8. Working with Enums

Contents

[1. Introduction 1](#_Toc18150)

[2. Conditional Logic 1](#_Toc19848)

[3. Relative Comparisons and Common Methods 2](#_Toc7462)

[4. Representing Math Operations as Enums 4](#_Toc14443)

[5. Processing Interactive Input in CalcEngine 5](#_Toc27180)

[6. Enum Types Are Classes 6](#_Toc9319)

[7. Summary 7](#_Toc14511)

# Introduction

=>slides: Pg. 2

Welcome to our next module, Working with Enums.

=>slides: Pg. 2

Oftentimes in our applications, we may want to have a pre‑defined list of values that we want to be valid particular fields or variables, and that's where enum types come in. So in this module, we'll start out, we'll look at how enum types solve this particular kind of problem. We'll also see how we can declare enum types, which allows us to define a type that only supports this finite list of values. As we work with these types, we're, of course, going to need to do conditional logic with it. So we'll see how conditional logic works when dealing with an enum type. And these kind of comparisons don't have to be limited to simple, equal to, or not equal to comparisons. So next, we'll see how we can do relative comparisons with enum types. So one of these values can be compared to another to see if it's greater than or less than that other value. Now enum types go beyond just simple values. Enum types also have methods. And whenever we declare an enum type, there are certain methods we automatically have access to. And in fact, enum knew types are actually classes, which means we can actually associate characteristics with each of the values within our enum type. So we'll finish up this module. We'll look at how we can provide our own characteristics for each of our enum types.

# Conditional Logic

=>slides: Pg. 3

As we mentioned, enumeration types are useful for when we define a type that has a finite list of valid values. In other words, enumeration types allow us to create a brand‑new type and then specify what the valid values for that type are. Now the way we declare an enumeration type is by using the enum keyword. So, for example, here we have a type named FlightCrewJob, and we specified that it's an enum. So as we declare this type, one of the things we want to do is provide the values that are part of the type. So we're going to do that with a comma‑separated list. Now, by convention, each of the values will be in all uppercase. So if we're going to provide a list of FlightCrewJobs that our application allows, we could say we have a FLIGHT\_ATTENDANT, we can have a COPILOT, and we can have a PILOT. Now once we declare this, FlightCrewJob is a first‑class type within our application, and what that means is if we declare a variable to have the type FlightCrewJob, the compiler will assure that only one of these values, FLIGHT\_ATTENDANT, COPILOT, or PILOT is ever assigned to that variable.

=>slides: Pg. 4

Now once we have our enumeration type defined in our application, in general, we're going to want to perform some kind of conditional logic. And enumeration types support equality tests. So we can use the == or != operators. So let's say we have a variable here, job1, whose type is FlightCrewJob. We've given it a value of PILOT. Now notice that when we assign this value, we can't simply say PILOT. We have to qualify it by the type. So we say job1 = FlightCrewJob.PILOT, not simply job1 = PILOT. So let's go ahead and declare another variable here, job2. We'll specify the value for job2 as FlightCrewJob.FLIGHT\_ATTENDANT. So now, once we've assigned values to the variables, we can do our conditional logic. So I can check job1 here against that value FlightCrewJob.PILOT. And, again, we're doing this comparison with the == operator. So in this case, we'd go ahead and print out "job1 is PILOT." Now we can do these equality or inequality comparisons using not just the constants, but also we can compare two variables. So I could say if(job1 != job2) which, in this case, they are unequal, we could then take an action, like print out "job1 and job2 are different." Now our enumeration types will generally have multiple values. So, oftentimes, we'll want to go beyond a simple == or +!. We'll want to have different branches in our code, depending on which particular value a variable has. And we'll do that using switch statements. So within an enumeration type, we can perform a switch and have a separate case for each of the values for that type.

=>slides: Pg. 5

So let's look at some code. We'll add a method displayJobResponsibilities. Notice it has one parameter named job of type FlightCrewJob. So inside this method, we can have a switch statement that switches on job. Then inside the switch statement, each case can be a value for FlightCrewJob. So I can have a case here for FLIGHT\_ATTENDANT. Now one thing to note, inside the switch statement, we don't have to type qualify the value. We don't say Case FlightCrewJob.FLIGHT\_ATTENDANT. We simply say case FLIGHT\_ATTENDANT. Then inside the case, we just do whatever work we need to do. So in this case, we'll print out the message "Assures passenger safety." Now, remember, at the end of each case, we want to have a break statement, and then we could provide our next case. So we'll have case COPILOT, and for COPILOT, we'll print out the message "Assists in flying the plane." And for the PILOT, we'll say, "Flies the plane." So as you can see here, by using an enumeration type, we've got really readable code. It's very clear what scenario each case is handling. All right, so now in our next section, let's take a look at relative comparisons.

# Relative Comparisons and Common Methods

=>slides: Pg. 6

As we deal with our enum types, it's important to understand that the values are not just a simple list of values. Those values are actually ordered. So one value could be considered greater than or less than another value. Now the first value on the list is going to have the lowest ordering. The last value on the list is going to have the highest ordering. And this allows us to do relative comparisons. So when we declare a variable to be of an enumeration type, that variable actually holds a reference to that value, so we can call the compareTo method against one reference to compare it to another reference. Now compareTo returns back either a negative value, zero, or a positive value. And this indicates the relative ordering in the comparison. So we call compareTo against one reference, passing in another reference. And return value indicates how the one you've called compareTo against compares relatively to the value you pass in. So if the return value is negative, the one you've called compareTo against is ordered lower than the value you've passed in. If the value is positive, the one you've called compareTo against is ordered higher than the value you passed in.

=>slides: Pg. 7

So let's see what this looks like in code. So we again have our FlightCrewJob enumeration. We have our three values, FLIGHT\_ATTENDANT, COPILOT, and PILOT. So since FLIGHT\_ATTENDANT is the first value on our list, it's considered to have the lowest ordering. Then we have COPILOT. So COPILOT would be considered to be ordered higher than FLIGHT\_ATTENDANT. And then PILOT we'd consider to be ordered higher than both COPILOT and FLIGHT\_ATTENDANT. So we also have a class here, CrewMember, and CrewMember has a field named job whose type is FlightCrewJob. Let's go ahead and add another field, name, whose type is String. We'll give CrewMember a constructor that accepts a job and a name and then assigns those values to our fields. So now we have our CrewMember class, which is leveraging our FlightCrewJob enumeration type.

=>slides: Pg. 8

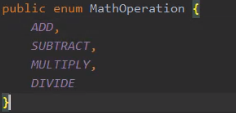
So let's go ahead and create some instances of CrewMember. So we'll create a new CrewMember here. We'll specify the job as PILOT, and the CrewMember's name is Geetha. Then we'll create another CrewMember whose job is FLIGHT\_ATTENDANT and name is Bob. Let's go ahead and call a method we'll create called whoIsInCharge passing in Geetha and Bob. So we have our whoIsInCharge method that accepts two parameters, both are CrewMembers. These parameters are member1 and member2. And the job of whoIsInCharge is to figure out which of these two members are theBoss. So we have this local variable here, theBoss, whose type is CrewMember. So what we'll do is say member1.getJob. So that would give us a reference to the job field for this CrewMember, and then we'll call compareTo against that job, and we'll pass in member2.getJob. So, basically, what we're doing here is seeing how the job for member1 compares to the job for member2. And we'll check its return value. So here we'll check and see if it's greater than 0. So if the return value is greater than 0, that means that member1 is ordered higher than member2, and that's a value we'll assign to theBoss. Otherwise, we'll assign member2 to theBoss. So now theBoss will have a reference to the CrewMember whose job is ordered higher. So we could then use theBoss to print out that CrewMember's name. So in this case, because Geetha was a PILOT and Bob is a FLIGHT\_ATTENDANT, in this case, we'd print out that Geetha is boss.

=>slides: Pg. 9

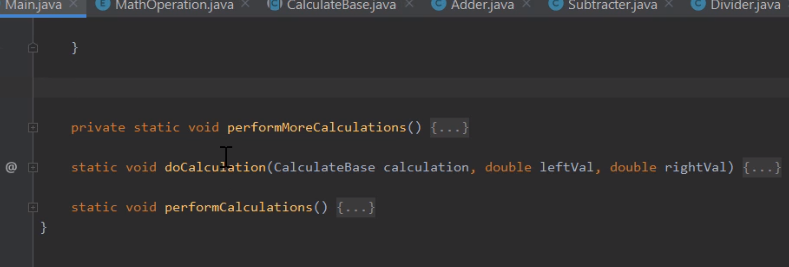
Now the capabilities of enumeration types go well beyond the values themselves. There's actually a number of methods that we have access to as part of an enumeration type. A couple that we use commonly are values, which will give us back an array containing all of the values for that enumeration type. And we also have valueOf. ValueOf can actually translate a string into the corresponding enumeration value. So if you type in a string with the value of that enumeration type, we'll get back the corresponding enumeration. Now it's important to note that valueOf is actually case sensitive. So the case of the string has to match the case that's used for that value. Since our enumeration values are normally all uppercase, that means the string would have to be all uppercase as well. So to give us a better understanding of all this, in our next section, let's get into our CalcEngine project and see how we can leverage enumeration types.

# Representing Math Operations as Enums

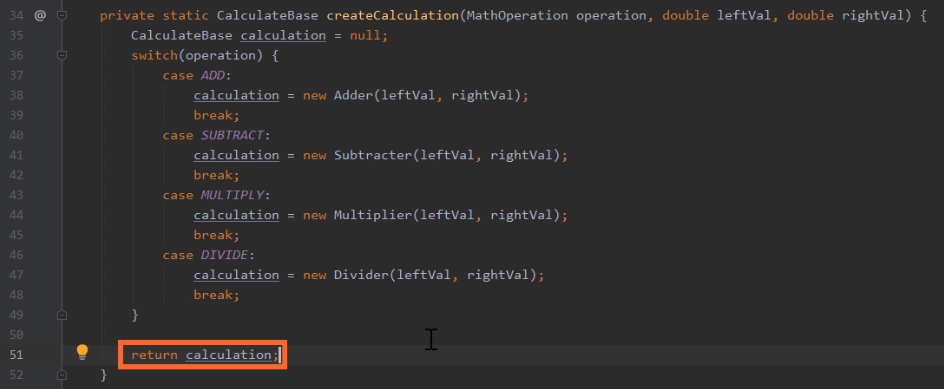
Here we are in STS, and what we want to do now is add some interactive capabilities into our CalcEngine project. We want to allow people to type in an operation and some values, and the application will perform that operation. And as part of creating this behavior, we're going to use an enumeration type. So to create our enumeration type, let's head over here to our project window. Then we'll right‑click on our package name. We'll head up to New.



Then even though we're creating a new enumeration type, we're going to choose Java Class. And you'll notice that when this New Java Class dialog comes up, it has a series of options down here at the bottom. One of those options is Enum. So we'll go ahead and choose that. And this enumeration type is going to contain values for the four basic math operations. So we'll name this type MathOperation. So go ahead and hit Enter. And then here in the enumeration type, we just need to add the values we want for this type. So our first value will be ADD. So now, remember, as we add these values, each value has to be comma‑separated, and by convention, the value name is all uppercase. So we've got our ADD value here. Let's go ahead and provide values for SUBTRACT, MULTIPLY, and DIVIDE. So now our MathOperation type has values for all four math operations. Now as you recall in our previous module, we added classes that could perform each of the math operations. We had an Adder class, a Divider class, a Multiplier class, and a Subtracter class. So we could, of course, use those classes to perform the operations that correspond to the value of our MathOperation enumeration type. Now remember that one of the things those classes had in common is they all inherit from a common base class, CalculateBase. So what we could do is have some code that uses the MathOperation value to create the appropriate class instance. And since all those classes have a common base class, we could put that into a method whose return type was CalculateBase. So let's add that code over in our Main class. So I'll go ahead and collapse our project window and head over to our Main class.

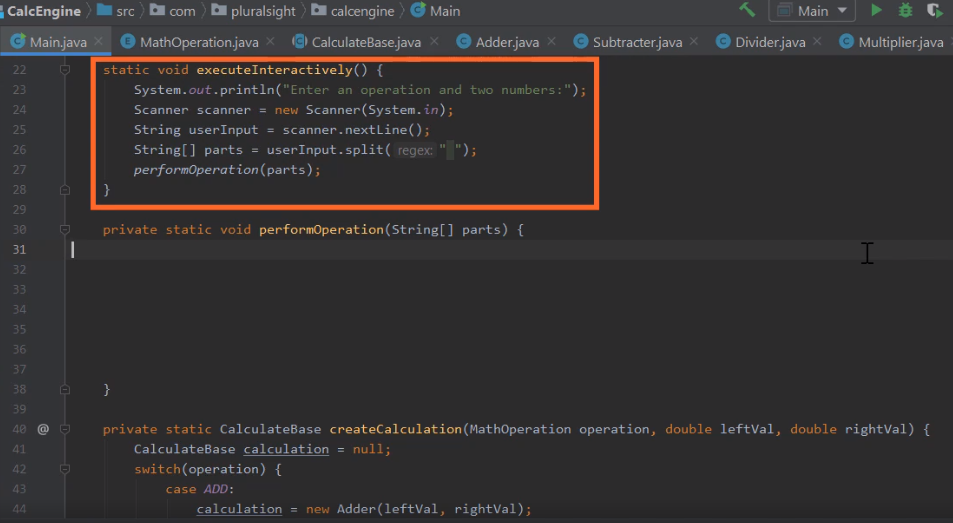


So now we're here in our Main class.

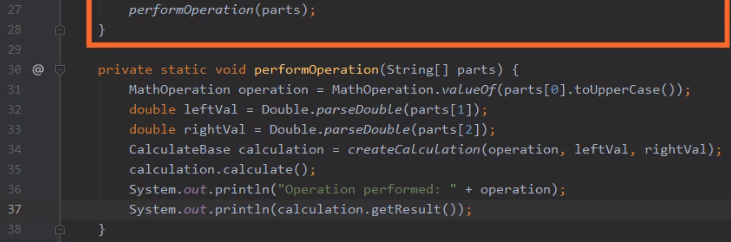


So let's go ahead and add a method named createCalculation with a return type of CalculateBase. So now our first parameter to createCalculation will be of type MathOperation, and we'll name it operation. So that allows us to specify what operation we want to perform. Now as you may recall, as we create each of these classes, one of the constructors they have supports passing in the leftVal and rightVal. So let's go ahead and include parameters for those two values. So now our createCalculation method accepts parameters that indicate the operation and the two values that we want to operate on. So now inside of here, we can do the work of creating the different types. So now the first thing we'll do is create a local variable named calculation of type CalculateBase, and we'll use our calculation variable to hold a reference to whichever class instance we create. Now remember that our MathOperation is an enum. And since it's an enum, we can actually perform a switch on it. So let's add in a switch for our operation parameter. Then within our switch, we can have a case for each of the values. So let's start with a case for ADD. And in the case of ADD, we'll create a new instance of our Adder class passing in leftVal and rightVal, and we'll assign it to our local variable calculation. Then after we create the instance of the Adder class, we'll put in a break statement. So with that, when we receive an operation of ADD, we take care of creating an instance of our Adder class. So now let's do the same sort of thing for SUBTRACT, MULTIPLY, and DIVIDE. So now with that, our switch statement handles all four values in our MathOperation type, and in each case creates an instance of the appropriate class and assigns that reference to a local variable calculation. So the last thing we'll need to do here is have our createCalculation method return calculation. So now our createCalculation method is complete. We passed in an operation and two values, and we get back a reference to a type that will perform that operation. So now we're ready to add our interactive capabilities, and we'll do that in our next section.

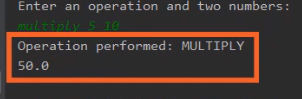
# Processing Interactive Input in CalcEngine



Here we are back in STS, and what we want to do now is finish the job of adding interactive capabilities to our application. Now I've actually added a little bit of code in here to get us started, and notice we have this method here, executeInteractively. And this method takes care of the details of prompting the user and getting that user's input back. And the technique I'm using here is the same technique we used in our earlier course, Getting Started with Programming in Java. Now let me just walk through it quickly as a reminder. Now as the first thing we do is print out a message of the user to tell them to add an operation and two numbers. And then we use this type named scanner to actually get back the input from the user. So we call this method nextLine. That gives back a string containing all the values the user typed in up until they hit the Enter key. So what we're expecting the user to type in is something like add 10 20, an operation name, along with two values, with each separated by a space. So then what we do is use the split method to break that string up into its three parts. So if the user types in add 10 20, this parts array will have three elements. Element 0 will be the string add, element 1 will be the string 10, element 2 will be the string 20. So we then take our parts array and pass it into a method we have here, performOperation. So we're going to do our work to process that input here in performOperation. So now the first thing we need to do is identify the operation that the user wants to execute.



So let's declare a local variable here named operation of type MathOperation. Now remember that a string identifying the operation is in part subzero. And as we mentioned in the slides, enumeration types can actually translate strings into the equivalent enumeration value, and we do that with the method valueOf. So let's call MathOperation.valueOf, passing in parts subzero. So calling valueOf will attempt to translate that string into the corresponding MathOperation value, but remember that valueOf is case sensitive, and all our operation names are all uppercase. So we want to make sure that this string that we're processing is also in uppercase. So to take care of that, we'll call toUpperCase against parts subzero. So now, no matter what case the user used when typing in the operation, we'll process it as uppercase, and that should give us back the appropriate operation. So now the next thing we need to do is translate the strings containing the values into doubles. So let's declare local variable named leftVal of type double. And to do the translation, we use the same technique we used in our previous course. Remember, there's a class named Double, and it's Double spelled with a capital D. Well, the Double class has a method parseDouble that can translate a string into its equivalent double value. So we'll call Double.parseDouble, passing in (parts[1]). So that will set our leftVal to the appropriate value. So we'll do the same thing for our rightVal with (parts[2]). So that gives us our operation and both values. So now we can use our createCalculation method, and we'll pass in operation leftVal and rightVal. Now remember, when we wrote the createCalculation method, we receive the operation in both values. We then create an instance of the appropriate class, passing back a reference to it as a CalculateBase reference. So let's assign the result of createCalculation to a local variable name calculation of type CalculateBase. So once we get back our reference into calculation, we can call calculate. Now remember when we call calculate, it performs the appropriate operation and stores the outcome of that operation into its result field. So that means we've actually done all the work at this point. The only thing we need to do now is print out the outcome. So let's print out a couple things. The first thing we'll do is print out the value in our operation variable. And then after that, we'll go ahead and print out the result, and that's all there is to it. Basically, the executeInteractively method took care of getting the input from the user and passing the three parts of the input into our performOperation method. Here inside performOperation, we determine the operation and the two values. We then got a reference to the appropriate type to perform that operation called calculate to do the work and then simply printed out the name of the operation and the result of the calculation. So with that all done, let's go ahead and run our application, and we'll verify that everything works.



So now if we look down here in the Run window, we can see our prompt to enter an operation and two numbers. I'll go and click in the Run window to give it focus, and that's going to enter multiply 5 10. So now we've our operation and both values with each part separated by a space, so I'll go ahead and hit Enter. And when I do that, we see the output is Operation performed: MULTIPLY, and the result is 50. So our application worked perfectly. So by using our enumeration type, implementing this behavior was very simple. We were able to easily get the operation from the user and then create the appropriate type that corresponded to that operation. Alright, so now in our next section, let's take a look at some of the ClassBase features of enumeration types

# Enum Types Are Classes

=>slides: Pg. 10

And although it may not look like it, when declaring an enum type, we're actually declaring a class, but there are some key differences from other classes we declare. One key difference is that they implicitly inherit from Java's Enum class, and this provides many of the special capabilities that we associate with enum types. Now there are many aspects of enum types that are similar to other classes, but there's some really key characteristics that are special about enum types.

=>slides: Pg. 11

Now in terms of similarity with other classes, one of the key things is enum types can actually have members. So that means that our enum types can have fields, our enum types can have methods, and our enum types can even have constructors, which means we can actually pass in initial state when we're creating an instance of our enum type.

=>slides: Pg. 12

But now one of the really key differences about enum types is the way we create instances of our enum types. It turns out that each value within an enum type is actually an instance of that enum type. So that means that simply declaring the value creates the instance. And as part of declaring that value, we can even leverage a constructor. So as part of declaring the value, we can pass initial state into that instance as we declare it.

=>slides: Pg. 13

Now this all may sound really confusing, so I think the best thing for us to do is look at some code. So we'll get a look at our FlightCrewJob enum type. Remember, we have three values here, FLIGHT\_ATTENDANT, COPILOT, and PILOT. Now, so far, we've only listed the values. So let's get ready to add some members to this FlightCrewJob enumeration type. Now one key thing to note. If we're going to add members to our enumeration type, here at the end of the list of values we need to provide a semicolon. So when we were just listing the values, we didn't need the semicolon, but if we're going to add members, we do need to have a semicolon at the end of the list. Now in terms of the members we want to put inside of this type, pretty much any members you would put into a class. So let's go and add a field to our enumeration type. So we have this field title, which will be the title we want to associate with a particular job. For example, the pilot's title might be captain. Now we could have more fields here if we needed to, but in this case, we just need one. Let's go and add a method, getTitle, so that will be a getter for our title field. And then we need some way to specify the title, so we'll do that with our constructor. So we have a constructor here that accepts title as a parameter, and that will set our title field. Now as we mentioned, the values that we list here are actually instances of our FlightCrewJob enumeration type. So it's these values themselves that are creating the instance. So with each of these values, we can actually call the constructor.

=>slides: Pg. 14

So in the case of PILOT, we want the title to be Captain. So after the value PILOT, we'll pass in the parameter we want to pass to the constructor, which is the string, Captain.

=>slides: Pg. 15

And then here for our COPILOT, we'll set that title to be First Officer.

=>slides: Pg. 16

And then here for our FLIGHT\_ATTENDANT, we'll set that title to be Flight Attendant. So now here within our FlightCrewJob, we have three instances of FlightCrewJob, FLIGHT\_ATTENDANT, COPILOT, and PILOT, and in each case, we've passed in the title we want to associate with that instance.

=>slides: Pg. 17-18

So now let's revisit some code that we wrote earlier. As you recall, we had that method we wrote, whoIsInCharge. And when we were using whoIsInCharge, we created one instance of a crew member who was a pilot name Geetha, then we created another instance of a crew member who was a flight attendant named Bob, and we called whoIsInCharge, passing in Geetha and Bob. Then down inside of whoIsInCharge, we used compareTo to figure out which one of those two members is the boss. And then once we determined who is the boss, we wrote out a message to that effect. So in this case, because Geetha is a pilot, we wrote out, Geetha is boss. So now let's see how we can leverage the new capabilities we've added to our type FlightCrewJob. So up here before we print out the name, we'll use our theBoss reference to get access to the job. Now remember, that'll give us back a reference to pilot, and pilot is an instance of FlightCrewJob, and FlightCrewJob has a method, getTitle. So this will give us back the title that's associated with that instance of FlightCrewJob, which in this case, would be Captain. Then after the title, let's go and concatenate a space. So now as we print out the message, rather than simply being Geetha is boss, the message will now be Captain Geetha is boss. So now, thanks to the enhancements we've made to our FlightCrewJob enumeration type, each job beyond simply being a value, actually has state associated with that value.

# Summary

=>slides: Pg. 19  
  
To wrap up, here are some of the key things you want to remember from this module. Remember that as we talk about enumeration types, we're talking about types that allow us to define a finite list of valid values. In other words, we're in effect creating our own type and specifying what values can be assigned to variables of that type. Now as we work with our enumeration types, we're going to need to perform conditional logic. We want to check to see what values are contained where. Now we can do simple equality tests or inequality tests, comparing a variable to one of the values in the enumeration type, as well as comparing two variables against one another. And oftentimes as we work with enumeration types, we want to use switch statements, and switch statements work really well with enumeration types because it allows us to branch our logic depending on which of our enumeration values are contained within a particular variable. Now remember that the values that we specify are ordered. The first value we list has the lowest ordering, the last value we list has the highest ordering, so we can perform order‑based comparisons. We do that by using the compareTo method.

=>slides: Pg. 20  
Also remember that enum types are actually classes. They implicitly inherit from Java's Enum class. And we can do many of the things with enum types that we do with other classes, for example, we can define members. So our enum types can have fields, they can have methods, they can even have constructors. Now when we want to create an instance of our enum type, we don't do that by using the new operator. Instead, we rely on the values that we specified for that enum type, because the values are actually instances of that enum type. So simply declaring a value within the enum type creates an instance of that enum type, and we can specify the state of that instance by leveraging our constructors. So as part of declaring the value, we can specify the parameters that we want to pass into that constructor. Alright, that wraps up this module. In our next module, we'll see how we can create abstract relationships using interfaces.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*