Lab 4b – GitHub Copilot

Contents

[1 Introduction 1](#_Toc174447585)

[2 Copilot4Eclipse Introduction 2](#_Toc174447586)

[3 Setup Copilot & Copilot4Eclipse 2](#_Toc174447587)

[4 Initialize Java Project 2](#_Toc174447588)

[5 Chat – Room Example 3](#_Toc174447589)

[6 Chat – Formal Testing 10](#_Toc174447590)

[7 Chat – OCSF Example 11](#_Toc174447591)

[8 Chat – To Learn Something 12](#_Toc174447592)

[9 Role of AI Assistants in CS 1302 Policy 13](#_Toc174447593)

[10 Submission Requirements 13](#_Toc174447594)

[Appendix 1 Gain Access to GitHub Copilot 14](#_Toc174447595)

To make this document easier to read, it is recommended that you turn off spell checking and grammar checking in Word:

1. Choose: File, Option, Proofing
2. At the very bottom, check: “Hide spelling errors…” and “Hide grammar errors…”

# Introduction

*(Read, no action required)*

[*GitHub Copilot*](https://github.com/features/copilot)is an AI tool that can be used to write, suggest, interpret, and test code. It lives at GitHub. Once you have a GitHub account, you can sign up for Copilot. As a student, it will be free, but you do have to go through a verification process. There will be instructions later for this.

“GitHub Copilot is an AI [pair programmer](https://en.wikipedia.org/wiki/Pair_programming) that offers autocomplete-style suggestions as you code. You can receive suggestions from GitHub Copilot either by starting to write the code you want to use, or by writing a natural language comment describing what you want the code to do. GitHub Copilot analyzes the context in the file you are editing, as well as related files, and offers suggestions from within your text editor…

GitHub Copilot is trained on all languages that appear in public repositories. For each language, the quality of suggestions you receive may depend on the volume and diversity of training data for that language. For example, JavaScript is well-represented in public repositories and is one of GitHub Copilot's best supported languages.”[[1]](#footnote-1)

The objective of this lab is to learn to use Copilot. You can use it on GitHub directly for a few things, but only with a Pro (paid) license. However, you can use it through an IDE. In this lab, you will use the [*Copilot4Eclipse*](https://www.genuitec.com/products/copilot4eclipse/)(Eclipse) plugin to access GitHub Copilot to generate code, *etc.*

**NOTE: The steps below may produce results that don’t match my screen shots (it is AI, after all!). Don’t sweat that. Just do your best to try to fulfill the *spirit* of the lab. And, feel free to experiment as much as you want. What you turn in does not need to match mine at all! Learn!**

# Copilot4Eclipse Introduction

*(Read, no action required, 20 min)*

*Copilot4Eclipse* is a plugin for Eclipse to use GitHub’s Copilot AI tool. In this section, you will read a few pages and watch a few short videos to get an overview of the features and how to use them in Copilot4Eclipse (C4E). Do the following:

1. Watch 2-minute [Copilot4Eclipse video](https://www.youtube.com/watch?v=q2dhnulEYFk) which overviews: installation, authorization, code suggestions, and customization/preferences. It does not discuss *Chat* which will be in another video. It was helpful for me to slow the video speed, something I don’t think I’ve done before.
2. Scan this overview of the [Copilot4Eclipse UI](https://www.genuitec.com/products/copilot4eclipse/docs/basics/overview/). You can stop at Section 2.1, you will see Menus later.
3. Scan this overview of [Inline Code Completions](https://www.genuitec.com/products/copilot4eclipse/docs/basics/completions/). Read these [tips](https://www.genuitec.com/products/copilot4eclipse/docs/basics/completion-tips) for generating better code suggestions.
4. Scan this overview of [Chat](https://www.genuitec.com/products/copilot4eclipse/docs/basics/chat/) and be sure and watch the video there.

(optional resource) [Best practices for using GitHub Copilot in VS Code](https://code.visualstudio.com/docs/copilot/prompt-crafting) – This is for the VS Code plugin, but it has a Chat feature that is virtually the same as Copilot4Eclipse. These are really good suggestions.

# Setup Copilot & Copilot4Eclipse

Follow these steps to access GitHub Copilot, Copilot4Eclipse, set them up, and connect them.

1. Obtain access to GitHub Copilot. **YOU PROBABLY ALREADY DID THIS STEP IN LAB 1, Section 2. If so, advance to number 2 below.**
2. Do the following to install the Copilot4Eclipse Plugin:
3. Verify that your computer meets the Copilot4Eclipse [prerequisites](https://www.genuitec.com/products/copilot4eclipse/docs/prereqs/prerequisites).
4. Follow these steps to [install](https://www.genuitec.com/products/copilot4eclipse/docs/installation/). Note that on the last step (4), the image shows the Copilot menu, which at the top says, “Status: Copilot4Eclipse ready”. Yours will **not** say that. Yours will say, “Status: GitHub Copilot4Eclipse sign in required”. After you sign in, it will say, “…ready”.
5. Follow these steps to have Copilot4Eclipse [sign in](https://www.genuitec.com/products/copilot4eclipse/docs/basics/signin) to GitHub Copilot.

# Initialize Java Project

Do the following in Eclipse:

1. Create a project named: *labAILastname* by doing the following:
2. Choose: File, New, Java Project
3. Supply the Project name: *labAILastName*
4. **Important:** Uncheck: Create module-info.java file.
5. Choose: Finish
6. Create a package named: *prob1* by doing the following:
7. Right-click the *src* node in the Project Explorer and choose: New, Package
8. Supply the Name: *prob1*
9. Create a class named: *Room* by doing the following:
10. Right-click the *prob1* node in the Project Explorer and choose: New, Class
11. Supply the Name: *Room*
12. The code should look like this (I’ve removed the line breaks for brevity).:

**package** prob1;

**public** **class** Room {

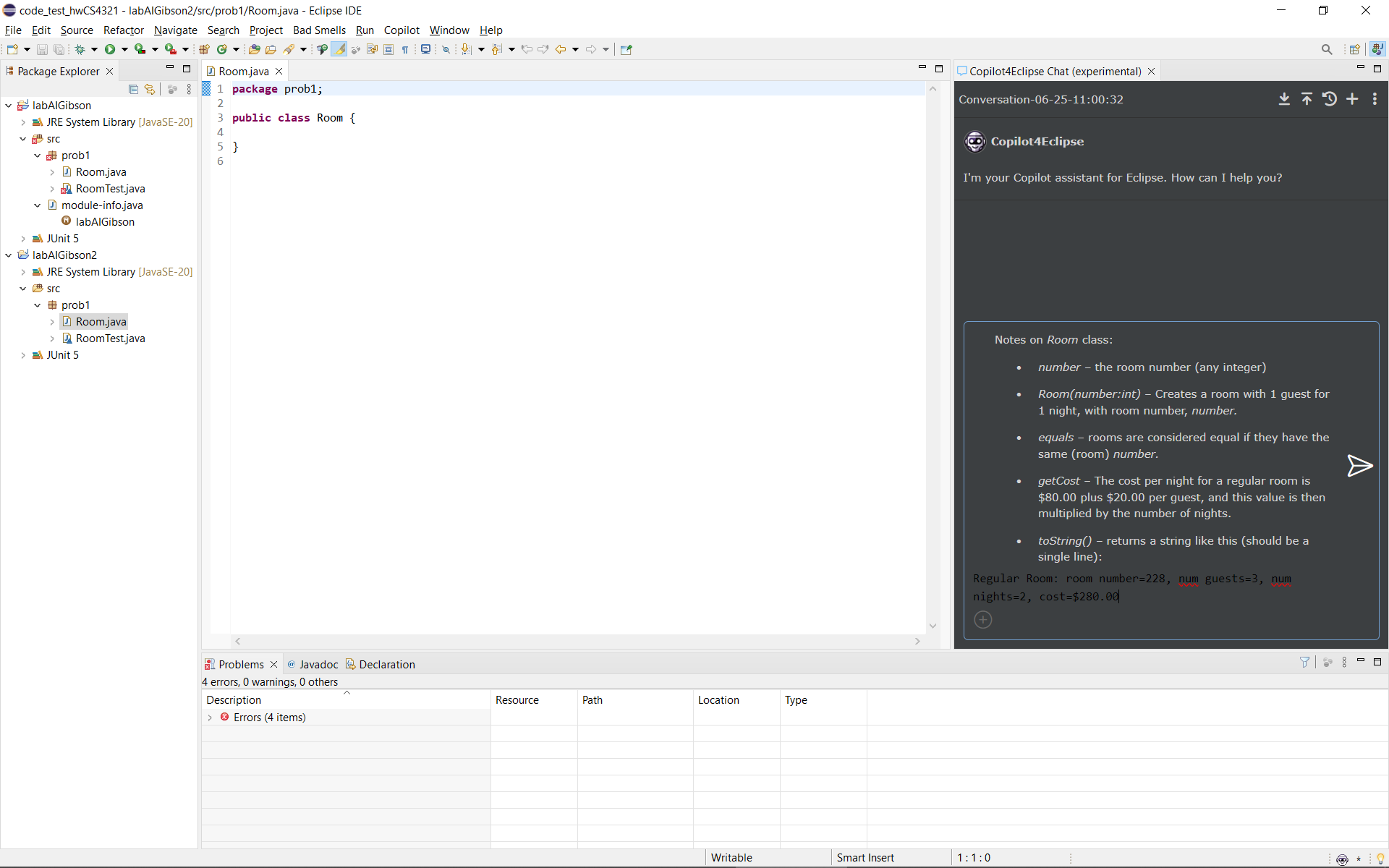
}

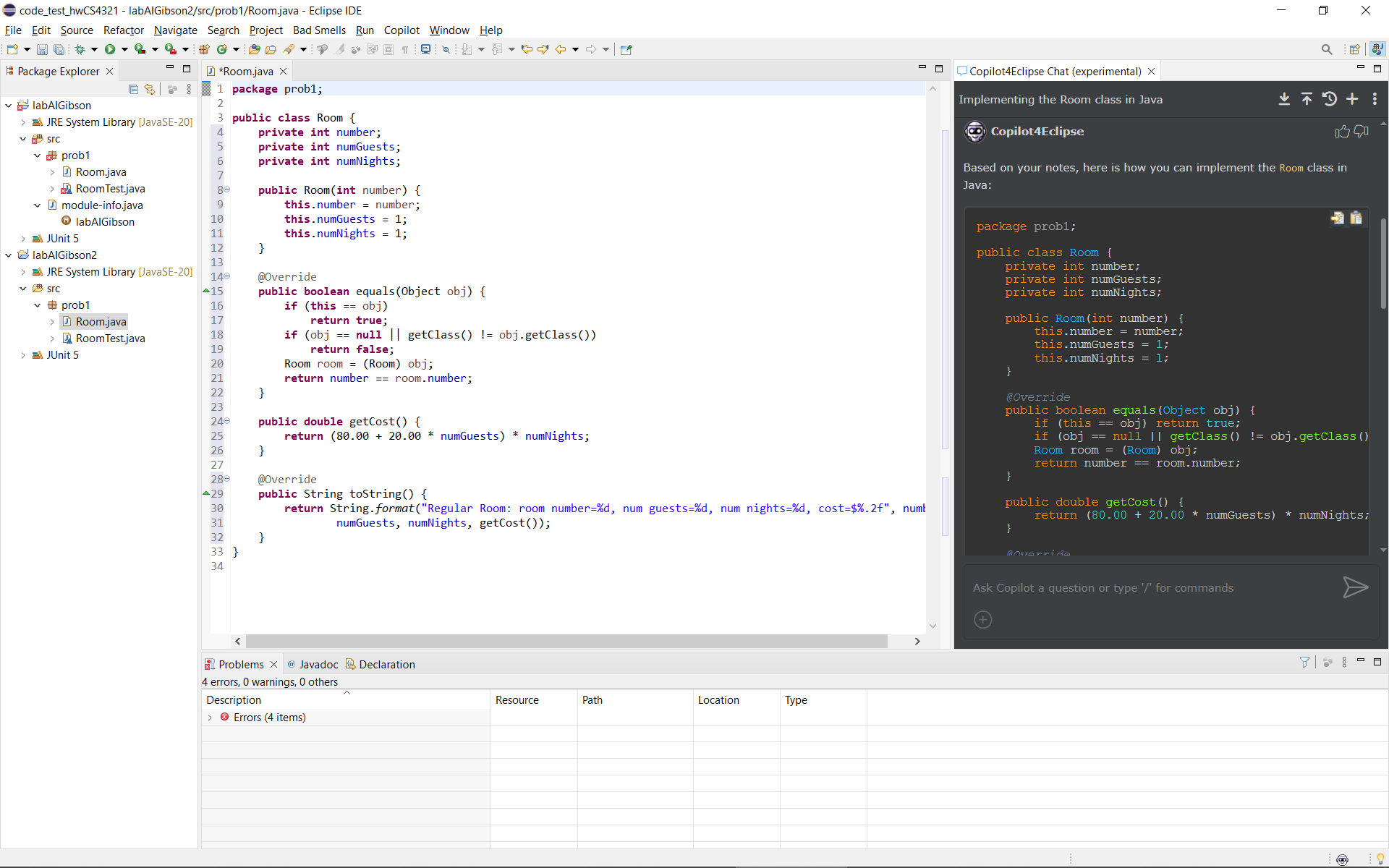
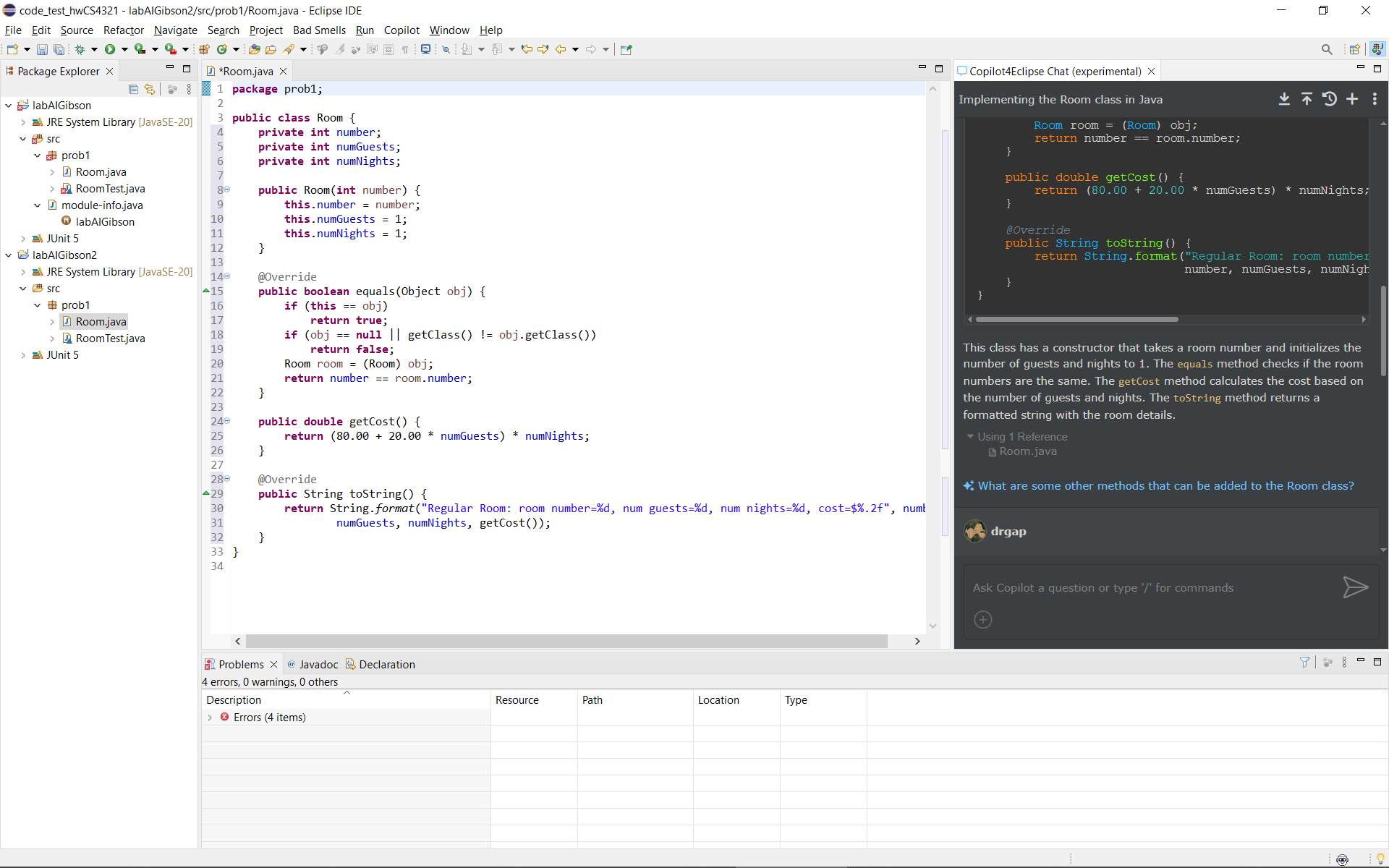
# Chat – Room Example

Do the following:

1. Open the Chat panel by choosing: Copilot, Open Chat Panel (experimental). It will look similar to the image immediately below.
2. Copy everything in the box below and past directly into Chat (as shown below) and press the Arrow icon.

|  |
| --- |
| Notes on *Room* class:   * *number* – the room number (any integer) * *Room(number:int)* – Creates a room with 1 guest for 1 night, with room number, *number.* * *equals* – rooms are considered equal if they have the same (room) *number.* * *getCost –* The cost per night for a regular room is $80.00 plus $20.00 per guest, and this value is then multiplied by the number of nights. * *toString() –* returns a string like this (should be a single line):   Regular Room: room number=228, num guests=3, num nights=2, cost=$280.00 |



1. (Read, no action required) Note the following:
2. Scroll to the top of the generated output and you’ll see that it creates a *Room* class, as shown on the right.
3. Hover your mouse over the two icons in the upper-right of the generated code window.
4. The icon on the right says, “Copy to Clipboard”, which of course you can use to copy the code into your actual code window. **Do not do this now, we will do it shortly.**
5. The icon on the left says, “Insert at editor cursor position”. Thus, you can put your cursor at the location where you want the code to be copied into and choose this icon
6. If you scroll down, you can see:
7. A description of the class.
8. A node that says, *Using 1 Reference*, which when expanded shows that the frame of reference for the generated answer is: *Room.java*.

* As discussed in one of the readings or videos above, Chat will answer your question utilizing a *frame of reference*. If not explicitly stated, it will typically just assume your active, open code window(s). The “+” at the bottom-left of the dialog (only shown in the figure immediately above) to explicitly define the context.
* A *conversation* occurs when you ask questions repeatedly about some context, and as I understand it, the context is not only the references you define or it infers, but also the conversation itself. In the upper-right of the figure on the right is another “+” which will start a new conversation. To the left of this, the reverse, circular arrow, shows the conversations. There is not a way to export entire conversations.

1. A suggestion (blue, in figure on the right at bottom)
2. Notice in the generated code window, there is an *equals* method. That is something we will study in a later chapter. Type the following phrase into Chat and press Enter (or the arrow icon)

*Remove the equals method*

Notice that the *equals* method is now removed.

From now on, in this lab, I’ll just say: “Tell Chat: ….” When I want you to type something in and enter.

1. Copy/Paste or Insert the Copilot generated code into your *Room* class, replacing everything that is there. You could just delete the code, leave your cursor in the empty window, then choose the Insert icon from the Copilot code window.
2. Tell Chat:

*Add a constructor that accepts all three instance variables and have the first constructor call the second*

Copy and paste the result in. I would just manually, copy/paste the two constructors into the code, overwriting the existing one. The 2 constructors it generated should look like this:

**public** Room(**int** number) {

**this**(number, 1, 1);

}

**public** Room(**int** number, **int** numGuests, **int** numNights) {

**this**.number = number;

**this**.numGuests = numGuests;

**this**.numNights = numNights;

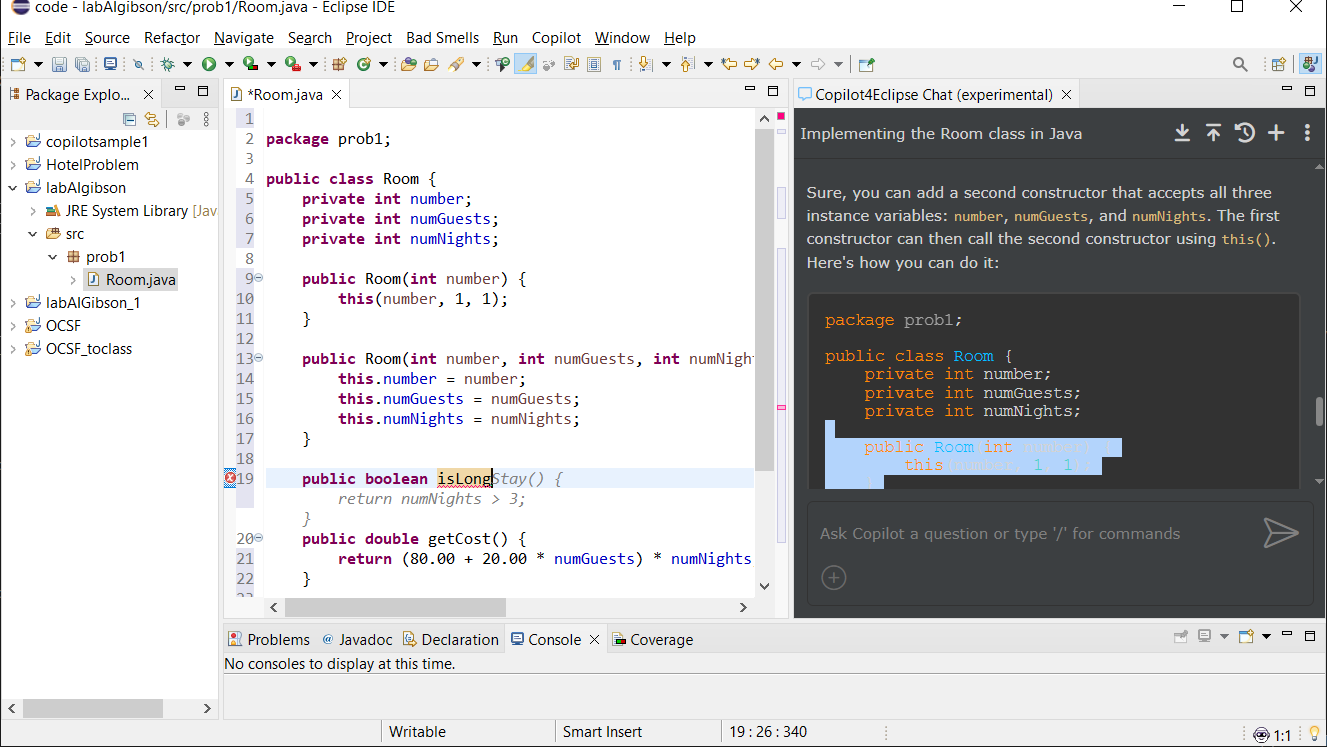
}

Note: Sometimes when you make a modification to a class that is in a conversation, it will leave things out. So, you have to be careful when you copy/paste a whole class

1. In the actual code window (not the Chat window), start a new method by typing (and noting the ghost text):

*public boolean isLong*

In my case, the ghost text appears as shown below. When I did this exact problem a month ago, it used “>2” instead of “>3”. Either way is fine!



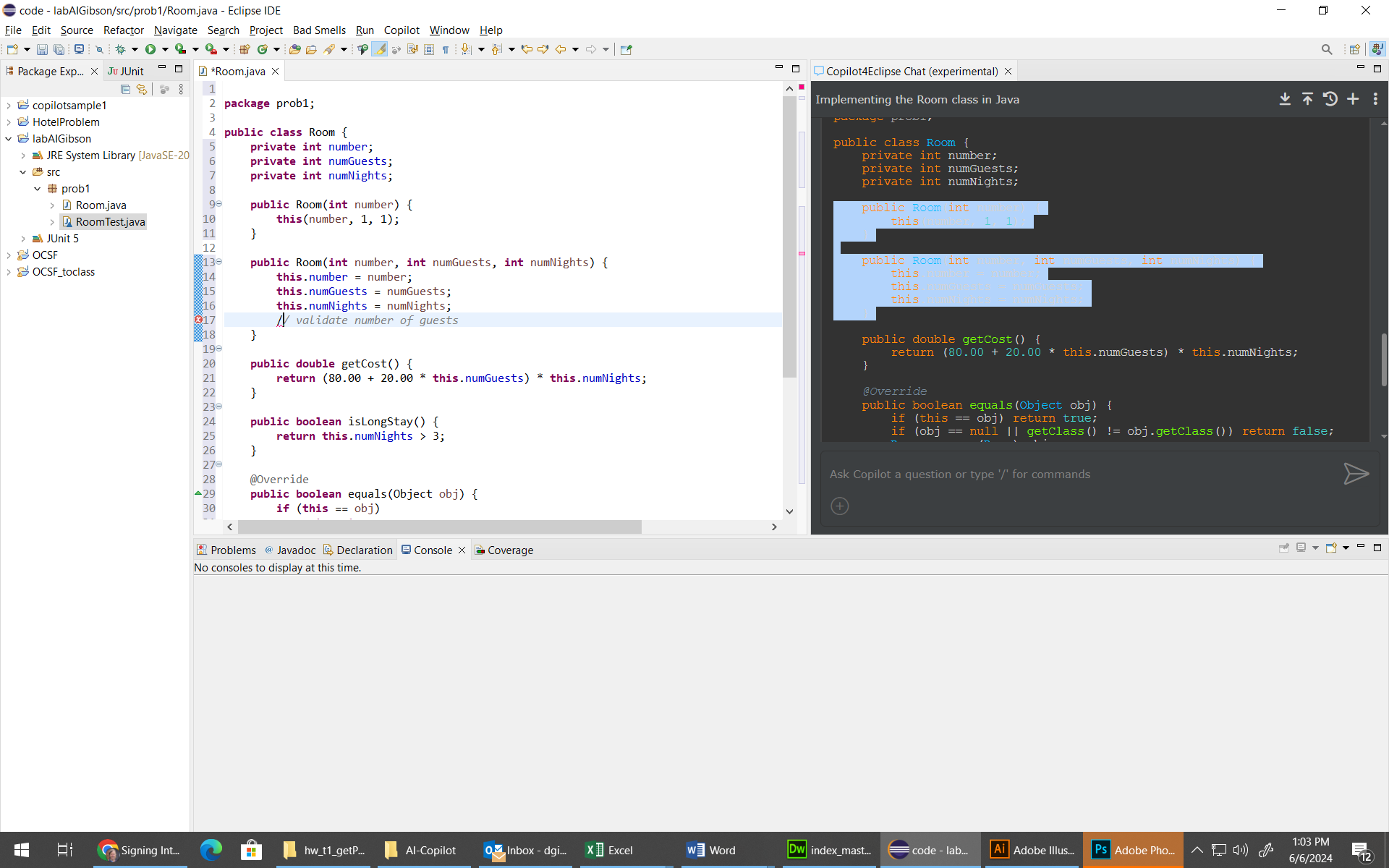
Press the Tab key to accept the suggested code.

**public** **boolean** isLongStay() {

**return** numNights > 3;

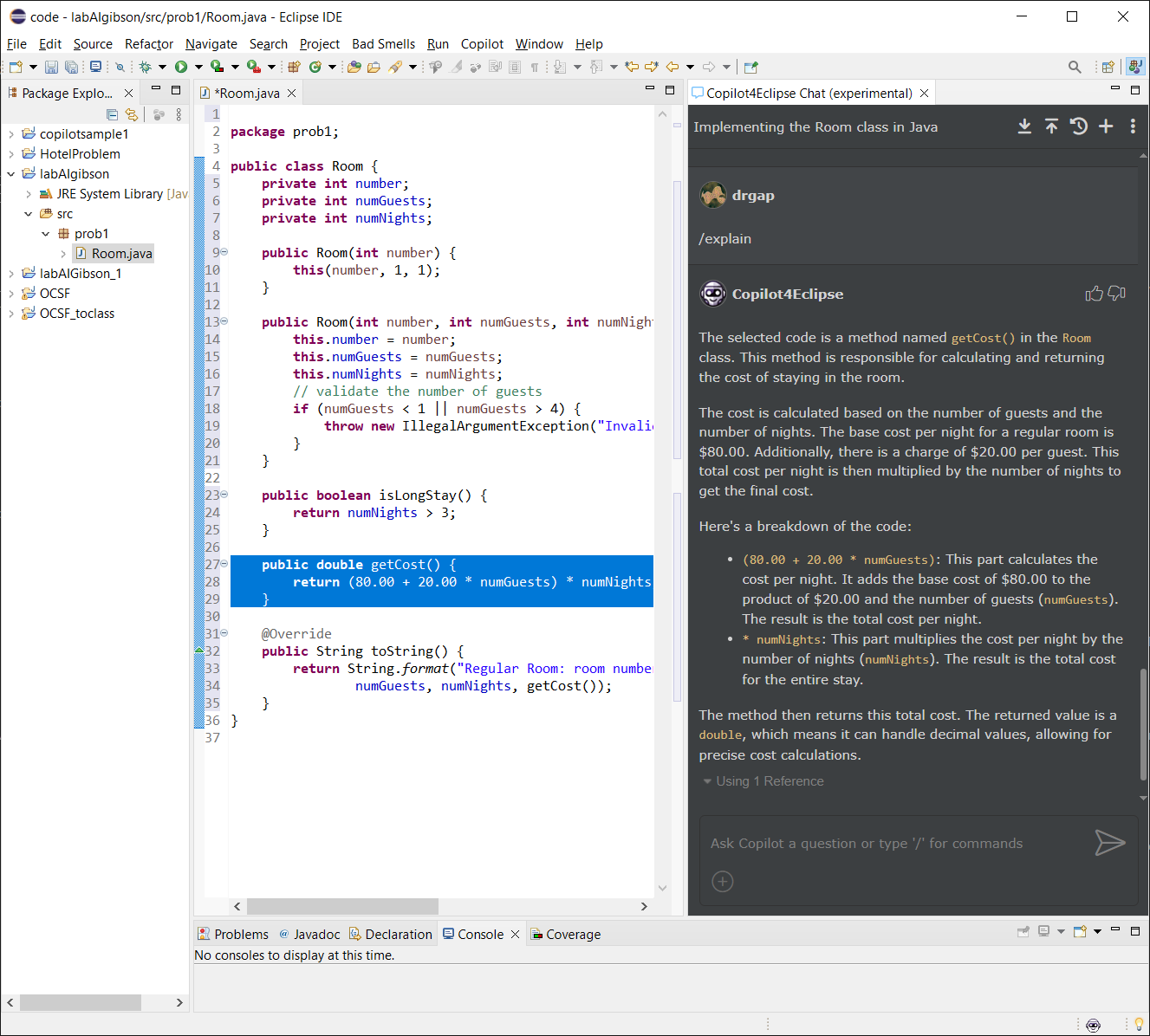
}

1. Go to the end of the *Room* constructor that defines three parameters. Type a single: “/”. This is a way to explicitly invoke Copilot. On my computer, it is as shown below. Whatever it shows on yours, accept it.



After accepting, your cursor will be at the end of the line. Press Enter and some suggested code will appear. Accept it. Note the difference in results when I have done this at different time. The suggestion on July 28 shows a technique we will learn later in the course.

|  |  |  |
| --- | --- | --- |
| June 6, 2024 |  | July 28, 2024 |
|  |  |  |

1. Do the following:
2. Highlight the *getCost* method in the code window.
3. Then, in Chat, type: “/” and you will see options. Choose: “/explain”. You could type something after this, but instead we will just press Enter in which case it is operating on the selected text.
4. Read the description.
5. Do the following:
6. Notice the *getCost* method $20 for each guest. We are going to change that.
7. Tell Chat:

*Change getCost so that the there is only a 20 charge when there are more than 2 guests*

1. On June 6, 2024, I got the result below. Your results may be different than mine. Read this regardless if yours are different and try the steps below, but don’t worry if it doesn’t exactly match the images below. If you think it gets messed up, or you just want to do it again, use the “+” in upper-right to start a new conversation. Experiment!

public double getCost() {

double guestCharge = (numGuests > 2) ? 20.00 \* (numGuests - 2) : 0.00;

return (80.00 + guestCharge) \* numNights;

}

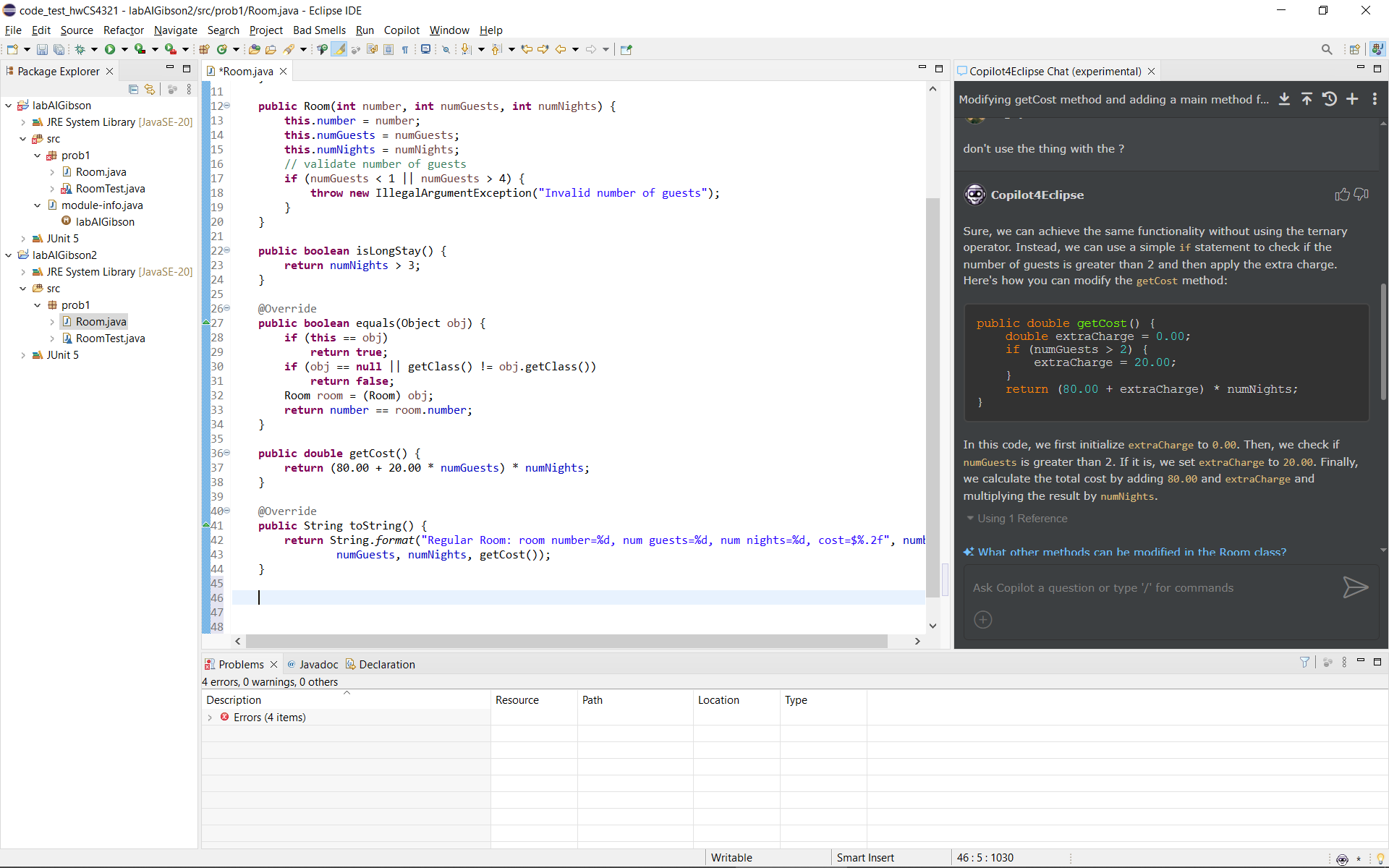
1. This code is not a good place to use the ternary operator (“?”). So, tell Chat: *simplify*. What I got was something similar:

public double getCost() {

return 80.00 \* numNights + ((numGuests > 2) ? 20.00 \* (numGuests - 2) : 0.00) \* numNights;

}

1. This was not what I wanted, so I told Chat, “don't use the thing with the ?” which gave me what I was looking for:



1. On July 28, 2024, I got the result shown below, which is fine. For fun, tell Chat to simplify. When I did this, it gave me the same result as from June 6, 2024 (with the “?” operator).

public double getCost() {

double cost = 80.00 \* numNights;

if (numGuests > 2) {

cost += 20.00 \* (numGuests - 2) \* numNights;

}

return cost;

}

1. Replace the existing *getCost* with the one you like (scroll up if necessary) from the Copilot code window.
2. Do the following:
3. Tell Chat: *Write a main method to do informal testing*

|  |  |  |
| --- | --- | --- |
| June 6, 2024 |  | July 28, 2024 |
|  |  | public static void main(String[] args) {  Room room1 = new Room(101);  Room room2 = new Room(102, 3, 2);  Room room3 = new Room(103, 4, 5);  System.out.println(room1);  System.out.println(room2);  System.out.println(room3);  } |

If you get one like the June 6 image, Tell Chat: *Don’t explicitly call toString.* The result might look like this:

public static void main(String[] args) {

Room room1 = new Room(101, 2, 2);

System.out.println(room1);

Room room2 = new Room(102, 3, 2);

System.out.println(room2);

Room room3 = new Room(103, 4, 2);

System.out.println(room3);

}

1. Tell Chat: *test the getcost method*

public static void main(String[] args) {

Room room1 = new Room(101);

Room room2 = new Room(102, 3, 2);

Room room3 = new Room(103, 4, 5);

System.out.println("Cost for room1: " + room1.getCost());

System.out.println("Cost for room2: " + room2.getCost());

System.out.println("Cost for room3: " + room3.getCost());

}

1. Tell Chat: *create a room with the other constructor*

In my case, it removed the other code that was in *main*. This is something to think about when you are continuing to change a method. So, you might want to copy into your code before making changes.

public static void main(String[] args) {

Room room1 = new Room(101);

System.out.println(room1);

double cost1 = room1.getCost();

System.out.println("Cost of room1: " + cost1);

}

1. Tell Chat:

*Create a main for informal testing, be sure and use both constructors and don't explicitly call toString*

I ended up with this. Inspect yours and make sure the tests make sense. Sometimes they don’t. In this case they do.

public static void main(String[] args) {

*// Using the first constructor*

Room room1 = new Room(101);

System.out.println(room1); *// Implicit call to toString()*

System.out.println("Cost for room1: " + room1.getCost());

*// Using the second constructor with 2 guests (no extra charge)*

Room room2 = new Room(102, 2, 2);

System.out.println(room2); *// Implicit call to toString()*

System.out.println("Cost for room2: " + room2.getCost());

*// Using the second constructor with 3 guests (extra charge for 1 guest)*

Room room3 = new Room(103, 3, 2);

System.out.println(room3); *// Implicit call to toString()*

System.out.println("Cost for room3: " + room3.getCost());

*// Using the second constructor with 4 guests (extra charge for 2 guests)*

Room room4 = new Room(104, 4, 2);

System.out.println(room4); *// Implicit call to toString()*

System.out.println("Cost for room4: " + room4.getCost());

}

1. Insert the generated *main* into your code and run.
2. Do the following:
3. Tell Chat: *change getCost so that the method accepts a tax rate and applies it to the cost*

public double getCost(double taxRate) {

double cost = 80.00 \* numNights;

if (numGuests > 2) {

cost += 20.00 \* (numGuests - 2) \* numNights;

}

cost += cost \* taxRate;

return cost;

}

1. Add the new *getCost* method to your code, replacing the old one.
2. Notice the *main* and *toString* methods don’t compile because the *cost* method accepts an argument now.

Tell Chat: *fix main so it compiles*

public static void main(String[] args) {

double taxRate = 0.05; *// 5% tax rate*

*// Using the first constructor*

Room room1 = new Room(101);

System.out.println(room1); *// Implicit call to toString()*

System.out.println("Cost for room1: " + room1.getCost(taxRate));

...

...

1. Tell Chat: *fix tostring*
2. Add these updatedmethods to your code, replacing the old ones.

# Chat – Formal Testing

1. Do the following:
2. Create a class named: *RoomTest* and make sure it is open.
3. Tell Chat: *Write test methods for the Room class that display expected and actual output but don't use JUnit*

package prob1;

public class RoomTest {

public static void main(String[] args) {

testGetCost();

testToString();

}

public static void testGetCost() {

double taxRate = 0.05; *// 5% tax rate*

Room room = new Room(101, 3, 2);

double expectedCost = 80.00 \* 2 + 20.00 \* (3 - 2) \* 2;

expectedCost += expectedCost \* taxRate;

double actualCost = room.getCost(taxRate);

System.out.println("Expected cost: " + expectedCost);

System.out.println("Actual cost: " + actualCost);

}

...

1. Tell Chat: *Add some more tests for getcost and put comments to explain them*

package prob1;

public class RoomTest {

public static void main(String[] args) {

testGetCostNoExtraGuests();

testGetCostWithExtraGuests();

testGetCostDifferentTaxRate();

testToString();

}

*// Test getCost() when there are no extra guests*

public static void testGetCostNoExtraGuests() {

double taxRate = 0.05; *// 5% tax rate*

Room room = new Room(101, 2, 2); *// 2 guests, so no extra charge*

double expectedCost = 80.00 \* 2; *// Base cost for 2 nights*

expectedCost += expectedCost \* taxRate; *// Apply tax*

double actualCost = room.getCost(taxRate);

System.out.println("Expected cost (no extra guests): " + expectedCost);

System.out.println("Actual cost (no extra guests): " + actualCost);

}

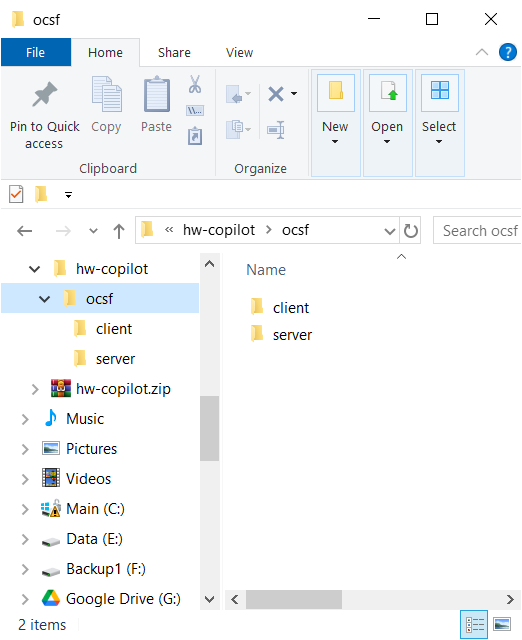
*// Test getCost() when there are extra guests*

...

1. Add the generated tests to *RoomTest.* Be careful, in my case it only generated the test methods, so you need to copy them in, not replace the entire file.
2. Some other things I have told Chat in my previous experimentation:

* Tell Chat: *the second test method is not properly named.*
* Tell Chat: *The third test fails, please fix the method*
* Tell Chat: *getCost doesn't handle the case where there are zero guests*

# Chat – OCSF Example

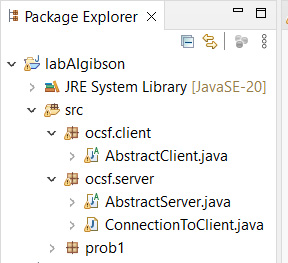
Next, we show an example of explaining a complex piece of code.

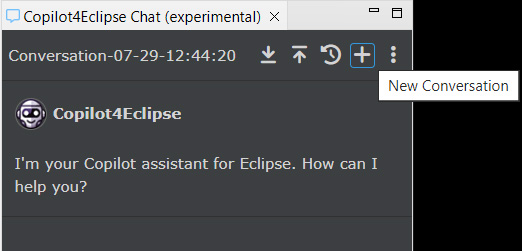
1. Close all open filesCreate a new project in Eclipse named: *OCSF*
2. Download and unzip, *hw-copilot.zip* from the Schedule. Inside, you will find an *ocsf* folder that contains two subfolders: *client* and *server.* Inside these there are a total of three Java files:

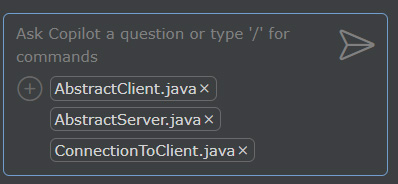
\client\AbstractClient.java

\server\AbstractServer.java

\server\ConnectionToClient.java

1. Drag the *ocsf* folder into the *src* node in the Package Explorer in Eclipse. The results will appear as shown on the right.
2. Open all 3 java files.
3. Start a new conversation in Chat by pressing the “+” in the upper-right.



1. Add references to the three classes by pressing the “+” at the bottom. You may have to type the names in, and then they will appear in a dialog and you can select them. The result is shown on the right.
2. Tell Chat: */explain what is the purpose of these classes?*
3. Read through the description thoroughly.
4. Tell Chat: *show an example of how to use these classes*
5. Read through the description thoroughly.

The resulting *Main* class will run and illustrate using the framework. This is beyond the scope of this class. I just wanted to show you how Copilot could explain a fairly complicated piece of code and provide a running example.

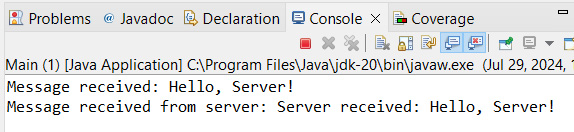
If you want to run it (you don’t have to), do the following:

1. Create a *MyServer* class in the *ocsf.server* package.
2. Replace the code in that class with the generated code (assuming the package statement and class name are correct). Resolve the compilation error by hovering your mouse on the red squiggly and choose something like: import java.io.IOException.
3. Create a *MyClient* class in the *ocsf.client* package.
4. Replace the code in that class with the generated code (assuming the package statement and class name are correct). If there are any compilation errors (don’t think there are), resolve them as above.
5. Select the *src* node in the Package Explorer and create a *Main* class
6. Replace the code in that class with the generated code. Resolve the compilation errors by hovering your mouse on the red squiggly and choose something like:

**import** ocsf.client.MyClient;

**import** ocsf.server.MyServer;

1. Open *Main* and run it. You’ll see the messages as below. The *MyServer* and *MyClient* are still running even though *Main* is finished (not a good design, but Copilot did its best, maybe!). If you ran it again, it would bomb. However, if you kill those processes by pressing the red square (shown in the Console window below), then you can run it again.



This OCSF package is for exchanging messages (objects) across a network. I’ve used it in CS 4321 – Software Engineering for the project.

# Chat – To Learn Something

There is nothing to turn in here. Just do these quick chats to illustrate how to learn something.

1. Start a new Chat (the “+” in the upper-right)
2. Tell Chat: *what is a class in java*

* Read the answer.

1. Tell Chat: *how do you create objects*

* Read the answer.

1. Tell Chat: *why are the properties declared private*

* Read the answer.

1. Play around with it some more!

# Role of AI Assistants in CS 1302 Policy

*(Read, no action required).* This is the policy we will use for this course for all homework and labs. This policy is identical to this one I found at [Georgia Tech](https://www.cc.gatech.edu/news/new-policies-navigate-role-ai-assistants-cs-courses), except that I removed 2 sentences and added a parenthetical statement about Copilot4Eclipse.

*We treat AI-based assistance, such as ChatGPT and Copilot, the same way we treat collaboration with other people: you are welcome to talk about your ideas and work with other people, both inside and outside the class, as well as with AI-based assistants.*

*However, all work you submit must be your own. You should never include in your assignment anything that was not written directly by you.* *If you are unsure where the line is between collaborating with AI and copying AI, we recommend the following heuristics:*

* *Heuristic 1: Never hit “Copy” within your conversation with an AI assistant. You can copy your own work into your own conversation, but do not copy anything from the conversation back into your assignment.*

*Instead, use your interaction with the AI assistant as a learning experience, then let your assignment reflect your improved understanding.*

* *Heuristic 2: Do not have your assignment and the AI agent open at the same time* (choose: Copilot, Sign out of GitHub Copilot)*. Similar to the above, use your conversation with the AI as a learning experience, then close the interaction down, open your assignment, and let your assignment reflect your revised knowledge.*

*This heuristic includes avoiding using AI directly integrated into your composition environment: just as you should not let a classmate write content or code directly into your submission, so also you should avoid using tools that directly add content to your submission.*

*Deviating from these heuristics does not automatically qualify as academic misconduct; however, following these heuristics essentially guarantees your collaboration will not cross the line into misconduct.*

# Submission Requirements

Submit your *Room.java* and *RoomTest.java* files in the *HW-Copilot* dropbox on Blazeview by the deadline

Appendix

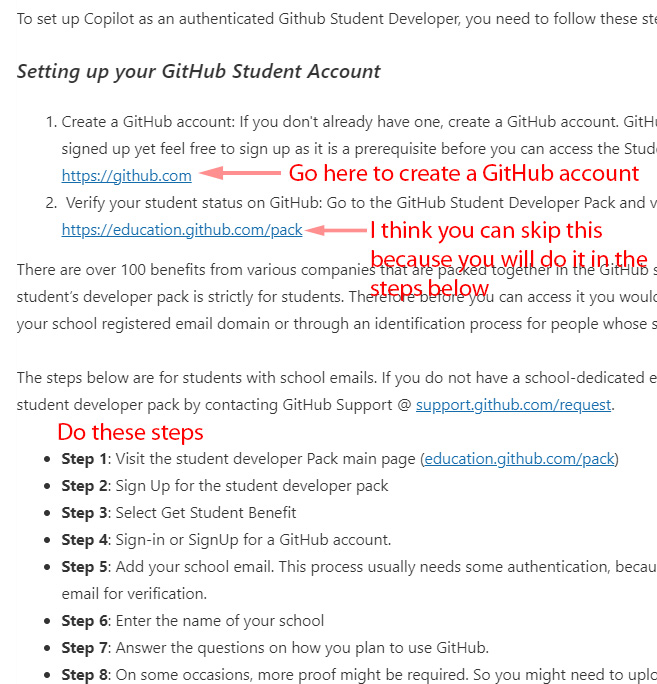
1. Gain Access to GitHub Copilot

This material is copied from Lab 1, Section 2.

Do the following to obtain access to GitHub Copilot. We will set it up and learn to use this in Lab 4b. You are doing this now because it takes a few days to verify your student status.

1. Visit [Setting Up GitHub Student](https://techcommunity.microsoft.com/t5/educator-developer-blog/step-by-step-setting-up-github-student-and-github-copilot-as-an/ba-p/3736279), and as shown in the image below (next page), do the things indicated. Notes:

* For item “1” at the top of image, create a GitHub account. You can use your VSU email or another as your primary email. If you don’t use your VSU email (which is fine), you will eventually have to supply your VSU email account as part of “Student Status Verification” and that will be your secondary email.
* I believe you can skip item “2”, as you will do that when you do the 8 “Steps” below that.
* For “Step 5”, you will enter your VSU email for verification. I believe it will ask you for more proof of student status. Notes:
  + The official documentation says, “…upload an image of your school ID, academic transcript, or other document showing proof of your academic status that clearly shows at least one date”
  + I believe your VSU ID does not work because it doesn’t show any year or current term in it.
  + Some approaches my students have used to verify their student status:
    - A screen shot of their schedule as shown Visual Schedule Builder. Interestingly, it did not show their name. However, this worked.
    - A screen shot of Degree Works and that worked. The student did not show me the screen shot, so I don’t know exactly what the screen shot looked like.
    - A screenshot of my digital VSU 1Card, as well as a screenshot of my personal information from the Student profile tab in the MyVSU Banner. Both contained my student ID# and the student profile page showed the same VSU email that was used in the GitHub account creation.
    - I requested a document from my advisor with the VSU letterhead and a date indicating that I am taking classes remotely.
  + From talking to students, it may take several tries at this to get it to go through.
  + There is a support email, so use that as a last resort. I can’t find it right now, but it is in the pages for this process.



1. Once your student status is verified (takes a day or two), you are done with this. We will set it up and learn to use it in Lab 4b. So, move on the next section!

1. <https://docs.github.com/en/copilot/copilot-individual/about-github-copilot-individual> [↑](#footnote-ref-1)