9. Creating Abstract Relationships with Interfaces

Contents

[1. Introduction 1](#_Toc1327)

[2. Understanding Interfaces 2](#_Toc19880)

[3. The Comparable Interface 3](#_Toc31524)

[4. Generic Interfaces 4](#_Toc5482)

[5. Implementing Multiple Interfaces 5](#_Toc30744)

[6. Declaring an Interface 6](#_Toc11902)

[7. Adding and Implementing an Interface in CalcEngine 7](#_Toc22315)

[8. Using the Interface in CalcEngine 8](#_Toc1013)

[9. A New Implementation of the Interface 9](#_Toc10709)

[10. Adding a Default Method to an Interface 10](#_Toc4456)

[11. Summary 11](#_Toc30020)

# 1. Introduction

=>slides: Pg. 1-2

Welcome to our next module Creating Abstract Relationships with Interfaces.Throughout this course we've been focused very much on concrete types, that is classes and using inheritance to extend those classes. But there are times where we need to move beyond concrete relationships and instead use abstract relationships. In other words, relationships that allow us to specify rules for conformance without giving you the details of how we actually implement those rules. And that's where interfaces come in. Interfaces allow us to define abstract relationships. In a sense, an interface allows us to define a contract for interoperability. So throughout this module we'll be looking at interfaces. So we'll start out and look at why we need more than inheritance to build on existing functionality. Why do we need these abstract relationships known as interfaces? From there, we'll take a look at what an interface is. We'll understand how we can implement an interface on one of our classes. The next thing we'll look at is what's known as generic interfaces. Generic interfaces allow us to specialize an interface for a particular type. So it allows us to have these abstract relationships that are tied to a specific type. We'll then look at how we implement multiple interfaces. One of the key powers of interfaces is one class can implement multiple interfaces. So one class is able to conform to multiple contracts for interoperability. Next, we'll see how we can declare interfaces. In other words, we'll see how we can specify our own contracts for interoperability. And one of the challenges we have when we specify interfaces is that sometimes we need to change those interfaces over time, which always runs the risk of breaking existing code. So that's a really serious challenge. So we'll finish up, we'll look at default methods, and we'll see how default methods allow us to overcome that challenge.

# Understanding Interfaces

=>slides: Pg. 3

As software developers, we want to be as effective as possible, and a key part of being effective as a software developer is relying on reusability. We don't want to spend the time and the effort to build something new when a good solution already exists. Now a key part of usability is, of course, class inheritance because class inheritance allows one class to leverage the implementation of another. So if you have an existing class that does mostly what you want, you can inherit from that existing class and then specialize it for your current need. But inheritance has some limitations. Remember that in Java, a class can directly extend only one other class. Now the class that it extends can, of course, extend other classes. But one class cannot directly extend two or more classes. You can only extend one class, and that constrains the amount of reusability we can access using only inheritance because with inheritance all our reusability would have to be in base classes, and these base classes could become unmanageably large if we had to put every feature and every function that we needed in the hierarchy of base classes.

=>slides: Pg. 4

And that's where interfaces come in. Interfaces allow to define a contract for behavior, and an interface simply provides a list of operations and the behavior that goes along with those operations. But interfaces do not focus on implementation details. They simply provide the methods and the contract for their behavior. Then from there, classes can then implement interfaces. So when a class implements an interface, it expresses that it conforms to a contract. But the details of how an individual class conforms to the contract is up to the class itself. The class implements the methods that are part of the interface in a way that's appropriate for that class.

=>slides: Pg. 5

I think really the best way to understand interfaces is with an example. So we're going to look at an interface called a Comparable interface. And the Comparable interface is a very widely used interface in Java. The Comparable interface is used to solve a particular challenge. That challenge is that objects often need to be ordered. In other words, we often need a way to sort different objects. But the rules we're ordering are different for each class. Depending on the data inside of a class and the purpose of that class, the way it's ordered is going to be different from other classes. So that's where the comparable interface comes in. The comparable interface provides a contract for ordering, and by using this contract, we get the key benefit of being able to have broadly reusable sorting utilities with those utilities knowing nothing at all about the classes that they need to sort. The simple fact that the class implements the Comparable interface is all that's needed for the class to be able to leverage one of these utilities.

=>slides: Pg. 6

So let's take a closer look at the Comparable interface. Now this interface has just one method. That method is the compareTo method. The compareTo method has one parameter, which is a reference to another object. And the purpose of the compareTo method is to indicate the relative relationship between the current object and the object that it receives as a parameter. In other words, it indicates the ordering between the current object and the received object, and the way it indicates this ordering is by the return value of the compareTo method. So if the compareTo method returns back a negative value, that indicates that the current object is ordered first. In other words, the current object comes before the received object. If the compareTo method returns back a positive value, that indicates that the received object is ordered first. In other words, the received object comes before the current object. And if the return value is zero, then the two objects are equal. So in our next section, let's take a look at an example of implementing the Comparable interface.

# The Comparable Interface

=>slides: Pg. 7

Let's take a look now at how we can leverage the capabilities of the comparable interface. And to do that, we'll use our Passenger class. So let's add some new fields to our Passenger class. Now one of the fields will be name, which is a passenger's name and we'll have another field, member level, which indicates what level of membership a passenger has in the airline's loyalty program. We often think of it as a frequent flyer program. Level three members of the highest level members, the ones most entitled to benefit, level one are the lowest level members, those least entitled to benefits. And in addition to the membership level, let's also keep track of how long a passenger has been a member. We'll track that information in terms of number of days. So let's go now to constructor to our Passenger class that accepts values for all those fields and then sets the appropriate fields. So now let's add some ordering capabilities for our Passenger class. We want to order passengers based on their membership information. Now the primary way we want to order members is based on membership level. Now remember that level three members are the highest priority members, so level three members should always be the first ones to receive any benefits. Then our level two members would come after our level three members, and at the end of the list would be all the level one members. But of course, one of the challenges we have is that we only have three membership levels, but a flight can have hundreds of passengers on it. So there is a very strong probability that on any given flight there will be multiple members with the same membership level. So the deal with ties in the member level will use member days. So that's our ordering behavior. I want to use the member level is the primary way we ordered things and then using member days as a way to break a time. And we want to be out of order our passengers by doing the least amount of work that we reasonably need to.

=>slides: Pg. 8

So here in our Passenger class, what we're going to do is implement the comparable interface and again, what this means, by implementing this interface, we conform to the contract to be able to compare a current passenger instance to another passenger instance and indicate which one is ordered first.

=>slides: Pg. 9

Now remember, the comparable interface has one method and that's the compareTo method. Now compareTo receives one parameter and we're going to receive that parameter as an object, but it's going to be a reference to another passenger. So the first thing we'll need to do is cast that object to be a passenger reference. So now once we have this passenger reference p, we can compare the current passenger instance to the passenger that's referenced by p. Now we could do these comparisons with a series of explicit ifs by comparing the fields between the current passenger and the passenger we received. But remember, the contract of the comparable interface is simply to return back a negative, positive, or a 0 value. With that being the case, there is a much simpler way to determine ordering of passengers than doing explicit ifs. We can actually use arithmetic. So what we could do here is declare a local variable named returnValue and then simply take the member level of the passenger we received and subtract from it the current passengers member level. Now let's think how that would work. Well if the passenger received at a member level of 1 and the current passenger has a member level of 3, well 1 minus 3 is ‑2. So if we return that value back, that would be a negative value, which correctly indicates that the current passenger is ordered before the passenger we receive. So this simple bit of arithmetic takes care of indicating which of the passengers is ordered first. But now what happens if the current passenger and the passenger we receive have the same member level? Well, in that case, we would calculate a return value of 0. So in that case, what we do is simply fall back to doing the arithmetic using member days. So with that in place, if the current passenger has been a member longer, we'll correctly return a negative value. If the passenger received has been a member longer, we'll correctly return a positive return value. So, as you can see, this simple bit of arithmetic takes care of doing the work of indicating ordering using a positive, negative, or a 0 value.

=>slides: Pg. 10

So now that we have our implementation of the comparable interface in place, we can now efficiently order passengers. So let's say we have an array of passengers here named passengers and we'll initialize it with four instances of our Passenger class. Now currently these passenger instances are in the array in no particular order and what we'd like to do is get them back ordered by our membership information. So to do that, we can use a class in Java named Arrays and the Arrays class has a method on it named sort, and then to our sort method, we're going to pass in our passengers array because if we were to look up the documentation on the sort method, it tells us that it will order the members of the array based on the comparable interface. So what that means is when Arrays.sort returns, the elements of our passengers array will be reordered and it will order them based on how we've implemented the comparable interface. So, looking at the array members, we have Ashanti who is the only level three member. So since she's the highest level member, the first element of the passengers array will be Ashanti. So next, we have Harish. Now Harish is the only level two member. So that means the next member of this sorted passengers array will be Harish, but then we have both Luisa and Jack and both of them are level one members. Well remember, the way we've implemented our compareTo method, if two members have the same member level, we use the member days. So since Luisa has been a member longer, Luisa will come after Harish, and then finally, Jack, who's only been a member for 90 days, will come at the end of the now sorted array. And as you can see, this really shows off the power of interfaces. By implementing the interface and conforming to the contract of the interface, we were able to leverage the capabilities of this arrays.sort method, even though that method knows nothing about our Passenger class. Okay, so now in our next section, let's take a look at something known as generic interfaces.

# Generic Interfaces

=>slides: Pg. 11  
As we're working with interfaces, it's important to note that some interfaces allow being specialized for a particular type, and they do this using a concept in Java known as generics. So if we were to take a look at the comparable interfaces declaration, we'd notice that after the interface name, there is a less than symbol, a T, and then a greater than symbol, and that indicates that this interface could be more strongly typed. And what that means is the method within the interface doesn't have to use an object type, you can actually use a more specific type,

=>slides: Pg. 12

So let's again look at our Passenger class and we know our Passenger class implements the comparable interface, so we have our compareTo method, and the compareTo method as we currently have it written, receives an object as a parameter. Now the name of that object is o, and because we want to compareTo passenger references, the first thing we have to do here is cast that object to be a passenger, and then once we do that, we can do the actual work we want to do of comparing the current passenger instance to the passenger instance we received. Well up here, we say we're implementing comparable.

=>slides: Pg. 13

Because the comparable interface supports generics, after the interface name, we can specify that we want to specialize on the type passenger, and by doing that,

=>slides: Pg. 14

the compareTo method here where we receive an object can instead directly receive a passenger reference, which means the first thing we're doing here where we cast one object to a passenger is no longer necessary. So our method compareTo is now more strongly typed, which means we get all the benefits of strong typing within the compiler. The compiler will make sure that we never passed anything other than a passenger reference into compareTo, and generics are a really powerful concept in Java. So once you've had a chance to work with Java for a while, I would encourage you to check out the Pluralsight class on Java generics to get a better understanding of this concept. You can do that by heading over to the Pluralsight search facility just typing in Java generics. Alight, so now in our next section, let's take a look at how a class can implement multiple interfaces.

# Implementing Multiple Interfaces

=>slides: Pg. 15  
Now as a class implements interfaces, it's not limited to implementing just a single interface. A class can implement multiple interfaces. Now to do that, we simply list the interfaces we want to implement separated by commas. It's important to note that there's no practical limit on the number of interfaces a class can implement, so a class can conform to as many interface contracts as is appropriate for that class. So let's see what it looks like to implement multiple interfaces.

=>slides: Pg. 16

So this time we'll take a look at our Flight class. Remember, our Flight class has a number of fields, including number of passengers and the number of seats. And let's assume that our Flight class has already implemented the Comparable interface. As part of implementing the Comparable interface, the Flight class would include the compareTo method. It's not important what the details are of how this particular class implements compareTo. The thing we're focusing on here is that it's already implemented one interface. So let's say we want to add some additional information to our Flight class. In addition to keeping track of the number of passengers, we also want to keep track of the individual passengers that are on the flight. So, effectively, we want to have a complete list of all the passengers that are currently on the flight. You'll notice that my passengerList field here has a type we haven't seen before, which is an ArrayList. An ArrayList is just a class in Java that could be used very much like an array, but it automatically increases its size as we add more members to it. So unlike a simple array, where you have to pre‑size it, the ArrayList grows automatically. So our passengerList here will have the full list of all the passengers on a flight. Now you'll notice that this field is currently declared as public. And that might be kind of dangerous because that means that any code can directly access our list of passengers, which means that additional passengers can be added to the list or taken away from that list without our Flight class ever knowing. So this isn't the sort of thing we want to make public.

=>slides: Pg. 17

We probably want to go ahead and keep this guy as private. Now that might create some challenges because something we're very likely to want to do is be able to walk through that full list of passengers that are currently on the flight, so we might want to use a for each statement to do that sort of thing. But with this field being private, that's not possible.

=>slides: Pg. 18

But we can address that by having our Flight class implement another interface, which is the Iterable interface because it's the Iterable interface that actually makes the for each statement possible. Now one thing to note here before we get into the details of this interface, remember that when we're implementing multiple interfaces, we want to separate those interfaces by a comma. So we're saying here that the Flight class implements both the Comparable interface and the Iterable interface. Now this Iterable interface also supports generics, so we're going to specialize this on the type Passenger. So that means we can actually walk through a list of passengers using the Flight class. Now this interface has one method, which is the Iterator method. This is responsible to return back a reference to something known as an iterator. Now, an iterator is something that knows how to walk through a list of items. Now, in our case, implementing this method is really simple. Remember that you can use a for each statement to walk through an array. You can also use a for each statement to walk through an ArrayList. So all we need to do is ask the passengerList field, which is an ArrayList, to return back an iterator. So what we've done here is provided the ability to walk through a list of passengers without directly exposing the passengerList. And we do that by using the Iterable interface.

=>slides: Pg. 19

So let's see how we could use this now that we've implemented the interface. So let's declare a Flight instance. We'll assign it to a reference f175. And we'll add a series of passengers to that flight. So we have a passenger named Santiago, another named Julie, another named John, and another name Geetha. So now that we've added the passengers to the flight, we can actually walk through the list of passengers with a for each statement. So as part of our for loop, we'll declare a variable name p of type Passenger. Then we'll simply specify our flight f175. And then inside the loop, we'll print out the name of each passenger, and with this code in place, we would print out Santiago, Julie, John, and Geetha. So now the question may come to mind, How does this code now walk through the list of passengers?

=>slides: Pg. 20

Let's take a closer look at the for loop itself. Let's go ahead and expand that to what actually happens under the covers. So let's add some pseudo code here that shows what goes on with the for each statement. Now to be really clear, the code I'm going to show you is not the exact code that's produced, but it demonstrates what goes on when we use the for loop to walk through a list of passengers. So the first thing we want to do is take our f175 reference, and we're accessing it as the Iterable interface. Remember, that's the interface that we just implemented on our Flight class. One of the methods on the Iterable interface is the Iterator method. So we call that Iterator method, which gives us back the iterator. So once we have that iterator, Java can then generate a loop that will walk through the list of passengers because an iterator has a method that tells you if there's anything more in the list, and that's the hasNext method. And if the hasNext method returns True, there's another method called next that will give you the next item from the list. So the next method will give us back a reference to the Passenger, which we can then print out the name of that passenger. And we got all this capability by implementing another interface on our class.

=>slides: Pg. 21

So the key thing to understand here is that, unlike classes where each class can only directly extend one class, when it comes to interfaces, a class can implement as many interfaces as it needs. So our Flight class implements the Comparable interface, and it meets the requirements of that contract by implementing the compareTo method. And it can also implement the Iterable interface with the contract associated with that interface by implementing the Iterator method. So this allows us to implement classes that can conform to the requirements of many different interfaces. Okay, so now in our next section, let's see what's involved in declaring our own interface.

# Declaring an Interface

=>slides: Pg. 22

As we're developing applications we can actually declare our own interfaces. In other words, we can actually declare our own sort of contracts that we expect other code to conform to. Now declaring an interface is very similar to declaring a class. The key difference is rather than using a class keyword we're going to use the interface keyword.

=>slides: Pg. 23

Now when you declare an interface, most commonly what you include is one or more methods. And when you specify the method you're generally just going to provide its name, the list of parameters, and its return type. We're not going to provide an implementation for the method because we expect the classes that implement our interfaces to provide their own method implementations. Now it's important to note that methods in an interface are implicitly public and interfaces are not limited to methods. Interfaces can also have fields, but those fields are constant fields. In other words, they're just named type values because any field we declare as part of an interface is implicitly not only public but also final and static, meaning that any value we provide as part of the interface declaration is a value that field will always have even as other classes implement the interface.

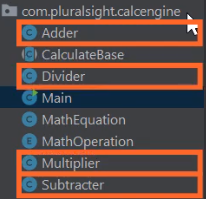
=>slides: Pg. 24

Now an interesting note on working with interfaces, one interface can extend another interface. And we do that by using the extends keyword. Now any given interface can only directly extend one other interface. And an important thing happens when one interface extends another. Any class that implements the derived interface is automatically considered to also implement the base interface, which means that that class is now conforming to two contracts, the contract specified by the derived interface, as well as the contract specified by the base interface. Now don't get too hung up on this, but it's just something important to keep in mind that one interface can extend another interface. So now to help us get a better understanding of declaring our interfaces, in our next section we'll jump back into our CalcEngine project. Well declare and use an interface of our own.

# Adding and Implementing an Interface in CalcEngine

Here we are in STS and what want to do now is see how we can leverage interfaces here within our CalcEngine project. And as you recall, the way our application currently behaves is we can enter in a command followed by two values with each separated by a space, and then we'll perform the appropriate operation. But the application is currently implemented in terms of a series of concrete types. 

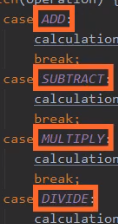
We have CalculateBase, which is a base class that models the idea of setting a leftVal, a rightVal, and performing an operation.

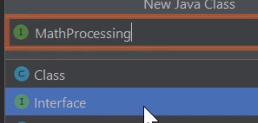


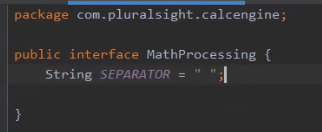
We then have a series of classes, Adder, Subtracter, Multiplier, and Divider that extend CalculateBase and implement the details of a particular operation.



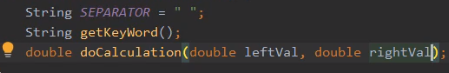
Then we have an enumeration math operation that has values for each of the operations,



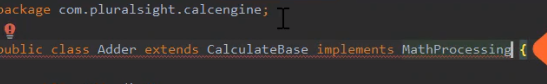
add, subtract, multiply, and divide. And this solution works great and we have total control over all the pieces within our application, but there are times where we don't want this tight concrete relationship. Instead, we want a looser, abstract relationship so that new components can be easily plugged in and that's where interfaces come in. So let's see how we can model the same behavior using an interface.   
So now to create a new interface, we'll go over here to our Project window and I'll right‑click on our package name, I'll head up to New, I'll head over the Java class, I'll choose that. Now let's head down here and choose Interface is what we want to create, and let's name our interface MathProcessing. Then, once we've named our interface, I'll go and hit Enter. So now that lays out our interface for us, so now we want to provide members for the interface. Remember that members can be a method or they can be a field. Now remember, a field within an interface is really just kind of a constant. So a good field for our interface might be the separator that we use to break apart the value that a user types in. Remember that the parts of the user's input are separated by a space.

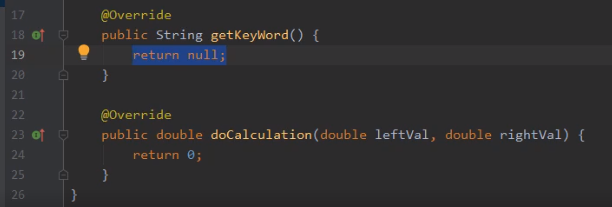


So let's create a field named separator of type string whose value is space.

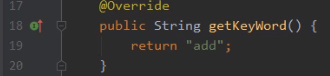


So now, once we have that, let's go and add the methods in that we want to be part of this interface. Well we're going to plug in new components, those components will need to identify what keyword they're associated with. So let's add a method named getKeyWord that returns a string. Then once a component identifies the keyword it supports, it needs to have the ability to do the actual calculation. So let's have a method here named doCalculation that returns a double and we'll have it accept two parameters, leftVal and rightVal. So what we have now is an interface that very simply models the idea of being able to do calculations. We've specified what the separator is we expect between the values, we can identify the keyword to the component handles, and the component has the ability to receive the values, do the calculation, and return the result. So now what we need to do is provide an implementation of this interface and let's do that on our Adder class. So I'll head over here and select our Adder class, and our Adder class already does all the work that we want to do in terms of the CalculateBase base class so we just need to enhance it to do the work in a way that conforms with the interface.

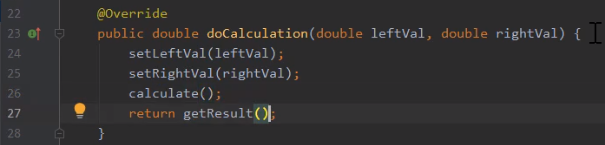
  
So let's have it implement our MathProcessing interface. Now, remember that part of implementing the interface means we have to implement the methods that are part of the interface and you'll notice that STS is showing us that red light bulb, which means there is some help they can offer for us. So to access that help, here on Windows, I'll hit Alt+Enter. You notice the first option is Implement methods so I'll choose that.

  
STS then confirms which methods I want to implement and I want both of them, so I'll again hit Enter, 

and you can see now that STS has stubbed out both of those methods. Now let me just scroll down a bit so we have some room to work.

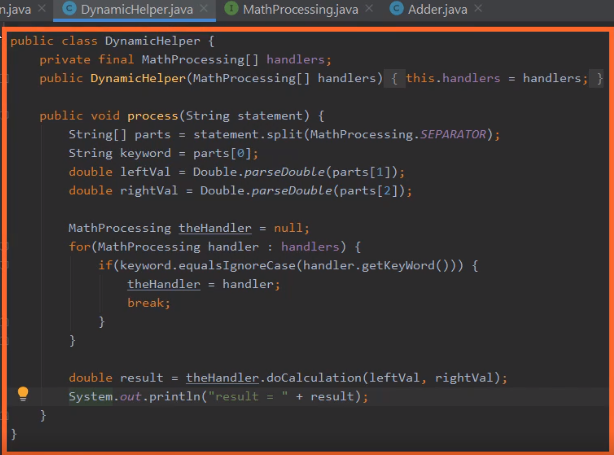


So first, we'll need to specify what keyword this component supports and we know the keyword that are our Adder class likes is add, so we'll return back the string add. Now once we implement that, let's head down here to doCalculation.



And here inside of doCalculation, we need to perform the steps that are necessary for adder to do the work. Well we know that the Adder class needs to set both the left and right values. So let's call setLeftVal passing in the value of our leftVal parameter, then we'll do the same sort of thing for rightVal. So now once we set the leftVal and the rightVal to do the actual work, we call our calculate method. Now, as you recall, calling a calculate method stores the result in the result field so we can then return getResult. So now that's all there is to it. Our Adder class is doing the same work it's already been doing, but now it's doing it in a way that conforms to our MathProcessing interface. So now in our next section, let's see how we can use this interface based capability.

# Using the Interface in CalcEngine



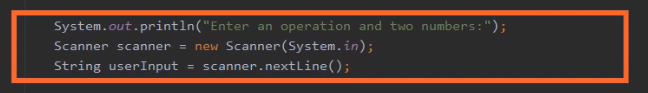
Here we are now back in our CalcEngine project, and what we want to do now is continue the work of using our MathProcessing interface. Now to make it a bit easier for us to work with the interface, I've added another class to the project. This is the DynamicHelper class, and this class takes care of the details of performing work based on our MathProcessing interface.



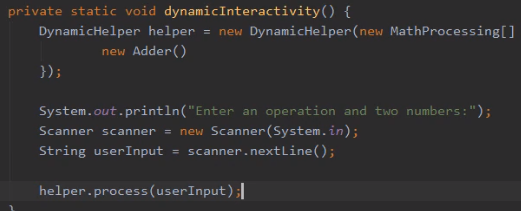
Now you'll notice that DynamicHelper has one field named handlers, which is an array of MathProcessing, which means it can hold an array of implementations of that interface. We have a constructor here that accepts the array, and then sets the field with whatever array is passed in. Once we have that array, we'll do the work with a method named Process. Now we'll look at the details of this method in just a moment. The key thing to understand at this point is that the process method has one parameter, which is a string, which is the statement that we want to process. So to see how to use our DynamicHelper class, let's head over to our Main class.



  
Now over here in our Main class I've added a method, dynamicInteractivity,

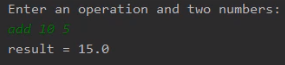


and you'll notice at this point, the only thing this method does is prompt the user for input, read the user's input, and store it into a local variable named userInput. So in order the process that input, we're going to need our DynamicHelper class.

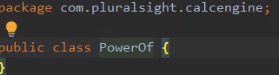


So let's declare a local variable named helper of type DynamicHelper. We'll assign it a new instance of DynamicHelper. Now remember that DynamicHelper constructor accepts an array of MathProcessing, so let's create a new MathProcessing array. Now because the type of this array is MathProcessing, it can be initialized with instances of any class that implement that interface, so that means we can initialize it with a new instance of our Adder class. So now by doing that, we've initialized the DynamicHelper class to use our Adder class as a way to process the user's input. So with that all in place, after we get the user's input, we can call helper.process, passing in userInput. So now whatever the user types in, something like add 10 5, that string is then passed in to our process method.

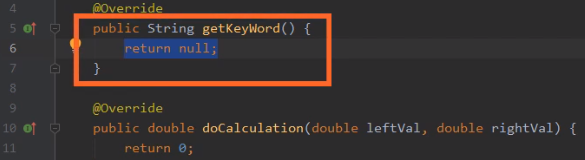
  
So to see how that process method works, let's head back to our DynamicHelper class. So now we're here in our process method and we've received the userInput. Now notice the first thing that we do is we split it into its parts, and we're using our MathProcessing interface's SEPARATOR field. Remember we set that to a space, so it's going to split the input based on the spaces. So then we should have our parts array with three parts in it, parts[0] will be the keyword for the operation, parts[1] and parts[2] will be the left and right values. So now the next thing we need to do is identify what handler understands that keyword. So we've declared a local variable, the handler of type MathProcessing, and then we have a loop that will walk through the handlers. For each handler, it takes the keyword, does an equalsIgnoreCase asking that handler what keyword it understands. Now remember, at this point the only keyword we understand is add, but when we find that match, we then set the handler to be that handler and break out of the loop. So once we leave that loop, we have the handler who understands the keyword, so now the work is really simple. We simply call doCalculation on that handler, passing in left and rightVal, and assign the result to our local variable named result, and then print out that result. So you can see what we've focused on here is having the classes implement the details of doing the work. The DynamicHelper class takes care of the details of splitting apart the statements, finding the right handler, and just asking it to do the work.

  
So let's run our program and let's verify that our add capability still works. So now once the program starts, down here in the run window you can see the prompt to enter an operation and two numbers. So I'll enter add 10 5, I'll hit Enter, and you can see our Adder did its work correctly, but it did it in a way that conforms to our MathProcessing interface. But here's the really cool thing about interfaces. Once we have all this set up, plugging in new capabilities gets really easy. Let's see how we can do that in our next section.

# A New Implementation of the Interface

  
Here we are back in STS. And what we want to do now is see how we can leverage our math processing interface to plug in a new feature into our application. So let's say we want a new feature that supports raising numbers to a power. Now none of our existing classes do that, so we'll create a new class to add that feature. So to do that, we'll head to the project window, then right‑click on our package name. We'll head up to New, then we'll go over to Java Class, and choose Java Class. Let's name this class PowerOf. Then to create the class, we'll hit Enter. So now once we create this class, in order to plug it into the existing system,

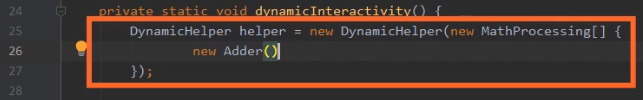
  
we'll need to implement our math processing interface. And by implementing this interface, it allows us to leverage all the code we already have in our DynamicHelper class. Now remember, as part of implementing the interface, we have to implement all the methods. So to do that, I'll hit Alt+Enter, then I'll choose Implement methods.

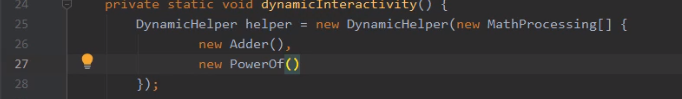
  
Then I confirm that I want both methods, so I'll hit enter. So now our PowerOf class just has to do the work that's necessary to conform to the interface.

  
So the first thing we'll do is go to getKeyWord. We'll need to return a string that identifies what keyword we understand. So our keyword will be power. So that indicates that this class can do the work associated with the keyword power, but now, of course, we need to do the work itself.

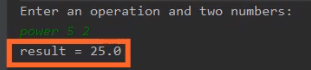


Well, it turns out that we can leverage a class provided by Java to do that work. There's a class in Java called Math that has a method named pow. So down in our doCalculation method, let's return back Math.pow. And pow accepts two parameters, the value and the power you want to raise it to, so we'll pass in leftVal and rightVal. And that's all there is to it. We've now built a class that conforms to our mass processing interface. Our getKeyWord method identifies the keyword that we're responsible for handling. Our doCalculation method does the work of receiving the two values, doing the appropriate work, and returning the value back.



So now in order to use these capabilities, we need to pass in an instance of our PowerOf class to our dynamic helper class. So we'll do that over in our Main class. So we'll head back there. So now here in our Main class, we have our dynamicInteractivity method, and up here at the top is where we create the instance of DynamicHelper. And as part of creating that instance, we pass in this array of MathProcessing implementations.  


So here after Adder, let's go ahead and create an instance of new PowerOf, and we're done. And that's the power of interfaces. Because all of our code inside of DynamicHelper works only in terms of the interface, MathProcessing, it doesn't know anything about the classes, doesn't know anything about Adder, doesn't know anything about PowerOf. So that makes it very easy for us to plug in new features by simply implementing the interface that DynamicHelper understands.

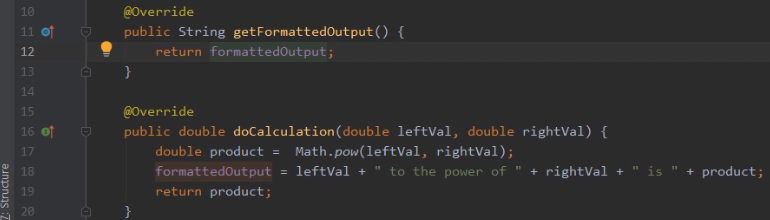
  
Now just to confirm that our PowerOf class works correctly, we're just going to run our code. So now down here in our Run window, I'll enter power 5 2. So that should raise the number 5 to the power of 2. And if that works correctly, we should get back 25. So as you can see, interfaces make it very easy for us to plug in new capabilities into existing code, but there is a challenge with interfaces we need to consider. Sometimes after you've already released an interface, you need to add new features to the interface. So in our next section, let's see the appropriate way to deal with that scenario.

# Adding a Default Method to an Interface

  
Now one of the challenges that often comes up with interfaces is that once you've declared them, as time goes by, you may want to add some new features to the interface. Now, for example, in our case, the way we currently display our output is pretty simple. We simply say something like results = 25.0. And maybe we want to provide the capability for classes to return back a more descriptive version of their results.

  
So let's say want to add a method here, getFormattedOutput, that returns back a string. Now that seems simple enough. But the problem is that there are already classes that implement this interface. If we start adding new methods to the interface, we need to go back and revisit all of those classes. Now, in our case, we're only talking about two classes. But in practice, there could be tens, hundreds, or even thousands of classes that have already implemented your interface. And so adding this method would break all of that code. So to help us with that, Java supports something known as a default method on an interface. Now with the specified default method, we put the default keyword before the method return value. And because it's the default method, we can actually provide a default implementation. So, like in our case, what we might do here is simply return back a null. So now by doing that, any class that implements the interface MathProcessing that doesn't explicitly implement getFormattedOutput will use this default implementation that simply returns back a null. But the thing to keep in mind is that no class is required to use the default implementation.

  
So let's say, for example, our PowerOf class wants to support getFormattedOutput. So let's head over to our PowerOf class. So now here in our PowerOf class, let's add a field named formattedOutput of type String, and then what we'll do is go down here to doCalculation, and as part of doing the calculation, let's set up our formattedOutput.



So here we returned the value of Math.pow. Instead, let's assign it to a local variable named product. So now once we have the product, let's go ahead an assign a nice formatted version of our output to our field formattedOutput. And then, finally, we still need to make sure we fulfill the requirements of doCalculation. So the last thing we'll do is return back the product. So now at this point, doCalculation does the work of preparing the formattedOutput and returning back the product as the return value of doCalculation. But we still haven't implemented the new method. Well, to do that, here on Windows, I compress Ctrl+O. You notice that one of the options I have now is that method that we've added getFormattedOutput. So I'll choose that. Then here in getFormattedOutput, I'll return back formattedOutput. So now with that, the PowerOf class supports all three methods in the MathProcessing interface. We have getKeyword, getFormattedOutput, and doCalculation. But if I head over to our Adder class, it still only supports the original two, getKeyword and doCalculation. And that's okay because I have default implementation for the getFormattedOutput method. That's a key thing to understand. As you evolve your interfaces, you want to be sure not to break existing code, and default methods give us a mechanism for dealing with that.

# Summary

=>slides: Pg. 25

To wrap up, here's some of the key things we want to remember from this module. Remember that an interface defines a contract, so it actually defines a contract for behavior, and as part of doing that, an interface provides a list of operations. Remember that these operations are not focused on implementation details, it's simply a list of methods and the behavior that's expected from those methods. So then it's up to classes to actually implement our interfaces. By implementing an interface, a class expresses that it conforms to a contract, and a key part of conforming to that contract is the class is going to have to provide all the necessary methods for the interface.

=>slides: Pg. 26

Remember that many interfaces are actually generic interfaces, which means it's an interface that allows stronger typing. So as part of specifying that we implement the interface, we can actually specialize the interface on a particular type, which allows the methods corresponding to the interface to be more strongly typed. And remember that a key aspect of interfaces is that one class can implement multiple interfaces, so following the implements keyword we can simply provide the list of interfaces that we support separated by a comma. And remember that there's no practical limit on how many interfaces a class can implement. Any class can implement as many interfaces as is needed for that particular class, which allows one class to conform to many different contracts for behavior.

=>slides: Pg. 27

And remember, as developers we're not just limited to implementing interfaces, we can actually declare our own interfaces. Remember that declaring an interface is very similar to declaring a class except we use the interface keyword rather than the class keyword, and remember that any members in an interface or implicitly public.

=>slides: Pg. 28

Now remember that our interfaces can have fields, but fields in an interface are not exactly like fields in a class. Remember that fields in an interface really are just named values. They're a way to have named constants as part of the interface. In general, when we're declaring interfaces we focus much more on the methods, but those methods will generally just be the name, the list of parameters, and the return type. The methods are generally not going to have a body, because the interface is not focused on implementation. We expect the classes to implement the methods. But remember, an interface can have default methods. In other words, there are methods in an interface that can have a body, and the reason we do that is because it allows the interface to be extended over time without breaking existing classes that implement the interface. So by using a default method, we can specify the behavior for the method for any class that doesn't explicitly implement that particular method. Alright, that wraps up this module. In the next module, we'll take a look at nested types and anonymous classes.

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