4. Also use javadoc to generate the documentation for **Airplane.java**.

5. Open the **Seaplane.java** file in jGRASP. It is wholly not documented. Use proper javadoc commenting standards to document that file, and generate the documentation using jGRASP.

For more information about javadoc, please refer to the following link: https://en.wikipedia.org/wiki/Javadoc .

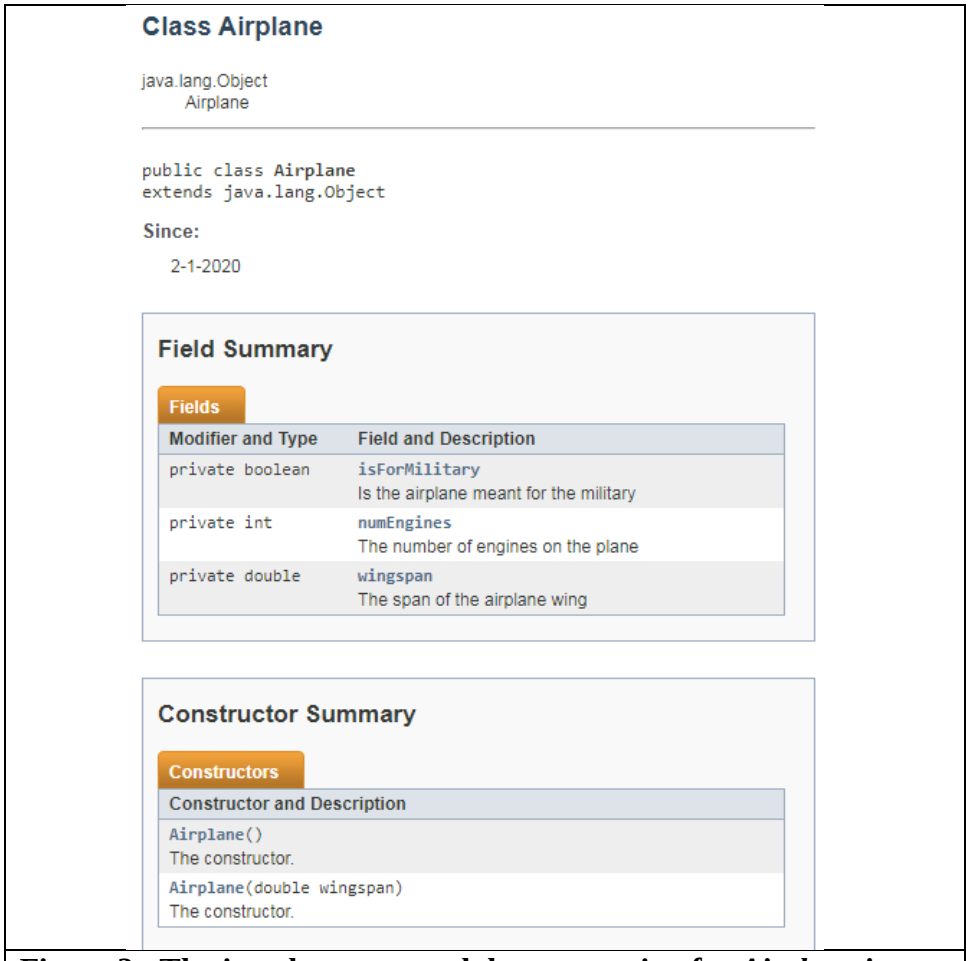


Figure 2 : The javadoc generated documentation for *Airplane.java.* Remaining portions of API documentation not shown.

II. Inheritance

Explore the UML diagram one more time to get familiar with the classes and methods. Write a new java class named **Airport**, it should only have the main method. Inside of the main method do the following:

- Create an object of type **MilitaryCargo** with the reference variable ***airplane1***. The constructor of the **MilitaryCargo** class requires an int value for the field ***missilePayload***, it should be 15. Set the number of engines for ***airplane1*** to be 8.

- Create an object of type **Seaplane** with the reference variable ***airplane2***. The constructor of the **Seaplane** class requires a double value for the field ***waterDepth***, it should be 30.0. Set the number of engines for ***airplane2*** to be 2.

- Create an object of type **Airplane** with the reference variable ***airplane3***. Use the constructor of the **Airplane** class which requires a double value for the field ***wingspan***, it should be 100.0. Set the number of engines for ***airplane3*** to be 4.

- Create an array of type **Airplane**, named ***airplanes.*** Place ***airplane1***, ***airplane2***, and ***airplane3*** into the ***airplanes*** array.

- Using a for loop print out the reference to each airplane, and how many engines it has. The for loop should go through the ***airplanes*** array. Use the getter method to retrieve the value of the field ***numEngines***.

- Check if ***airplane2*** is able to land on water by calling the method ***getLandOnWater().*** Print out if the plane can land or if it should look for a spot where the water is deeper.

Compile and run your program. Sample output can be seen below in **Figure 3**.

```
 ----jGRASP exec: java Airport
MilitaryCargo@2b1be57f number of engines : 8
Seaplane@34780af5 number of engines : 2
Airplane@351775bc number of engines : 4
Seaplane is able to land.

 ----jGRASP: operation complete.
```

Figure 3: Sample Output

III. Upload your work to Canvas

For this lab, make sure that you upload the following files to the Lab 5 assignment in your Canvas account:

Seaplane.java
Airport.java
Screenshot image containing the output

Rubric

File / Lab	Points
Class *Seaplane* has been properly documented using the style that is required for javadoc	35
Class *Airport* has the main method which creates 3 objects (MilitaryCargo, Seaplane, and Airplane). Number of engines is set for each object. An airplane array is created and populated correctly. A for loop is used to print out number of engines for each element in the array. Code written to check if the object of type Seaplane can land.	50
Variables names are descriptive, and code is indented, good names and commented properly so that it is easy to read	10
Screenshot of the output	5
Total	100