## 2:07:00 Data Classes

So now let's return to our sealed class hierarchy for a second and dive more deeply into what

data classes are. So you see here both easy and medium and hard are all defined as data

classes. Data classes are Kotlin's way of providing very concise, immutable data types.

By defining a class as a data class, it means that it is going to generate methods such

as equals hashcode into string automatically for you. What this allows us to do is perform

a quality comparisons on instances of these data classes and treat them as equal if the

data they contain is equal. So here's an example. Let's explore what this looks like. So we

can say Val entity one equals entity factory that create and will create an easy entity.

And then we're going to create another version of this. And then now we can check their equality

comparison.

So you can say if entity one equals entity two per DeLeon, they are equal else per Delon,

they are not equal to. Now if we run this, what will we see? They are not equal. And

that's to the expected. That's because if we come back up to our factory, we'll notice

that we are creating different unique ideas each time. So even though that the name is

the same, the unique ID is different. So now let's update this and see what it looks like

if we pass the same data in. So in this case we could create an easy directly and this

case will pass in ID comma name and then we will duplicate this for entity two. And so

now if we run this, we're going to expect to see you. They are equal and of course they

are. So this is really convenient. This allows us to represent data within our applications

and compare this data no matter where it comes from.

And as long as those properties are all the same, we're going to be able to evaluate these

as true. Now another really interesting thing that data classes give us are effective copy

constructors. So we can create an instance of entity two by copying entity one entity,

one dot copy. And because this is a direct copy, if we run this once again, we're going

to see they are equal. However, we could also use named arguments with the copy constructor

to change the value. So let's say we only wanted to change the BAME and you could say

name equals new name. And once again, if we rerun this, we're going to see they are not

equal. So you could see changing a single property and the data class is going to impact

whether or not two instances evaluate to true or not when compare.

Now one thing to notice is this is comparing the value of the data themselves. If we wanted

to U S referential comparison, we hit add a third equal sign here and this will check

whether or not it's the exact same reference or not. So in this case they are not equal.

However, this isn't all that surprising since the data was also equal. So what about if

we revert this and make this an exact copy again? So before if we were just using two

equal sign, the data would be the same. So it would print, they are equal. However, by

using three equal signs and using referential equality, we see they are not equal. That's

because it's not the same exact reference of the object. If we updated this to be entity

one equal equal equals entity one and run this, now we'll see they are equal.

So that's just one way in which we can check whether or not we have the exact same object

or if it's two different objects that have the same data. Now also keep in mind that

these equality comparisons are working off of the generated equals and hash code methods

generated by the compiler when indicating a data class. However, we could update this

to change how the equals or hash code is evaluated and to do that we would do it like any other

class. We could add a class body and then we could simply override equals and or hash

code. Now as in Java best practice, if you're going to override one of these, you should

really override both of them and you have to follow the same rules, but you have that

freedom if you would like to.

## 2:12:25 Extension Functions

Another really useful feature in Kotlin is the ability to define extension functions

or extension properties on an existing class. This is particularly powerful if you're working

with classes that you can't control but would like to modify the way in which they're used.

You can define your own properties and methods and defined kind of a new API around the existing

class. So an example of this would be adding a new method to the medium class without actually

defining that method within the definition of the medium class. So to do that, let's

come down here and you can start off by typing the fun keyword. And then instead of directly

typing the method name, we can reference the class name. Dot. And this case will type print

info.

And then we can define our function buddy. So in this case we'll just say medium class

with the ID and that'll be it. And so if we wanted to come down here and now create an

instance of entity dot medium directly, we could do that. And then we could call that

print info method. And if we run that code, we'll see a medium class and then that ID

printed out. So this is great if we know that we have that exact type that we're working

with. And in cases where we don't know if we have that direct type, we could rely on

smart casting. So if we update this to you, their factory say entity factory, create entity

type medium. Now we can say if entity two is medium entity, now we can reference that

new print info method.

This is done because the if statement will only evaluate to true if that cast is successful.

So anywhere within that context it will automatically perform the smart cast for us. And like I

said before, not only can we define extension methods, but we can also define extension

properties as well. To do that, we could start off by saying Val or VAR. In this case we'll

say vow and then again we'll reference the class type. So medium dot we'll say info will

be this property name string equals some info. If you do that, notice that we have this air.

If you look, it says extension property cannot be initialized because it has no backing field.

So to actually create an extension of property for an existing class, you need to rely on

backing fields. Thankfully the IDE can generate this forest, convert extension property initializer

to a getter.

So once we do that and notice here that we have still defined our property but now we're

relying on this custom getter for that property and so now if we come back down here within

our, if statement that's doing our smart cast for us, we could reference that new info property

directly. So this is how extension functions and properties work. You could use these anytime

you want to add additional functionality to existing class. You might notice within the

Kotlin standard library that many functions and operations work by using extension functions

in classes be are particularly effective when using them with template ID types

because it allows you to define the same common functionality across any type that matches

that template.

## 2:16:37 Advanced Functions

Now up until this point, we've covered a lot of things. We've looked at the

basic type system of Kotlin, how to work with different variable types, how to work with

basic functions and then diving into more advanced functional topics like named arguments

and default parameter values. And then we took a deep dive into modeling data with Kotlin.

So now I'm going to circle back to functions and specifically take a look at higher order

functions and how to work with functional data types. Now what are higher order functions?

Higher order functions are functions that either return another function or that take

functions as perimeter values. Now much of Kotlin standard library is built on top of

higher order functions and it's what really allows us to write highly functional code

by leveraging that standard library.

So let's take a look at how we can write our own higher order function. To start we have

a new Kotlin file and we're going to define a new function. So we'll call this fun. And

then we're going to call this print filtered strengths. And now the first argument to this

is going to be a list of strings. So we'll call this list and then define it as list

of string. And now the next thing we're going to do is define a parameter which will in

fact be a function. That function will take in a string and return a bullying. We can

then use that to filter out values in the past collection. So to define a function parameter,

you could start off by defining the parameter name as usual. And this case, we'll name it

credit kit, followed by colon. And now you have to define the type as you normally would

to define a functional type.

You can start by adding your parentheses. This will define the parameters of the function

being passed in to your other function. So in this case we are going to take a string.

He'll then add the arrow, and then you want to define the return type. So in this case,

that will be bullying. And now we'll add the open and closed curly braces to define our

block body. So now we have a parameter called predicate, which will be a function that takes

in a string parameter and returns a Boolean. Now we can implement our print filtered strings

function to make use of that predicate function to filter out any strings in that past list.

So to implement this function, first off, we want to iterate over each string in the

past list. So to do that, we could say list doc for each and now we will be able to iterate

over each of those strings.

So now what we want to do is evaluate the predicate for each stream in the collection.

So we can call the predicate function in several different ways. So to start we'll say if,

and then the easiest way to invoke the predicate is to simply say predicate open and close

parentheses and pass in the parameter value. A parameter that is a functional type can

be called as if it was a regular function. As long as you can satisfy the required arguments.

So in this case we can say if predit kit returns true, then we can print out that string. Now

to test this, we'll come down here and we will add a main function and we will say vow

list equals list of, and then we can say something like Kotlin, Java C plus plus Java script.

And now we could call print filtered strings pass in our list.

And now we need to pass in a function as the second parameter to print filters, drinks.

So we can do that by specifying a Lambda, and in this case we will say it starts with

K. so this Lambda is going to evaluate to true if any of the past strings begins with

a K. now if we run this function, we'll see only Kotlin print it out to the screen. If

we were to update this to print things out, that started with a J, well now see Java script

and Java. Now one thing to notice is it in our invocation of print filtered strings,

we've passed our Lambda within the parentheses of that function in vacation. However, this

is something that we don't have to do. As we mentioned earlier, we can take use of Landus

syntax, which says that if the last parameter of a function is a function, you can specify

that as a Lambda outside the function body. So we can restructure our function to look

like this. We can pass in the list first and then specify or Lambda outside of the parentheses.

So this is actually very similar looking to the for each function which we called up above.

And in fact if you look at the implementation of for each is in fact a higher order function.

The Lambda that we specify after invoking for each is a function which will operate

over each string and that list. Now if we come back up here to our implementation notice

we are calling the function parameter directly as if it was a regular function. So this works

absolutely great in most situations. However, if we were to make this function, type a NOLA

ball type by wrapping it in parentheses and adding new question Mark. Well now see an

error in our implementation of print filtered strings. That error basically says that you

cannot invoke that function parameter by using the parentheses directly. If it's a nullable

type to get around this, we can make use of the invoke method on that functional type

and then we can make use of the safe call operator and now, but updating this to do

a safe invoke call on the predicate function.

We can handle this rather not the predicate is no calling invoke will invoke the function

just as it would any other indication of a function. So now down here nothing has changed

and how we can call print filtered strings. However, we could also pass it in list and

now we could pass in no as a no function. So we've seen how we can treat functions as

parameters to other functions and these function parameters are really treated as tight. Just

the same as let's say integer or string. Caitlyn has this idea of functional types. It's a

first-class part of the language. This means that we could define a variable of a functional

type and then pass that variable in any time. We needed a function parameter that matched

that function signature. So an example of this might be something like vow credit kit

and then we will define our function type to match that of our print filtered strings

function.

So in this case it'll take a string and return bullion and now we'll define our function

the same way that we were doing it before. By saying if the string starts with aJ , go

ahead and return true. Now instead of invoking print filters, strings with a landed pass

to it, we can pass in our predicate variable directly. And now if we run this, we'll see

the same output as we would before. So this allows us to store function as variables.

This can be really useful for things like optional input handling. For example, maybe

you have a view on some screen and you want to be able to specify a ClickList center for

that view. You could define that as a Lambda property on some class and allow client code

to set that ClickList center as needed. As we mentioned before, higher order functions

include functions which take other functions as parameters, as well as functions that return

other functions.

So let's define a function called get print predicate and it'll take no parameters, but

we defined its return type as a function which takes a string and returns a bullion. And

now we can return that value by saying return. And then we could pass a Lambda and say it.

That starts with J. So we're passing essentially the same type of Lambda that we've been using

in these other examples. But now we've wrapped it in this other function and so now and so

then passing predicate directly or instead of defining a new Lambda as our function parameter,

we could instead call get print predicate as a function on its own, which will then

return a function which then can be used as the predicate for print filtered strings.

And if we run this once again, we'll see that our output hasn't changed though. So higher

order functions can work as both inputs and outputs and Kotlin allows you to define properties

with functional types.

So through this function's really become a very powerful and first-class part of the

language that can start to replace a lot of other instances. For example, you might find

yourself relying more heavily on functions to define things like event or a ClickList

centers rather than defining concrete interfaces for those same types of functionality. Now

this was recently mentioned. Much of the Kotlin standard library is built around higher order

functions and especially a higher order functions defined with generic types. So if we look

at the implementation of four each, well notice that this is actually an extension function

as well as a higher order function. So for each works on generic Iterable type and takes

in a function parameter that takes in that generic type and returns unit. So this essentially

allows us to iterate over each element in the collection and then call that action on

it and it doesn't have to return anything.

And similarly for each index takes in a single function parameter as well. But this one takes

in an event to represent the index as well as the generic type. This allows us to iterate

over each element in the collection while incrementing a counter and then passing that

counter into the function parameter as the index. The power of generic types, extension

functions and higher order functions allows us to write single implementations of these

methods and then reuse them over any type that we can think of. Now this is very powerful

and can allow us to write much more functional code without having to redefine these methods

and functions for all of our different types. So let's take a look at example of how we

can combine some of these different functional operators to perform complex operations with

very little code. We'll come into this new main function here and we'll start off by

defining a list of strings.

Once again. Now let's look at some ways in which we can chain these functional operators

together to do more interesting things. So as we've seen before, we can do a simple for

each to iterate over each item in this collection and print it out. And if we run it, we'll

notice that we see all of the programming language printed out to the console. Now what

if we wanted to print out only the strings that start with J plus similar to the functions

we were working with before, we could do that by making use of a filter operation. So we

have a lot of options to choose from. In this case, we will just choose a generic filter

and then we will use a predicate which says it starts with J and now if we run this was

he, he had only Java and Java script printed out. Now, what if our collection included

some no values?

So as soon as we add, no, we see now here in our filter operation, it's warning us that

Hey, this value might be no, you need to add a safe call weld in Kotlin. Oftentimes we

don't want to work with no, we want to try and hide no as much as possible. And so we

could make use of another functional operator called filter not know. What this does is

immediately filter out any no values up front. So everything past that in the functional

chain will be guaranteed to be not. No. So as soon as we added filter, not know, we no

longer had to deal with a possible no string. And if we run this once again, we'll see only

Java and JavaScript printed out.

Now what if we wanted to change the type of this? Let's say we wanted to convert this

from a string to an integer, which represents the length of that input string. We could

do this type of transformation using a map function. The map function will take in whatever

the previous type is in this case string, but it'll allow us to return any other type

we want. So in this case, we might define our map function as simply returning the length

of the string. As soon as we've done that. Now below that in the for each, the type has

changed from string to end. And now if we print this out, we'll see four and 10 printed

out for representing the four characters in Java and 10 representing the 10 characters

in Java script. Now let's remove this mapping and let's remove the filter. And instead,

let's imagine that we want to take only a certain number of items from this collection.

So we can do that by using the take function and passing in. Let's say three. What that'll

do is we'll take the first three items from that collection and then we'll be printing

out each of those three names. So you see in this case we're getting Kotlin, Java and

C plus plus. Alternatively, if we didn't want to take the first three elements in the collection,

we could use take last today, the last three. So in this case we see Java C plus plus and

Java script and it has skipped over Kotlin since that was not one of the last three elements.

We can also do other transformations such as associating the input values with some

other value to return a map. So let's create a map that essentially maps the string to

the number of characters in that string. So to do that we could say associate, and then

in this case we could say it to it dot length. And so now in our, for each function, instead

of iterating over strings, we're iterating over map entries of string and event. So in

this case we can now use a template string and say it got value comma it dot key.

And if we print this out, we'll see the length comma followed by the name. This makes it

really easy to map all of the input strings to some other value and then iterate over

that map. Now, what if we didn't want to iterate over the map but instead just wanted to hold

on to that in a variable? Well, instead of using a fork each at the end, we could assign

this to a variable just like this. The continent standard library also provides a variety of

functions to help us pull out individual elements from a collection to demonstrate that that's

created a variable called language. And then we're going to perform different operations

on our list to grab a single language string from our list. So we could do that in a number

of ways. We could say list dot first and if we print this out, we'll expect to see Kotlin

as that is the first language in the list.

Alternatively, we could say we'll start last and in this case you'll see that it's actually

printing out. No, since [inaudible] was the last value in that list. Now, if we didn't

want to retrieve a null value from our list and instead wanted the Alaskan non-male value,

once again, we could add the filter, not no function, which we used previously. And now

if we rerun this, we'll see Java script printed out instead, since this is the last non no

value. Now what if we wanted to find a specific item in the list? Let's say we wanted to use

the find function and in our predicate we'll say it got starts with and we'll pass in Java

as a street. So this is going to find the first value in this list that starts with

Java. So in this case it actually returns us Java and alternatively we could use find

last to find the last element in the collection that matches this predicate, in which case

it's going to return JavaScript.

Now what happens if we are searching for a string which doesn't match our predicate?

We can test that by looking for a string which starts with food. If we then run this, we'll

see no print it out to the console. This is because there is no matching string. So fine.

Last is going to return. No. And then the print line statement, we'll print out. No

if it has a null value. Well what if we didn't want to work with no? What if instead we wanted

to use an empty string as the placeholder? Well, strings in Kotlin have a useful function

called or empty. So we can actually chain that directly off of find last here and call

or empty. So at this will do is return either a nano string or a static empty string. So

now if we run this once again, instead of no, we're just seeing empty, we're not printing

anything out.

So this is one way in which you could default your collections or your strings to an empty

value as opposed to a no value. And this is something you might want to consider doing

more and more of in Kotlin as you start to move away from relying on null. So as we've

seen, Caitlyn has first-class support for functions including functional types and higher

order functions, and the Kotlin standard library builds upon those tools and provides a rich

set of functional operators for us to use. This allows us to build powerful functional

chains to transform our data and make complex workflows much simpler. All right, that's

it for this tutorial. You now have a good understanding of the fundamentals of Kotlin

and how to work with it, and you're now ready to start taking that knowledge and applying

it to other domains. Until next time, devs.