CST-239 Activity 1 Guide

Contents

Part 1: Tools Installation and Validation

Overview

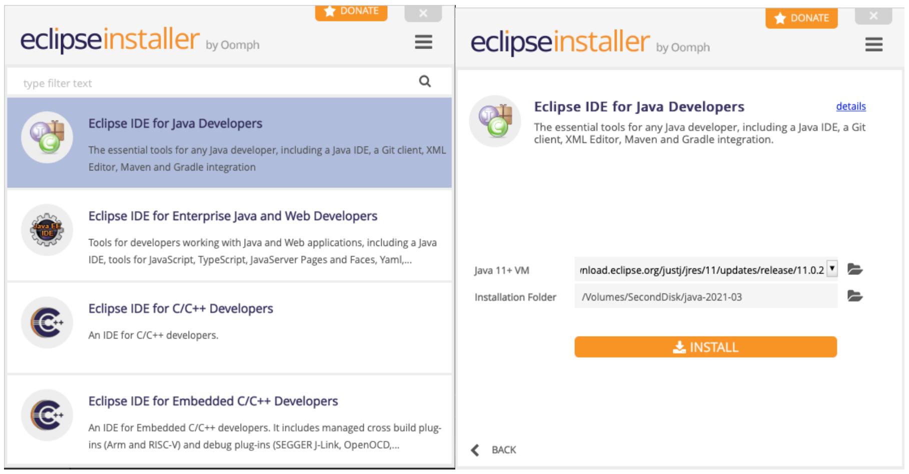
Goal and Directions:

In this activity, you will install the latest version of Eclipse for Java and validate your locale environment by developing a simple "Hello World" Java class in Eclipse. Note, these activities are written assuming the use of the Eclipse for Java IDE. With approval from your instructor, you are free to explore the use of the Microsoft Visual Studio Code IDE in this course as long as you are comfortable with finding alternative steps to complete the activities. See Appendix A for developer notes if you will be using the Microsoft Visual Studio Code IDE. Complete the following tasks for this activity:

Execution

Execute this assignment according to the following guidelines:

1. Install the latest version of Eclipse for Java by going to https://www.eclipse.org and clicking the Download button. This will download the Eclipse Installer.
2. Run the Eclipse Installer as shown below, click the Eclipse IDE for Java Developers option, select the version from the Java VM dropdown list as suggested by your instructor, browse to a folder where you want the Eclipse IDE installed, and finally click the Install button.



Eclipse IDE for Java Developers
The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Maven and Gradle integration

Eclipse IDE for Enterprise Java and Web Developers
Tools for developers working with Java and Web applications, including a Java IDE, tools for JavaScript, TypeScript, JavaServer Pages and Faces, Yaml,...

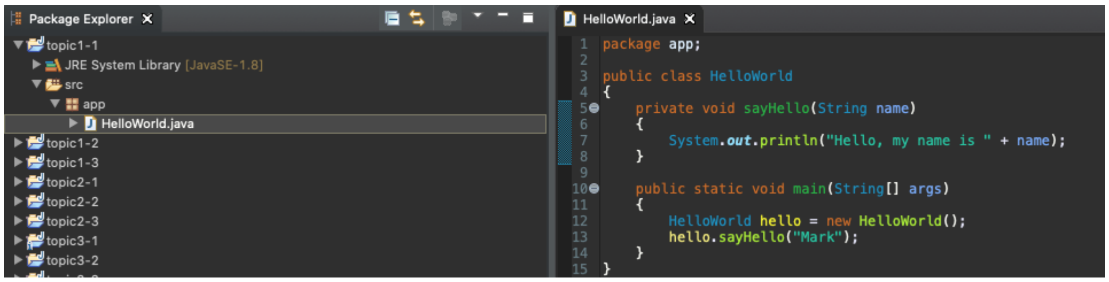
Eclipse IDE for C/C++ Developers
An IDE for C/C++ developers.

Eclipse IDE for Embedded C/C++ Developers
An IDE for Embedded C/C++ developers. It includes managed cross build plug-ins (Arm and RISC-V) and debug plug-ins (SEGGER J-Link, OpenOCD,...

Java 11+ VM: nload.eclipse.org/justj/jres/11/updates/release/11.0.2
Installation Folder: /Volumes/SecondDisk/java-2021-03

3. After the installation is complete start the Eclipse IDE. Take a screenshot of the About Box of Eclipse for Java.
4. Set up Java 11 in your Eclipse IDE by going to your Eclipse Preferences and drilling into the Java → Compiler menu items and selecting Java 11 from the Java Compliance Level. Then select the Java → Installed JREs → Execution menu items, select Java 11, and select the desired JRE from the Compatible JREs list.

   NOTE: these Activities were designed and have been tested with Java 11 and it is not recommended to not use any other versions than Java 11 for this course unless approved your instructor.
5. Set up your Java Coding Style, which can be set up by going to your Eclipse Preferences and drilling into the Java → Code Style → Formatter menu items. You can export this setup and share with your team, so everybody is coding using the same conventions.
6. Create a simple "Hello World" class using the Eclipse IDE:
   a. Start Eclipse.
   b. Create a new Eclipse Workspace name *workspaceCST-239*, which will be used for all activities. You should make sure you back up your workspace!
   c. Create a new Java Project using the File → New → Java Project menu items and name your project *topic1-1*. Also validate that your project is configured to use the JavaSE-11 JRE. Click the Finish button.
   d. Right click on the project folder within the Project and select the New → Class menu options. Name your class *HelloWorld* and put in a package named *app*. Click the *public static void main*() method creation option. Click Finish.
   e. Add a private method named *sayHello()* that returns void and takes a *String* as a method argument named *name*. In the implementation, print a string *Hello my name is* and concatenate the *name* method argument to the string. Print the string to the system console.



```java
package app;

public class HelloWorld
{
    private void sayHello(String name)
    {
        System.out.println("Hello, my name is " + name);
    }

    public static void main(String[] args)
    {
        HelloWorld hello = new HelloWorld();
        hello.sayHello("Mark");
    }
}
```

f. In the *main()* method, create an instance of the *HelloWorld* class and call the *sayHello()* method.
g. Run the HelloWorld class.
h. Take a screenshot of the console when executing the HelloWorld class.

<u>Deliverables</u>:

The following need to be submitted as this part of the activity:

a. Screenshot of the Eclipse About Box.
b. Screenshot of console output when running the HelloWorld class.

GRAND CANYON UNIVERSITY

Part 2: Designing, Coding, and Documenting a Person Class

Overview

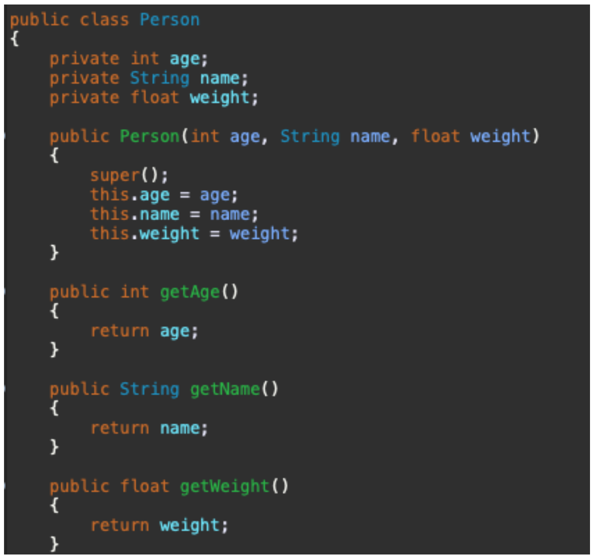
Goal and Directions:

In this activity, you will design and implement a Person class based on a UML class model. Complete the following tasks for this activity:

Execution

Execute this assignment according to the following guidelines:

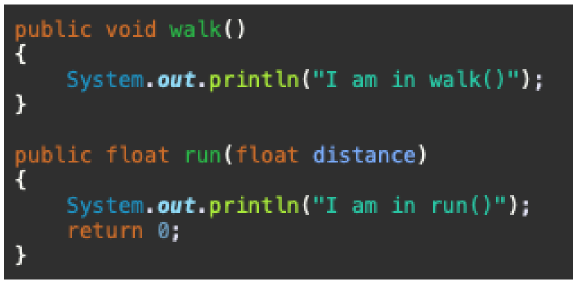
1. Create a UML diagram for a Person class by using any number of UML drawing tools that include, but are not limited to: Microsoft Word, Microsoft Visio, and Draw.io. Create a variety of properties and methods that describe what you want the person to be able to be and do. Students who are taking this course on campus should do an in-class activity with the whiteboard to model a Person class.
2. Create a new Java Project using the File → New → Java Project menu items and name your project *topic1-2*. Also validate that your project is configured to use the JavaSE-11 JRE. Click the Finish button.
3. Right click on the project folder within the Project and select the New → Class menu options. Name your class *Person* and put in a package named *app*. Click the *public static void main*() method creation option. Click Finish.
    a. Add the state variables that were modeled as class private member variables.
    b. Create a non-default constructor that initializes your Person state variables.
    c. Create a getter and setter method for all of the state variables using Eclipse Refactoring (use the Source → Generate Getter and Setters…. menu options).



```java
public class Person
{
    private int age;
    private String name;
    private float weight;

    public Person(int age, String name, float weight)
    {
        super();
        this.age = age;
        this.name = name;
        this.weight = weight;
    }

    public int getAge()
    {
        return age;
    }

    public String getName()
    {
        return name;
    }

    public float getWeight()
    {
        return weight;
    }
```

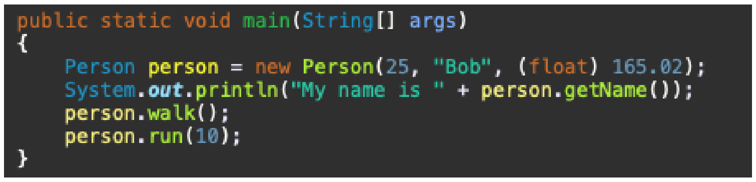
d. Add the behavior methods that were modeled as public class methods.

e. Add console print statements to each method that displays appropriate testing messages.



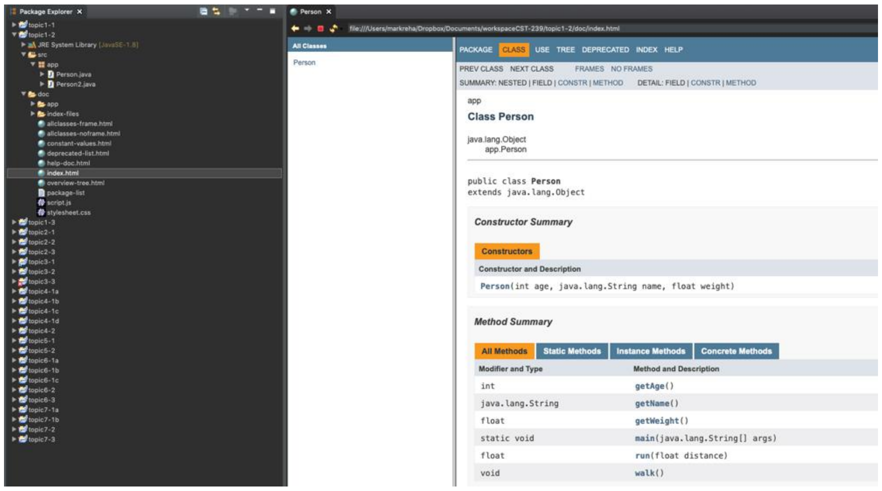
```java
public void walk()
{
    System.out.println("I am in walk()");
}

public float run(float distance)
{
    System.out.println("I am in run()");
    return 0;
}
```

f. In the *main()* method, create an instance of the *Person* class and call each of the public methods.



```java
public static void main(String[] args)
{
    Person person = new Person(25, "Bob", (float) 165.02);
    System.out.println("My name is " + person.getName());
    person.walk();
    person.run(10);
}
```

4. Run the Person class.
5. Take a screenshot of the console when executing the Person class.
6. Provide a brief (3- to 4-sentence) description of how and why the output was displayed.
7. Using JavaDoc conventions using Eclipse (enter /** [return] above each class method), document your class and all methods. Generate the JavaDoc using Eclipse with following steps:
    a. Select the Project → Generate Javadoc… menu options.
    b. Select your project.
    c. Use the standard doclet.
    d. Set the location where you want the JavaDoc generated. The default location is the doc folder inside of the project.
    e. Click the Generate button.
    f. Validate the documentation by opening the *index.html* generated by JavaDoc.



file:///Users/markreha/Dropbox/Documents/workspaceCST-239/topic1-2/doc/index.html

app

Class Person

java.lang.Object
    app.Person

```
public class Person
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor and Description
`Person(int age, java.lang.String name, float weight)`

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`int`	`getAge()`
`java.lang.String`	`getName()`
`float`	`getWeight()`
`static void`	`main(java.lang.String[] args)`
`float`	`run(float distance)`
`void`	`walk()`

8. Create a UML class diagram of your Person class.

<u>Deliverables</u>:
The following need to be submitted as this part of the activity:

a. ✓ Theory of operation write-ups.
b. UML class diagram of the Person class.
c. Screenshot of the console output when running the Person class.
d. Submit a zip file of your source code, including the generated Javadoc files.

GRAND CANYON UNIVERSITY™

Part 3: Designing, Coding, and Documenting a Race Car Class

Overview

Goal and Directions:

In this activity, you will be given a set of functional requirements for a Race Car game. You will then design/model and implement the game. The following are requirements to complete this activity:

Execution

Execute this assignment according to the following guidelines:

Problem Statement
We need to design a Racing Car that can be used in a game, and the Car must be able to be started, run, and stopped.

Requirements

1. The Car must have an Engine and Tires.
2. The Car must have 4 Tires.
3. The Car must have 1 Engine.
4. The Engine must be able to start and stop.
5. Before the Engine can be started, all Tires must have at least 32 psi.
6. The Car must be able to start, run between 1 and 60 mph, stop, and also be restarted.

Modeling

- You must do object-oriented analysis for the requirements.
- What Objects will you need (hint, pick the nouns out of the requirements)?
- Make sure each Object has the proper design:
  - States (nouns from requirements)
  - Behaviors (verbs from requirements)
  - Relationships:
    - If Object A has a relationship to (or contains) Object B, then it must be included as a member variable in Object A.
- **Create a UML class diagram** of your complete model. Use a tool such as Draw.io or lucidchart.com if you wish. LucidChart.com has a good video tutorial for how to draw UML diagrams on their website, as well as on YouTube: https://www.youtube.com/watch?v=UI6lqHOVHic

GRAND CANYON UNIVERSITY™

Implementation

- ✓ Create a new Java Project using the File → New → Java Project menu items and name your project *topic1-3*. Put all classes into appropriate packages (common car related classes into a car package, derived car classes into a racecars package, and the game class under a game package). Also validate that your project is configured to use the JavaSE-11 JRE. Click the Finish button.
- ✓ You need to code the logic and rules to properly start, run, stop, and restart the Car.
- ✓ You need to code all the classes per your models.
- ✓ You must be able to exercise your Car from another "driver" script.
- ✓ Take a screenshot demonstrating that you can start, drive, and stop your Car.

<u>Deliverables</u>:
The following need to be submitted as this part of the activity:

a. ✓ UML class diagram of your complete model.
b. ✓ Screenshots demonstrating that you can start, drive, and stop your Car.
c. ✓ Submit a zip file of your source code, including the generated Javadoc files.

Part 4: Using the Debugger

Overview

Goal and Directions:

In this activity, you will be given some tutorials and various debugger operations, such as setting breakpoints, inspecting variables, stepping thru code, and inspecting the call stack, that you will practice using code developed in this activity. The following are requirements to complete this activity:

Execution

Execute this assignment according to the following guidelines:

1. Read the Eclipse Debugger tutorial at http://www.vogella.com/tutorials/EclipseDebugging/article.html. Open the Debug Perspective in Eclipse. Inspect each of the Debugger Icons by hovering your mouse over each of the icons, as well as the Debug menu items located within the Run menu.
2. Open the project from *topic1-2* project.

    1. Setting Breakpoints:
        a. Open up the *Person* class.
        b. Set a breakpoint on one of the statements in the *main()* method and another breakpoint on one of the classes methods.
        c. Run the application in debug mode by clicking the Debug icon from the Eclipse toolbar.
        d. Take a screenshot of the first breakpoint stopped in the debugger.
        e. Click the Resume icon from the Eclipse toolbar.
        f. Take a screenshot of the second breakpoint stopped in the debugger.
        g. Remove all breakpoints.
        h. Click the Resume icon from the Eclipse toolbar.
    2. Inspecting Variables:
        a. Remove the breakpoints that were previously set.
        b. Set a new breakpoint on a different method in the class.
        c. Run the application in debug mode by clicking the Debug icon from the Eclipse toolbar.
        d. Inspect all of the class member variables in the Variables Window.
        e. Take a screenshot of the Variables Window in the debugger.
        f. Remove all breakpoints.
        g. Click the Resume icon from the Eclipse toolbar.
    3. Stepping Into a Function, Step Over Function, and Step Return from a function:
        a. Set a breakpoint on one of the statements in the *main()* that calls a method in the *Person* class.

    b. Run the application in debug mode by clicking the Debug icon from the Eclipse toolbar.
    c. Verify that code stopped at the breakpoint.
    d. Click the Step Into icon from the Eclipse toolbar to step into the method.
    e. Verify that you are in line 1 of the method.
    f. Click the Step Over icon from the Eclipse toolbar and inspect all local variables during each step. Continue clicking the Step Over icon until you have reached the last line of the method.
    g. Click the Step Return icon from the Eclipse toolbar.
    h. Verify that you are in the method callback in the *main*() method.
    i. Take a screenshot of the debugger.
    j. Click the Resume icon from the Eclipse toolbar.
    k. Remove all breakpoints.

4. Inspecting the Call Stack:
    a. Set a breakpoint on one of the statements in the *main()* that calls a method in the *Person* class.
    b. Run the application in debug mode by clicking the Debug icon from the Eclipse toolbar.
    c. Verify that code stopped at the breakpoint.
    d. Click the Step Into icon from the Eclipse toolbar to step into the method.
    e. Verify that you are in line 1 of the method.
    f. Click the Step Over icon from the Eclipse toolbar and inspect all local variables during each step. Continue clicking the Step Over icon until you have reached the last line of the method.
    g. Inspect the Debug (Call Stack) Window.
    h. Take a screenshot of the Debug (Call Stack) Window.
    i. Remove all breakpoints.
    j. Click the Resume icon from the Eclipse toolbar.

<u>Deliverables</u>:
The following need to be submitted as this part of the activity:

a. Screenshot from the Setting Breakpoints task.
b. Screenshots from the Inspecting Variables task.
c. Screenshots from the Stepping task.
d. Screenshot from the Inspecting Call Stack task.

Research Questions

1. Research Questions: For traditional ground students, in a Microsoft Word document, answer the following questions:
    a. Your text discusses a relationship between classes called "association." Think about the class project you are currently designing. What associations exist between the Salable and the Shopping Cart class? What is the multiplicity of these associations?
    b. What is meant by the statement "class abstraction is the separation of class implementation from the use of a class"? Illustrate your answer with a Java class.

Final Activity Submission

1. In a Microsoft Word document, complete the following for the Activity Report:
    a. ✓ Cover sheet with the name of this assignment, date, and your name.
    b. ✓ Section with a title that contains all the diagrams, screenshots, and theory of operation write-ups.
    c. ✓ Zip file with all code and generated JavaDoc documentation files.
    d. Section with a title that contains the answers to the Research Questions (traditional ground students only).
2. Submit the Activity Report and zip file of the code and documentation to the Learning Management System (LMS).

Appendix A: Using the Microsoft Visual Studio Code IDE

The Microsoft Visual Studio Code IDE has adequate support for programming in Java. The following are the recommended extensions to be installed to support programming in Java for this IDE.

1) Java Extension Pack

The Microsoft Visual Studio Code IDE does not have the exact same concept of Workspaces like you would find in the Eclipse IDE. However, to mimic the notion of projects in your Workspace, simply create a top-level directory where you want all your projects to be held. Then use the *Add Folder to Workspace* menu option and then save your Workspace by selecting the *Save Workspace As* menu option. You can then create project folders underneath the top-level directory. Any number of projects can be placed in the Workspace. You can even copy one project to another by using the *Copy* and *Paste* menu options on the project directory while working in the IDE.

It should be noted that the Microsoft Visual Studio Code IDE does not have all the convenient wizards for creating Java packages, Java classes, Java interfaces, etc. that you will find in the Eclipse IDE. You will need to look up the alternative menu options to perform many of these common tasks. If you are not comfortable completing these tasks without the wizards, then it is highly recommended that you complete this course using Eclipse IDE.

Until further notice, Java 8 is the only approved version of Java that is used in all the Java courses.