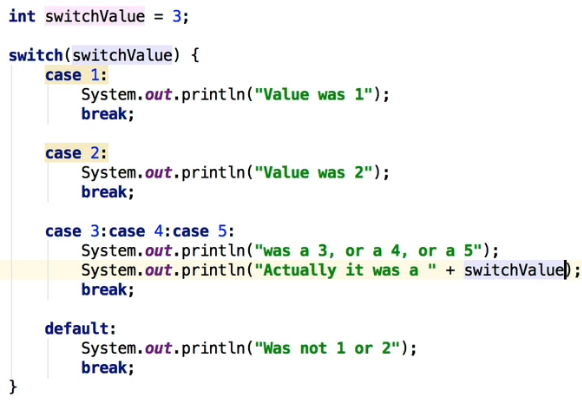
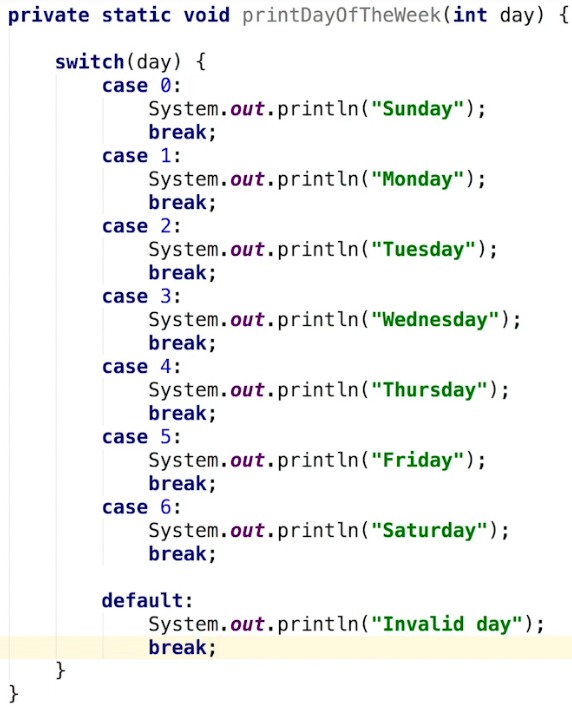
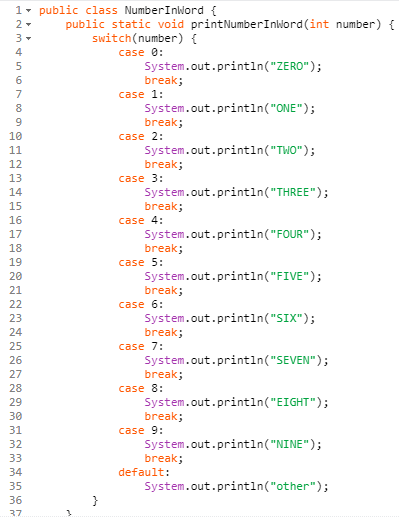
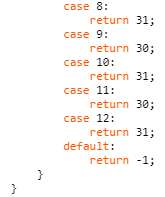
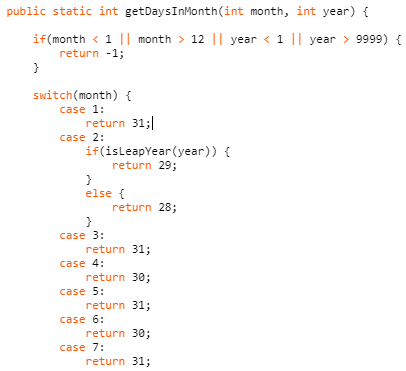
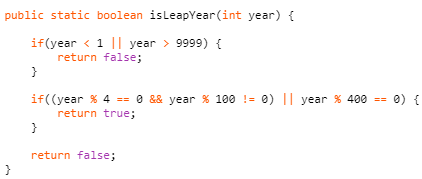
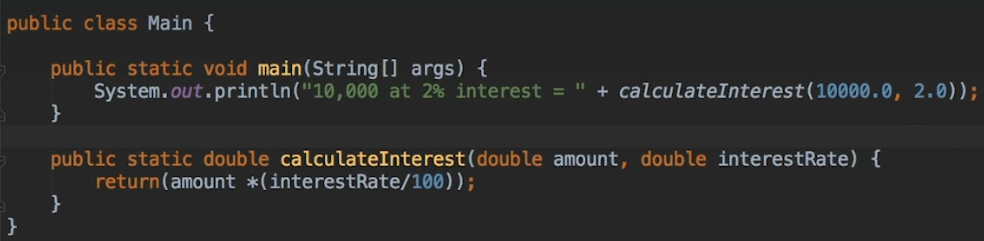
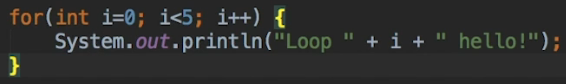
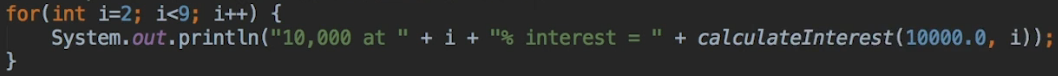
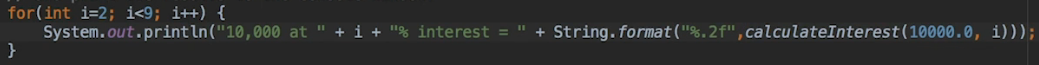
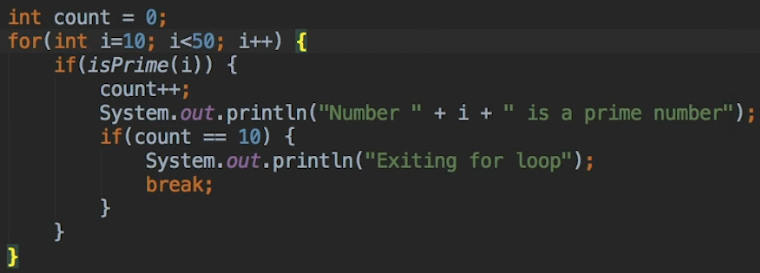
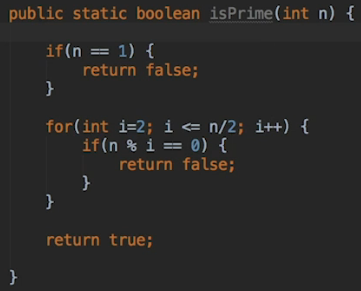
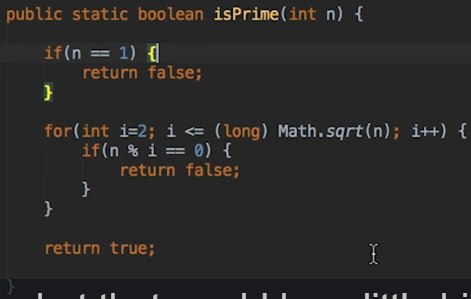
**Introduction**  
\* We’ll learn about **switch**, **for**, **while**, **do while** statements.

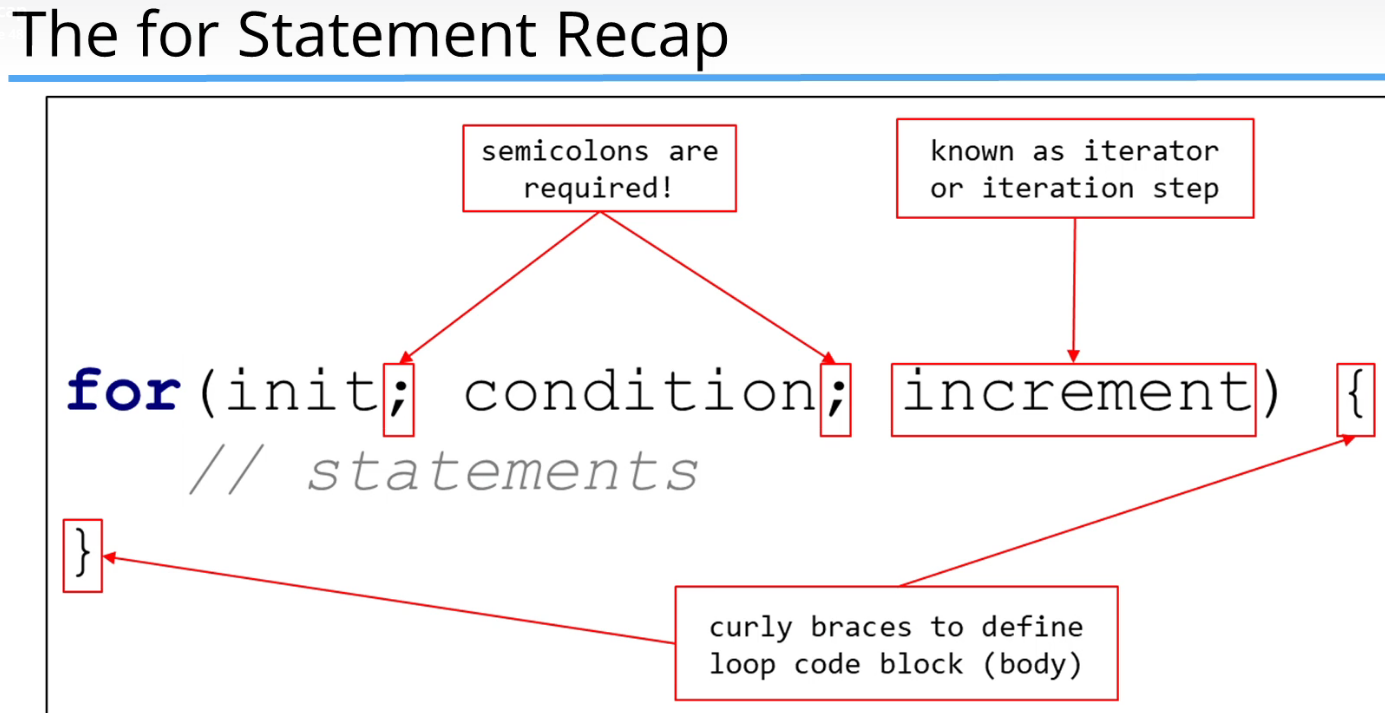
**The SWITCH statement (+ Challenge Exercise)**  
\* In IntelliJ go to File => Project Structure => 8 - Lambdas, type annotations etc. or 11  
\* It’s often a matter of style to choose the IF statement or the **SWITCH** statement because both can really achieve the same thing.  
\* The if statement is a little bit more flexible in what we can test for, we don’t have to use the same test criteria. In SWITCH we’re just testing that 1 same variable.  
\* You can even do something like this:  
  
\* You don’t have to use `break` in the `default` because it’s the last line anyway.  
\* The SWITCH statement can be used with 4 primitive types: **byte, short, char, int**.  
\* In JDK version 7 they added the capability of using the SWITCH with **Strings**.  
\* name**.toLowerCase()**. It’s attached to the String class.  
\* name**.toUpperCase()**.

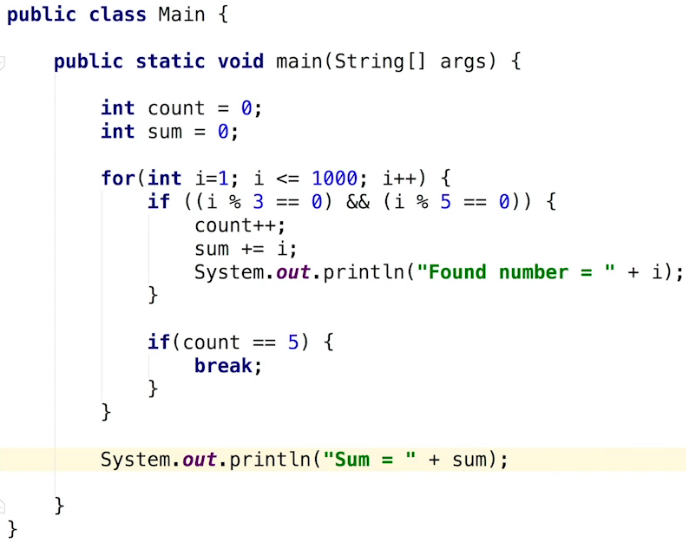
**Day of the Week Challenge**  


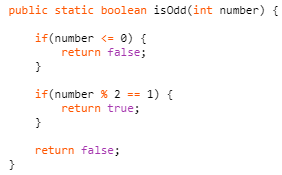
**Coding Exercise 12: Number in Word**  
  
**Coding Exercise 13: Number of Days in Month**  


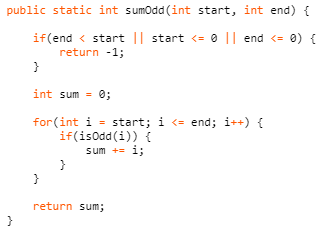
**The FOR Statement (+ Challenge Exercise)**  
 **\* This is the basic format of the FOR statement:  
\* FOR loop.** **=> init means the code that’s gonna be initialized once at the start of the loop.  
=> termination means we tell the for loop how/when/at what point we want to exit the loop, once it’s false, it’ll exit the loop, kind of like WHILE.  
=> increment is an expression that’s invoked after each iteration of the loop.**  
\* Java is smart enough to convert a whole number int to a double so it’s doing that for us. You could cast it (double).  
  
\* Double sometimes gives you many decimals at the end so to keep it to 2, you can use:  
**String.format(“%.2f”, variable)**  


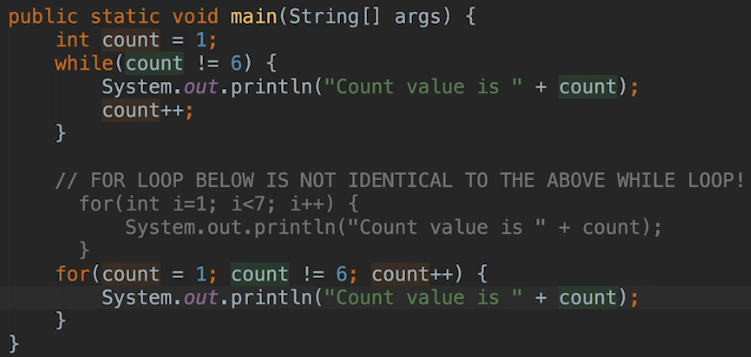
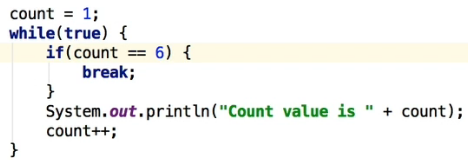
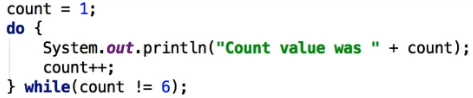
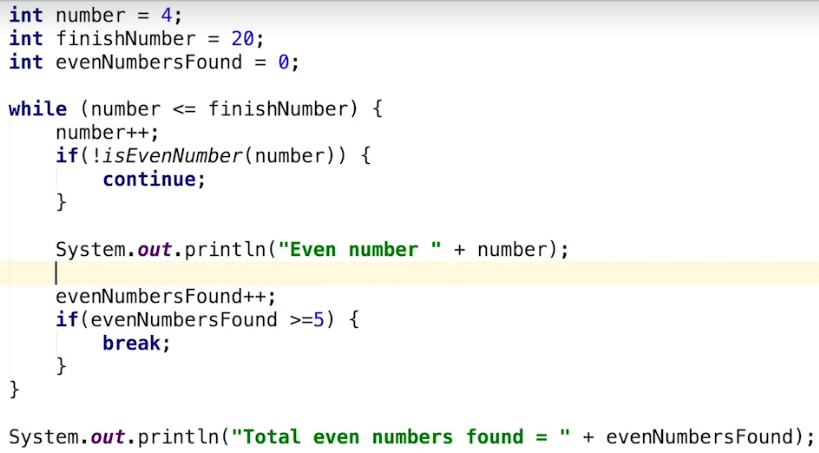
\* We can also optimize the isPrime() function more:  
  
<https://primes.utm.edu/lists/small/1000.txt>   
\* There’s also a **FOR EACH** loop, we’ll see that later in the course.

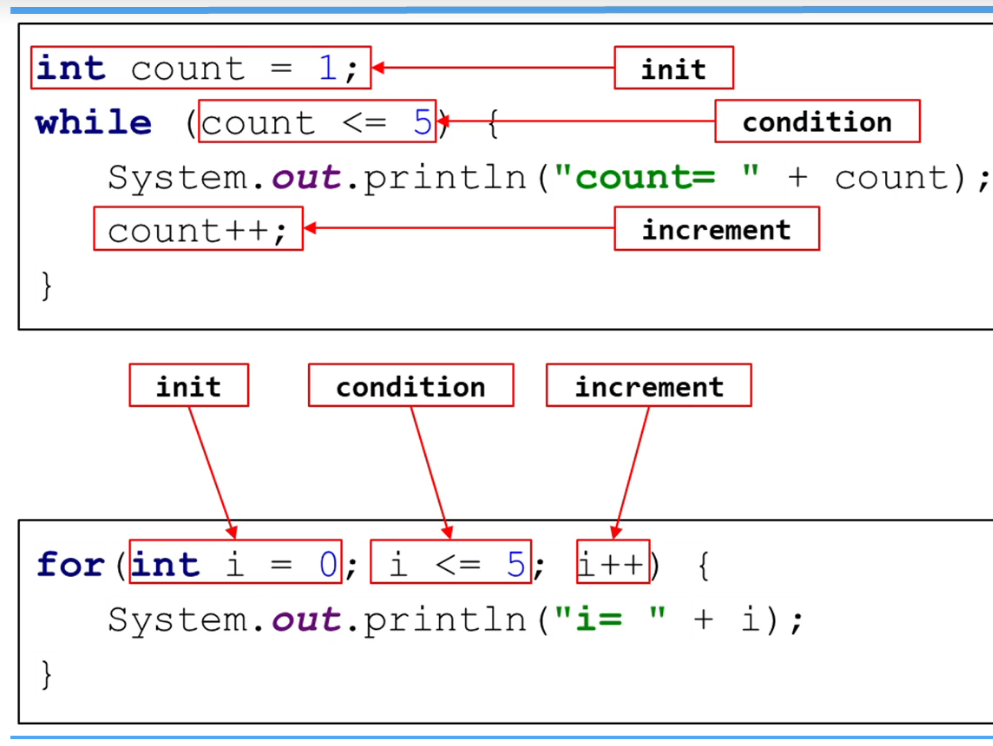
**For Loop Recap**  


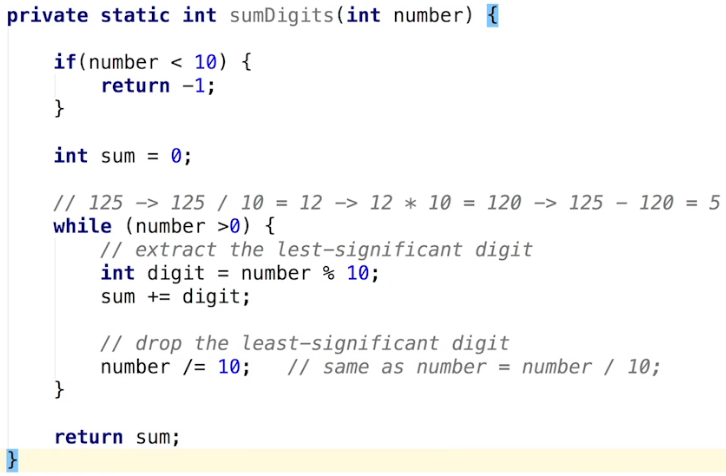
**Sum 3 and 5 Challenge**  


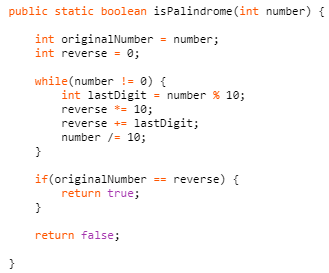
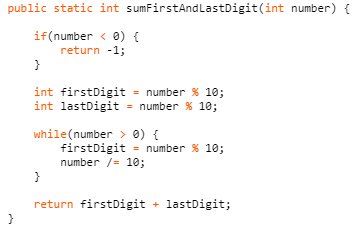
**Coding Exercise 14: Sum Odd**  


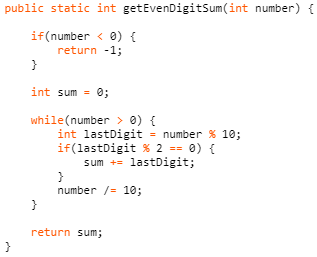
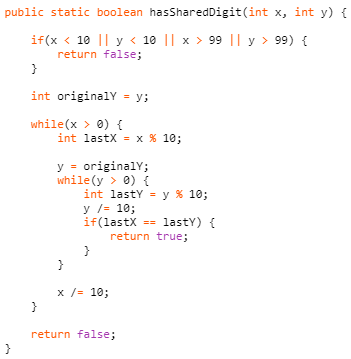
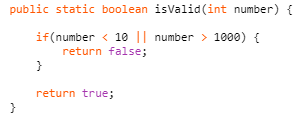
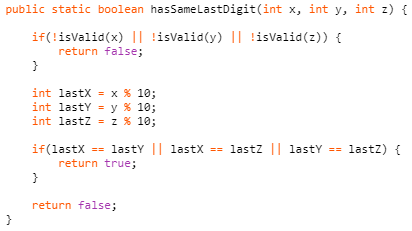


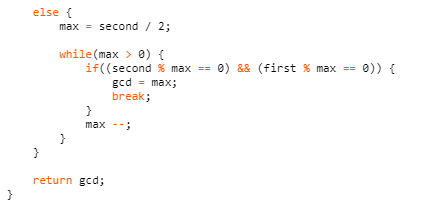
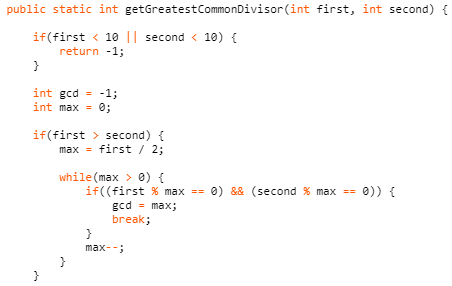
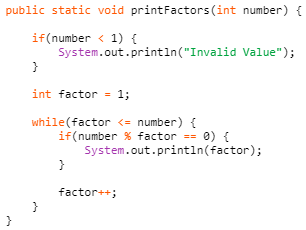
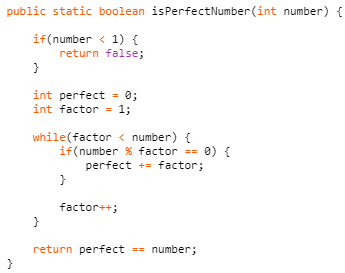
**The WHILE and DO WHILE statements (+ Challenge Exercise)  
WHILE**  
\* Another way to do it:  
  
\* **DO WHILE always executes at least once**. **Semicolon after `while` is required here**.  
\* **You can use break; in while/do while loops**.  
  
\* We can also use **continue;**.  


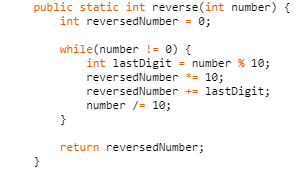
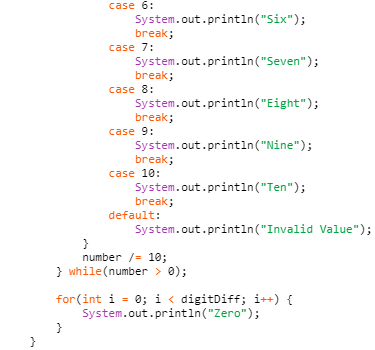
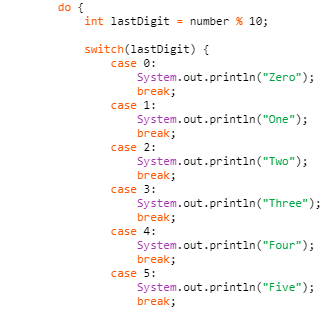
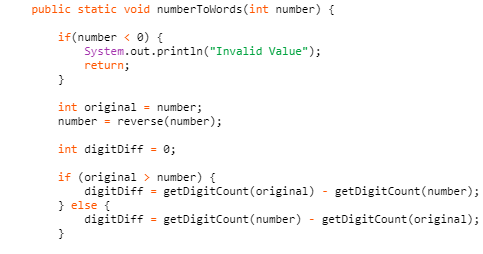
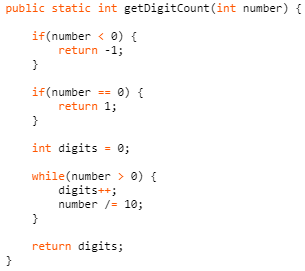
**While and Do While Recap**  


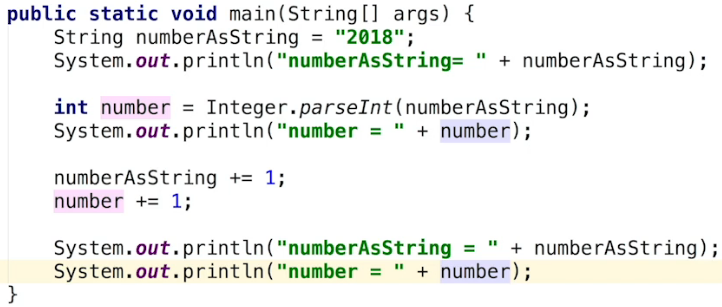
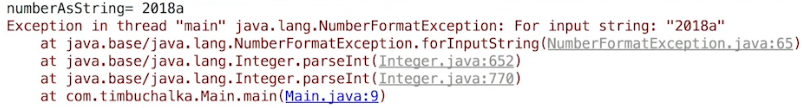
**Digit Sum Challenge**  


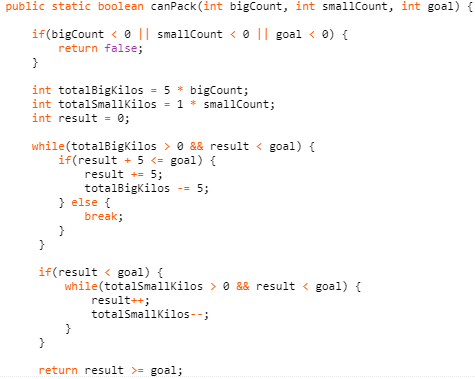
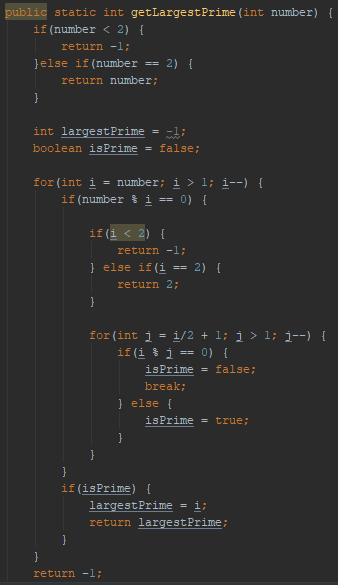
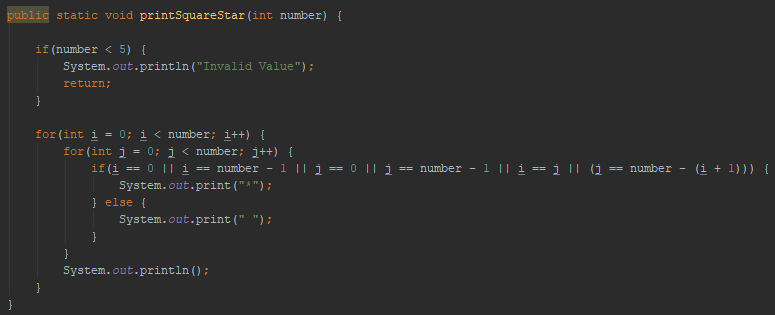
**Coding Exercise 15: Number Palindrome**  
  
**Coding Exercise 16: First and Last Digit Sum**  


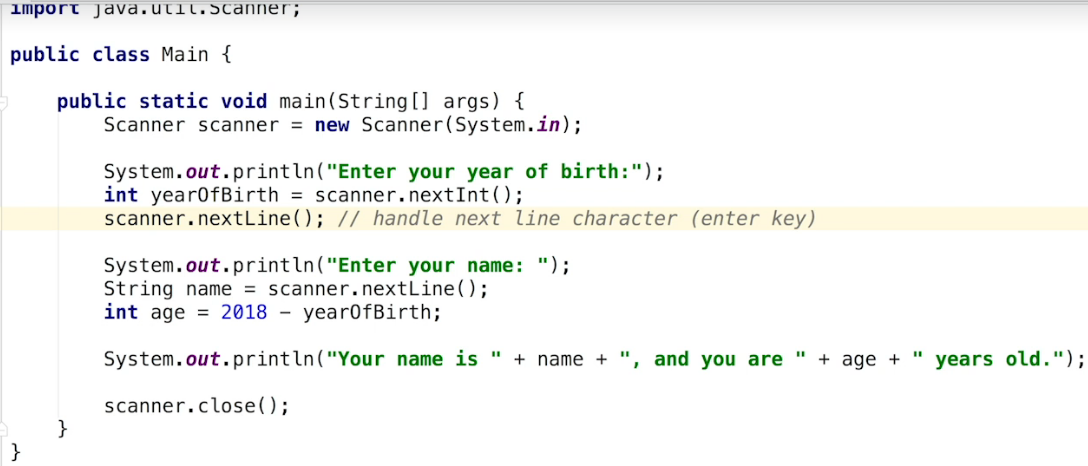
**Coding Exercise 17: Even Digit Sum**  
  
**Coding Exercise 18: Shared Digit**  
  
**Coding Exercise 19: Last Digit Checker**  


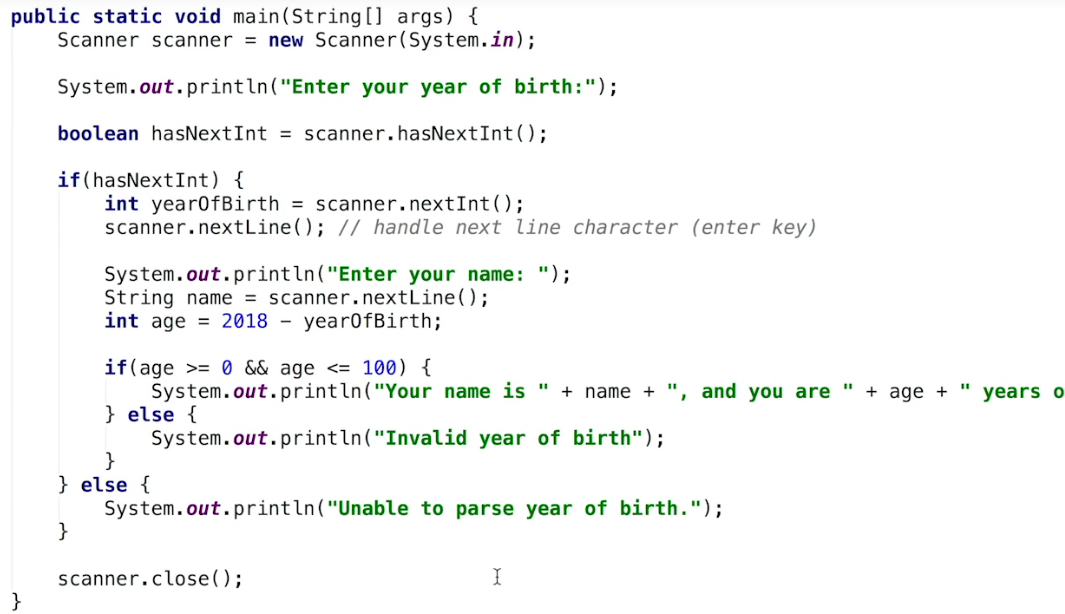
**Coding Exercise 20: Greatest Common Divisor**  
  
**Coding Exercise 21: All Factors**  
  
**Coding Exercise 22: Perfect Number**  


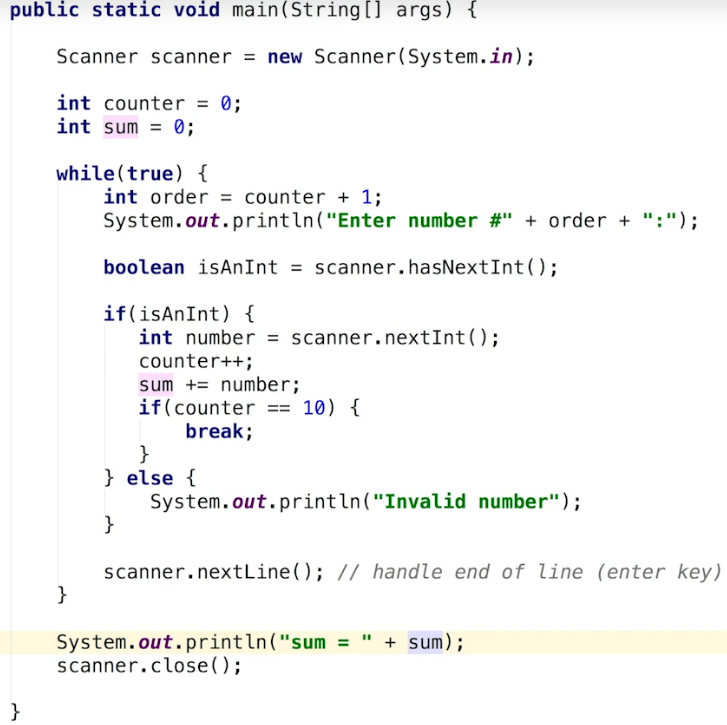
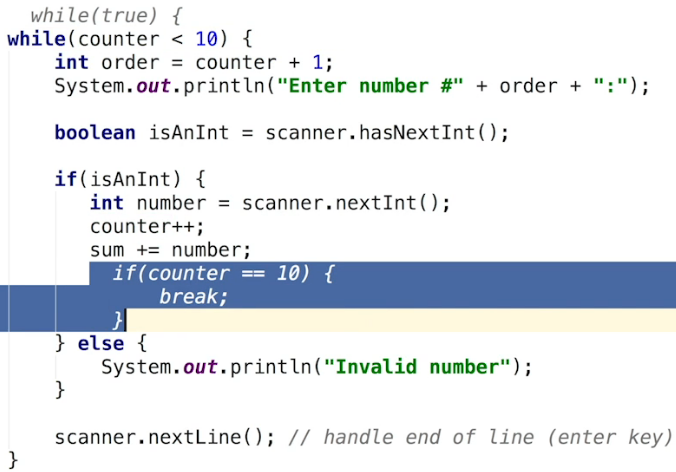
**Coding Exercise 23: Number to Words**  
  


**Parsing Values from a String**  
\* In many cases you’ll have to parse the value from a string, meaning that you’ll have to convert the string into some other data type such as an int.  
\* One use case is when reading input from the user (from console or user interface).   
\* One way to convert a string into a number is by means of a **parsing method**. With these methods we can convert a string into various primitive types depending on the specific parse that we use.  
  
\* We’re using a **CLASS** called **Integer**. This is a **wrapper class** for the primitive type `int`.  
\* It will try to convert the string that we’re passing as an argument into an integer. If it’s successful, it is converted into an integer, if it fails the conversion, we’ll get an error.  
\* This **parseInt()** method is a static method that we can use to convert a string type into a number.  
\* **stringNumber += 1 will convert the `1` into a string and concatenate it. “2018” + 1 = “20181”**.  
  
  
\* If the string is not a valid number:  
  
\* There are a couple of ways to handle this problem.   
**1)** **One way is to handle the actual Exception but we’re going to be covering handling exceptions later in the course**.  
**2)** **We could also create our own method as an alternative but that too will come up later on in the course**.  
\* We can do similar conversions with different data-types.  
\* **Double.parseDouble()**.  
\* This adds .0 to 2018 => 2018.0.  
\* There’s a similar parsing process that can be used for other types like float or long but keep in mind that when we call methods such as parseDouble(), the associated class must be used before calling the method.

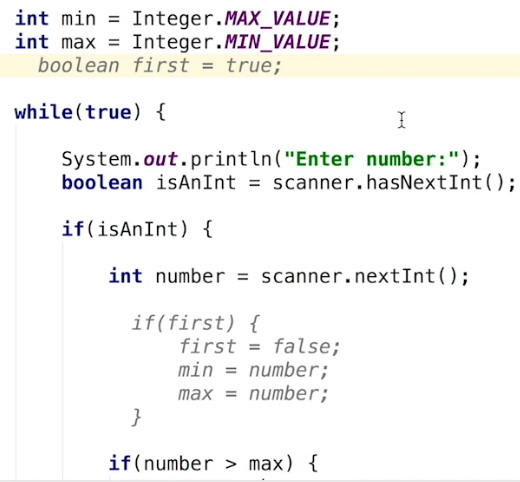
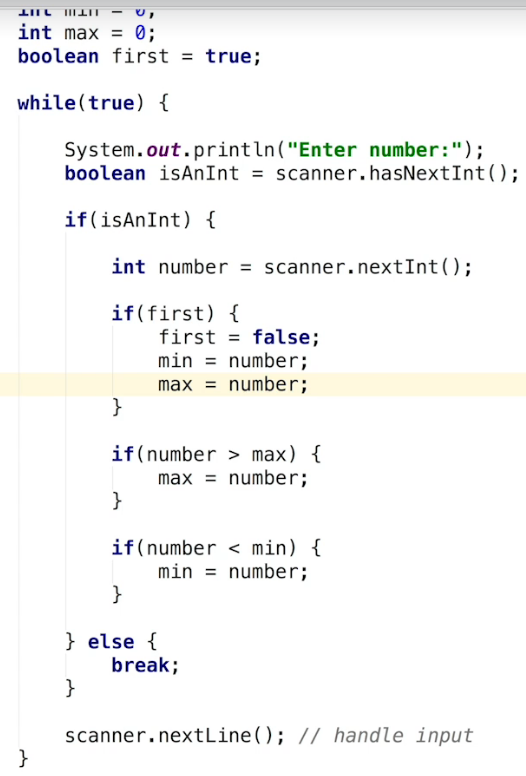
**Coding Exercise 24: Flour Pack Problem**  
  
**Coding Exercise 25: Largest Prime**  
  
**Coding Exercise 26: Diagonal Star**  
  
\* **System.out.print(“”)** => to print on the same line, it prints a space and does not “move”.  
\* **System.out.println()** => to move to the next line.

**Reading User Input**  
\* **Scanner** scanner (**java.util**) = **new Scanner**(**System.in**)  
\* **Scanner can parse both Primitive types and Strings**.   
\* **So basically Scanner uses methods like parseInt internally**.  
\* We’re using the `new` keyword to create an Instance of Scanner, meaning that we’re creating a new object of type Scanner.  
\* **scanner.nextLine()**.  
\* **scanner.nextInt()**.  
\* **scanner.close()**.  
\* After we’re finished using a scanner, we should close it using the close method. Note that after closing the scanner, calling methods like .nextLine or .nextInt will cause errors so we should really ensure we’re done using the scanner before closing it. So it’s recommended to close the scanner when we don’t need it anymore as it then releases the underlying memory that scanner was using internally.  
\* You could also use the .nextLine() to retrieve the year and then convert that to an integer.  
\* **Whenever we read a number from the scanner, we press the ENTER key on the keyboard and doing that ends the line. So it’s recommended after nextInt to call nextLine again without assigning it to a variable. In other words: to handle the ENTER key issue, we have to call .nextLine so that the scanner works as expected**. Now internally the scanner is checking for a line separator and when we hit ENTER, we’re typing a line separator, so that’s interpreted as input. When we reach the .nextLine method, that retrieves the name, the input for the name becomes ENTER, so it’s essentially skipped.  
  
\* So keep this in mind for the future - after you read a number with a scanner, there must be a .nextLine method call to handle the ENTER key.

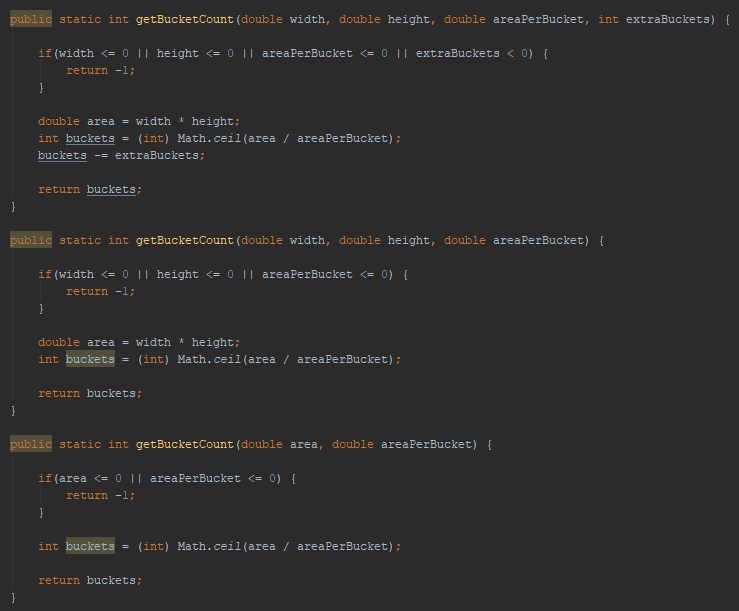
**Problems and Solutions**  
\* There are a few potential problems in the last video.  
\* What if the user enters a negative number for the year of birth or if they enter some letters instead of only numbers. -20 would give us 2038 years old.  
\* So we can add some conditions to check if the birth year is positive or even better - check a valid range. If it’s not in the range, we’ll print an error message to the user.  
\* If the user times letters as the age, we get an Exception - an error and the program exits.  
\* **scanner.hasNextInt()** - checks to see if the next input entered is a number, however, it won’t remove it from the scanner so in other words it’ll ask the user for input and check to see if it qualifies as an int. If it doesn’t quality, the method will return false so it allows us to avoid generating type errors when using .nextInt() but we need to add some extra code to add to the flow.   
\* We’ll add an IF statement but don’t include the scanner.close() in it.  


**Reading User Input Challenge**  
  
\* We can do this instead of WHILE TRUE => use counter < 10 and we can remove the BREAK.  


**Min and Max Challenge**  
\* **We can use a boolean flag to check if the user is entering the first number and set the min to it**.   
\* Boolean is often referred to as a **FLAG** - you can only have 2 states true/false on/off.  
\* **Another way is to set the min extremely high --- set max to the minimum value that an INT can hold, now we can do this because the user will only be able to enter a number that is >= to the minimum int value**. Now the maximum INT value is approximately positive 2 billion while the minimum is approximately negative 2 billion.  
\* We can use CONSTANTS   
=> **Integer.MAX\_VALUE**  
=> **Integer.MIN\_VALUE**  
\* **Every Primitive Number numeric type has these MAX\_VALUE and MIN\_VALUE constants in their respective wrapper class**.



**Coding Exercise 27: Input Calculator**  


**Coding Exercise 28: Paint Job**  


**Resources**  
x