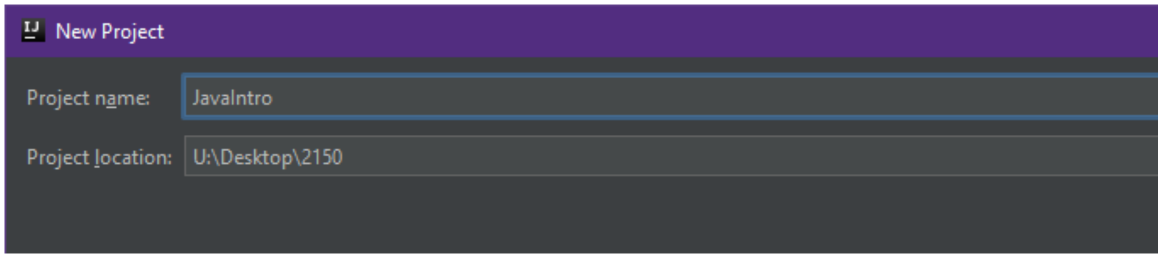
CPSC 2151

Lab 1

Due: at the end of your lab

In this lab you will be working in IntelliJ and with Java for the first time. Don't worry, we're going to go through this step by step. You should have watched the videos before lab and have IntelliJ and Java installed on your machine already. If not you'll have to refer to those instructions on Canvas.

1. Locate the IntelliJ program on your machine and run it. It takes some time to get going. Note: the actual name is much longer than IntelliJ, but searching IntelliJ should be enough to find it.
2. Select Create New Project.
3. At the top of the window, there is a box that says "Project SDK" which may not be found. If that's the case, continue to step 4. Otherwise if IntelliJ already knows where the SDK is, skip to step 6
4. Hit the "new" button next to the SDK box and select "JDK." A normal windows file navigation window will appear.
5. Navigate to the directory where you installed Java. On Windows machines the default is C:\Program Files\Java, but you could have placed it anywhere. In that directory there should be two files, one that starts with "jre" and one that starts with "jdk." Select the file that starts with "jdk." The screen will return to the new project screen that we were at before, but there will be an SDK selected. **NOTE 1:** make sure you installed Java 8 to avoid issues when you move to Unix. **Note 2:** On MacOS machines the default is /Library/Java/JavaVirtualMachines/ and there should only be one JDK folder.
6. There are a lot of options on this screen. On the left you have a list of types of projects, such as Java, JavaFX, Android, etc. This should already have Java selected, but make sure that it does.
7. In the middle of the screen it asks if you want to include any additional libraries or frameworks. Do not select any additional libraries or frameworks
8. Hit next
9. Now it will ask if you would like to create the project from a template. Do not select any templates. Hit next
10. Now it will ask for the name of your project. Your project name can be anything, it's just the name of the folder that will be created to hold all the files for this project. It does not need to match any of the code files. You can name it whatever you want, but I suggest "JavaIntro."
11. In the box below, it asks where you would like to save this project. You may even want to create a folder just for this class to keep all your assignments in one place.

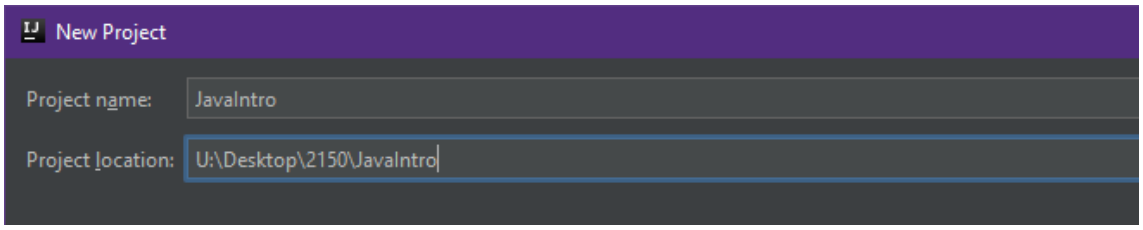


Project name: JavaIntro

Project location: U:\Desktop\2150

12. When you change the folder, IntelliJ gets a little confused. I've selected my folder I made for this course, but IntelliJ wants the project name to match the folder. If I hit Finish now, it will make

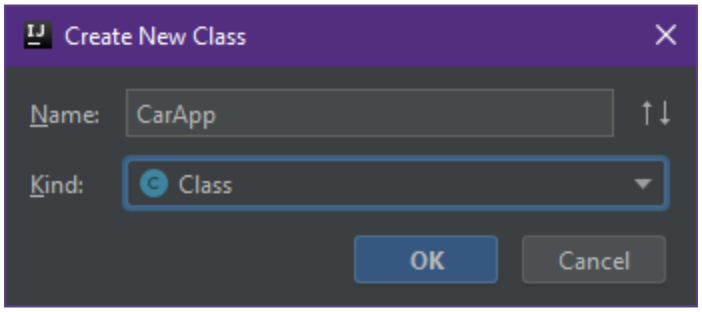
my U:\Desktop\2150 folder into my project folder, and I don't want that. I'll go ahead and type in JavaIntro so IntelliJ knows to create that folder for me.



Project name: JavaIntro

Project location: U:\Desktop\2150\JavaIntro

13. Now I will go ahead and hit Finish, so IntelliJ can create my project. Hit OK if it asks if you want to create the project directory.
14. On the left side of the window you have your project navigation pane. If you expand the Java intro project, it will show 3 things
    a. The .idea folder. This folder exists to hold most of the files that IntelliJ needs. These aren't necessary for our java code, only for the IntelliJ project, so we'll ignore these
    b. The src folder is where our actual code will go. It is currently empty, but we will add to it
    c. A file that has your project name and ends in .iml. This is another IntelliJ file that we can ignore
15. Right click on the src folder and select to add a new package. Packages are structures in java to group related classes, and we'll talk about them later in the semester. Go ahead and name your package "cpsc2150.cars". The naming of the package is important to us, so for every assignment you'll want to make sure the package name matches exactly what is given in the prompt. Otherwise it will take longer for the TAs to grade. You will even lose points for not using the correct package name.
16. Now inside of your src folder you have the cpsc210.cars package. Right click on the cpsc210.cars package and add a new class. Name it CarApp. There are other options under "kind" but we'll talk about them later.

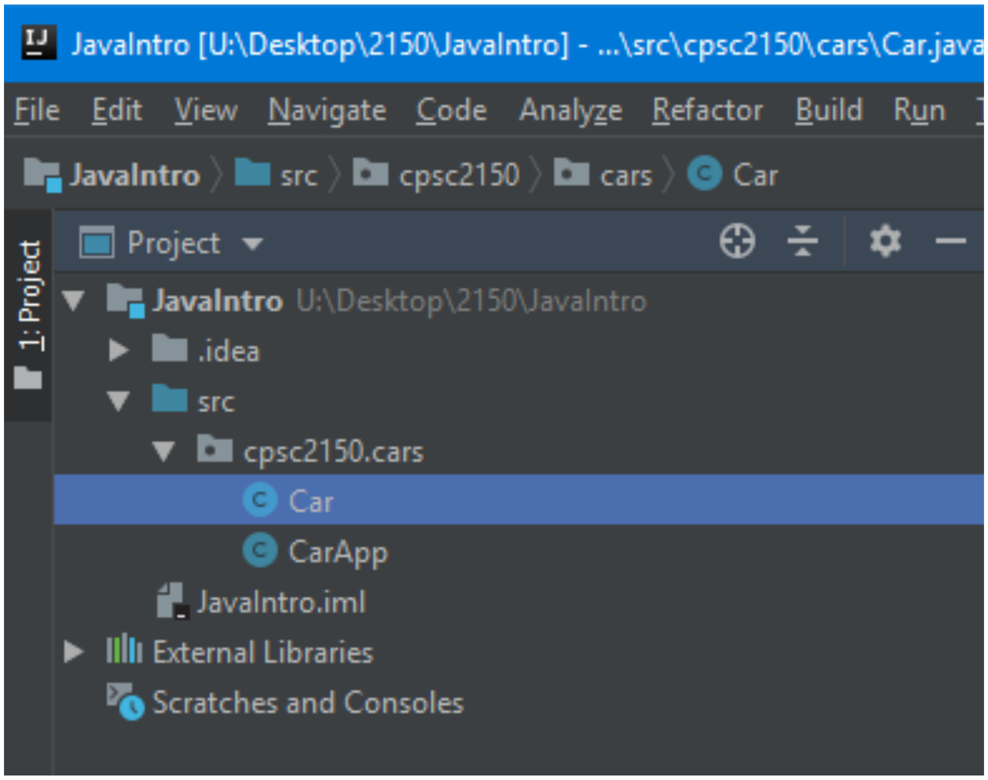


Name: CarApp

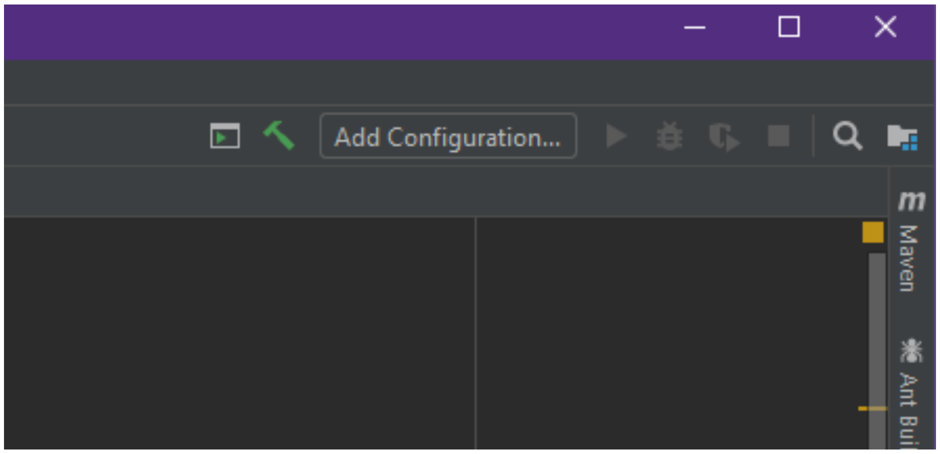
Kind: Class

17. IntelliJ has now created the class for you. Go to canvas and grab the "CarApp.java" file I have provided. Copy the code into your class. **Note:** my code has the package statement and the class declaration in it, so don't copy that in as well. You don't want those lines duplicated. Make sure to get those import statements!
18. We have some errors, because our code is referencing a class Car that does not exist. We'll need to add that. Right click on the cpsc210.cars package and add a new class called Car. Get the code from the Car.java file from Canvas as well.

19. Your package structure should look like this:

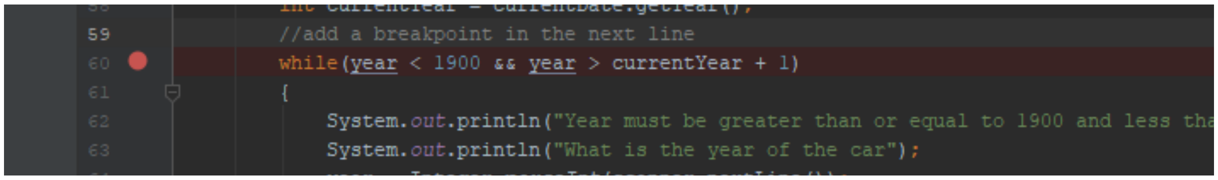

JavaIntro [U:\Desktop\2150\JavaIntro] - ...\src\cpsc2150\cars\Car.java
File Edit View Navigate Code Analyze Refactor Build Run
JavaIntro src cpsc2150 cars Car
1: Project
Project
JavaIntro U:\Desktop\2150\JavaIntro
.idea
src
cpsc2150.cars
Car
CarApp
JavaIntro.iml
External Libraries
Scratches and Consoles

20. Now we're ready to compile our code. We'll have to add a build configuration to do this. Click on Add Configuration near the top right corner



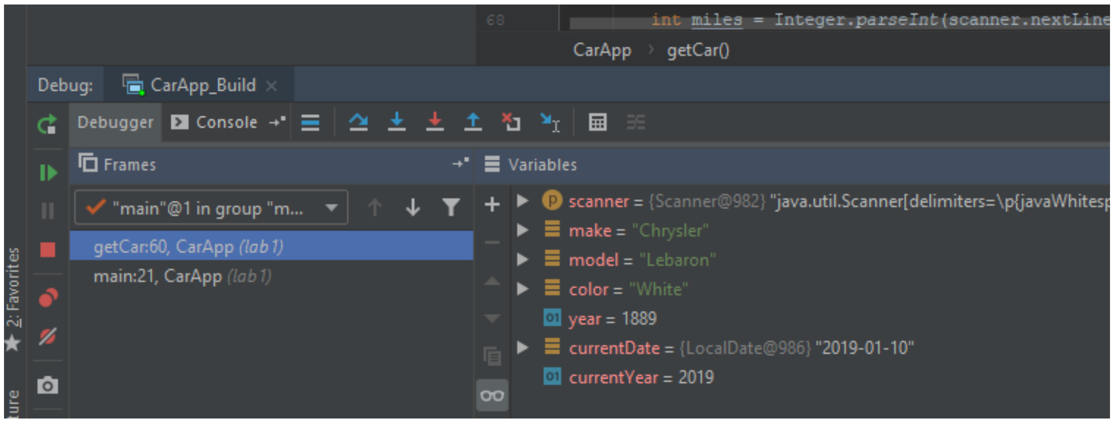
21. Click on the + button to add a new configuration, and select application
22. You can name the build whatever you want. I'll call it CarApp_Build
23. Click the button with three dots to the right of main class. This is asking which class in our code has the "main" function in it. Remember, in java every class can have its own main function, so we have to tell IntelliJ which main function to use if there are more than one. Select CarApp, since that has our main function in it.
24. Leave all other settings the same and hit "OK"
25. Now click on the green arrow in the top right (right next to the add configuration spot that nor says CarApp_Build) to build and run our program.

26. Our code works. But try to add a car with an invalid year, such as a year before 1900 or a year after 2020.
27. It accepts the bad year, but we have code in CarApp.getCar to check to make sure it's a valid year. Since our code compiles, but does not do what we expect it to do, we can debug the code to see what happened.
28. If the program is still running, go ahead and stop it either by completing the program and selecting 3 to quit at the menu, or by hitting the red stop square by the play button.
29. Now before we start debugging. We need to set a break point. A break point is where we want our code to stop while it's running. When the code is stopped, we can see the values of all the variables. Let's add a breakpoint on the while statement for the loop that checks the car year. We can add a breakpoint by clicking just to the right of the line number.



```
        int currentYear = currentDate.getYear();
        //add a breakpoint in the next line
        while(year < 1900 && year > currentYear + 1)
        {
            System.out.println("Year must be greater than or equal to 1900 and less tha
            System.out.println("What is the year of the car");
```

30. Now, instead of clicking the green play button to start our program, click on the green bug icon to debug it.
31. Enter the car information like you did before, again, using an invalid year. Now the code will stop at the breakpoint and the debugging window will appear

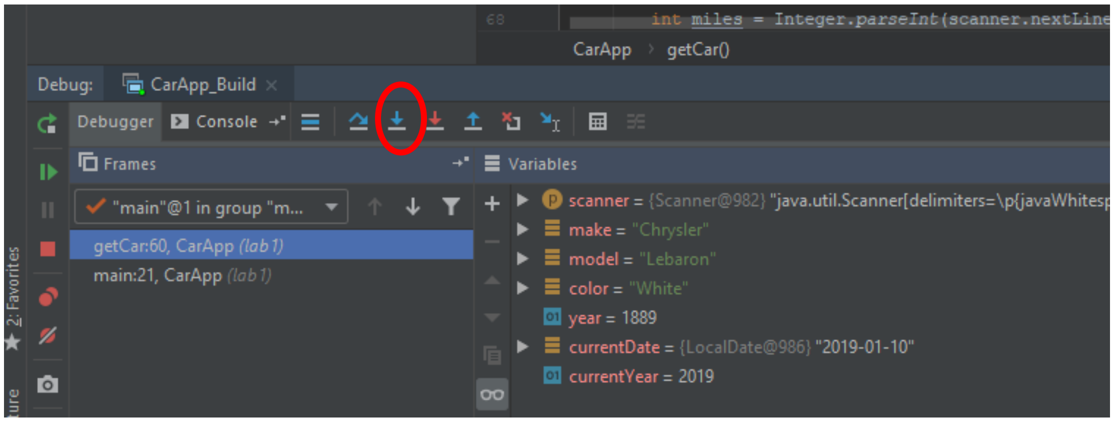


```
        int miles = Integer.parseInt(scanner.nextLine
```

```
getCar:60, CarApp (lab 1)
main:21, CarApp (lab 1)
```

```
scanner = {Scanner@982} "java.util.Scanner[delimiters=\p{javaWhitesp
make = "Chrysler"
model = "Lebaron"
color = "White"
year = 1889
currentDate = {LocalDate@986} "2019-01-10"
currentYear = 2019
```

32. Now we can see the value of all the variables in our program. If the variable is a class (like scanner) we can even click on the arrow next to it to expand it and see all of it's data, but we don't need to do that here.
33. We can use the "Step into" button to move to the next line of code. When we do that, it skips the loop entirely. Can you figure out why? Fix it, then debug it to make sure it works.
    a. The Step into button

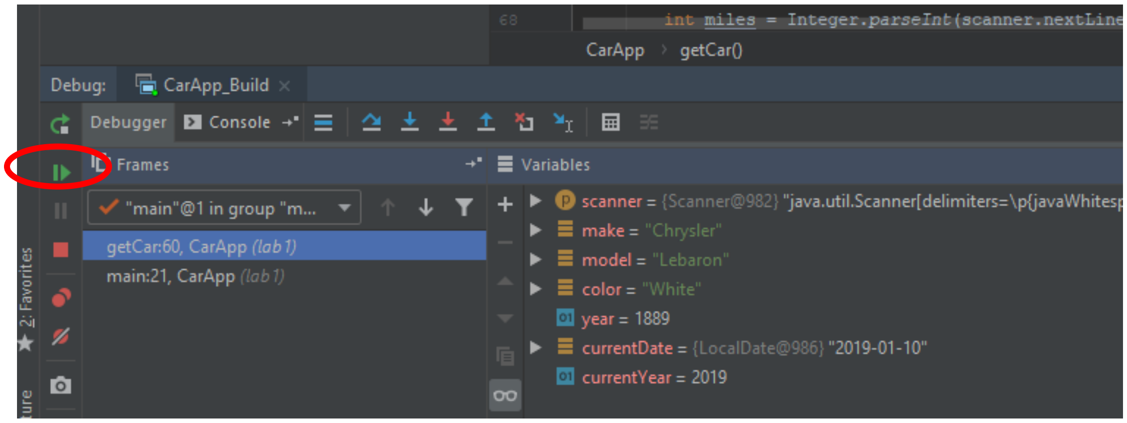


```
int miles = Integer.parseInt(scanner.nextLine
```

```
"main"@1 in group "m...
getCar:60, CarApp (lab 1)
main:21, CarApp (lab 1)
```

```
scanner = {Scanner@982} "java.util.Scanner[delimiters=\p{javaWhitesp
make = "Chrysler"
model = "Lebaron"
color = "White"
year = 1889
currentDate = {LocalDate@986} "2019-01-10"
currentYear = 2019
```

34. You can continue using the Step Into button to go line by line in your code and see the changes to the data. When you want to continue as normal, click on the "Resume" button. You can also set multiple breakpoints to allow you to stop at multiple areas. Unfortunately when debugging you can't go backwards, so you'll want to go slow and be careful so you don't miss anything.
    a. The Resume button

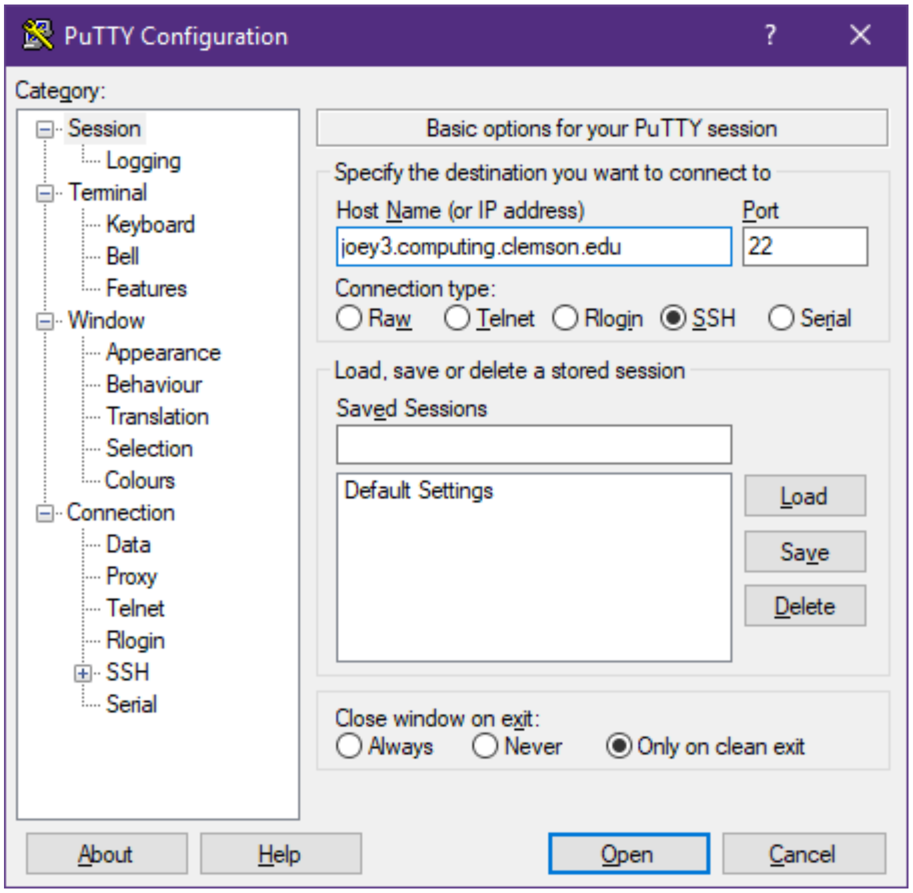


```
int miles = Integer.parseInt(scanner.nextLine
```

```
"main"@1 in group "m...
getCar:60, CarApp (lab 1)
main:21, CarApp (lab 1)
```

```
scanner = {Scanner@982} "java.util.Scanner[delimiters=\p{javaWhitesp
make = "Chrysler"
model = "Lebaron"
color = "White"
year = 1889
currentDate = {LocalDate@986} "2019-01-10"
currentYear = 2019
```

35. Now that we have fixed our program, we're ready to test it on Unix. Make sure you know where the files are saved, the path is in the top left of the screen. Save and exit IntelliJ.
36. In File Explorer, go the where your IntelliJ project was saved. In the project directory you'll see that src directory that we had in IntelliJ. Go into that folder, and you'll see your package directory. Inside that directory you'll see your code files.
37. Now we want to connect to the Unix machines. Use whatever program you are used to for that (I prefer Putty for Windows and Terminal for MacOS). Use an SSH connection, any unix machine you want as the host name (such as joey3.computing.clemson.edu) and use port 22. Remember, you don't need to use access if you are on campus. If you aren't on campus, make sure to login through access, then from access ssh into whatever Unix machine you want to use. You won't be able to compile and run your code on access.



Specify the destination you want to connect to

Host Name (or IP address): joey3.computing.clemson.edu

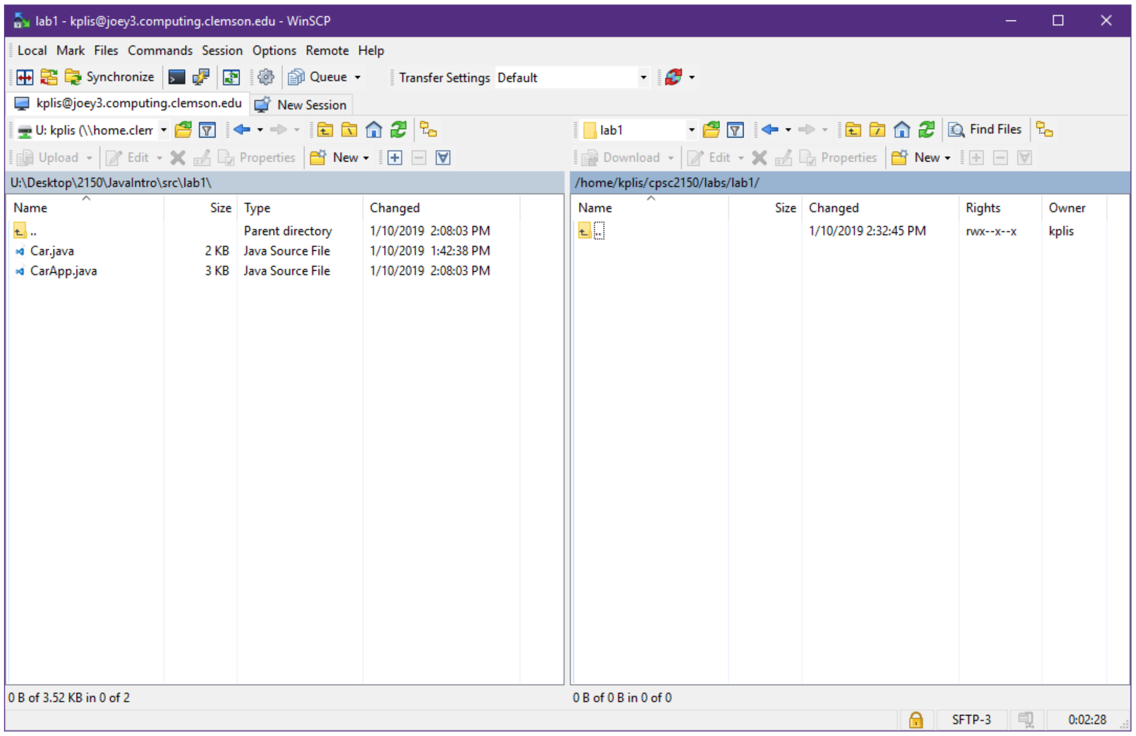
Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Close window on exit: Always Never [✓ Only on clean exit]

38. Use your account to log in. Make a new directory for work for this class (cpsc 2150), and inside that make a directory for labs. Inside of that directory make another directory called "lab1". This is a lot of directories, but it will make it easier to keep track of each lab and project we work on in this class
39. We need to make sure our directory on Unix matching the directory structure in our source folder. So in the lab1 folder, make a directory calls "cpsc2150" and inside of that directory add a directory called "cars" to match our folder structure that IntelliJ made. Remember that folder name came from the name of our package.
40. Now we're ready to move our code into our cars directory.
41. Use your preferred method to transfer code to the unix machines. I prefer WinSCP and I've included instructions for that if you need them. If not, skip to step 46. **Note:** Cyberduck/Filezilla are available for MacOS systems and should be very similar to WinSCP.
42. To set up the connection, you want SFTP as the file protocol. Since you are off campus, you'll have to connect to access.computing.clemson.edu. Use port number 22 and use your username and password to connect. Once we transfer our files to access, they are available on all the Unix machines.
43. WinSCP will have two main windows. On the left side is our local machine, which is the computer we are sitting at. On the right side is the remote machine, the Unix machine we logged into. On the left side navigate to the spot on the machine that has your code.

44. On the right side, navigate to the directory we just created.


lab1 - kplis@joey3.computing.clemson.edu - WinSCP
Local Mark Files Commands Session Options Remote Help
Synchronize
Queue
Transfer Settings Default
kplis@joey3.computing.clemson.edu
New Session
U: kplis (\\home.clem
lab1
Find Files
Upload
Edit
Properties
New
Download
SFTP-3
0:02:28

U:\Desktop\2150\JavaIntro\src\lab1\

Name	Size	Type	Changed
..		Parent directory	1/10/2019 2:08:03 PM
Car.java	2 KB	Java Source File	1/10/2019 1:42:38 PM
CarApp.java	3 KB	Java Source File	1/10/2019 2:08:03 PM

0 B of 3.52 KB in 0 of 2

/home/kplis/cpsc2150/labs/lab1/

Name	Size	Changed	Rights	Owner
..		1/10/2019 2:32:45 PM	rwx--x--x	kplis

0 B of 0 B in 0 of 0

45. Select both files in the left window and click on Upload, and the files will be copied to Unix.
46. In putty, we can now type `ls` to see the files in the cars directory.
47. To compile the files on Unix we can either
    a. Be in the lab1 directory and use the command `javac CarApp.java Car.java`
    b. Go back outside the package directory (cpsc2150/cars) and use the command `javac cpsc2150/cars/CarApp.java cpsc2150/cars/Car.java`
48. Now we can run the program. To run the program we HAVE to be outside the cpsc2150/cars directory and use the package name. However, we don't need file extensions, and we only need the name of the file that has the main function. The command is `java cpsc2150.cars.CarApp`
49. Congratulations! You have now written a program in Java in IntelliJ, debugged it, moved it to Unix and compiled it there. Now let's setup a makefile!
50. Grab the makefile I have provided on Canvas and use WinSCP to transfer it to the Unix machine. Remember, the makefile needs to go outside of out package directory (cpsc2150/cars). Note: Canvas does not like not having a file extension. You'll need to remove the ".txt" for the makefile to work.
51. Now that my makefile is there, I can go to putty and type "make" to run the makefile on the default target.
52. Well, that didn't work. I'm getting an error saying "missing separator." I made a common mistake in my makefile. When I created it on my computer, my computer replaced the tabs with 5 spaces, and makefiles have to have tabs! We can open up the makefile on unix using vim, and delete the spaces and replace them with tabs.
53. Now I can try to compile my code with the command "make". It works! I can use "make run" to run my program as well.

54. Now that everything is working I am ready to zip up my code and submit. I want to keep my current package structure, so I need to change my directory to be outside of the lab1 directory.
55. You can use the command "`zip -r Lab1.zip lab1`" to zip up all the code. Don't forget the `-r` to make sure it includes the subdirectories as well. **Note:** copying and pasting this command may cause issues, because Windows encodes the `-` symbol differently from Unix.
56. When you zip it up, you should see it list the files being compressed one by one. If you don't see a list of files, then it didn't compress everything. Probably because you left off the `-r`. You might think I'm talking too much about this `-r`, but in the past a lot of students forget about it and submit an empty lab directory. Even worse, they'll continue to forget about it on homework assignments!
57. Now you have you zipped directory in Lab1.zip. You can use WinSCP to transfer it to your machine. Now you can go to Handin (handin.cs.clemson.edu), go to my cpsc2150 course and the lab 1 assignment, and upload your zipped folder to submit your assignment.
58. Before finishing, let's just do one quick check to make sure we uploaded the correct files. On your machine, use 7zip (or another tool) to unzip the zipped directory and make sure everything is there. If you don't see all the code files, then something went wrong when you tried to zip everything up. Go back to step 54 and try again. Don't worry, Handin allows multiple submissions.
59. Once you have submitted your code, you are all done!