Best Practices
Version 1.3 (new items in **red, bold font**)
Week 03

Naming Conventions:

- Package names follow the conventions:
  - All lowercase letters to avoid conflict with the names of classes or interfaces
  - Use the 'reversed Internet domain name' to being the package names
    For example: `edu.westga.cs6311`
- Classes always start with a capital letter (and are the *only* identifiers that start with a capital letter) For example: `String`, `Integer`, `Random`
- Methods and variables will always have more than 1 letter
- Methods and variables always start with a lowercase letter
- Always give your identifiers (packages, classes, methods, and variables) meaningful names. Instead of using variable names like `'num'`, please type out what the variable represents, like `'numberOfWords'`
- Always give constants names using all capital letters, with each word separated with an underscore. For example, TOTAL_NUMBER_OF_STUDENTS
- The `'this'` keyword
  When referencing instance variables inside your class, it is a best practice to refer to them using the `'this'` keyword, as in `this.totalPurchase`. Doing so helps to eliminate the confusion between instance variables (that use `'this'`) and local variables (which do not use `'this'`).

- **JUnit test classes will always begin with the word 'Test', followed by a description explaining what part of the code is being tested (like TestCreateCar or TestTruckTurnLeft)**
- **JUnit test methods will always begin with the word 'test' and include a descriptive ending to explain what is being tested and the expected result (like testBallWithColorYellow or testMovingRight1SquareFrom2IsSquare3)**
  REMEMBER: The goal of JUnit testing is that we're writing automated tests – LOTS of them. This means that we'll be able to click a button and see the results of the tests. If the pass, great! But if they fail, we're going to want as much information as we can get as to why it failed. The test class and method names will be BIG clues to us as to what's going on.

Testing

- **There should always be enough tests for every piece of functionality in your model classes to confirm they function correctly.**
- **Test classes should be in their own package that ends with 'testing' and the name of the class being tested**
- **Do not include any output in your test method code (automated tests mean that no human needs to look at the console to inspect the results)**
- **Do not test any methods that require user input (again, they are supposed to be automated tests)**
- **When writing test methods, you should have exactly 1 'assert statement' per test method**

Writing Classes

- Although the order of the instance variables and methods listed in the class doesn't matter, Java convention says that we list our instance variables first, followed by the constructor(s), followed by the other methods
- The concept of encapsulation says that we are to expose what the class can do, not how it is done. This means that typically we're going to declare our instance variables as private members and provide public methods to access them, unless you have a good reason to do things differently (so far we don't have any such reason). More details about encapsulation can be found in the textbook.
- Avoid storing duplicate information in multiple class instance variables. For example, if you have a class called `Circle`, you would want to keep up with the `Circle`'s radius with a variable called, `radius`. But don't declare instance variables `diameter`, `circumference`, or `area` as each of these can be calculated based on the radius. Storing duplicate information is a BIG source for bugs.
- Your author suggests you replace 'magic numbers' with static final constants, declared along with the instance variables. I will ask that you please only do this if that constant is needed in more than one method inside the class. (If it is only used in one method inside the class, then it should be a local constant, declared for use just in that one method)
- Methods should always begin with error checking to confirm that the preconditions are confirmed

User Input

- Please use the `Scanner` class to get user input from the keyboard. I appreciate that there are other classes available for use, however those classes were the only ones available for us to use prior to the `Scanner` class in Java 1.5.
- Please follow my request and ALWAYS use the `Scanner`'s `nextLine` method for ALL user input. Note that because this method will return a `String` value, you will be forced to explicitly call a method to convert this input `String` to a numeric value. I promise you that it is in your best interest to use `nextLine` in this course.
- If you are writing a console application, please stay consistent and use console input (a `Scanner` object) and console output (`System.out.println`)
- If you are writing a GUI application, please stay consistent and use GUI input and GUI output (the `Alert` class).
- Please convert String values to whole numbers using the `Integer.parseInt` method.
- Please convert String values into floating point numbers using the `Double.parseDouble` method.

Indenting and Whitespace

Including appropriate indenting is really going to make the code much easier to follow. Remember, the compiler doesn't care about whitespace, so if you wanted you could type your entire program on a

single line. But don't do this – it'll make the code nearly impossible to read (and you'll get a 0 for the exercise!). Indenting is something that can take some practice. Basically the rules are:

- Start the new line so it is left-aligned with the line of code above it, unless there's a reason not to. So far, those reasons are:
  - It's the first line inside a class
  - It's the first line inside a method

Remember, Eclipse will help you with us if you'll use Control-Shift-F

If Statements (these guidelines are enforced by Checkstyle)

- Always use curly braces to surround your if clauses and your else clauses (even if they are just a single statement)
- Always indent the code 'inside' the if clause (the true path) and inside the else clause (the false path)
- Always place a single blankspace character before the opening parenthesis and after the closing parenthesis of an if statement (see example below)
- Always place the opening curly brace of an if or an else clause on the same line as the if or the else keyword (see example below)
- Always place a single blank space, then the keyword else, and the single blank space, and finally the opening curly brace ALL on the same line for an else clause (see example below)

```
if (some Boolean expression) {
     // some things to do when the expression is true
} else if (some different Boolean expression) {
     // some things to do when this second expression is true
} else {
     // some things to do when none of the expressions evaluate to true
}
```

switch Statements

- Appropriate for situations where there are a small, discrete number of cases.

```
switch (some expression)  {
      case value1:
            1 or more statements
            break;
      case value2:
            1 or more statements
            break;
      ...
      default:
            1 or more statements
}
```

for Statements

- Use a for loop (only!) for counted loops – loops where you know at design time the exact number of iterations the loop body will perform
- Always use a blank space between each piece of the for statement header
- Always place the opening curly brace on the same line as the for header

```
for (initialization; some Boolean expression; condition update) {
     // some things to do when the expression is true
}
```

while Statements

- Use a while loop (only!) for pre-test, conditional loops – loops where you do not know at design time the exact number of iterations the loop body will perform
- Always use a blank space between each piece of the while statement header
- Always place the opening curly brace on the same line as the while header

```
while (some Boolean expression) {
     // some things to do when the expression is true
     // Don't forget to update so the expression can change
}
```

do / while Statements

- Use a do/while loop (only!) for post-test, conditional loops – loops where you do not know at designed time the exact number of iterations the loop body will perform, but you want to require that the loop body run at least one time
- Always place the opening curly brace on the same line as the 'do'
- Always place the closing curly brace on the same line as the while statement footer
- Always use a blank space between each piece of the while statement footer
- Do not forget to include a semicolon after the while's condition in the footer

```
do {
     // some things to do
} while (some Boolean expression);
```

foreach ('Enhanced for') Statements

- Use a foreach loop any time you want to process *every* element in a collection (`ArrayList` or simple array)
- Be sure to place a single blank space character between each piece of the for statement header

```
for (variable declaration : collection variable name) {
     // some things to do with each element in the collection
}
```

Your textbook uses the variable name '`element`' for its local variable in the foreach loop. Myself, I like the variable name '`current`', because it seems to help remind me that (1) I'll be looking through every single element in the collection and (2) I'm only currently looking at exactly one element. Honestly, you're free to use whatever variable name you like here – just make sure that it's meaningful (and `i` or `j` is not meaningful in a situation like this)

Collection Types:

- Be sure to use a simple array for collections where you know exactly how many elements are needed at design time
- Be sure to use an `ArrayList` for collections that may grow or shrink in size at runtime.
- Be sure to use a foreach ('enhanced for') loop when you want to process every element in a collection.

Commenting:

- Every class will start with a Javadoc comment listing:
  - The purpose of the class
  - An `@author` tag, followed by your first *and* last name
  - An `@version` tag, followed by today's date (at least the date that you first started the exercise)
- Every method (including `main`) will start with a Javadoc comment listing:
  - The purpose of the method
  - For each parameter:
    - An `@param` tag
    - The parameter's variable name
    - A brief description of what the parameter variable is holding

NOTE: For the main method, the Javadoc comment will almost always be:

```
/**
 * This method is the entry point of the application
 *
 * @param args  Command-line arguments, not used
 */
public static void main(String [] args) {
```

Be sure that your Javadoc comments line up with the method header they are describing. Using Eclipse's Control-Shift-F will do this for you!

Submitting File to Moodle:

- Eclipse files that are submitted must be Zipped (because there are many, MANY files/folders created by Eclipse) using 7-Zip (you'll know why to use 7-Zip a little later)

- Zipped file names should be 6311*YourName*Lab*XX*.zip
  For example: `6311SamSmithLab01.zip`
- Most Moodle assignments are set up to accept exactly 1 file. The good news is that you can upload multiple times. When you do so, the previous submission is overwritten. This is useful in that you can use Moodle as another backup or as a way to transfer files from one computer to another. Because of this you are encouraged to submit early and often (only the last file submission will be seen or graded).
- After you upload your file(s) to Moodle, you will then be able to click the assignment link again (as you did at the start of the submission process) and see the actual files that you uploaded. It's recommended that you download this file and double-check that the correct thing was uploaded.

Version Control

- All development should be done using the version control software so that individual 'commits' can be stored.
- Developers should stop and commit the changes to their repository when any significant changes have been made to their code.
- Each commit should contain an appropriate commit comment describing the changes that were performed. Note that it is *not* appropriate to state something like, "Completed Step 10b" or "Fixed bugs". Instead, the comment should describe to an outsider what modifications were made.
- Although time/date stamps will be stored with each commit, these will *not* be valid as proof that work was completed on time. Only work submitted on time though Moodle will be accepted. Please refer to the Course Syllabus for more details about the Course Late Policy.