**Formatting**

You should follow standard conventions when writing code. It makes it easier for others to understand what your code does and that's important in the real world.

Formatting resources:

~ Following your textbook examples in naming and formatting classes, fields, and methods is the simplest way to go.  
~ If you want to start being professional follow Google's specification: <https://google.github.io/styleguide/javaguide.html>.

Whichever model you choose, be consistent and look up what's right when you are unsure.

Here is a list of the mistakes I see most frequently (with page references to the above Oracle document for more information).

* All names you invent for classes, variables, methods, etc, should suggest the purpose of the named element. For example distancePerSecond is a better name than d. It is okay to use single letters such as i, j, and k for loop indices if the loop index has no other meaning.
* Class names should use CamelCaseWithInitialUpperCase. Method and variable names should use camelCaseWithInitialLowerCase. Class constants should use ALL\_UPPER\_CASE\_WITH\_UNDERSCORE\_SEPERATOR names.
* Each open brace indicates a new block and everything in the block should be indented one more level until the matching close brace.
* No line should be more than 100 characters long. If you need more than 100, wrap it at natural place in the line and continue it on the next line indented.

**Javadoc**

It's been said that a majority of the time a professional programmer spends with code, it is with code originally written by someone else. This means that when you write code, you should keep this reality in mind, and write your code to be easily understood by someone else. In general, this means:

* Follow standard formatting conventions (ie, capitalization, indentation, etc).
* Give methods and variables meaningful names.
* Write code simply rather than cleverly.
* Write inline comments describing any non-obvious code.
* Write documentation describing the purpose and function of every class and public method.

In Java, this last step is done using the javadoc tool. If you've ever looked at the Java documentation for any of the built-in classes (String, Random, Scanner, etc) then you've seen what javadoc output looks like. It's remarkably easy to produce.

To learn how to write javadoc:

1. Look at some examples. Google search "javadoc string" and "javadoc random" to see what kind of information is in a thorough javadoc. We won't be as thorough in this class, but seeing Java's own documentation exposes you to what a really good javadoc looks like.
2. Read some tutorials. Google search "javadoc tutorial" and look over the top couple of hits. You'll see examples and descriptions of the various parts.

What's required in this course? The rule-of-thumb is to write as much as would be required to satisfy *you* if you were a client of the software. Ask yourself if you'd be happy with this level of documentation of a class you were trying to use.

* Include class documentation (just before the public class Foo). This should explain at a high-level the purpose of the class and how it's used. If there is something interesting or important about how it is implemented, this can be discussed here. Look at some good examples of Javadoc to see what kinds of things are mentioned.
* Every public method (except main) should have an entry describing the purpose of the method and how it's used. Again, if there is something interesting or important about how it is implemented, this can be discussed here. Look at some good examples of Javadoc to see what kinds of things are mentioned.
* Use @author, @version, @param, @return, and @throws tags every chance you get.
* The javadoc should compile without warnings or errors.

**Note:** On jGrasp you can generate javadoc by bringing the coding window forward and clicking on the icon that has a book on it.

Here is an example class with javadoc:

import java.util.Random;

/\*\*

\* A class representing a single Die. Upon creation, the user must pass to the

\* constructor the number of sides on the Die. Each Die object keeps track of

\* both the number of sides the Die has, and the current value of the Die.

\* Possible values for a Die are 1 through to the number of sides. Subsequent

\* calls to a Die's roll method cause the Die to take on a new random value,

\* which can be accessed through either the getValue or toString methods.

\*

\* @author Ted Krovetz

\* @version 2 September 2015

\*/

public class Die {

private int value;

private int sides;

private Random rand;

/\*\*

\* Initializes a new Die with its number of sides. The die can then take

\* on any value 1..sides. An initial random roll takes place too.

\*

\* @throws IllegalArgumentException if sides is not greater than 0.

\* @param sides sets the number of sides on the Die. The sides have values

\* 1 through sides.

\*/

public Die(int sides) {

if (sides < 1) {

throw new IllegalArgumentException("sides must be positive");

}

this.sides = sides;

rand = new Random();

value = rand.nextInt(sides) + 1;

}

/\*\*

\* Sets the Die's value to a new randomly chosen one. The value will be in the

\* range 1 through the number of sides on the Die.

\*/

public void roll() {

value = rand.nextInt(sides) + 1;

}

/\*\*

\* Returns the current value of the Die as an integer.

\*

\* @return the current integer value of the Die.

\*/

public int getValue() {

return value;

}

/\*\*

\* Returns the current value of the Die as a String (eg, value 2 is returned as "2").

\*

\* @return a String representation of the Die's current value.

\*/

public String toString() {

return ""+value; // Trick to produce a String from an int

}

// Simple embedded unit test.

public static void main(String[] args) {

try {

Die doesntWork = new Die(0);

System.out.println("Shouldn't get here!");

}

catch (IllegalArgumentException e) {

// Do nothing -- this exception is expected

}

Die d1 = new Die(6);

Die d2 = new Die(6);

for (int i=0; i<10; i++) {

System.out.println("Die 1: " + d1 + ", Die 2: " + d2 + ".");

d1.roll();

d2.roll();

}

}

}