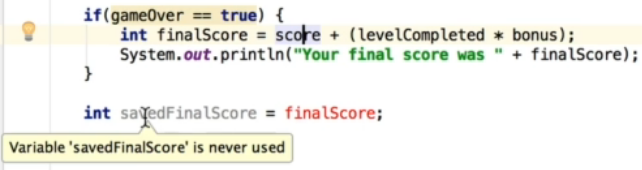
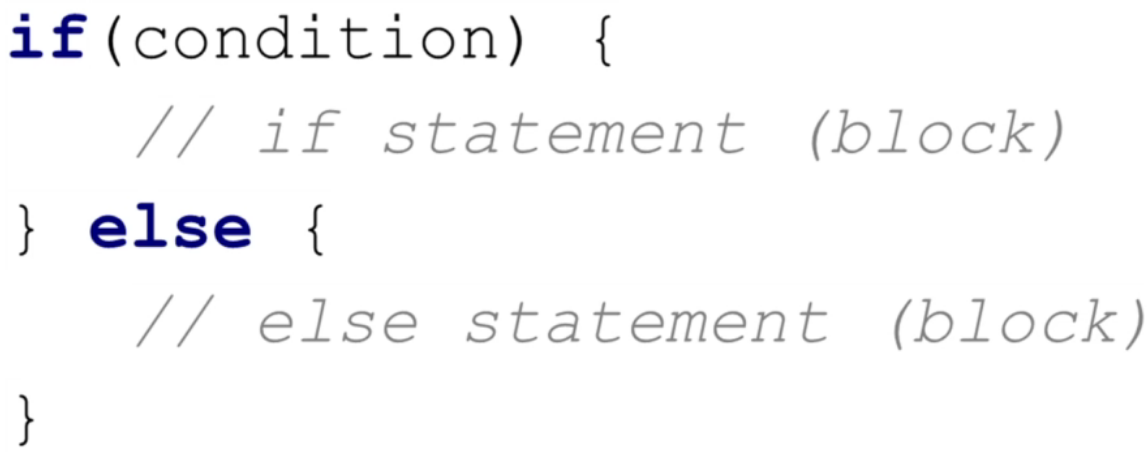
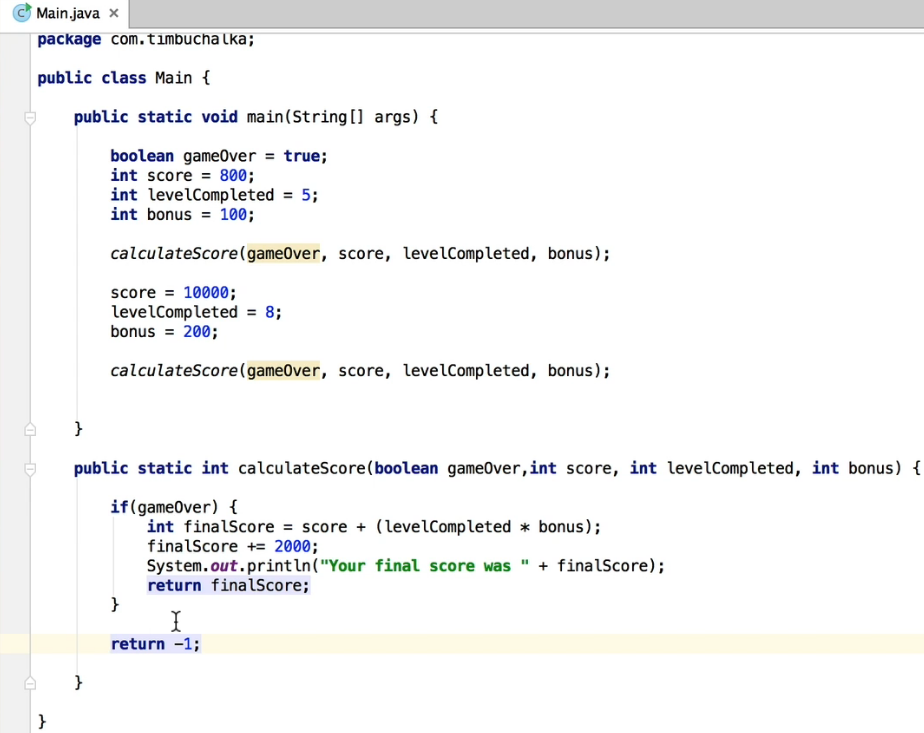
**Introduction**  
\* This section is dealing with 4 really important concepts of Java:  
1) **Expressions**  
2) **Statements**  
3) **Code** **Blocks**  
4) **Methods**  
\* These are four key components of Java that you really need to get a good understanding of and that’s the goal of this section, to give you that understanding.

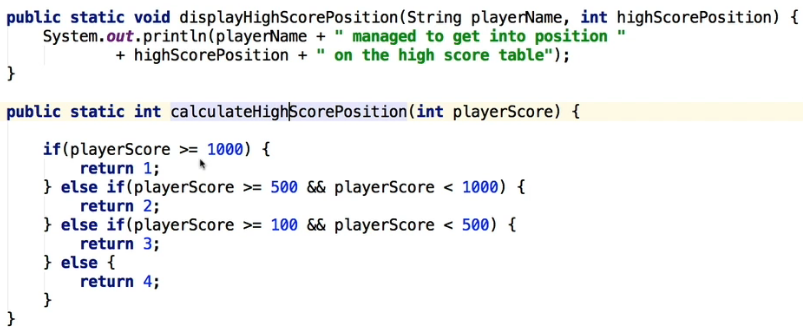
**Keywords And Expressions**  
\* **Keywords**.  
\* Just has actually 50 reserved words that are used for keywords in your applications.  
\* We’ll go over all of the keywords in the course.  
\* **Expressions**.  
\* Expressions are essentially building blocks of all Java programs. You’ll be using expressions a lot.  
\* Expressions are built with values, variables, and operators but also with **methods calls** which we’ll be discussing later in the course.  
\* For example:  
\* double **kilometres = (100 \* 1.609344)**;   
=> That’d be a valid statement.  
=> For that line, the actual expressions is the yellow part. So the Data Type doesn’t form part of an expression, but everything else on the line typically forms or is part of the expression.  
\* So the expression component is variables (kilometres), values (100, 1.609344), and operators (=\*).  
\* **By adding the Data Type** we’re creating a valid Java **Statement** with the **;** and the end of the line.  
\* Another example:  
int **highScore = 50**;  
if (**highScore == 5**) {  
 System.out.println(**“This is an expression”**);  
 **score = 0**;  
}  
\* The yellow part is the expression, within the parenthesis.  
\* So the keyword for if and the parenthesis (brackets in UK?) and the braces don’t form part of the expression.  
\* The other component if the brackets is actually calling a method, and we’ll be talking about methods in a future section. The main idea is just to get your head around what components of the entire line is an expression and what an expression actually is.  
\* So if you’re using a method, anything in their brackets () essentially is also an expression.

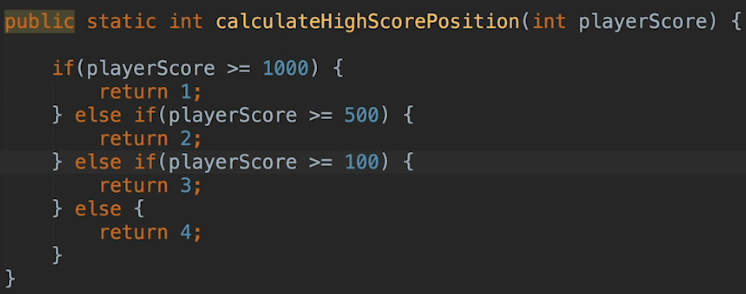
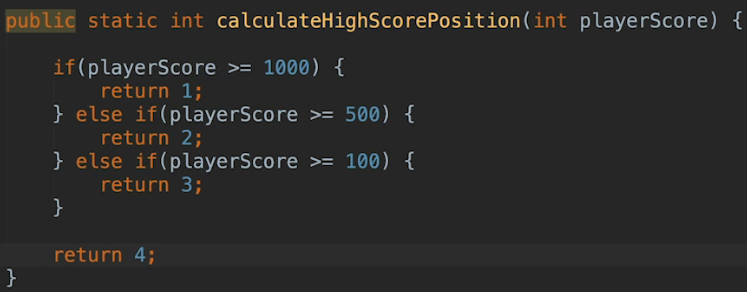
**Statements, Whitespace and Indentation (Code Organization)**  
\* The **Statement** is the entire line: **int x = 50;**\* So by adding the Data Type at the start to our expression and then finishing off with a semicolon, we’ve made a valid Java Statement.  
\* So Java Statements can be assignment expressions.  
\* **x++;** is also a Statement.  
\* Also method calls which we haven’t talked about but will much more in the future is a Statement.  
\* **System.out.println(“This is a Statement too”);**\* It has a method call and a semicolon at the end with the parameters passsed.  
\* Semicolon is needed to complete a Java line, to make it a Statement in most cases but there are exceptions to that, and we’ll talk more about those.  
\* The other thing to keep in mind with Statements is that they don’t have to be on one line, you can break them over multiple lines if you want. If there’s no semicolon at the end of the line, Java continues on the next line, so it collectively adds all of it together and effectively creates the line just as if we haven’t broken it over multiple lines.  
\* You can also have multiple statements on the same line but it can get confusing.  
\* **Whitespace** is the space inbetween some of your expressions, your operands and so forth.  
\* Java deletes out the spaces internally.  
\* It’s a good idea to use spaces to clarify the meaning of particular things.  
\* Another common place to put a space is between the = and the variable name and again between the = and the literal value.  
\* So in general you can do whatever you like with the Whitespace and Java will happily ignore the Whitespace.  
\* **Indentation** is indenting your code so that it’s more readable.  
\* Generally as you’re typing in different code blocks, it’s actually going into the next level of indentation. IntelliJ uses 4 spaces. The concept of indenting is to make it easier for you to see the logical flow.  
\* **In IntelliJ you can click Code => Reformat Code…** re-indents the code for you.

**Code Blocks And The If Then Else Control Statements**  
\* **if (condition) {}**  
\* **You can omit the {} if you’ve only got 1 line, 1 Statement**.  
\* **I recommend using a Code Block {} even if you have only one line, because I believe it makes the code clearer and easy to understand**.  
\* **if (condition) {} else if (condition) {} else {}**  
\* **As soon as it’s found something, it’s going to ignore the else if and any other subsequent elses along the way**.  
\* If we create a variable in a Code Block, you need to remember that the Code Blocks are able to access variables that were created outside of the Code Block, but once that section of code has finished - the computer has processed it - it’ll automatically delete any variables that it creates in that Code Block.  
\* While you can access variables created outside of the Code Block, in the Code Block what you can’t do is access variables created in the Code Block outside of the Code Block as you can see in this scenario here:  
  
\* **Another abbreviation tip is that in the condition you can just type the boolean variable name**.  
\* **The concept of variables inside a Code Block is called** **Scope**.  
\* And Scope is something that requires its own videos, we’ll be talking more about Scope later.  
\* Scope really deals with accessibility of variables in certain situations or in certain places in the code, for example in a Code Block. The bottom line for now is that in a Code Block if you create a variable, you can’t access that variable outside of that Code Block. You can of course still access variables that were created previously and prior to going into the Code Block while you’re in the Code Block itself.  
\* **You can remove duplication by using the concept of Methods**.  
\* **Methods are taking the concept of a Code Block to the next level and it effectively allows you to reuse code**.

**If then else Recap**  
 **\* Example:**

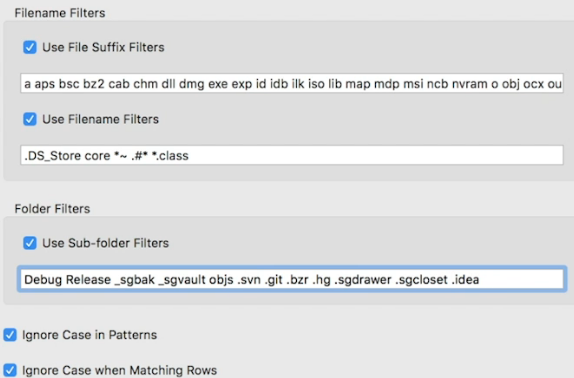
**Methods In Java**  
\* The `public static void main(String[] args) {}` is actually a method too. It has the name of main and all the code in the middle of it is the statements that form the method.  
\* You can’t put a method within another method.  
\* It has to be within the `public class Main`.  
\* If there’s a method which isn’t used, IntelliJ uses a light gray color for it.  
\* `**parameter**` **is defined in the method definition**.  
\* `**argument**` **is the actual values that you pass to the function**.  
\* You can pass the values into the method or you can create variables which you then pass to the method to make it easier to read and know which argument represents what, instead of hardcoded values.  
\* We can also get our method to do some calculations much like it’s doing now, but we can send back the result of that calculation back to the code that had called the method in the first place.  
\* `**void**` **means nothing is returned**.  
\* We can use the datatype that we want the method to return.  
\* You use the `**return**` keyword.  
\* When you’re sending some data back with a method, you’ve gotta make sure that no matter what, that some values gets sent back. It’s not okay to just use `if` and only return something there. So all the program variations have to be accounted for and we gotta return something everytime if we specify that we are gonna be returning some data.  
  
\* One final point - if you’re wondering why we’re returning -1 here: in programming terms, -1 is conventionally used to indicate an error. And in searching algorithms, -1 indicates an invalid value or value not found.

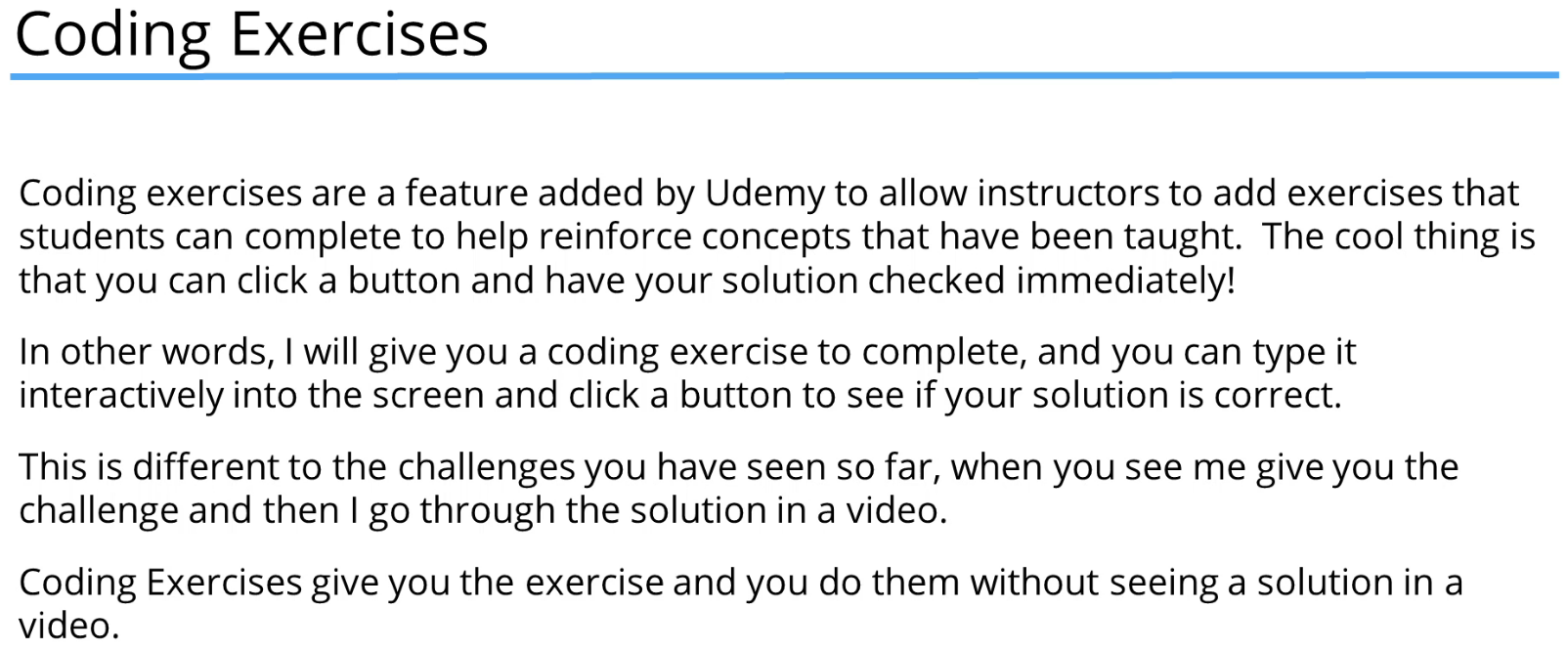
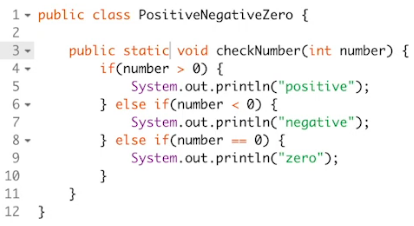
**More On Methods And A Challenge**  
\* A **void method** can also be known as a **Procedure**. And a method generally speaking can also be known as a function.  


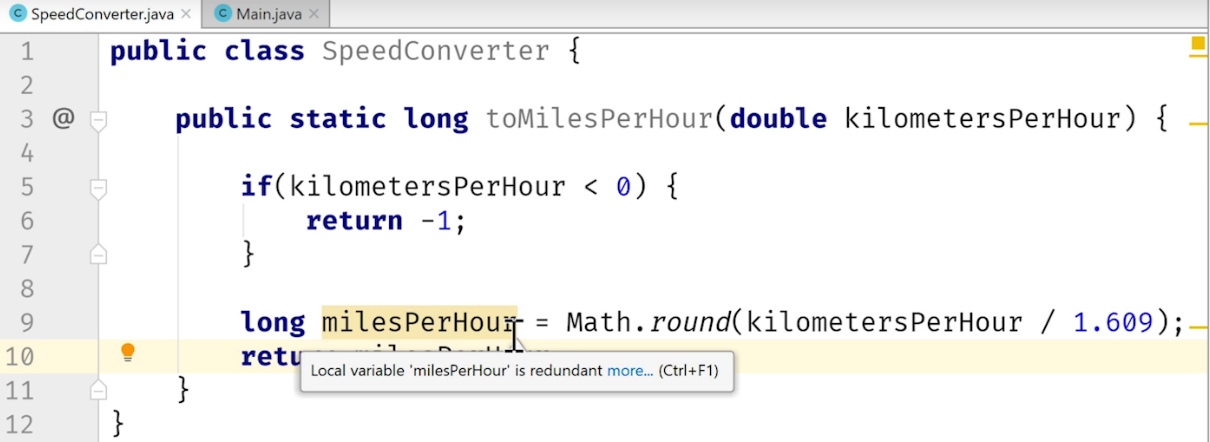
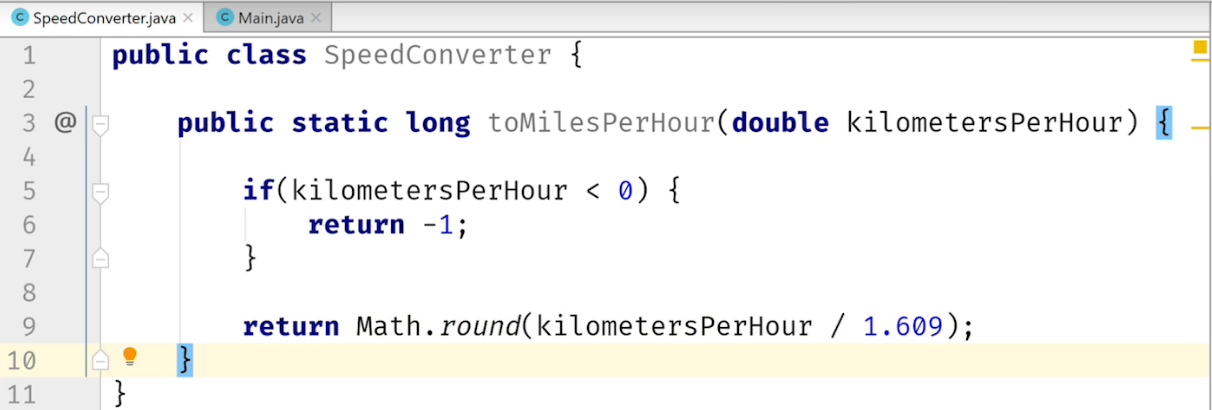
**Method Challenge - Final Code Changes**  
\* We can change the conditions a bit because we have some redundant testing because it’s already tested in the condition above it.  
  
\* You could also remove the `else` and just use a return statement under the `if` statements.  
  
\* I’m wanna show you one more solution that actually eliminates multiple return statements. Multiple return statements can be confusing for people starting out in a programming language so let’s instead make the code easy to follow and use only one `return` statement.  

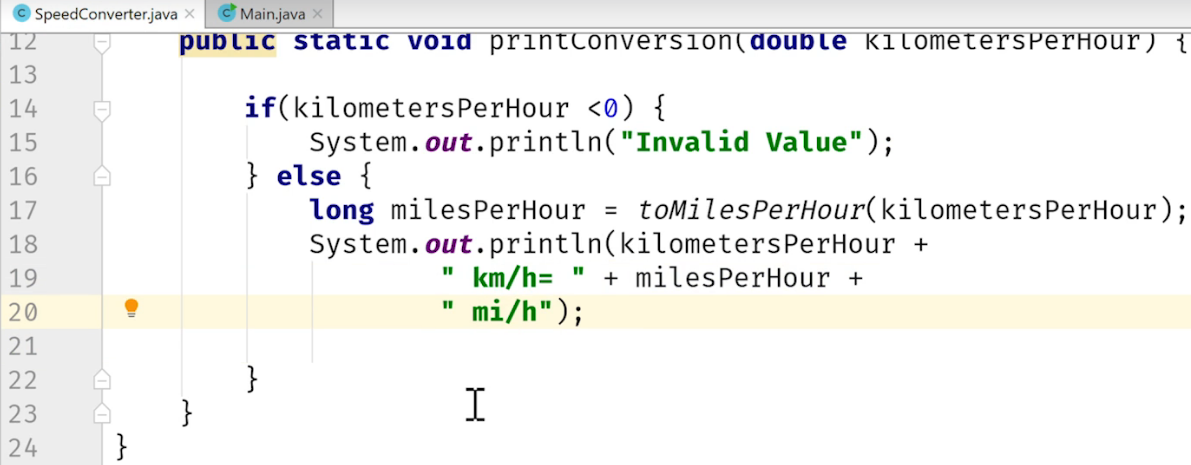
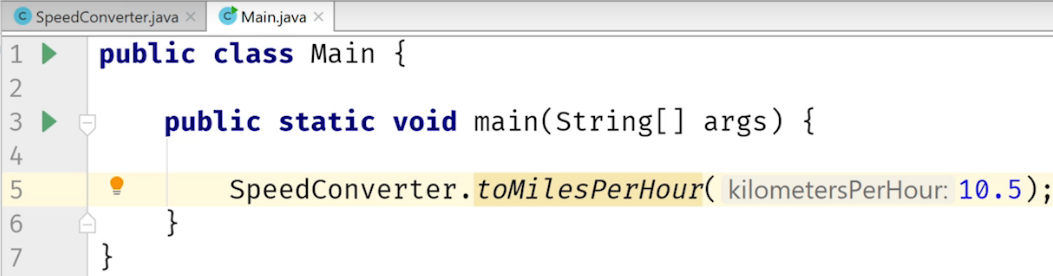
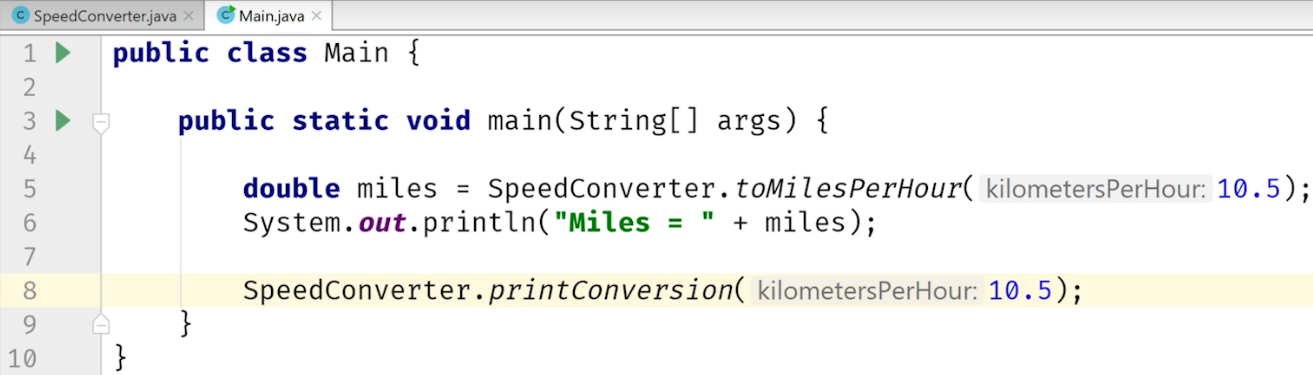
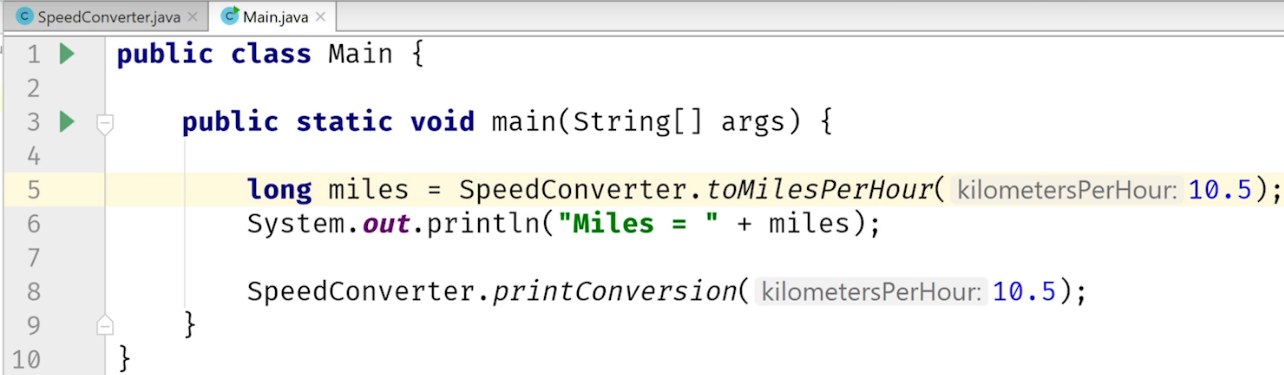

**DiffMerge Tool Introduction**  
\* **DiffMerge is a program that will help you to visually compare and merge files on any operating system**.  
\* It can be very helpful with programming as well.  
\* Some other tools like code repositories use similar merge operations, so this will also help you in the future when you start working with code repositories.  
\* When you code along with the videos in course it is easy to make a typo, and get stuck, and this is where DiffMerge will help you!  
\* Every lesson has the code attached to it that I typed in the video and you can download a zip file with all that code.  
\* I will explain and show you later how to download and use that code. It’s pretty easy.  
\* By downloading code from the video and comparing it with yours you will be able to easily find any errors/typos made using DiffMerge.  
\* With DiffMerge you can compare a single file or even the whole folder and visually see the differences side by side.  
\* In the next video we will install the DiffMerge program and I will then show you how to use it.

**Install DiffMerge**  
\* **sourcegear.com/diffmerge => downloads => DiffMerge =>** .

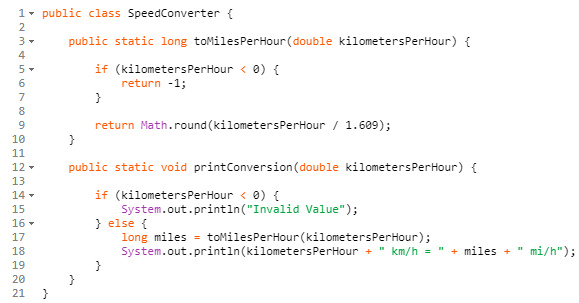
**Using DiffMerge**  
\* By default it only shows files that are different.  
\* The lines that are different will have the differences highlighted in red.  
\* It also shows blue vertical lines when you click on the line.  
\* You can click the red icons on the left scroll bar to navigate through the different diffs, usually one for each file that is different.  
\* You can also use the icons at the top right side to jump between different lines.  
\* Now how do we fix the errors? What we can do is we can choose to copy the changes from the left-hand window to the right-hand window. Generally that’s what you’d want to do and that’s because we’ve identified that the code on the left-hand side is the original code from the course and it’s the correct one.  
\* There’s a button at the top that says `Apply change from Left`. When you click that, you can see a black underline and bold text and also \* asterixes in parts that changed.  
\* To save those changes, we need to is click on the Save icon at the top.  
\* DiffMerge => Preferences => Folder Filters => we want to set it up to ignore some common file times.   
\* Class file is a compiled version of a Java source file so that’s not something we wanna check.  
\* The other thing we can do is come down to the Sub-folder filters.  
\* Now on the left you in the DiffMerge you can still see .idea folder and that’s the IntelliJ’s folder that it uses to track changes to code, so we usually wouldn’t have a need to check if those are different so we can add it to the Sub-folder filter.  


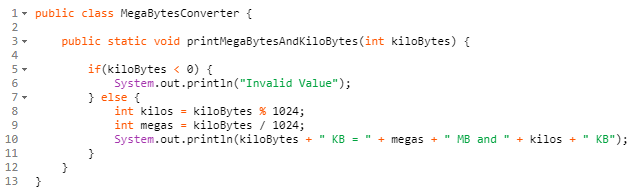
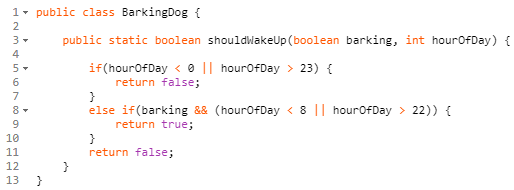
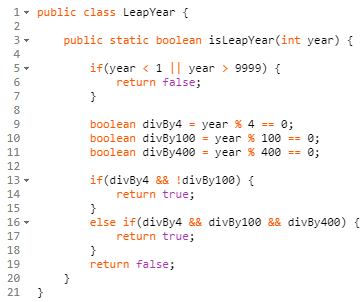
**Coding Exercises**  
  
\* **In IntelliJ you can type psvm for `public static void main`**.  
\* **In IntelliJ you can type sout for `System.out.println() `**.  


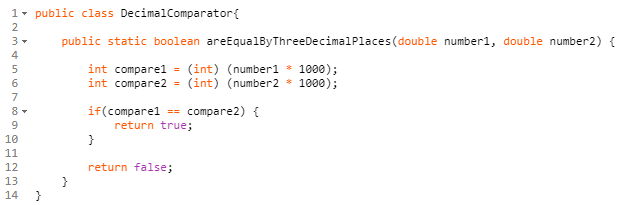
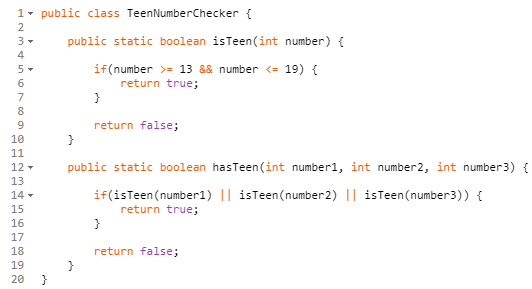
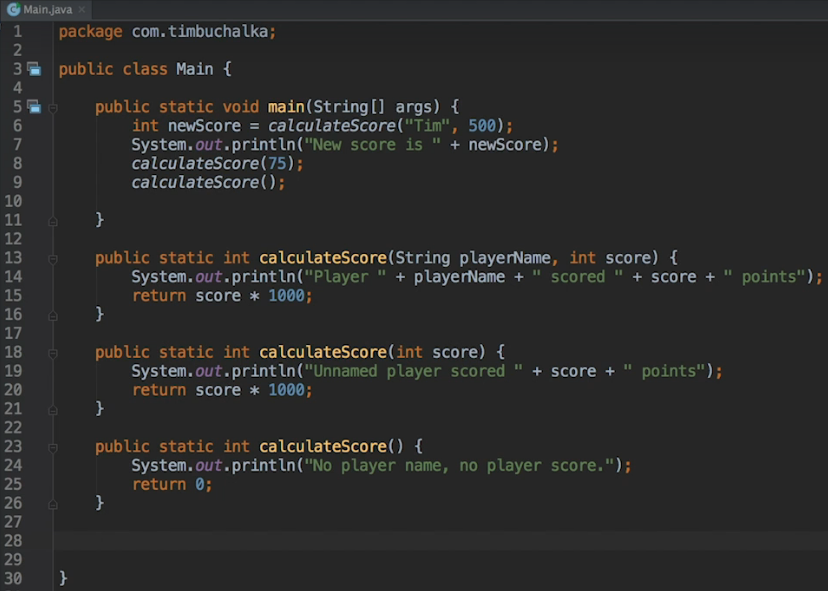
**Coding Exercises Example Part 1**  
\* Here it’s saying that this variable is redundant.  
  
\* That’s because we’re only using this local variable for returning the result of our calculation. Instead of using a variable, we can use the return statement right away with a calculation or an expression. Now that will also return the result but the code is shorter and then we don’t need a local variable.  


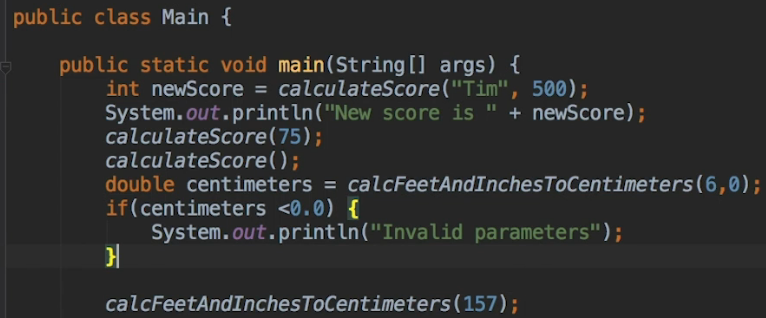
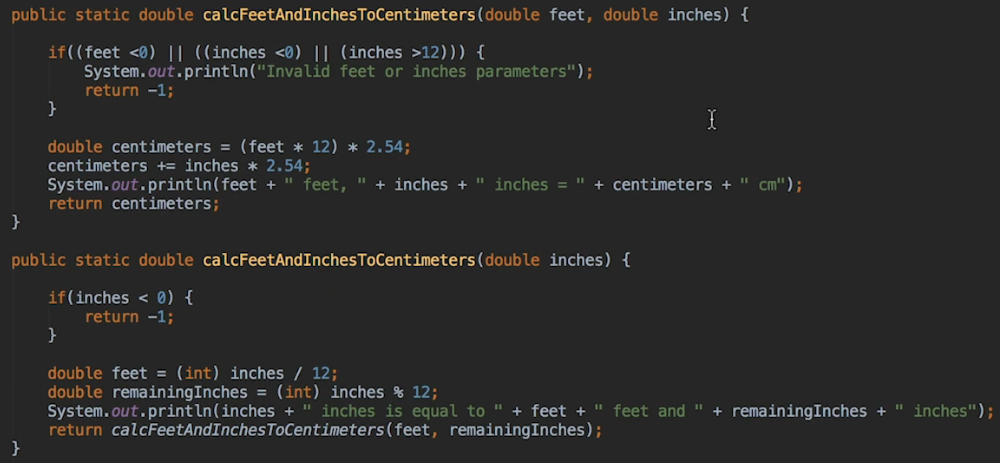
**Coding Exercises Example Part 2**  
  
\* We’ll use Main class to make it easier for us to test and also not to accidentally copy extra {} which could happen if we were adidng a main right next to our other methods in 1 file.  
  
\* IntelliJ is telling us that the result of this method is ignored. If we run it, there’s no output from the program. The reason it’s not outputting to the screen is that we haven’t told IntelliJ to print anything to the console, we’ve only told it to call that method to invoke the code. IntelliJ is pretty smart here and basically saying to us: okay, you’ve called this method but you’re not doing anything with the returned value so it’s just giving us a warning.  
  
\* There’s a mistake - toMilesPerHour uses `long` but in Main it’s stored in `double miles`. It still worked because Java was actually smart enough to convert it over, but we should change it to `long`.  
\* Also it returned 7.0 instead of 7.  


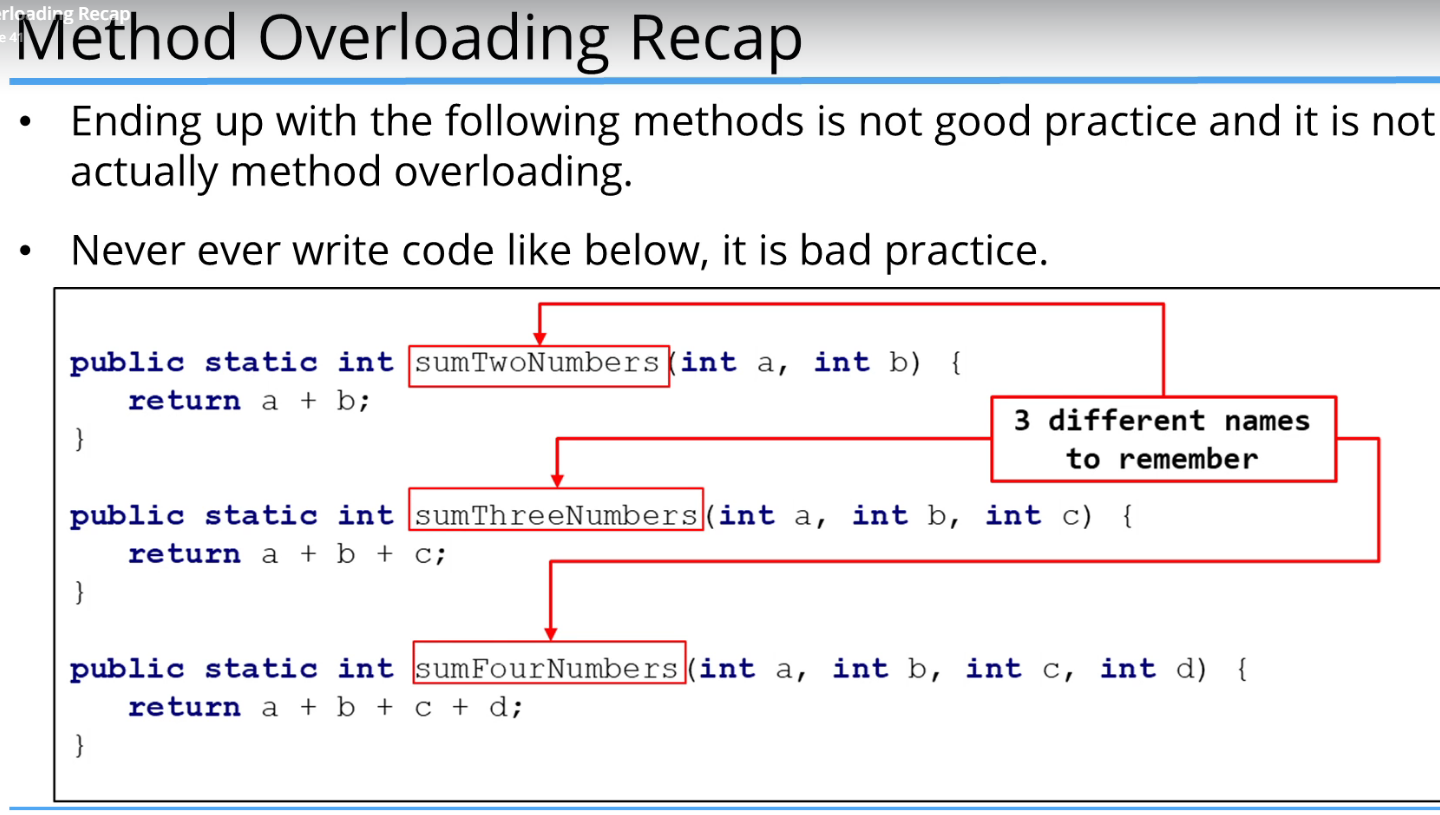
**Coding Exercises Example Part 3**

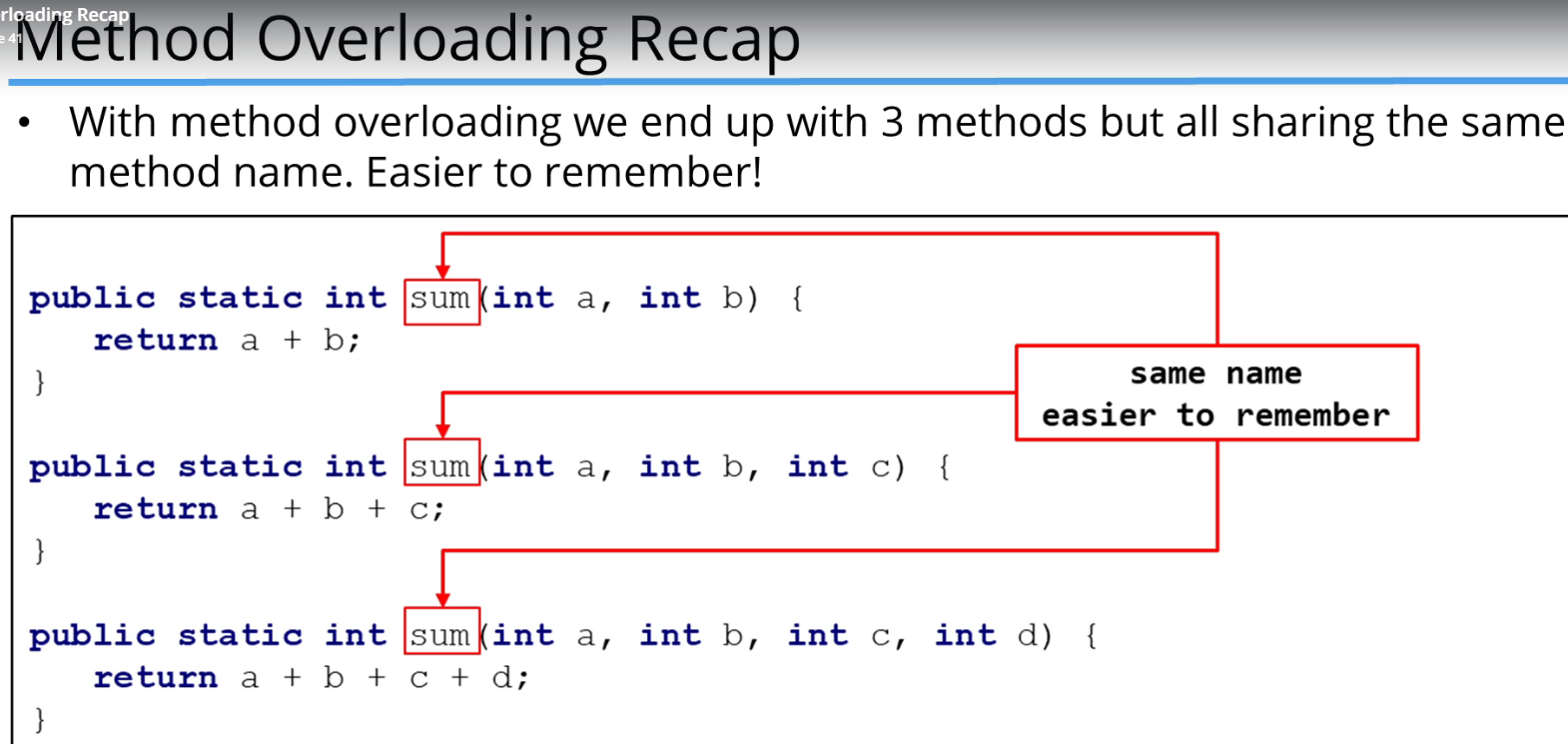
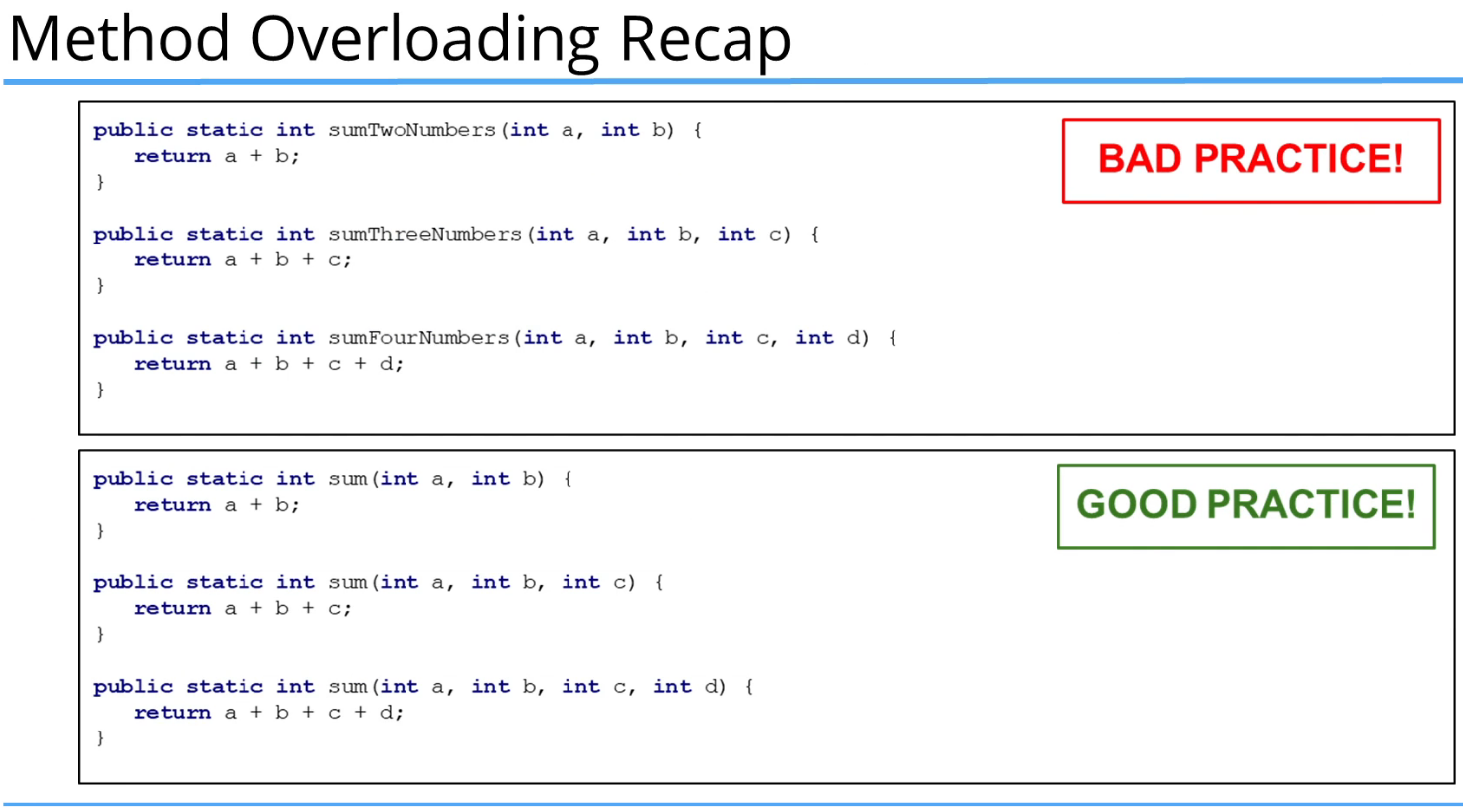
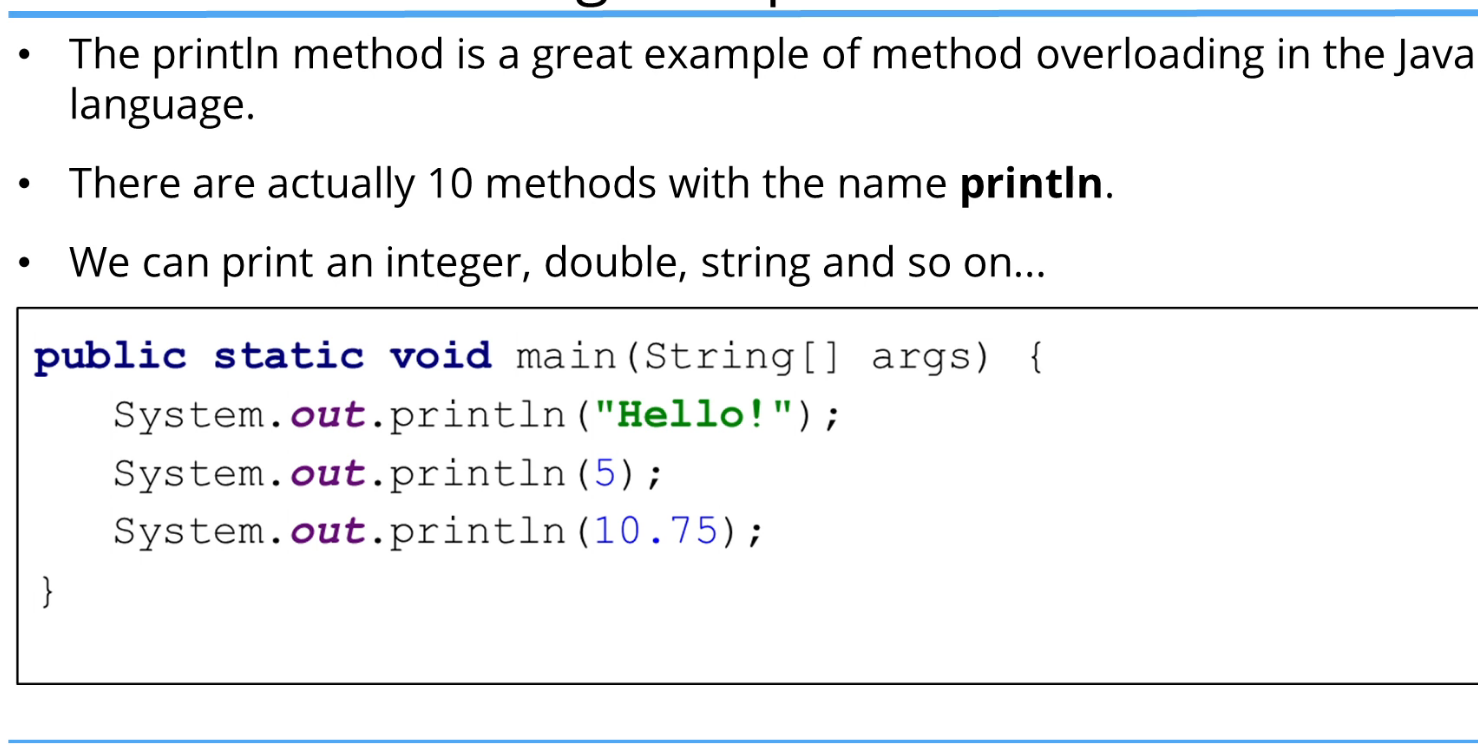
**Coding Exercise 1: Speed Converter**  


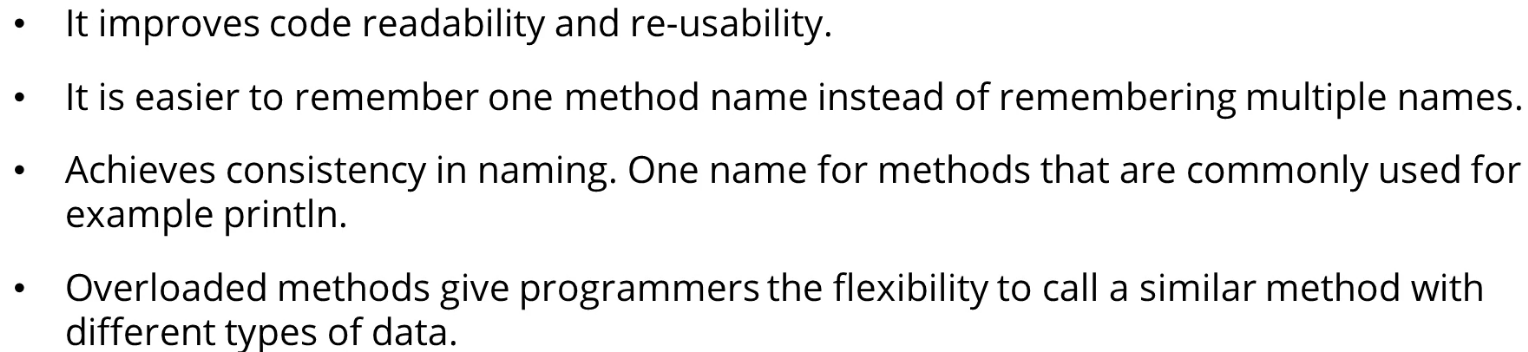
**Coding Exercise 2: MegaBytes Converter**  
  
**Coding Exercise 3: Barking Dog**  
  
**Coding Exercise 4: Leap Year Calculator**  


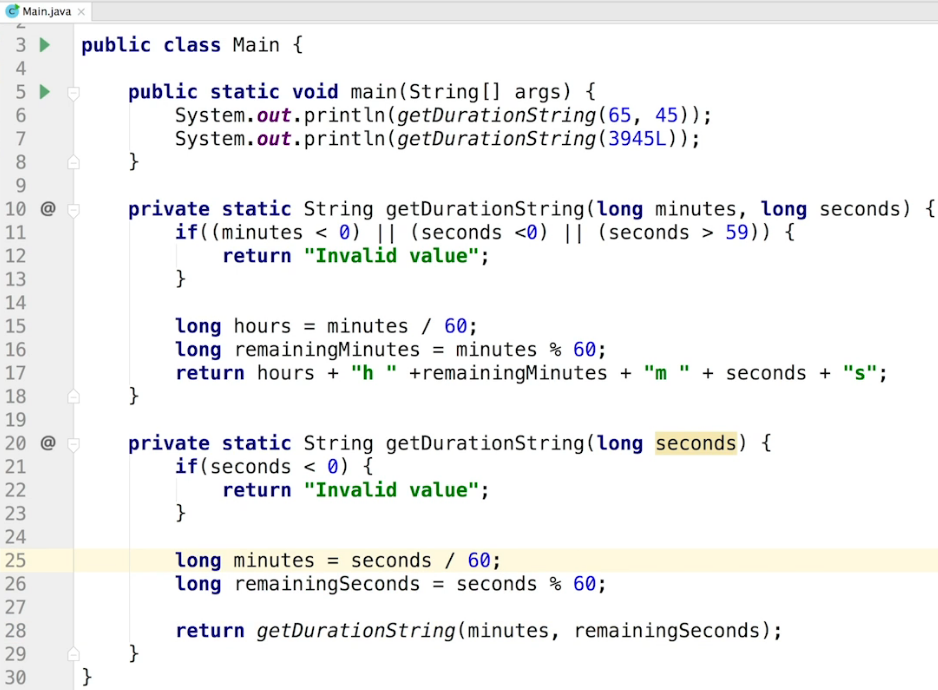
**Coding Exercise 5: DecimalComparator**  
  
**Coding Exercise 6: Equal Sum Checker**  
**Coding Exercise 7: Teen Number Checker**  
**Method Overloading**  
\* Method overloading is very commonly used in Java programming and in other programming languages and it’s the option where you use the **same method name** but with **different parameters**.  
\* When overloading a method we need to create a unique method signature which is essentially the Unique part => **METHOD NAME** + **PARAMETER TYPES**  
\* **(the return type doesn’t make it unique)**.  


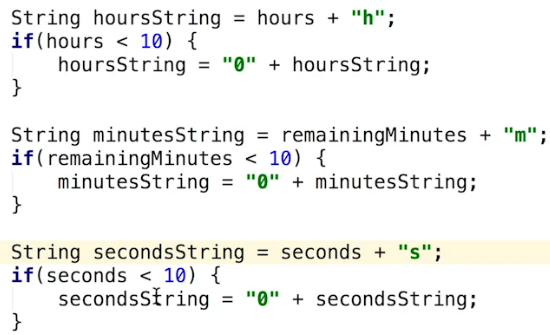
\* **Calling an overloaded method within an overloaded method**.  
  
  
\* Overloaded method are very common in production Java code.

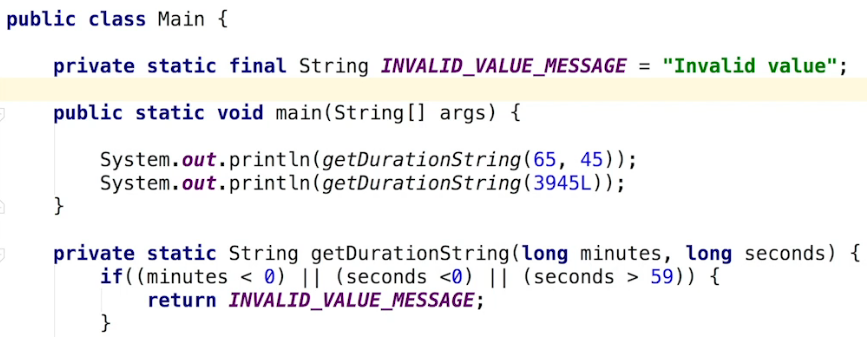
**Method Overloading Recap**  
  
\* **It’s a BAD practice to write methods like this**. You have to remember multiple method names that do practically the sum.

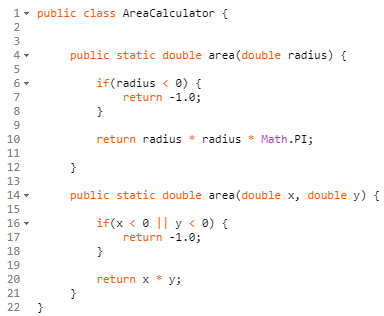
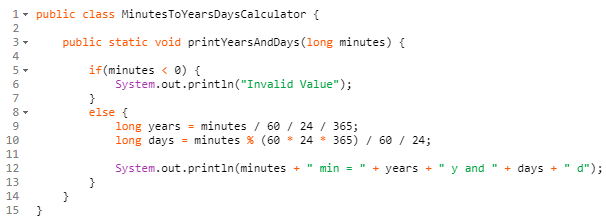
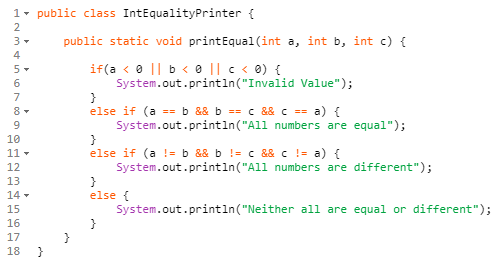
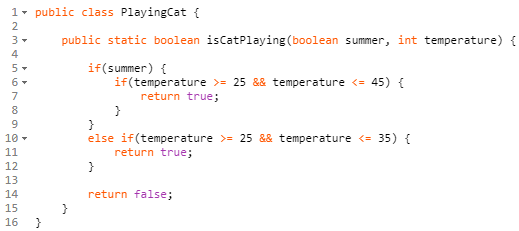
\* Instead, do it like this:  
  
  
\* **Example of Method Overloading**:  


\* **Benefits:**  


**Seconds and Minutes Challenge**  
  
\* A good idea would be to use L after the number to indicate `long`.

**Bonus Challenge Solution**  
\* There are many ways to format and add leading 0s but let’s use what we already know.  
\* One way to add leading 0s:  
  
\* Let’s use a **CONSTANT**.  
\* To use a constant in both methods, it needs to be declared outside of the methods.  
\* `final` keyword is used to make it so that once we’ve assigned the value, it can’t be be changed.

\* We also use `static` because we’re going to use that variable in static methods.  
\* CONTANTS are usually typed in upper case so that they’re easily identifiable as CONSTANTS.  
  
\* Another useful thing about using a CONSTANT is that it can help you prevent a typo. If you’re typing the same String multiple times into a program, it’s easy to make a typo. Also if we change the CONSTANT, it will be affected everywhere we use the CONSTANT.

**Coding Exercise 8: Area Calculator**  
  
**Coding Exercise 9: Minutes To Years and Days Calculator**  
  
**Coding Exercise 10: Equality Printer**  
  
**Coding Exercise 11: Playing Cat**  


**Resources**  
**List of Java Keywords:**<https://en.wikipedia.org/wiki/List_of_Java_keywords>   
**Feet to Centimeters**  
<https://www.metric-conversions.org/length/feet-to-centimeters.htm>