Section 1 - General Homework Requirements

This **module** contains important information you'll need to get full credit for your programming labs. In addition to this **module**, read

- the entire handout Downloading and Configuring Your Compiler, and
- any prior module sections pertaining to compiling, running, and preparing homework code for submission.

LABS.1.1 General

There is a lot of "*don't do this or else*" in this module; I have to protect the transferability and accreditation of the course. So get ready to hear all the bad things that will happen if you don't follow directions. We'll get it over with now and get back to positive things shortly.

Cheating

The most important thing to know about homework programs is that, in my course, they are to be done by you, alone. You can ask questions in the public forums where I and others can give you carefully considered answers.

Warning

Do not ask friends, fellow students, or get help from on-line answer sites or you will end up in the dean of students office with the possibility of college expulsion. You will, of course, receive a 0 for that assignment, and I will go back and re-examine past assignments to see if I missed any prior copying.

I send multiple students to the dean for cheating every quarter, and I look forward to the day when I can get through a quarter without having to do this. Most of you will welcome this policy since it prevents less qualified people from gaining an advantage over you for transfer and job placement.

The PSME tutorial center is the second acceptable choice for help, but the course discussion forums are preferred.

Accessing Lab Assignments

To view an assignment, click the **Assignments, Tests and Surveys** (***ATS***) tool on the left navigation bar and click *Begin* under the desired assignment name. From there you may be asked to check an *Honor pledge* and *Begin/Continue* to see the assignment. You will be able to read the assignment, come back to it as often as you wish, and eventually submit ***one attached file*** as your work. You cannot change anything once you submit (*Finish*), and *you can only submit one file*.

The File You Submit

Submit one attached file, only. There are buttons called **"Browse"** and **"Upload"** near the bottom of the assignment page. Click on **Browse** and locate the file that you want to attach (described next), then click **Upload** to upload it. If you make a mistake and want to attach a different file, first remove the old one, then reattach the new one. I only allow one attached file.

Important

The one and only file you attach should be a plain text file, like a **Notepad** file, and ***always*** must have a **.txt** extension**.** *It must **not*** be an MS Word file, .rtf file, .zip, etc. I will not accept those.

Exception: In some assignments of some advanced classes I accept a second Excel or Word file with mathematical analyses to support the code, but this will be stated in the assignment.

Most, if not all, of your homework programs will be console assignments (as opposed to GUI assignments). Preparing and submitting a console assignment is slightly different than handing in a GUI project.

LABS.1.2 Submissions for Console (i.e., Non-GUI) Programs

The file you submit will contain:

1. The ***program source***, and
2. The ***program run*** copied and pasted from your console window

in a single file. You must have ***both*** the ***source*** and ***run***. Paste the ***source*** followed by the ***run***, into your submission file, with separators (dashes) between the two. For example:

```
public class Foothill
{
   public static void main(String[] args)
   {
      System.out.println( "Hello World!");
   }
}

--------- OUTPUT -----------

Hello World!

----------------------------
```

Name the main public class **Foothill**, always. You may have many other classes like **Property, Agent, ClientProfile, Point**, etc., but your main class -- the one with the public static void main() method --should be named **Foothill**

LABS.1.3 Submissions for GUI Programs

In the rare case you are in a course in which you will be submitting a GUI for one or two assignments, you do not have to submit a run for GUI programs. I will run your GUI myself. Place all of your source code into a single file, *and make sure it runs as one file*.

Code Running as a Single File Means:

1. You will have only one public class (all others being default or private access).
2. All import statements must be at the top of the file.

LABS.1.4 Indentation, Removing Tabs and Style

Always remove tabs (by replacing each with 3-spaces) and double check manually, using the trick in the ***removing tabs module***. Just because it looks right on your screen does not mean tabs are not present, nor does it guarantee it will look correct on another system or in another platform. Also, check your indentation.

Note

You will lose a point for *each* tab or style error.

Refer to the sections pertaining to removing tabs in the module on Downloading and Configuring Your Compiler as well as the sections on style rules for details. These important sections are in different places depending on the course, but they are all required reading, so please find and read them.

I do not want you to lose a single point for style, so I provide lots of support to help you remove the tabs and get the indentation correct. If you are not able to follow through on learning those items, here is a full disclosure on what to expect.

Penalties for incorrect style are

- -1 for the first assignment,
- -2 for the second assignment,
- -3 for the third assignment,
- ...
- -9 for the ninth assignment.

To avoid this, simply read and follow the tab-removal and indentation rules.

LABS.1.5 Difficulty Levels

Some weeks you will have the option of doing one of at least two versions of the assignment.

- **Easy Option A**
- **Intermediate Options B**

You can only submit one option. There is no point difference among the options . They are there only to give you a way to adjust the difficulty to your personal level. (Exception: in some advanced classes I give extra points for certain B options.) If you find option A too easy, and you wish a more challenging task, try one of the B options if they are present.

You cannot use the options for extra credit (unless you are in an advanced class and they are so designated). The only way to to get extra credit is by handing in the bonus assignment at the end of the quarter.

Section 2 - Specific Requirements and Point Penalties

In the last page, I gave you exact point deductions for incorrect style. I know students like clarity, so I am including more details about the programming assignment grading in this section.

LABS.2.1 Factors that Influence Grading and Point Deductions

You can lose points for several reasons. Normally, if you ignore a written rule or recommendation in the modules or in the assignment description, you will lose multiple points. As stated earlier, tabs or style rule infractions are also costly.

Below, I've listed some other common reasons that students have been penalized in the past. If you refer to this list frequently, *especially just before you submit*, you can avoid losing points for these same issues. If you don't understand a bullet point below, ask me or a fellow student in the public lounge.

- **Not Working.** You will not get any points for a non-working program. If you can't get the program to run perfectly, keep trying to debug it, ask for help, and hand it in late. On-time non-working = zero. Late, working = many points. The correct choice should be obvious.
- **Don't Edit Your *Run*.** Paste your ***run*** directly into the end of your text file for submission. You can set it off from the ***source*** by adding a line of dashes or comments, but do not change or add anything to the actual lines in your ***run***. I want to see them, untouched, as they appeared in your console. Any additional spaces or characters added to the ***run*** of the program will result in a large penalty.

- **Late**. The late penalty is 1 point per day, (*1 hour past the **due time of 2 PM** it is 1 day late*) with a maximum of 50% taken off for late points. After you have lost 50% for late, I will not take off any more points for late.
- **Extensions.** I do not normally give extensions. In extreme cases I might consider one, ***but only if you contact me before the assignment due date*** and have a very good story. The most important thing is to not wait until after the assignment is due and then try to convince me that you could not spare 90 seconds in the previous week to send me a message about your emergency. You need to inform me before the due date if you want to have any chance of getting an extension. Otherwise, you can still hand in the assignment late, but it will get the usual late points deducted.
- **Time.** Some of you will take one hour to do an assignment, others will take ten. Most of you will spend an amount of time between those two extremes. *The amount of time it takes to do an assignment has no impact on your grade*.
- **Proof of Correctness.** Sometimes you will need to hand in more than one run to demonstrate a program runs correctly. Put your program through its paces. Also, sometimes you have to interact with your program during the run to give it various sample input. You are expected to give your program a good "work out" to prove that it is correctly designed. This is part of the assignment. An inadequate test run will incur point penalties.
- **I/O and Calculations.** Don't mix input/output (I/O) with calculations. Do not do calculations in screen output statements. Functions (methods) that compute should not do output or input. Functions (methods) that do input or output should not calculate (except for filtering bad input). This is a source of multiple point loss.
- **Informative Output.** Screen output should be informative. For example, displaying numbers without accompanying words that describe what those numbers mean is not informative. Asking the user to enter a value without giving him or her the valid choices or a description of what kind of value you want from him or her is not informative. Communicate fully with your user through your program's output statements.
- **Write to Spec.** Programming is not a class in creative writing. Programmers have to learn to follow the specification. Therefore, if you deviate from the instructions and submit something that looks really *cool*, but doesn't do what was originally requested, you will lose many points. If you want to deviate from the assignment for some good reason, you can clear it with me beforehand. Otherwise, the penalty for not following the specification is usually in the 6 - 12 point range.
- **Test your program vigorously.** Before submitting an assignment, test it on various types of input. For example, If your input is a string, test with a short (even empty) string, a long string, upper case and lower case and check your output. This will point to hidden errors in your code.
- **Tabs.** Remove all tabs (and replace with three spaces each) as described in the section Configuring Your Compiler for Proper Style in the Obtaining Compiler module. Again, the penalty for not doing this matches the assignment number.
- **Style**. Obey all style rules as described in the section Style Rules and/or the recommended booklet on style near the end of the modules List.
- **Make code readable.** Make sure your code (and output) is readable to the human eye. This means putting spaces between words in your code and using descriptive variable names in addition to using good programming style.

- **Cheating.** If I detect that you have changed another student's -- or my -- solution and submitted it to me as if it were your original work, you will not only get a zero, but I will report the issue to the dean of students. If two (or more) students hand in practically identical work, both (all) get zeros. The innocent among you may worry that you might *accidentally* think of and submit the same solution as another student. After years of doing this, I know which assignments lend themselves to coincidental clones, and I don't penalize for those. I can usually tell (as can my division dean and the dean of students) when two or more papers originated from one *master-solution* and are nothing more than that master-solution with minor, cosmetic changes.
- **Ask questions.** If you are unsure about any part of an assignment, use the discussion board to ask questions. Not only will you have your question answered quickly, you will help other students who needed the same information.

To get you started, I'll give you a few very specific rules now. These are examples of the kinds of things I might say in a module and expect you to follow on pain of point loss. A few of them won't mean much to you until we get to the material, but I'll bet most will make sense.

- **Simplicity.** If there is an easy way and a hard way to do something, and you choose the hard way, you will lose a point or more. It is important in programming not to make unnecessary work for yourself or the computer. Always think about the simplest solution to any problem.
- **Single Letter Names.** Don't use single letter variable names like a, b, or x for your variables, except in the case of something traditional (like k for a loop counter). The module lessons have examples.
- **Descriptive Names.** Use descriptive names for variables like temperature, numDimes or userGuess, not cryptic names like var1 and z.
- **Filtering in Mutators.** When you get to class methods called *mutators*, (set() methods), you are required to filter out bad data by testing the parameters for legal, expected values.
- **No Error Messages in Mutators.** Don't send out error messages to the user (i.e., do not do any input or output) from inside *mutator* methods.
- **Mutators Return Booleans.** *Mutators* always return either **true** or **false**, to indicate to the client whether the set was successful. The client may, or may not, use this information, but the *mutator* must supply it.
- **Keep data private.** Be sure that data that should be private is specifically labeled private. Use mutators and accesors to get and set these variables.
- In **Java** classes, name your main class **Foothill** (does not apply to C++ classes).
- **Avoid material we have not covered in this class or assigned reading.** I know it is sometimes tempting to demonstrate that you know more than I have presented, but if you do that -- and do so incorrectly (*as defined by my upcoming modules*) -- you will lose points. Classic examples are the use of *loops* or *arrays* before we get there. Students who know the mechanics of these, but have never had them correctly presented, sometimes use the wrong kind of loop (*while* vs. *for* vs. *do*), incorrect indentation of the loop, a (never-allowed) *label* in the loop, or an array index with a *literal* int and without the proper accompanying loop. There are other examples. So it's safest to use only the tools included or suggested in the material presented up to the assignment date. (4 - 10 point penalty if not-yet-covered material is used incorrectly).

- **Never use GOTO statements (C++) or use labels (C++ or Java**). You may read about these, but never use them. (-10 points)

Other juicy facts about assignments that affect your standing in the class are:

- **Drop for Non-Participation.** Two (2) consecutive zeros will result in a drop from class. Three (3) zeros, even if they are not consecutive, will result in a drop. This is to protect you from forgetting to drop when you stop attending, but it is also a rule that I will follow. This means if you fall more than two weeks behind in your assignments, without prior notification, I can drop you. I might do so without sending a notice (because of my own deadlines and the many students that I have to track). If you make arrangements with me before the assignments are due, I can usually help. Please don't wait until after, because it will be too late.
- **Final Day.** No assignments are accepted after the Friday of the 11th week (before the Final Exam). There is absolutely no exception to this rule - the course is over and my grades are "in the mail."
- **Don't Submit Too Early.** If you finish early, hold on to the assignment in case someone asks a public question and gets an answer that causes you to want to modify it. Once you hand it in, it's mine. No multiple submissions.
- **Grading Time.** Allow one week from the due date for grading. If you hand in an assignment late, I may not get to it with the original batch of returned assignments, so it could be two weeks before you get it back.
- **Help.** I am happy to give you assistance if you ask me specific questions. Avoid "I don't know where to begin" but take time and figure out where you are lost, exactly.

I know -- this seems like a lot to digest, but it is not as hard as you might think. Your programs are small in the beginning, so there isn't much to check. And by the third or fourth week, when the programs get larger, most of this will be automatic. The rules are here because I am training you to fit in with the professional programming community, not because I am looking for a reason to deduct points.

Section 3 - Style Rules

LABS.3.1 Why Style Rules?

Meditate On This

- A Working Program Is Not Necessarily a Good Program.
- Proper style is more important than a working program.

Most programs do not work the first time and must be debugged. Even working programs are renovated and improved. If a program has a clean, understandable style, then a future programmer can easily debug or augment it. However, if it works but is randomly or improperly indented, then it may be difficult and expensive to modify later. The logic flow of the program may be completely obscured by misalignment of statements.

Most students learn style by observing the way the author indents his code in the examples. This is perfectly acceptable. I have dozens of example programs and fragments for you to use as examples, as does our text.

You are all responsible for proper style from day one. You will lose between one and nine points for style infractions. The later in the course the style errors occur, the greater the penalty. I want to help you avoid losing even a single point.

LABS.3.2 A Perfect Fragment

Here is an example of a perfectly indented code fragment. In the next two pages, I will list the rules that you must apply in order to always end up with style like this.

```
// constructor for class initArray that takes int size ---
public initArray( int size )
{
   int k;

   // bounds check the size
   if (size == 0 || size > MAX_SIZE)
   {
      undefine();
      return;
   }

   // allocate a new string array of size: size
   data = new String[size];
   this.size = size;
   for (k = 0; k < size; k++)
      data[k] = "default string";

   // set instance variable so we know array is valid
   defined = true;
}
```

If you are a beginner, you will not understand this code yet, but the style rules I am about to present should make sense. I'll explain each line today from a style perspective. Then, when we get to a particular coding topic tomorrow, next week or in a few weeks, I will restate the relevant rule in context.

For example, there is a ***for loop*** in the code fragment above, but you don't know what a ***for loop*** is yet. When we cover it in the third or fourth week, we will focus on the style rule for just that construct.

Let's list ***11 important rules*** that apply to this code fragment -- and your programming assignments -- and give an example of how to apply each one.

Section 4 - Rules One through Five

LABS.4.1 Rule One

Rule 1

Do not cram code on the left margin.

Bad

```
// method initArray() takes int size, does error check,
// allocates memory and sets array
public initArray( int size )
{
int k;

// bounds check the size
if (size == 0 || size > MAX_SIZE)
{
   undefine();
   return;
}
// etc ...
```

Better

```
// method initArray() takes int size, does error check,
// allocates memory and sets array
public initArray( int size )
{
   int k;

   // bounds check the size
   if (size == 0 || size > MAX_SIZE)
   {
      undefine();
      return;
   }
   // etc ...
```

The only things that should be on the left margin are the function/method headers (public initArray(...), or class headers:

```
public class Fragment
{
   ...
```

LABS.4.2 Rule Two

Rule 2

Indent *for/if/else/while/switch* bodies.

Bad

```
// bounds check the size
if (size == 0 || size > MAX_SIZE)
{
undefine();
return;
}

for (k = 0; k < size; k++)
data[k] = "undefined";
```

Better

```
// bounds check the size
if (size == 0 || size > MAX_SIZE)
{
   undefine();
   return;
}

for (k=0; k < size; k++)
   data[k] = "undefined";
```

LABS.4.3 Rule Three

Rule 3

Do not indent for no reason.

Bad (In first fragment, comment should be aligned with code below)

```
// bounds check the size
   if (size == 0 || size > MAX_SIZE)  <--bad here
   {
      undefine();
      return;
   }

// allocate a new string array of size: size
data = new String[size];
this.size = size;
   for (k = 0; k < size; k++)     <--bad here
      data[k] = "undefined";
```

Better

```
// bounds check the size
if (size == 0 || size > MAX_SIZE)
{
   undefine();
   return;
}

// allocate a new string array of size: size
data = new String[size];
this.size = size;
for (k = 0; k < size; k++)
   data[k] = "undefined";
```

Bad

```
// allocate a new string array of size: size
   data = new String[size];
this.size = size;
```

Better

```
// allocate a new string array of size: size
data = new String[size];
this.size = size;
```

LABS.4.4 Rule Four

Rule 4

Do not randomly stagger code which should be aligned.

Bad

```
  // allocate a new string array of size: size
data = new String[size];
 this.size = size;
  for (k = 0; k < size; k++)
          ...
```

Better

```
// allocate a new string array of size: size
data = new String[size];
this.size = size;
for (k = 0; k < size; k++)
        ...
```

Bad

```
   int k;

 // bounds check the size
   if (size == 0 || size > MAX_SIZE)
      {
         undefine();
        return;
   }
```

Better

```
   int k;

   // bounds check the size
   if (size == 0 || size > MAX_SIZE)
   {
      undefine();
      return;
   }
```

LABS.4.5 Rule Five

Rule 5

Do not use excessive indentation or extra blank lines.

Bad (too much indentation)

```
   // bounds check the size
   if (size == 0 || size > MAX_SIZE)
   {
          undefine();
          return;
   }
```

Bad (meaningless blank lines)

```
   // bounds check the size
   if (size == 0 || size > MAX_SIZE)

   {

       undefine();

       return;

   }
```

Bad ("double indentation" or "lonely braces")

```
// bounds check the size
if (size == 0 || size > MAX_SIZE)
   {
       undefine();
       return;
   }
```

Better

```
if (size == 0 || size > MAX_SIZE)
{
    undefine();
    return;
}
```

Section 5 - Rules Six through Twelve

LABS.5.1 Rule Six

Rule 6

Do not continue typing on the same line as a "{".

Bad

```
// bounds check the size
if (size == 0 || size > MAX_SIZE)
{  undefine();
   return;
}
```

Better

```
// bounds check the size
if (size == 0 || size > MAX_SIZE)
{
   undefine();
   return;
}
```

LABS.5.2 Rule Seven

Rule 7

Do not create long lines that will cause wraparound in some editors (use 80 chars max).

Bad

```
   if (!(poly = createStartingPoly(theNurb, uCount, uNewKnots, dim, dim,
uControlPoints, uDegree) ))
   {
      putUpAlert("Couldn't allocate starting poly. No non-uniform-rational-b-
spline surface drawn.  Please check your values", FALSE);
      postNurbError(theNurb,GLU_OUT_OF_MEMORY);
      freeLists();
      return 0;
   }
```

Better

```
   if (!(poly = createStartingPoly(theNurb, uCount,
      uNewKnots, dim, dim, uControlPoints,
      udegree) ))
   {
      putUpAlert("Couldn't allocate starting poly. No non"
         + "-uniform-rational-b-spline surface drawn.  "
         + "Please check your values", FALSE);
      postNurbError(theNurb,GLU_OUT_OF_MEMORY);
      freeLists();
      return 0;
   }
```

LABS.5.3 Rule Eight

Rule 8

Don't use tabs in this class.

Tabs expand awkwardly in email or on browsers and create problems. Always substitute about 3 spaces for every tab character you have in your code, or just avoid using the tabs in the first place.

You can either set your editor to automatically use three spaces per indentation level or, after you are done writing, do a global search and replace.

You can tell the difference between a space and a tab in a document by click/dragging the mouse across the region in question. If it selects the whitespace gradually, you have spaces. If nothing selects until you move a large distance, then suddenly the entire region is selected with a ***snap***, then you have a tab.

```
Try selecting this:    and then this: .
```

Which one was a tab and which was a string of spaces?

By now you have seen this rule at least four times: *use tabs, lose points*. I have supplied steps in your ***compiler handout***, and a special section (***next page***) on how to avoid and/or replace tabs in your code.

LABS.5.4 Rule Nine

Rule 9

One space is insufficient indentation.

Use at least three and no more than five per indentation level. Whatever you use for one level, use for all.

Bad (1 space not enough)

```
// bounds check the size
if (size == 0 || size > MAX_SIZE)
{
 undefine();
 return;
}
```

Bad (inconsistent: 3 spaces in one place, 5 in another)

```
for (k = 0; k < size; k++)
{
   data[k] = "undefined";
   for (j = 0; j < k; j++)
       bean[j][k] = "tracking";  <--not consistent
}
```

Better

```
for (k = 0; k < size; k++)
{
   data[k] = "undefined";
   for (j = 0; j < k; j++)
      bean[j][k] = "tracking";
}
```

LABS.5.5 Rule Ten

Rule 10

Use whole words, not single letters, for variables.

Variable names like dimes or temperature are far better than d or t. Do not use single character names. Exceptions are loop counters (n or k are typical) or mathematical or physical constants like *e*, *x* and *y*, if this is a math or physics program.

I use single letter variable names sometimes to help keep the explanations clean and free of specific details. You should almost never use single letter variable names. Your variables should be named to help the reader understand what they stand for.

Good

```
while (roomTemp < desiredTemp)
   roomTemp++;
```

Loop counters, also good

```
for (n = 0; n < 10; n++)
{
   <statement 1>;
   <statement 2>;
   <statement 3>;
}
```

LABS.5.6 Rule Eleven

Rule 11

Use camelCase for methodNames() or variableNames and UPPER_CASE for constants.

Every working group has a convention for naming variables and method names. The most common is **camelCase**, in which the first word is not capitalized, but every other word starts with a capital letter. Symbolic constants (values that you designate using the keyword **final** as not being changeable at run-time) use UPPER_CASE with, or without an underscore between words.

Bad

```
for (k = 0; k < size; k++)
{
   student_name[k] = "undefined";    <--not good variable style
   for (j = 0; j < k; j++)
      score[j][k] = GetScore(j, k);  <--method capitalized
}
```

Better

```
for (k = 0; k < size; k++)
{
   studentName[k] = "undefined";
   for (j = 0; j < k; j++)
      score[j][k] = getScore(j, k);
}
```

Bad

```
final int maxStudents = 100;  <-- constant should be ALL_CAPS

for (k = 0; k < maxStudents; k++)
   studentName[k] = "undefined"
```

Better

```
final int MAX_STUDENTS = 100;

for (k = 0; k < MAX_STUDENTS; k++)
   studentName[k] = "undefined"
```

Exceptions - We will see exceptions to these rules as we proceed, but they will be very easy to understand. For example, class names are capitalized: **CityCenter**, and the constructors for such classes have the same name as the class, **CityCenter()**.

LABS.5.7 Rule Twelve

Rule 12

Supply comments according to your professor's requirements.

As with all of style rules, this one only applies within the confines of this course; when you take another class or join a particular software group, they will have their own guidelines which will almost certainly be different from these, not to mention different from one-another's. This is particularly true regarding commentary.

Typical Comment Requirements of *Other Groups*

These are not necessarily the requirement of this course, but common rules for *other* courses. To see this course's requirements skip ahead to "**This Course's Requirements.**"

1. **When Not to Comment.** If variables or statements are self-explanatory, do not add a comment. "*assign **b** to **a***", or *"print the result*" are examples of this type of unnecessary documentation. The more self-documenting code with well-named variables, the better.
2. **Parameters and Returns.** Above each method declaration, list the ***parameters*** that the method takes and their meanings. Also include the purpose of the ***functional return***, if any, and other lasting effect the method has on the caller (client).

3. **Explanation of Code Fragments.** Above each several lines (two-to-ten, typically), include a brief comment (one or two lines at most) that explains what that code fragment does.
4. **Individual Line Comments.** If a single line is particularly and technically unusual, add a brief comment on the same line indicating its meaning.

***This Course's* Requirements**

In this class, I prefer *minimal* commentary. You are more likely to lose points if you supply too many comments. A well written method with no commentary is perfectly acceptable in my class.

1. **When Not to Comment (*same as above*).** If variables or statements are self-explanatory, do not add a comment.
2. **No Comments for Most Methods or Fragments.** Well named methods with well named parameters and variables should be free of commentary. Code fragments that are clearly designed do not need comments above them. When you feel a fragment needs qualification, the comment above should be restricted to ***one line***.
3. **Comment Alignment.** A Comment for a code fragment must be aligned with code below just as it would be were it an ordinary (e.g. assignment) statement.

You will get excellent practice and guidance with the first set of rules ("*Other Groups*") in other instructors' courses. Here we want to de-emphasize comments in order to encourage well designed and self-documenting code. My experience in many years of teaching coding at this introductory level is that the more comments students include in their code, the more complex and incorrect the actual code is. Specifically, it appears that students often lean on comments as a way to excuse variable and method names that are sub-optimal or logic that is inscrutible without the explanation.

You will get no points for excess commentary, but will lose points for unnecessary, distracting comments or if your code is unclear on its own merit.

LABS.5.8 Final Check

After you paste your homework into a text file for final submission, you should review it to be sure your compiler and text editor have not conspired to add any strange formatting. If they have, you will still be responsible for those errors. Manually fix the errors in your source code.

For more information on style, you should look closely at the examples.

Section 6 - Detecting and Killing Tabs

Most of my students are able to use the previous modules to avoid tabs. However, there are always a few who, for some reason, can't get the settings right, or, worse, believe that because the code looks correct on *their* screens, it means there are no tabs or that it should look right on

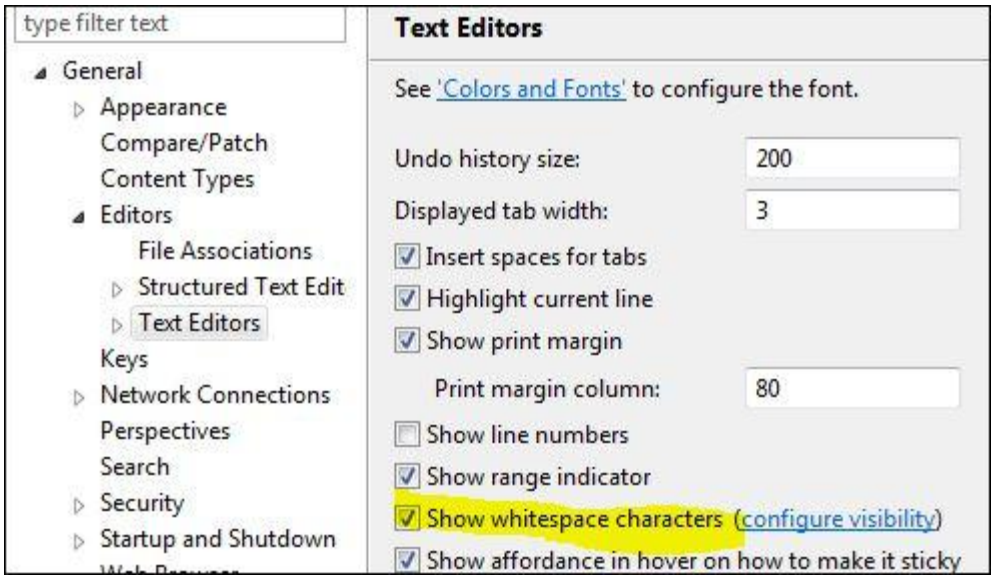
all screens. I have included this section to help you ***find the tabs that you might not believe are there, and kill them manually***, if they have snuck in.

The steps below should not normally be needed. But if you are getting dinged for tabs, you'll need to see them with your own eyes and manually get rid of them. Here's how to do it.

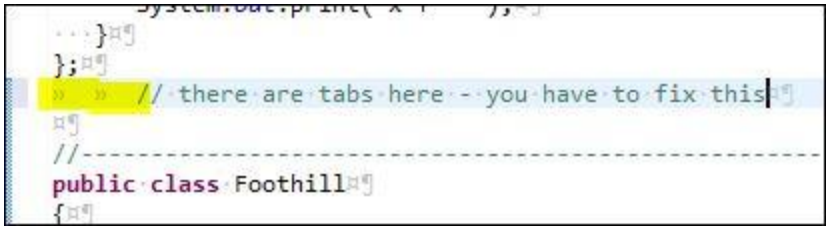
LABS.6.1 The Steps for Eclipse

This works on Eclipse, but if you are using a different IDE or editor, you'll have to learn the corresponding settings for the application you have chosen. Any good editor will have these options.

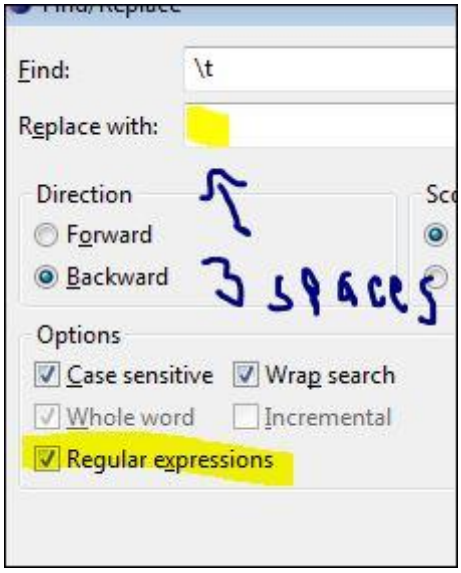
1. Turn on the "view whitespace" option in your editor so you can clearly see the tabs. Click on **Window → Preferences → General → Editors → Text Editors**, then check ***Show whitespace characters***.



2. Look carefully to see if you have any tabs. They appear as "»" characters. On the other hand, three spaces, for example, will appear as "·". Look at this screen shot which shows the tabs (bad) highlighted, but spaces (good) appearing a couple lines above:

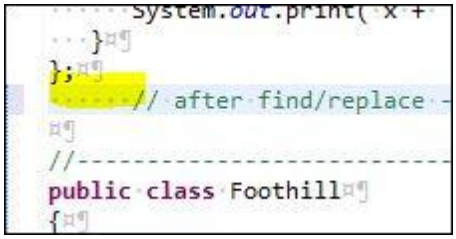


3. Do an **Edit → Find/Replace** and find tabs ("**\t**") to be replaced with three or four spaces (" "). *Crucial Detail: for this to work, you must check the **Regular Expressions** box*:



(Make sure you have the forward or backward bullets correctly set.)

4. Go back to your source and see the result:



5. When you have killed all the tabs, uncheck the "view whitespace" box, to get back to normal view.

LABS.6.2 Choose to Improve, Rather than Argue

If you're having difficulty following the earlier rules, and especially this solution, that's an indication that you are stuck on small problem -- no shame, but you should want to work through it. It has nothing to do with tabs, and everything to do with attitude: accepting ones own mistakes and being willing to make the adjustment. All successful people embrace finding and fixing their errors. (For example, at the end of every quarter, I take all my students' reports of typos and mistakes, and address every one, without hesitation.) Let this module, and any follow-up questions in the forums, be your keys to unlocking the door that may be holding you back in programming and in your career.