Indentation

Properly indented code is much easier to read than sloppily indented code. Your TA will appreciate indented code when grading, and you'll appreciate not losing points for indentation (: Indent your code using the following rule. Most structures in Java start and end with curly braces { } (classes, methods, loops, conditional statements, etc). Everything within a set of curly braces should be indented one level further than the curly braces were. For example a class has no indentation, a method in a class has 1 level of indentation, a loop in a method in a class has 2 levels of indentation and so on. A level of indentation is usually 3 or 4 spaces (It doesn't matter which you pick, just be consistent. jGrasp default is 3). With 3 space indentation code should look like this:

```
public class BeautifulIndentationExample {
   public static void beautifullyIndentedMethod() {
      for (int i = 0; i < 5; i++) {
         if (i % 2 == 0) {
            System.out.println("Inside 4 braces, Indented 12 spaces");
         }
      }
   }

   public static void anotherMethod() {
      System.out.println("Inside 2 braces, Indented 6 spaces");
   }
}
```

It is possible to omit curly braces in some cases. You must still indent as if they were there. The following could be substituted for the blue lines above:

```
      for (int i = 0; i < 5; i++)
         if (i % 2 == 0)
            System.out.println("2 omitted braces, Still 12 spaces");
```

Spacing

Proper spacing helps with readability as well. Most java programmers follow the following conventions. It's easiest to show them through examples

Good spacing	Bad spacing
`int i = methodCall() * j + j % 4;`	`int i=methodCall()*j+j%4;`
`for (int i = 0; i < 10; i++) {`	`for(int i=0;i<10;i++){`
`while (sum < target) {`	`while(sum<target){`
`public void myMethod(int x, int y) {`	`public void myMethod (int x,int y){`
`methodCall(param1, param2);`	`methodCall (param1,param2);`
`public class Foo {`	`public class Foo{`
`int[] myArray = new int[x + 1];`	`int[] myArray=new int[x+1];`

Commenting

Comments can make code easier to understand, especially for someone who did not write the code. Place a comment at the start of each file with your name, the date, the class (ie CSE 142 AE), your TA's name, and the assignment number/title. Also include a description of what the program does. Comment each method with a high level description of what the method does. In particular describe legal values for each parameter, if there are any special values that could be returned, and any conditions that could cause an exception to be thrown.

Good comments describe what a program or method does (behavior), but not how the method or program accomplishes that goal (implementation). A client of your code doesn't care and shouldn't need to know how your code works (ie does it use a for loop or a while loop). In addition, adding implementation details makes it difficult to change your code later. If you omit implementation details, however, you can switch the implementation without having to change the guarantees that your comments make to the client.

There are two ways to write comments in Java. Single line comments start with // like this:

```
// This is a comment.
```

Multi line comments start with a /* and end at the first */ like this:

```
/* This is a comment that
   spans multiple lines */
```

Bad comment:

```
// This is the method that prints the values in the ArrayList
// using a for loop.  Should throw exception if the list is null
public static void print(ArrayList list) { ...
```

Why this comment is bad:

- Language. We already know that it's a method, so just say what it does instead of "this method does." Also, don't say the method "should" do something – if you programmed it correctly, it "will" do it.
- Implementation details. Maybe later you learn how to use an iterator and decide to change how the method works. But since you've documented that the code uses a for loop, you can't go back and change it. It is therefore better to tell what the method does rather than how it does it.
- Output or return value. The user probably wants to know how the results look - does it print each value on separate lines or print it in reverse order? That makes a big difference.
- Exceptions. The user also wants to know how to avoid breaking the method, so it's good to tell them precisely what will cause an exception. In case they do break the method, they also want to know what type of exception it is in order to handle the error in their program

Good comment:

```
// Prints the contents of list as comma-separated values.
// Throws an IllegalArgumentException if list is null.
public static void print(ArrayList list) { ...
```

Chapter 1

- Naming conventions:
  - Classes – Start with a capitol letter, each subsequent word starts with a capitol letter.
    - `MyAwesomeClass`
  - Methods – Start with a lowercase letter, each subsequent word starts with a capitol letter.
    - `myAwesomeMethod`
- Methods
  - Use static methods to break programs into reusable pieces. Reduce redundancy!
  - Avoid "trivial methods," or methods that do too little.
  - `main` should be a concise, high-level summary of your program. No details in `main`.
  - Leave a blank line between method definitions.
- Printing
  - For a blank line, use `System.out.println();` not `System.out.println("");`
  - Combine `print` statements.
    - `System.out.print("**");` instead of 2 calls to `System.out.print("*");`
    - `System.out.println("*");` instead of `System.out.print("*"); System.out.println();`
  - Choose the appropriate print method
    - `System.out.println("hello");` not `System.out.print("hello\n");`
- General Conventions
  - Java doesn't care about spacing, but we do. Place each statement on its own line.
  - Each line should be less than 100 characters.

Chapter 2

- Variables
  - Choose the correct type for variables:
    - Use `int` when you only need whole numbers. Use `double` when you need decimal precision.
  - Declare variables in the smallest scope possible. If you only need a variable inside of a loop, then declare it in the loop.
  - Choose variable names that succinctly describe what the variable represents. `average` is a better variable name than `a`. It lets anyone reading your code know what the variable represents without having to read the code that computes an average.
  - Don't rely on context to give meaning to your variable names. The context of your variable may change as you edit your code.
  - Variables follow the same naming conventions as methods.
    - `camelCaseYourVariableNames`
- Constants
  - Use a class constant when you want to define a single value that won't change during the execution of your program.
    - The naming convention for constants is `CAPITOLS_WITH_UNDERSCORES`
    - Constants are declared with the keywords `public static final`
- `for` Loops

- Use `for` loops to repeat code a specific number of times. Reduce redundancy!
- You don't need a loop that runs only 1 time.

Chapter 3

- Parameters
  - Use parameters when you can to make methods more general. Don't include unnecessary parameters in a method declaration; every parameter should be used.
  - Don't pass class constants as parameters, they're accessible everywhere within the class.
- Returns
  - Use return statements to move data from a method to its caller.
  - Declare a method as void if you don't need to return anything.
- Objects
  - Create a single `Graphics` object or `Scanner` object and pass it to your methods.

Chapter 4

- `If/else` statements
  - Use the appropriate conditional structure
    - `if/if` – any of the statement bodies could execute, or all, or none
    - `if/else if` – 0 or 1 of the statement bodies will execute
    - `if/else` – exactly 1 of the statement bodies will execute
  - Factoring
    - Common code at the beginning of each branch of a conditional statement should be pulled out before the condition
    - Common code at the end of each branch of a conditional statement should be pulled out after the condition
- Exceptions
  - Construct Exceptions with a useful error message

Chapter 5

- `while` Loops
  - Use `while` loops to repeat code an unspecified number of times.
  - A `do/while` loop runs the loop body before checking the condition
- Fenceposting
  - Sentinels – You may need to prime a loop to get it to execute the first time
  - You also may need to pull the code for one iteration of the loop outside of the loop
    - This is preferable to solving your fencepost problem by using a conditional in the loop
- Random
  - Create a single `Random` object and pass it to your methods.
- `boolean` zen
  -

Bad	Good
if (myVar == true) {	if (myVar) {
if (myVar == false) {	if (!myVar) {
if (myVar == true) return true;	return myVar;

else return false;	

Chapter 6

- Using a `Scanner` on a `File`
  - Be careful mixing `next()`, `nextInt()`, and `nextDouble()` with `nextLine()`. It's easy to create bugs.
- You may not want to initialize variables in the smallest possible scope in one case. If initializing your variable could potentially throw an exception (such as initializing a `Scanner` on a `File`), you may want to do it in `main` and pass the variable into each method instead of adding a `throws` clause to each method.

Chapter 7

- Arrays
  - Use arrays to store a sequence of data that all has the same type
  - The easiest way to loop over an array when you need the index of each element:
    - `for (int i = 0; i < myArray.length; i++) {…}`
  - The easiest way to loop over an array when you don't need the index of each element:
    - `for (int element : myArray) {…}`
  - Sometimes it's appropriate to return an array, but don't use it as a hack for returning two unrelated pieces of data
  - Also don't pass an array of unrelated data to avoid declaring extra parameters
  - Rather than using many lines of code to initialize an array, shorthand it like this:
    - `int[] myArray = {1, 2, 3, 4, 5};`

Chapter 8

- Objects
  - Create `private` fields with getters/setters rather than leaving fields public.
    - This allows you to make sure your object is always in a valid state.
    - Also you can change the implementation later without breaking client code.
  - Helper methods should be private as well.
  - Use constructor overloading to reduce redundancy within your objects. Reduce redundancy!

Chapter 9

- Inheritance
  - Use inheritance to share common functionality across classes. Reduce redundancy!
    - This also lets clients re-use code to interact with different types of objects.
  - Use `protected` fields to let a subclass interact with your fields.
- Composition
  - Inheritance is not always appropriate, sometimes you need to use composition.
  - This is the difference between an is-a relationship and a has-a relationship.