9. More About Data Types

Contents

[1. Introduction 1](#_Toc27761)

[2. Understanding Time 1](#_Toc5512)

[3. Local and Global Date/Time 2](#_Toc4350)

[4. Adding Date Arithmetic to CalcEngine 3](#_Toc6169)

[5. Displaying CalcEngine Date Results 4](#_Toc15680)

[6. Formatting Dates 5](#_Toc27024)

[7. Primitive Wrapper Classes 5](#_Toc28611)

[8. CalcEngine with Word and Number Support 6](#_Toc24695)

[9. Classes and Interfaces 7](#_Toc22119)

[10. Summary 8](#_Toc26272)

# Introduction

=>slides: Pg. 1

Welcome to our next module, More About Data Types. All right, so as our applications become more sophisticated, we're going to increasingly rely on data types to help us work more effectively. So in this module we're going to start by looking at some specific data types, then move on to some more general data type issues. So the first thing we'll look at is time‑based data types. We'll first look at a data type that allows us to represent the time of events. And then from there we'll move on to dealing with human‑friendly time. And as part of our discussion about human‑friendly time, we'll see how we deal with formatting date and time values. From there, we'll move on to primitive type wrapper classes. As you recall, early on in this course we looked at our primitive types, and these primitive types are the foundation of much of what we do in Java, but those primitive types also have some limitations. So we need some wrapper classes to help overcome some of those limitations. And then we'll finish up, we'll look at classes and interfaces and see the important role that they play in helping us move our applications to that next level.

# Understanding Time

=>slides: Pg. 3

So let's take a look at this idea of time, and we'll start out with what seems like a very simple question, what time is it? And although this may seem like a simple question, in reality, it's not really a simple question at all because the details of time can be complex. How we represent time depends on what we want to do with that time value. So when it comes to representing this idea of time, there are three general categories of how we deal with time.

=>slides: Pg. 4

One is time of events, and this is a scenario where we're primarily interested in sequencing or timestamping things. And those needs are different from the idea of human‑friendly time. And this human‑friendly time is really even broken out into two categories. One is this idea of a local human‑friendly time. In this local human‑friendly time, I'm dealing with just a date or a time of day or maybe that combination. But there are situations where we need to deal with time in a global manner. And in this scenario, we need to keep track of everything there is to know about a time value. So we, of course, need the date and the time of day, but we also need to deal with time zones and be sure that our representation understands the differences in time zones.

=>slides: Pg. 5

Now in Java, when it comes to tracking time of events, we use what's called the Instant class, and the Instant class is optimized for timestamping events. It works well for situations where we want to sequence things based on time values and do relative comparisons of time values. But even though it's optimized for these scenarios, if we do need to convert it to one of more complex date/time types, it does have that ability. So let's take a look at some code.

=>slides: Pg. 6

So we have a method here, checkRelationship. What I want. this method to do is accept an instant as a parameter, and we've given it that name otherInstant. We want to see how that other instant compares to the current moment in time. So the first thing we'll do here inside of checkRelationship is get that current moment in time. So we call the Instant class' now method and assign it to a local variable named nowInstant of type Instant. Now once I do that, I can start doing comparisons. So I'll put an if statement here that calls the compareTo method on otherInstant and passes in nowInstant. Now, the compareTo method will return a negative value, a 0 value, or a positive value to indicate how these two instants relate to one another. If compareTo returns a positive value, that tells us that otherInstant comes after nowInstant. Now we can also do the opposite comparison. So if compareTo returns a negative value, that tells us that otherInstant is in the past. And if neither of those are true, that means that both of these instants represent the same moment in time. Alright, so now on our next section, let's take a look at human‑friendly time.

# Local and Global Date/Time

=>slides: Pg. 7

Now, as we deal with local human‑friendly time, we have a lot of control over what level of detail we want to keep track of that time. So we have one class, the LocalTime class, that tracks just the time of day. It doesn't keep track of the date portion at all. You can display a LocalTime as a string, it comes out in a format like this, a two‑digit hour, colon, a two‑digit minute, colon, a two‑digit second, followed by fractional seconds. And this string representation of the LocalTime is consistent with the international standard for how we represent that time. Now, in addition to LocalTime, we also have a LocalDate. LocalDate tracks just the date portion. You display LocalDate as a string, you get a four‑digit year, dash, a two‑digit month, dash, a two‑digit day. But now there are some cases where we need both date and time, so we have the LocalDateTime class. It has both the date and time value, but note it still doesn't have a time zone in it. None of these types track time zone. Now if we display the LocalDateTime as a string, it will display as a combination of the values, so the date information, the letter T, and then the time information. Now, again, these types focus on the date and/or the time value.

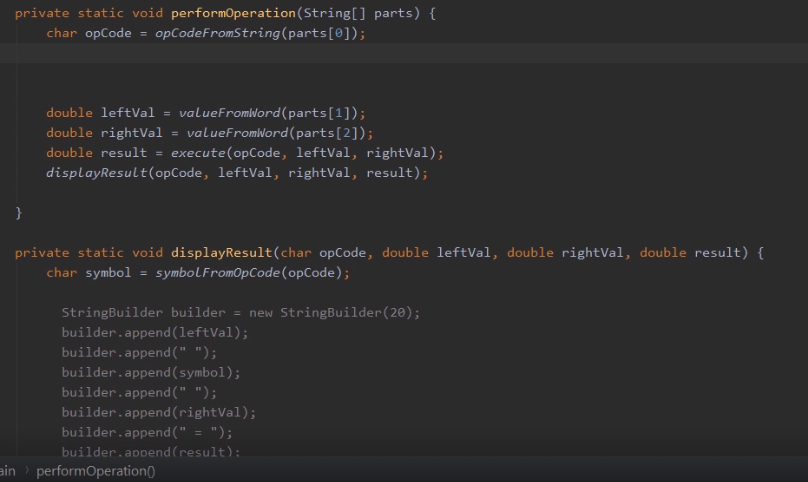
=>slides: Pg. 8

They don't include any aspect of the time zone. So, of course, you only want to use these types in scenarios where you don't need a time zone. Now, these types do provide a number of common operations, and that's really important, because if you've ever tried to do date math or time math manually, it gets really complicated really fast. So these types allow us to do things like find differences between two values, allow us to increase or decrease a value, in other words, you can add a number of days to a date or subtract a number of hours from a time, that sort of thing. We can also manipulate the content within these types, and we can also convert to a string or from a string. So you can give it a string value, the type can parse that string to convert it into the time value.

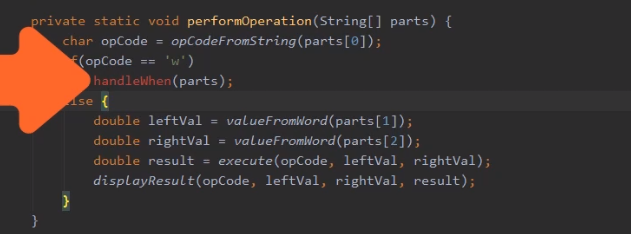
=>slides: Pg. 9

Now we do commonly encounter situations where we need to represent time in a way that works globally, and that's where the ZonedDateTime class comes in. It provides all the date and time operations you would expect, very much like LocalDateTime does, but it also understands time zones. Now, what I want to be clear about, and that's strong time zone support. It doesn't just track the basis of a time zone, which would be the time zones in relationship to universal time, but it knows all the key aspects of that time zone. It knows when daylight saving time starts. It knows when daylight saving time ends. And because it has a strong knowledge of time zone, it allows us to do date work across time zones, and even allows us to convert between different time zones. So, as you can see, this variety of time classes gives us a lot of control over how we represent time and how we work with time. So now in our next section, let's jump into our CalcEngine project and we'll add date math support to our application.

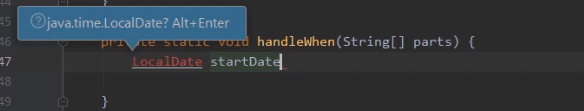
# Adding Date Arithmetic to CalcEngine



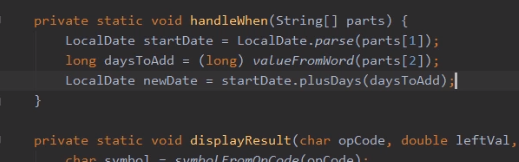
Here we are in STS, and we're again looking at our CalcEngine project. And what we want to do now is add date support to our application. Now as you'll recall, up until now, our application has been focused on basic math operations. Well, now I want to introduce a new command, and we'll call it the when command. With the when command, we'll provide a date and a number of days, and then the application will display the date that results by adding that number of days to the starting date. So let's think about how we might implement this in our code. So we're looking at our performOperation method.



And up until now, because everything we were doing was math operations, we just needed a single path through this method. We would first get the opCode. Once we had the opCode, we'd get the left and right values as doubles. And then we would execute the arithmetic. Well now we're going to need an alternate path when the user types in when. So what that means here, when we call our opCodeFromString method, we get our opCode back. If that opCode is w, because remember our opCode is always the first letter of the command, then, in that case, we want to call a separate method to do our date work. So we'll put an if here to check to see if the opCode is w. If it is w, we'll call a method we'll create named handleWhen. When we call handleWhen, we'll pass in our parts array. So that's what we want to do if the opCode is w. But if it's any other opCode, we just want to do what we always did. So we'll take all that code and place it in an else. So we're all set here in performOperation, so now we're ready to implement our handleWhen method.



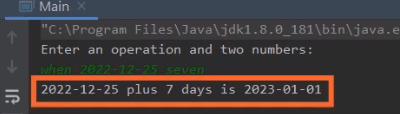
So I'll go up here and place my cursor on handleWhen. I'll hit Alt+Enter, Then I'll choose our option, Create method. Once we do that, I'll tab through the return value and all of our parameter information. So now we're ready to start writing the code to do our date work. So the first thing we'll need is a date. So let's declare a local variable named startDate of type LocalDate. Now you'll notice that our LocalDate is highlighted in red, and we've got this message here asking us if we want to use java.time.LocalDate. Remember, this is that issue we talked about earlier in the course regarding import statements. We need to provide an import to specify exactly which type named LocalDate we want. And the one that's being suggested is the one we want. So I can simply press Alt+Enter. And you'll notice that when we do that, LocalDate is no longer highlighted in red because STS is taking care of adding that import statement for us. If we were scroll back to the top of our source code, we would see import java.time.LocalDate.



All right, so now that that's all set, we want to parse the value that the user typed in. So we're going to call LocalDate.parse, passing in parts sub 1. So what this will do is take the string value that the user typed in for date. As long as the date is formatted correctly, that parse method will translate that string into an instance of LocalDate and assign it to our startDate variable. So now the next thing we need to do is get the number of days that the user wants to add. So let's declare a local variable named daysToAdd of type long. Now we need to translate that value the user typed in. And as you recall, we have our valueFromWord method that we wrote earlier. So we'll call valueFromWord, passing in parts sub 2. So now you'll notice that line is underlined in red, and that indicates to us that there's a problem. And the problem has to do with way valueFromWord is implemented versus the way our variable is declared. All right, that valueFromWord returns back to type double. And as you recall from our discussion earlier in the course, Java cannot automatically convert a double to a long. But remember, we can explicitly make that conversion by using a cast. Now I could type the cast manually, but you'll notice we have the red light bulb there. And that tells us that STS is suggesting a solution. So I could either click on that red light bulb or here on Windows, I can just press Alt+Enter. And you'll notice that one of the choices I have here is Cast to long. So I'll go ahead and choose that. And as you can see, STS automatically added that cast for me. Okay, so now we have our start date and the number of days we want to add. So now we're ready to do the date arithmetic. So what we'll do is use our startDate variable and call it plusDays method, passing in daysToAdd. And that plusDays method will give us back a new LocalDate. So we'll assign the result of plusDays to a variable named newDate of type LocalDate. So now with that, we have all the code in place to get the input from the user and do our date arithmetic. So the next thing we need to do is display our results, and we'll do that in our next section.

# Displaying CalcEngine Date Results

Here we are back STS, looking at our CalcEngine project. And what we want to do now is display the results of our new date feature in our application. So to display our result, let's go ahead and use string formatting to do that. So I'll declare a local variable named output of type String, and then I'll call the String.format method, specifying a format string, our start date, days to add, and new date. So let's go ahead and take a look at our format string and how we're matching up our format specifiers with the individual arguments. So let's start here with our second argument, daysToAdd. Notice that we're using a %d format specifier. Well that makes sense because daysToAdd is a long, and a long is an interval type. But now what about the other two arguments, startDate and newDate? Those guys are both of type LocalDate. But in both cases, we're using the %s format specifier. And %s means that we want to display things as a string. And those guys aren't strings. So what happens here in this case when we use %s, each of our date types will automatically convert themselves to a string. And it's that string value that gets displayed. So let's go ahead and run our code, and let's see how things look.



So our code is up and running, so we'll head down here to our output window, and we'll enter our new command, when. So once we type in our when command, we can specify the date. So that gives us the date of Christmas Day in 2022. So then we'll go ahead specify the number of days we want to add, and I'll specify 7. So if our program works correctly, we'll now add 7 days to December 25th, 2022. And when we do that, we see our result displayed. December 25th, 2022 plus 7 days gives us January 1st, 2023. So our code worked perfectly. It did exactly what we wanted it to do. But of course, key to this future working is the user typing in the date in this exact format. Well sometimes, we want to use different date formats, and we'll see how to do that in our next section

# Formatting Dates

=>slides: Pg. 10

Now as we've talked about, when we work with date and time values, we often need to present them as strings or often need to parse them from strings. But buy default, each of the date and time types is limited to a single string format, which means that if we want to present any of those values in any format other than the default, we're going to need some kind of help. And that's where our DateTimeFormatter class comes in. The DateTimeFormatter understands formatting of dates and times, so it allows us to describe a date or time format, and it also includes several predefined formats. And we can use the DateTimeFormatter to convert date or time types to a string using the format we specify in the formatter. In addition, we can use the DateTimeFormatter to parse strings using the format that's described in the DateTimeFormatter. So let's take a look at some code.

=>slides: Pg. 11

So we have a local date variable here named today, and we've set it to today's date. So if I print out today, it's going to print out in the default format. Now although that is a completely valid date, that's not necessarily a form that people are used to seeing. For example, here in the United States, we generally expect the month, then the day, then the year. So if I want to present this date following that US format, I'm going to need a DateTimeFormatter. So I've got this variable here, usDateFormat, and I'm going to assign it the result of calling DateTimeFormatter.ofPattern. And what this will do is create an instance of DateTimeFormatter that understands the pattern that I specify. I'm going to specify the pattern 2‑digit month, dash 2‑day, dash 4‑digit year. So now once we have that DateTimeFormatter in place, we can display the date using that format. So to do that, when we display today, we're going to call its format method, and then all we have to do is pass in usDateFormat. That will then format that value following that pattern. So the result will be 2‑digit month, dash 2‑digit day, dash 4‑digit year.

=>slides: Pg. 12

Now the cool thing about DateTimeFormatter is that it not only works when presenting dates and times of strings, but actually works when parsing dates or times from strings. So let's say we have this string value, which has the date July 4th, 2022 in standard US format. Well, if I then try the parse that date value, because it doesn't conform to the standard format that LocalDate expects, I'll actually get an error. The parse will actually fail. So if I want to parse a date of that format, I'm going to need some help. So we again can use our DateTimeFormatter. So here I've created a DateTimeFormatter exactly as we did in the previous slide. So once I have the formatter, I can now parse my usDateString. So we'll start out by calling a LocalDate.parse, just as we did before. We'll again pass in usDateString, but this time we're also going to pass in usDateFormat. And by passing in that DateTimeFormatter, we'll now be able to parse that value from our usDateString. Okay, so now in our next section, let's take a look at something known as primitive type wrapper classes

# Primitive Wrapper Classes

=>slides: Pg. 13

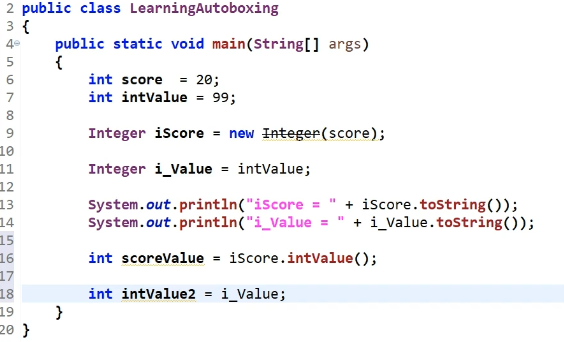
Now if you'll think way, way back to the very beginning of this course, you'll recall we talked about something known as primitive types. I remember that primitive types are the fundamental way that we represent data within our applications, and as we talked about, there are four categories of primitive types. We have are integral primitive types, our floating point, our character type, and our Boolean type. And as we talked about at that time, primitive types are really the foundation of how we represent all data within our applications. But now there are some issues with primitive types. One key issue is primitive types only represent data. They are unable to provide methods for operating on that data. So to work effectively with primitive types, we need to move beyond the primitive types themself. So that's why we have what are known as the primitive type wrapper classes or simply the primitive wrapper classes because the primitive wrapper classes enhance some of the capabilities of the primitive types, now they still hold the primitive data values, but one of the key things they can do is provide methods.

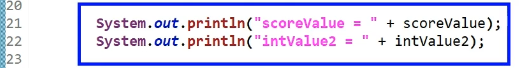
=>slides: Pg. 14

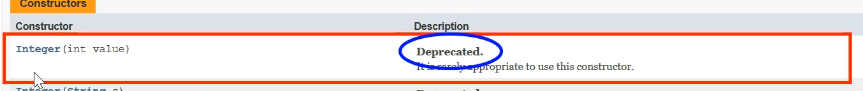
In addition, they enable compatibility with some of the richer aspects of the Java type system, and this is a really cool and powerful concept and we'll actually talk about this concept in detail in the next course in this series. For now, it's okay to think of the wrapper classes as simply providing methods that will help us with the primitive types. Now each primitive type has its own wrapper class, so we have wrappers for the integral types, floating point, character, and Boolean, and you'll notice that for most of these types, the wrapper class type is the same name as a primitive type itself, except the first letter is capitalized. There are two exceptions to that. Notice that integer and character are fully spelled out in the wrapper classes, where in the primitive types, both of those have shorter names.

=>slides: Pg. 15

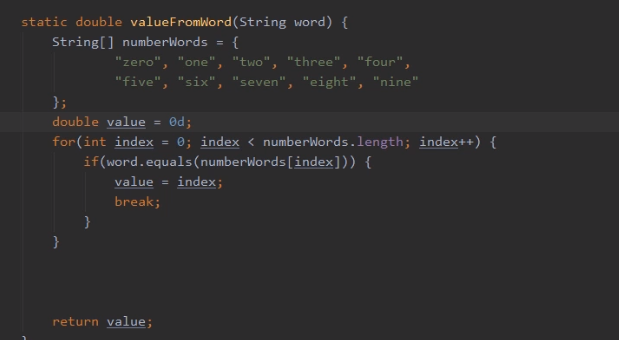
Now, these wrapper classes have a boatload of methods and they handle many of the common operations that we want to do with these types, so they give us things like converting to or from other types, extracting these values from strings, utility methods, simple things like finding the min or our max between two values, and a whole bunch of others because each of the wrapper classes has methods that are appropriate for their own type. For example, the character wrapper class allows you to check and see if a character is an uppercase character or a lowercase character, those sort of things. And there are a whole bunch of other cool things like that in these wrapper classes. So now, with all that in mind, in our next section, we'll jump back into our CalcEngine project and we'll use one of the primitive wrapper classes to enhance our applications ability to deal with numeric values.



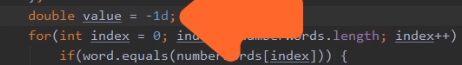




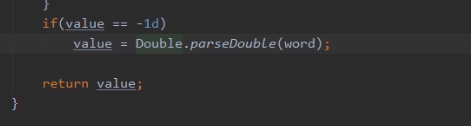
# CalcEngine with Word and Number Support



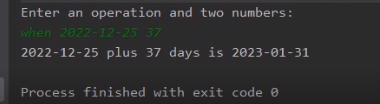
So here we are back in STS, and we're again looking at our CalcEngine project. And what we'll want to do now is enhance our application so that we can receive numeric values as numbers, not just as words. Now remember, we always receive our values as a string, and it's this method here, valueFromWord, that receives that string value and is responsible to convert it into its numeric equivalent. Now currently, the way this method is implemented is we have this array here that lists the numeric names that we understand. Then we have a for loop that loops through that array, trying to match up the word that the user typed in. When we find the matching word, we take the index of that word, assign it to our local variable value, and then return value. So to enhance this method so we can receive those numeric values either as words or as the actual numeric value itself, we'll need to figure out which way we received the information. So what we'll need to do is determine after we go through this for loop, whether we actually matched on a word or not.



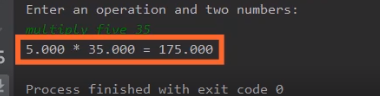
So to do that, let's change our local variable value, so it's initially set to ‑1 rather than 0. So by doing that, that means that value starts out with something that's not actually valid.



So if we go all the way through the for loop and value is still set to ‑1, that means we never matched on any of the words that we understand. So in that case, we'll make the assumption that we actually received a number, and we'll do that conversion. So here, after the for loop, let's go ahead and put an if statement in, to check to see if value is still set to ‑1. So if value is still ‑1, that tells us that the string we received is not one of the words we understand. So we'll need to parse that number out of the string, and we'll use one of the primitive wrapper classes to do that. Remember, the valueFromWord returns a double, so we'll use the double wrapper class. So remember, I mentioned that one of the things we commonly use the wrapper classes for is parsing values from strings. So the double wrapper class has a method called parseDouble, so we'll call that, passing in our word parameter. Then we'll take the result of parseDouble and assign it to value. And believe it or not, that's all there is to it. Our valueFromWord method can now handle numeric values as the words 0, 1, 2, all the way through 9, or as the numeric values themselves. So let's go ahead and run our app, and we'll verify that everything works as we expect,



So now we have our app up and running. So let's head down here to our run window, and let's issue a when operation. So there we have our when operation. So our starting date is December 25, 2022, and we're asking what the date 37 days later is. Now that 37 is an actual numeric value, which means we're going to rely on the double wrapper class' parseDouble method. So I'll go ahead and hit Enter, and let's see what happens. And you notice that when I do that, our application did exactly the right thing. It gave us the date 37 days later, which is January 31, 2023. So let me go ahead and launch the application again.



So now, just to verify that our application handles both words for numbers, as well as numeric values, let's do a math operation that mixes the two. So I'll do a multiply operation, and you'll notice in this multiply operation, we've identified the numeric values in two different ways. The first number of 5 is identified as a word. The second number, 35, is using a numeric format. So I'll go ahead and press Enter, and let's see what happens. You'll notice the app did exactly the right thing. It translated both numeric values appropriately, and that's really cool, because if you think about it, we now have an app that can do basic math operations, and date arithmetic, can deal with numeric values, both as words, as well as numbers themselves, and can accept input from the user as both command line arguments, as well as interactively. And we were able to build this entire app from scratch in just a few sessions. Alright, so now in our next section, let's take a look at classes and interfaces.

1. Classes and Interfaces

=>slides: Pg. 19

So now we're off to a great start in building up the skills we need to be effective Java developers. We've covered the topics that are critically necessary for us to be able to have a solid foundation as Java developers. But with that foundation underneath of us, we're now ready to go on to the next level. We're now ready to start thinking in terms of how do we build more sophisticated Java applications? And in order to do that, we'll need to understand how to create and use our own complex types. And when we talk about creating and using our own complex types, in general what we're talking about is classes. Because if you think about some of the most sophisticated types we've worked with over these last few modules, things like string, StringBuilder, LocalDate, DateTimeFormatter, all of those were classes, and classes allow us to move our thinking to a whole other level because classes have the ability to contain state. In other words, they can have data and actually multiple pieces of data that work together. In addition to storing state itself, classes can also contain the code to manipulate that state. So that means our data and the code that operates on it can be housed together. And this idea allows us to create our own custom data types. And a key concept that goes along with classes is interfaces. Interfaces allow us to model a type's behavior without getting into the details of how that behavior actually occurs. And interfaces are a key part of different data types being able to work together because interfaces create contracts for how different data types interact. Now a statement I just can't make strongly enough. Understanding classes and interfaces is essential to working in Java because understanding classes and interfaces allows us to create rich applications. Because these concepts simplify the process of modeling and implementing complex problems. In addition, understanding classes and interfaces enables us to get the most from the Java language itself because understanding those enables us to utilize some of the most powerful features of Java. In addition, no application stands alone. We're going to commonly have to leverage other libraries to do the work within our applications, and understanding classes and interfaces helps us leverage the vast universe of Java libraries that are out there. So with all that in mind, I want to strongly encourage you to go on to the next course in this series, Working with Classes and Interfaces in Java. That course will build on the foundation of knowledge that you've built up in this course and will give you the understanding of classes and interfaces that you need to go on and take advantage of the full capabilities provided to us by Java.

# Summary

=>slides: Pg. 20

To wrap up, here's one of the key things you want to remember from this module. We started out by looking at the date and the time types, and these provided operations for dealing with date and time values. But, remember, there are different ways to represent time, and the best way to represent time depends on what you want to do with those time values. So Java provides three broad categories of date and time types. We have the type for time‑stamping events. That was the instant class. We also had the local date/time values, and those were types like local time, local date, or local date/time. And these focus on just the date or time values themselves. These types have no awareness of time zones. But then there are situations where we need to have global time, which means that each instance has to know everything there is to know about time. And this was our zoned date/time type. Zoned date/time includes time of day, the date, as well as a strong knowledge of time zones. But now one of the challenges of our date and time types is that each of these types, by default, only supports one string representation, and there are times that we want to deal with different string representations. And that's where the DateTimeFormatter class comes in because the DateTimeFormatter class allows us to describe the formatting that we want to use. Once we describe that formatting, we can convert our date or time type into a string representation using that format. In addition, we can also use the DateTimeFormatter to parse data or time values to create our date or time instance.

=>slides: Pg. 21

Then we have the primitive wrapper classes. Remember, these are classes that enhance the capabilities of the primitive types. So these classes can hold the primitive data values. But they go much further than that because the primitive wrapper classes also enable compatibility with some of the richer aspects of Java's type system. And that's actually something we'll look out more closely in the next course in this series. And then we finished up with an introduction to classes and interfaces. As I mentioned, that's what we'll focus on in the next course in this series, Working with Classes and Interfaces in Java. So that wraps up this course. I hope you've enjoyed watching it as much as I enjoyed making it, and I look forward to seeing you in the next course in this series, Working with Classes and Interfaces in Java.

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