Important guidelines and other things that we assume you learned in 10A

1. Don't cheat! Occasionally students are expelled or otherwise penalized because they submit problem set solutions that they found online or because multiple students submit essentially identical code.
2. Put your name at the top of each submitted file.
3. Faithfully follow conventions:
    a. Classes' names begin with an upper-case letter.
    b. Methods' names and non-final variables' names begin with a lower-case letter.
    c. Final variables' names are all upper-case.
4. Indent consistently.
    a. Four spaces is a good unit of indentation.
    b. To ensure that other people see your code as you intend it to be seen, configure your text editor to use spaces instead of tabs.
5. Align curly braces consistently. There are two curly brace styles. Pick one and use it consistently.
    a.
```
class Foo {
    public static void main( String [] args ) {
        System.out.println( "Hello World." );
    }
    ...
}
```
    b.
```
class Foo
{
    public static void main( String [] args )
    {
        System.out.println( "Hello World." );
    }
    ...
}
```
6. Programs must contain comments.
    a. Per the syllabus, put your name at the top of each file.
    b. Every class must be preceded by a comment that describes the class.
    c. Every method except `main()` must be preceded by a comment that explains what the method does.
    d. For a, b, and c it would be a very good idea to use Javadoc-style comments.
    e. Within a method, use `//` comments to explain sections of code whose purpose and logic is not readily apparent.
7. Use lucid variable names and method names. Avoid one-letter names except for `for` loop indices.

Important guidelines and other things that we assume you learned in 10A

8. Put a few blank lines between methods. Be consistent. Put one or more blank lines between a method's logical sections of code. Consider a method's logical sections of code to be analogous to an essay's paragraphs.

9. There are two kinds of Java classes:

    a. *Template classes* are classes that model something. In a 10A context, **`TicTacToeBoard.java`** is a template class that models a Tic Tac Toe board. All knowledge about the board is encapsulated in the **`TicTacToeBoard`** class. That means that that class knows which spaces on the board have X's, which have O's, and which are unoccupied. That means that that class can answer questions like "X wants to claim a square. Is that Ok? If it is, update the board accordingly. If it's not, let me know." "O wants to claim a square. Is that Ok? If it is, update the board accordingly. If it's not, let me know." "Was there a winner? Was it X? Was it O?" "Was there a tie?"

    b. *Program classes* are applications that are executed by the user. In a 10A context, **`TicTacToe.java`** is a program class, because the user executes that class to play the Tic Tac Toe game. The program class manages the game by communicating with the user and by communicating with the **`TicTacToeBoard`** template class to ask the questions mentioned in 9a.

    When a user executes a program class, execution begins in a method that has this signature:
    `public static void main( String [] args )`

    c. Template classes should be designed to simplify the design of the program classes that use the template classes. E.g., the **`TicTacToeBoard`** template class should present an easy-to-use interface that answers the kinds of questions that make it easy to write the **`TicTacToe`** application.

10. In program classes, **`main()`** should either be the class's first method or the class's last method.

11. An application's **`main()`** method should not contain much application logic. Instead, the bulk of the application's work should be outsourced to other methods.

12. A program class should not use non-**`final`** class-level variables, because such variables make the program harder to debug and modify.

13. A template class's class-level variables can be *instance variables* or they can be *static variables*.

    a. Instance variables are identified by the omission of the keyword **`static`**. Instance variables define the state of an individual instance of the template class. Whenever a new instance of the class is created, it's given its own area of memory (casually referred to as the instance's *blob*) to hold its instance-specific copy of the class's instance variables.

    b. Static variables are identified by the keyword **`static`**. They are not associated with a particular instance of the class; instead a single copy of the variable is accessible to *all* instances of the class.

    c. Class-level variables should not be **`public`**, because otherwise you have no control over what your class's users can do to them. An exception to that rule is that **`final`** class-level variables can be **`public`** if that provides a benefit to your class's users.

Important guidelines and other things that we assume you learned in 10A

14. A template class's methods can be *instance methods* or they can be *static methods*.

    a. Instance methods are identified by the omission of the keyword **`static`**. Instance methods are executed in the context of a specific instance of the method's template class.

```
Foo bar = new Foo();
double rate = bar.getInterestRate();  // getInterestRate() sees
...                                   // bar's instance variables.
class Foo {
    private double interestRate;

    public double getInterestRate() { // Return instance's copy
        return this.interestRate;     // of interestRate.
    }
}
```

        i. When instance methods reference their class's instance variables, it adds clarity to qualify the reference using dot notation with **`this`** to refer to the *implicit argument*.[1]

           `this` can be used to differentiate a reference to an instance variable from a reference to a local variable with the same name. That can be useful, e.g., in a constructor:

```
class Foo {
    private double interestRate;

    public Foo( double interestRate ) {
        this.interestRate = interestRate;
    }
}
```

        ii. It's ok for instance methods to access their class's instance variables directly instead of going through accessor (getter) or mutator (setter) methods.

    b. Static methods are identified by the keyword **`static`**. Static methods are not associated with a particular instance of their class.

        i. Static methods can access their class's static variables or static methods without doing anything special, but …

        ii. Since they are not associated with a particular instance of their class, **`this`** is undefined inside of static methods and they cannot access their class's instance variables or instance methods.

    c. Methods should be **`static`** unless they need to access instance variables or instance methods, or they need to access other methods that need to access instance variables or instance methods.

    d. Only make a template class's methods **`public`** if there is a specific benefit to exposing the method to the class's users. A template class's constructor methods are usually **`public`**. A template class's helper methods are usually not **`public`**.

    e. Program classes' methods are usually not **`public`**, except for **`main()`**.

---

[1] The implicit argument is the class's instance through which the referencing method was invoked. In 14a, **`getInterestRate()`**'s implicit argument is **`bar`**.

Important guidelines and other things that we assume you learned in 10A

15. When an instance of a class is referenced in a context that expects a **`String`**, then the value returned by the class's **`toString()`** instance method is used.

```
Foo bar = new Foo();
System.out.printf( "%s", bar ); // %s = bar.toString()
...
class Foo {
    public String toString() {
        return "string representation of this";
    }
}
```

16. Template classes should always explicitly include a zero-argument constructor.

17. If a template class has multiple constructors, minimize code repetition by using the **`this( ... )`** mechanism for referencing another constructor.

```
class Foo {
    private double interestRate;

    public Foo() {
        this( 0.0 ); // Default interest rate is 0.0%
    }

    public Foo( double interestRate ) {
        this.interestRate = interestRate;
    }
 }
```

18. Every constructor should cause all of the new instance's instance variables explicitly to be initialized. Classes with multiple constructors can best facilitate this requirement using #17.

19. There are two styles of **`switch`** statement case labels:

    a. "colon case" labels:

```
switch( switchVar ) {
    case val1:
        ...
        break;
    case val2:
        ...
        break;
    ...
    default:
        ...
}
```

    When using "colon case" labels, always terminate each **`case`** with **`break;`** unless the **`case`** unconditionally executes a **`return`** statement or unless you actually want a **`case`** to flow into the next **`case`**.

b. "arrow case" labels:

```
switch( switchVar ) {
    case val1 -> { ... }
    case val2 -> { ... }
    default   -> { ... }
}
```

Regardless of the case label style, always include the `default` case.

20. Switch expressions let you embed a switch statement into an expression:

    a. Using "colon case" labels:

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY; }
static int lettersInDayName( Day whichDay ) {
    int numLetters = switch( whichDay ) {
        case MONDAY: FRIDAY: SUNDAY:
            yield 6;
        case TUESDAY:
            yield 7;
        case THURSDAY: SATURDAY:
            yield 8;
        case WEDNESDAY:
            yield 9;
        default:
            yield 0;
    };
    return numLetters;
}
```

    b. Using "arrow case" labels:

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY; }
static int lettersInDayName( Day whichDay ) {
    int numLetters = switch( whichDay ) {
        case MONDAY, FRIDAY, SUNDAY -> { yield 6; }
        case TUESDAY                -> { yield 7; }
        case THURSDAY, SATURDAY     -> { yield 8; }
        case WEDNESDAY              -> { yield 9; }
        default                     -> { yield 0; }
    };
    return numLetters;
}
```

21. Arrays are Java's analog to Scratch lists. A 3-element array whose elements are Strings can be declared and initialized via:

```
String [] foo = new String[3];
for( int i=0; i<3; i++ ) { foo[i] = "String" + i; }
```

    or by using Java's "array initializer" syntax:

```
String [] foo = { "String0", "String1","String2" };
```

Important guidelines and other things that we assume you learned in 10A

22. This course emphasizes stepwise refinement. Stepwise refinement is to software engineering as multiple drafts is to prose. The goal of stepwise refinement is to make your code as lucid and well-organized as possible.

    a. An important characteristic of lucid code is that it has been decomposed into reasonable-sized methods. Strive for methods that fit onto at most two screens of text.

    b. A method should generally accomplish one primary task.

    c. In achieving its primary task, a method should not cause side effects to occur. E.g., avoid "oh, by the way, when you execute `printResultsTable()`, everybody in the table receives a 5% raise."

    d. Sometimes it's easier to craft complicated solutions to problems, than it is to craft simple solutions to problems. Also, sometimes you might be tempted to provide a "tricky" solution or an unnecessarily "fancy" solution, instead of a simple, straight-forward solution. This course places high value on creating simple, straightforward solutions!

23. Recursive methods must define at least one base case, which is when the method returns the answer without calling itself.

24. Recursive methods also define a recursive case, which is when it doesn't encounter a base case. The recursive case computes the answer by calling the method with arguments that get the method closer to a base case in order to guarantee that the method will eventually reach a base case.

25. Recursive methods must be self-contained.

    a. That means they must not use non-final class-level variables.

    b. That means that they must not depend on processing outside of the method, e.g. to remove non-letters from a `String` argument. Occasionally a problem will specify exceptions to this requirement.

26. If you are asked to write a method, the problem usually asks you also to write a program that convincingly demonstrates the method by executing the method with a variety of arguments that shows the method to work for all relevant cases.

    a. Unless the problem set says otherwise, you must not require the person running the test program to create the test cases. Instead, your demo program must exercise the method with test cases chosen by you.

    b. Your demonstration program and its output must be implemented so that we don't need to examine your code to understand the demo program's output. I.e. the output should identify the test case that produces a particular result.

```
String result = arrayPrint( removeLowest( 59, 92, 93, 47, 88, 47 ) );
System.out.printf( "removeLowest( 59, 92, 93, 47, 88, 47 ) = %s\n",
                   result );
```

    c. When the assignment provides sample test cases with sample output, you should include those test cases (perhaps with additional test cases) in your testing. Your code should produce the same output that is shown in the assignment. E.g., in LowestGrade.java you should demonstrate that `removeLowest( 59, 92, 93, 47, 88, 47 )` returns an array that `Arrays.toString()` or your `arrayPrint()` would render as `"[59, 92, 93, 88, 47]"`.

Important guidelines and other things that we assume you learned in 10A

27. Although you are allowed to use Integrated Development Environments (IDEs) in this course, be sure that your code can be compiled and executed **independently** of any IDE-induced dependencies, because we will simply copy your code to an arbitrary folder on our personal system, compile it with no options, and execute it. If an IDE-induced dependency prevents your code from compiling or executing on our personal system, you will lose points. And as is pointed out on the pset submission pages the penalty for non-compilation is severe.

    a. For example, the NetBeans IDE inserts `package` statements in the code it produces, that reflect the folder structure and compiler options used by NetBeans. Our test environment will not reflect those things and the `package` statement will cause the compilation to fail.

28. Faithfully follow the specifications in the problem set.

    a. Be sure to use the specified filenames, method names, argument lists, etc. Remember that Java is case-sensitive, so, e.g., a method named `Foo()` is different from a method named `foo()`.

    b. If a problem doesn't strictly dictate a method's specific signature, you need to exercise judgement to choose the signature that's best for the end user. If your signature is unfriendly to the end user, you can lose points.

    c. Be sure that the programs and methods do exactly what is specified. Not less; not more.

    d. Do not use untaught material to solve homework problems. If you're unsure about using something, ask.

    e. Even if a problem says something like this ...

        Here are some details regarding how we suggest you organize your program. If you decide to deviate from this design, please carefully document and explain what you have done.

        ... **do not** deviate from the suggested design. This is important because our test harnesses assume that your submission will adhere to the design in the problem. If you deviate from the suggested design, you **will** lose points.

29. Unless a problem explicitly specifies otherwise, if a problem says to create a method `foo()` then executing `foo()` should not generate screen output.[2] E.g., `isPalindrome()` should not generate any screen output. The method is only supposed to return a `boolean` value. Imagine if a program called that method 100,000 times. The user would not want to see 100,000 unexpected lines of output from that method.

30. Always use curly braces, even if the compiler doesn't strictly require them. E.g., instead of this:

    ```
    if (foo) i += 5;
    ```

    say this:

    ```
    if (foo) { i += 5; }
    ```

    Doing so makes your code easier to understand. It also makes your code more reliable, because frequently you'll decide later to add statements that necessitate including the curly braces. If they're already there you won't accidently omit them.

[2] Unless, of course, we're talking about `main()`, because program classes almost always produce output.

Important guidelines and other things that we assume you learned in 10A

31. Instead of this:

```
if ( foo == true ) { ... }    or    if ( foo == false ) { ... }
```

say this:

```
if ( foo ) { ... }            or    if ( !foo ) { ... }
```

32. Instead of this:

```
if ( foo ) { return true; }
else       { return false; }
```

say this:

```
return foo;
```

33. Instead of this:

```
if ( foo ) { bar = true; }
else       { bar = false; }
```

say this:

```
bar = foo;
```

34. Instead of this:

```
i = i+1;    or    i = i-1
```

say this:

```
i++;        or    i--;
```

35. Instead of this:

```
i = i+n;   or  i = i-n;   or  i = i*n;   or  i = i/n;   or  i = i%n;
```

say this:

```
i += n;    or  i -= n     or  i *= n     or  i /= n;    or  i %= n;
```