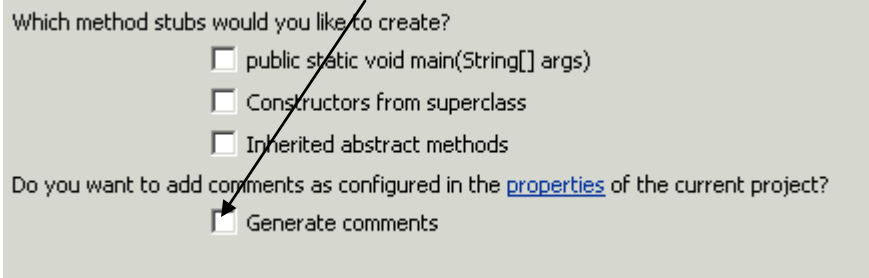


Initial Coding Guidelines

This handout specifies coding guidelines for programs in ITK 168. You are expected to follow these guidelines precisely for all lecture programs, and for lab programs. As your knowledge of Java increases, we will add more required elements to the guidelines.

Opening Comment Blocks

All code files should contain 2 opening comment blocks that specify information about the class or program defined in that file. The first opening comment block holds general information that is of interest only to someone looking at the code (and will appear green in Eclipse). The second block will be a Javadoc comment (and will appear blue in Eclipse). This will contain information for the external documentation of the program or class. This comment block will appear at the top of the class API. You can get Eclipse to generate a starting comment by checking this box (at the bottom of the new class window):



General comment blocks start with `/*` and end with `*/` while Javadoc comments start with `/**` and end with `*/`. Notice, the extra `*`'s used to start each line are not really necessary, but help make the comment block stand out. Follow this example carefully when writing your opening comment blocks. Use your lecture section and instructor for lecture programs and your lab section and instructor for lab programs.

```

/*
 * Filename: HelloWorld.java
 * Created on August 31, 2004
 *
 * ULID: mecaliff
 * Course: ITK 168
 * Section: 7
 * Instructor: White
 */

//Please place import statements between these 2 comment blocks
import something.somethingElse.*;

/**
 * This is a very simple program to demonstrate
 * the basic structure of a Java program and how
 * to print to the console window.
 *
 * @author Mary Elaine Califf
 */

```

Notes: In order for your Javadoc API to be created properly, the import statements must appear above the class Javadoc comment block. Also within the Javadoc comment block the `@author` tag (and `@version` tag if used) must appear at the bottom of the block with a good class description at the top of the block. This description is what users will read to understand how and why to use your classes so make them as clear and complete as possible.

In addition to opening comments, each **public** method (or service) **except main** must have a Javadoc comment. The Javadoc comment should have a good description of the method (fully explain what it is used for, but do not discuss how it is coded). If the method has one or more parameters your Javadoc must use @param for each parameter. The name following this tag **must match the parameter name** and should be followed by a description of the expected contents. Remember Java doc comments start with /** and end with */. Here is an example:

```
/**
 * Special Constructor for objects of class Employee
 *
 * @param number - a string built from their first initial + last initial
 *                + 4digits
 * @param name - a string holding the full name formatted Last, First
 *              case is not important
 * @param pay - an integer holding their yearly salary (full dollars only)
 *
 */
public Employee(String number, String name, int pay) {
    ...
}
```

If the method has a return type (other than void) use @return to describe the returned value. If the return value could be one of a few choices, you should describe all possible return values – however you can only code @return one time. Here is an example:

```
/**
 * compareTo method compares 2 Student objects based on their GPA
 *
 * @param stu - a Student object to compare to the calling object
 *
 * @return a negative number if the calling Student has a lower GPA or zero
 *         if the 2 Students have the same GPA or a positive number if the calling
 *         Student has a higher GPA
 *
 */
public int compareTo(Student stu) {
    ...
}
```

Inside your methods you should add internal comment lines to clarify your code as necessary. These comments are used to help explain complex code or divide code into blocks. A good programmer writes comments before they ever write a line of code! Single line comments start with // and do not need a closing tag - but these comments cannot wrap to multiple lines. If the comment covers multiple lines, each line must start with //. There are many examples of inline commenting throughout your text.

Braces and Indentation of code

Curly braces are used in Java to enclose blocks of code: classes, methods, control blocks. There are other ways to format these braces, but this is what we expect. Everything inside the braces should be indented one level.

```
public class ClassName
{
    public ClassName()
    {
        // constructor
    }
    public void method()
    {
        // method code
    }
}

if(condition)
{
    // if statements
}
else if(condition)
{
    // if statements
}
else
{
    // else statements
}

while(condition)
{
    // while statements
    if(condition)
    {
        // if statements
    }
    // other statements
}
```

You can and should let Eclipse work for you in maintaining clean code. First set up Eclipse to conform to our expected format rules. (See the document “Let Eclipse Work for You” on the class web site help page. <http://www.itk.ilstu.edu/FACULTY/sawwhite/itk168/help.htm>.) When you add and remove code from your programs you will often end up with unaligned code. Eclipse will automatically clean up your code for you using the following steps:

- **Select all using ctrl-a**
- **Right click anywhere in your document**
- **Choose source -> format**
- **-or- simply use ctrl-shift-f as a single keystroke shortcut**

Make sure you clean up your code often. Do not wait until you are ready to submit (but be sure to do it just prior to submitting also). Keeping your code clean and properly aligned will help you debug your control structures. It is easy to miss a close curly brace and that one missing brace will completely destroy your code!

General Programming Hints:

Errors considered minor on early programs, will later be considered major errors and will cost you greatly. If you lose points for something on a program, make sure you understand what was wrong and how to fix it. Do not continue to lose points for the same mistakes. Here is a list of common errors from past programming assignments that you should avoid:

- Missing or incomplete comment blocks
 - Remember you must have 2 opening comment blocks as described at the start of this document
 - Every **public** method (including constructors, but excluding main) must have a Javadoc comment block with @param for each parameter and @return if the return type is anything other than void
 - Every **private** and every **protected** method should have a general **non-javadoc** comment block. These methods are not made public, are not included in the API, and therefore should not have a Javadoc comment.
 - Longer or complex methods should have a reasonable amount of internal commenting
- Poor variable names
 - Choose descriptive variable names that make sense for all variables created in your projects
 - Do not use one letter or one letter + one digit for any variables other than the counter in a for-loop (and then we typically choose **i** as the control variable)
 - Remember you do not need to give Walls or Things names as they can be created without a calling variable using
`new Wall(ny, 3, 5, Direction.EAST);` // where ny was previously declared as a City object
- Missing or incomplete class diagrams (**CD**)
 - Class diagrams will be discussed in class often and should be easy points so do not forget to include them. They are to be drawn using MSWord’s drawing tools or similar tool – not hand drawn!
 - Class diagrams have 3 sections – even if the middle is empty because we are not adding data, it should be drawn
 - All 3 sections are connected – not separated
 - Class diagrams must match your final classes so update them just prior to submitting
 - You do not need to complete any diagrams for code that is given to you (unless you were required to change or add to it) – this includes the becker.jar classes
 - You **do** need to indicate inheritance – **is-a relationships** – although just the immediate parent of your class. If extending RobotSE, simply draw a rectangle **above** your class diagram, write RobotSE in it and draw an arrow pointing **from your class to the Becker class**
 - In addition to inheritance relationships you will be expected to indicate **has-a** and **uses-a** relationships as necessary in later programs.

- Poor design
 - You should be applying everything you learn in class and on previous programs to the current program design.
 - Do not write duplicate code
 - Not within a single method
 - Not within multiple methods of a single class
 - And not within 2 child classes of the same parent class
 - Take advantage of your super class. If extending RobotSE, do not call the move method in consecutive statements when you should call the move(int) method; do not code 2 left turns when you should call turnAround; etc
 - Think before you code so you are writing concise code. Do not take 12 lines of code when you can accomplish the same thing in 3. For example

Do This:

```
//move until you are
//standing on a thing that
//can be picked up
while(!this.canPickThing())
{
    this.move();
}
```

NOT this:

```
boolean answer = false;
while(answer == false)
{
    if(this.canPickThing() == true)
    {
        answer = true;
    }
    else
    {
        this.move();
    }
}
```

- **Do NOT compare predicates to the words true or false!!**

Do This:

```
while(!this.canPickThing())
{
    this.move();
}
```

NOT this:

```
while(this.canPickThing() == false)
{
    this.move();
}
```

- Use stepwise refinement **to design your program before you code!**
 - You should have small methods that do one thing
 - drawSide – draws one side of a square
 - These methods should be called by larger methods that do one slightly more complex thing
 - drawSquare – calls drawSide 4 times turning between calls
 - Then these methods should eventually be called by a single method that completes the problem
 - drawDiamond – calls drawSquare 4 times turning between calls
 - If you code without thinking through the design, I guarantee one (or both) of the following will be true
 - Your design will be poor and you will lose points
 - You will have to redesign your program several times before it is finished, taking much longer to code than necessary

- **General Comments:**

- **Do not create a package name – place all files in the default package for easier grading**
- **Do not zip a folder – select all required files (remember to select the .java files NOT the .class files) and zip the files using the built in zip utility**

Pseudocode

Pseudocode is simple English phrases NOT Java code. However, pseudocode resembles Java code – one statement per line, indentation and white space used to make it easy to read, specific format requirements.

There are lots of ways to write pseudocode, but I expect you to follow these rules:

- Use start and end tags for all methods, IF blocks, and Loops
- Body of methods, and control structures should be indented one level
- When making a call to another method in your class, write the exact method name as a call and place it in a rectangle. If you have difficulty drawing rectangles you can put the call inside square brackets like this ... [methodCall]
- Write descriptions that correspond to existing method calls or use the actual method calls for Java or Becker classes
 - Move 1 forward **or** move **but not** jump one
 - Otherwise I would expect to see a method in your class called jumpOne
- Use white space between method blocks and make the method names **bold** so they stand out
- **Do not** use any parenthesis or curly braces
- Do not use any shorthand used in coding
 - Write “increment num by 1” **NOT** “num++”
 - Write “assign the sum of num and total to total” **NOT** “total = total+num” **or** “total += num”
- Use the generic term LOOP not WHILE, since loops can be coded in a number of formats
 - Typically LOOP UNTIL is coded as a WHILE loop and
 - LOOP # TIMES is coded as a FOR loop
- When using variables, tell their type toward the top

Here is a sample method...

calculateAverage

use integers named num, count, and total

initially set total and count to zero

use double named average

request a number from the user using -999 to quit

get user entry and store in num

LOOP until num holds -999

 add num to total

 increment count

 request a new number using -999 to quit

 get user entry and store in num

end loop

calculate average as total divided by count

print average to screen with a label

end calculateAverage