# Transcript 1

in this video we're going to take a look at a concept in object-oriented programming called inheritance I'm going to start this off assuming that you have uh a sense of object-oriented programming on just a very basic level uh you need to know what classes and objects are and how they relate to one another you're going to need some basic vocabulary that we use when talking about um about classes and objects so if you're not quite to that point yet maybe go check out some other resources and then come back here when you're ready to uh take a look at inheritance uh and the other thing upfront is that this video is going to talk about uh inheritance as an idea and as a um a feature of objectoriented programming we're not going to get into writing code in this video because that's going to depend on the programming language that you're using and you'll move on to that um soon enough when we talk about object-oriented programming we talk about the four pillars of o op uh inheritance is one of those along with abstraction encapsulation and polymorphism uh we're not going to get into those um those other three in this video and as I like to do we'll start with just the the very basic question what is inheritance so in our life outside of programming inheritance is a term that refers to you know things that you get from your parents so maybe you inherit a big pile of money or or a big house um but that kind of stuff really depends on the circumstances obviously there are some things that you inherit no matter what uh for example you might inherit your parents hair color somebody says hey where'd you get that great hair oh I inherited it from my parent in that case you've inherited a characteristic or an attribute uh you might also notice that you have some kind of um habit but you know some mannerism from your parents maybe uh you tap your fingers in a certain way and you notice that your mom or dad does that in that case you've kind of inherited a behavior in programming um as a kind of a technical definition of inheritance we're going to say that uh inheritance is the ability to base a class on a different class and it turns out that just like in real life in uh in our programming in oop um our class or our objects can inherit attributes and behaviors too let's take an example of a bank account uh so there are going to be some attributes and behaviors that will be common to all bank accounts they're going to have an account number an owner name and a balance that we need to keep track of uh we're going to have the ability to make deposits and withdrawals and we're going to need to be able to to get the balance and obviously all kinds of other stuff too but for a simplified version of it but it turns out that there are different kinds of bank accounts you might have a checking account and a savings account for example you can see that all three of these classes all three of these things have a lot in common inheritance allows us to say that a checking account is a type of bank account so it can inherit the behaviors and attributes that any bank account has and since a savings account is also a type of bank account it can inherit those same things now a key benefit here to using inheritance is that anything that's inherited by a class doesn't have to be reimplemented in other words you don't have to write code for those things you've already written the code in the bank account class instead we can just add the attributes and behaviors that are specific to those kinds of accounts now a checking account has all the attributes and behaviors of a bank account plus those things that are specific to a checking account only like the last check number that was written uh the ability to write a check and the ability to reorder checks um similarly a savings account is going to inherit all of those attributes and behaviors that are already in a bank account and then we can add on uh the interest rate and an a method to calculate and add interest into the balance now we're going to need some vocabulary to be able to talk about these relationships right this hierarchy of relationships that we are establishing um so at the at the very top of this the class that everything is going to be in uh inheriting from uh we can refer to that as the parent class and any class that inherits from that we can refer to that as a child class now when we talk about implementing these in our code we also are going to start using some other terms so C coders might refer to these as a base class and a derived class and in Java we might call them a super class and a subclass and those last two are the ones that I tend to use a lot um when we're talking about the design phase which is what we're doing in this video it's probably more correct to use the terms parent and child but honestly I use them all pretty interchangeably and as far as I can tell so does everybody else the uh the next step in describing this is when we say that um one class inherits from another the word we use is extends so we say that a checking account extends bank account and that makes sense because a checking account takes what the bank account has and can do and it adds on to it or it goes beyond a bank account it essentially extends the functionality of a bank account same thing for a savings account which is going to extend the function ality of a bank account as well one thing we can do to sort of selfcheck and make sure that we are um thinking about our inheritance correctly when we're do when we're doing our design is what we call the is a test and uh the idea here is that inheritance is used when one class is a specific type of another class so we could say a checking account is a bank account that makes sense so a checking account extends bank account we wouldn't say that a savings account is a checking account right a savings account you can't write checks and and do all that kind of stuff so it wouldn't be appropriate to set up um with savings account extending checking account a savings account is a type of bank account so it's going to extend the bank account class just like checking account does and we can go beyond that I didn't show it here but we can have a VIP savings account which maybe has some other features and a higher interest rate or something like that that could extend savings account which extends bank account and so we can have multiple levels of this and in fact it turns out that there are uh some other kind of variations or or other ways to look at inheritance relationships especially when we get into the idea of polymorphism which is another one of those pillars of object-oriented programming that um that I started started with um that's really beyond the scope of what we're doing here but just understand that um this is the simplest example of inheritance and there can be other uh other things that we can do with it but for now I think this gives us uh enough of an understanding of what inheritance is to go off now and see how to implement that in our own programming language that's the next step for you but for now if you have any questions be sure and reach out

# Transcript 2

all right I am ready to take what we've learned about the idea of inheritance and try implementing that in Java by extending our first class now I am going to assume that you understand the basic idea of inheritance and what it is and you can watch the video that is an introduction to inheritance if you haven't done that yet in this video with code that we write we're going to build on one of the examples from that video and we're going to implement a bank account class and a couple of sub classes based on that now we're gon to not uh do the entire class diagram we're not going to implement everything that we used in that other video but just enough to show you the idea so let's go ahead and jump over to vs code and start working on our class and to demonstrate extending a class I've got a couple of files that I'm going to work with here and you can download these out of canvas I've got first a very simple implementation of that bank account class from uh from the graphic there and it is like I said very straightforward it's got some Fields it's got some Getters for those fields a couple of Setters and then it has a deposit method and a withdraw method and about the only thing interesting uh with these two that you might not expect is that the withdraw method has some logic in it to just make sure there's enough money before it does the subtraction so it checks to see if they have enough money and then uh if it does it reduces the balance and then it's going to return a Boolean it's going to return a true or false indicating whether or not the withdrawal was successful I also have a test program for that and it's not a comprehensive test it doesn't check everything in the bank account class but it instantiates an object it puts in 500 bucks now it's going to try to withdraw 700 and output an appropriate message whether or not that succeeds and then because I know that's really not going to succeed there's not enough money I also then try withdrawing $150 with again a message um outputting whether or not that is successful and finally I'm going to Output the balance at the end just to to make sure that that worked so let's go ahead and run this all right so it does fail to take out $700 but then it successfully withdraws 150 and when I check the balance it leaves us with $350 the first thing I'm going to need to do is I'm going to make a new Java file and the convention in Java generally is every time we have a new class we make a new file I know that we don't necessarily have to do it that way but I'm going to stick with the convention and the first subass I'm going to create is going to be a checking account so I'm going to call this checking account. Java and visual studio code is going to help me out by um putting in that class statement and in order to implement this idea of inheritance since the keyword in Java is extends so um we're going to extend the checking account class or sorry the checking account class is going to extend the bank account class I started to say that backwards if we were talking about this we would say that we extend the bank account uh to a checking account but we're in checking account so we are uh that is going to extend bank account and in fact let's just stop there and try that and see what what uh how this works back in my test program here I'm going to comment out uh a bunch of that stuff so that I don't get confused by it but I might want it later and I'm going to create a checking account object so checking account I'll call this checking one equals new checking account and now I'm going to just um uh try depositing some money into that I'll put in $50.7 and now I'm going to print out the balance and let's just run that even though I haven't written really written any code let's just see what happens okay always hard to see in the uh output window here uh but that does work correctly let's go ahead and clean up my workspace here a little bit and the idea here is literally all I've done is add that keyword extends and indicated what class I am extending so the rule that we say is a checking we use the is a test right a checking account is a bank account and so I can use it just like a bank account in other words it gets to inherit all of the um the term we use as all the members uh all of the variables and uh methods and anything else that is part of that bank account class automatically gets inherited by checking account so without even writing any additional code I'm able to use that checking account now let's start extending the functionality of that bank account uh into what a checking account can do and and what it keeps track of so I'll start with some my Fields here my instance variables and the the idea here is I don't need to indicate things like balance and the owner's name because we've already inherited that all I have to add in this class is stuff that is specific to a checking account so um what we keep track of with a checking account is I'm going to have an integer for the last check number in other words what's the most recent check that we've written and uh we'll also keep track of how many checks we have remaining remaining this isn't um by the way the most realistic or possibly useful uh checking account class but I'm just trying to keep the examples really simple and it's going to be easy to use integers that we can just add one to or subtract one from that kind of thing and in order to since these are Vari uh these variables are private in order to do anything with them I am going to need a couple of getter methods for that so public uh int get last check Nome and that's just going to return this dot as check num and then same kind of thing for checks remaining and I'll clean that up and with that new functionality in place let's go over to our test program and give that a shot so uh now I've got the ability to also get the last check number so number equals or checks from uh last last check number okay now um as you're probably going to guess here the last check number is going to be zero because we haven't set we haven't set that to anything but that demonstrates that this is indeed working now just to make this clear and I hope it kind of already is but if we come back up to here to our we've got this bank account instance called account one I can't get a last check number from account one because that's not a checking account that's a bank account so the idea is we're extending a bank account into a checking account and we're adding some stuff that stuff doesn't apply to a bank account I think that's kind of implied or I hope it is but I always like to make that clear um not because you won't understand it but because when you're writing code that's an easy thing to get confused about is wait a second am I am I using a a bank account here or a checking account here uh so the things that we add to a subass are not available to the super class let me go ahead and fix that get us back to where it's working so we want to add some methods to our checking account as well not just these uh fields that we've added so let's go back to that account and take a look at what we got and the first method we want to add is going to be a method for writing a check and this is going to be what I would call a modifier method because it's going to modify um one or more pieces of data within our class so modifier methods and this is going to be public and for now we're going to say that a a check writing a check doesn't return a true or false at the time you write a check we don't you know have any way of checking what the balance is and knowing if it's valid or not it might overdraw it might balance right so we're going to write a check and for for our purposes here we're going to say that we need to specify the amount of the check and what the check number is um as I mentioned this is not the most realistic uh checking account approximation here because you know we're not we're not going to keep track of who the check is to or what the date was or anything like that I'm guessing it doesn't matter because probably a lot of the people watching this video have never written a check in their life because you do everything online now um but it's still a good analogy I hope for for this inheritance idea and the first thing we're going to do is we're going to update the last check number uh variable here because if we have written a check then now the last check number is going to be equal to whatever check number we've passed in here uh this this integer that's a parameter so that updates that the other thing we need to update of course is the balance and so we can say um this. balance minus equals amount and we've run into an error okay the problem is Visual Studio code doesn't doesn't uh know what balance is and here's why when I created my bank account class I very correctly specified balance to be private because we always make our instance variables private um but that causes a problem private means that no other class can access it well checking account is another class and it kind of needs access to that variable so we have a few different ways we can address this we have um one way of addressing this that we absolutely should not do and then we have a couple options that are okay and so let's start with the one we should not do um and one that I see students do every semester is they come over to the bank account class and oh this is easy enough I'll just make that public this is particularly popular with python programmers because in Python you don't run into this kind of restriction python just lets you do whatever you want and I come back over here and hey great it works this is all good it's not all good because objectoriented programmers heads just exploded um remember we never let anyone touch our privates these variables are private for a reason we want them to be protected we want them to be um made safe from people messing around with them uh and the reason here is that if somebody shouldn't just be able to come along and manipulate the balance that's the whole reason we have um these methods in this class if you want to change the balance you make a deposit or you make a withdrawal and that's how you change how much money is there you can't just come along and change that variable but in checking account when we write a check that does change this variable so again there are a couple different ways we can address this and one of them is to go into our bank account and up until now we've only had public or private but there is another access modifier we can use in uh in Java it's called protected so private doesn't give access to anybody any class public gives access to every other class protected is in the middle and what I wish I was about to tell you was that the protected keyword gives access to any subass of bank account but that's not quite how it works and in fact I mess up sometimes when I explain this and I explain it that way in Java the protected keyword gives uh package level access so it gives access to any class within the within this package and so it's not quite as good as I would like it to be but um uh but I always have to say that carefully because I'm used to kind of saying well that gives access to the sub classes um it gives access to more than that unfortunately but um this is a this is an appropriate way to solve that problem okay an object oriented programming making an instance variable protected so that subclasses can access it is the way to go all right and let's go ahead and test this let's go over to our test accounts and uh so we've made a deposit let's say check oops check one. write check I'm going to say write a check um for $50 and it's going to be check number 13 and so when I run this now we should um see that we have deposit of deposited 150 we've now written a check for 50 bucks and it's check 13 and so the balance is $100 and then that crazy U floating Point math there that gives us that that obnoxious decimal uh afterwards and then we've got the check number is 13 I'm not going to bother rounding that off because this is not the point of the video we're just uh learning about inheritance uh but this seems to be working now let's go back to over to our checking account I did say that there is another way to do this and um that is that we can use the methods provided in our super class if those help us out and what I mean by that is that when we write a check we are kind of making a withdrawal right we're we're there's some other stuff associated with it like a check number but ultimately this line of code here where we're subtracting the amount that's a withdrawal so so what I can do is I can just use that method from the super class so I can say um this because this is this is an instance method this. withdraw amount and so now when I run this code that that writes a check it's going to bounce into this checking account method it's going to update the the last check number and now it's going to call the withdrawal the withdraw method and you have to understand how these method calls work in Java and I'm going to explain this in kind of layman's terms here when we have a method call in Java it's going to First Look within the current class it's going to have a look around and say hey I need to I need to call a withdraw method and if it has one then it runs it if it doesn't have one it moves up The Inheritance tree or up the inheritance hierarchy and it goes to the super class and it says oh okay a checking account extends bank account so let's go look there and it goes into bank account and it says hey I'm looking for a withdraw method do you have one yep we have one right here and it runs it if I try and call a method that's not in either of these so let's say I call this dot takeout I'm account then it looks around here and says well I don't see a takeout method in checking account so it goes up and says well that extends bank account so let's check there hey do you have a takeout method no we don't have a takeout method and it gives us an error it won't compile because it can't find an implementation of that um we're going to learn a little later that there's a little more to it than that because uh there's one more check that it does it checks the object class to see if there's an implementation there but there's not so um so that line of code doesn't work this approach of using the superclass method has a couple of advantages one of them that I like because I'm nerdy about objectoriented programming and encapsulation is that now I can go back and make this private again right because I don't need to access the variable directly I can call the modifier method that is intended to edit that for us so that I'm I'm still protecting um maybe the wrong word to use since I just got rid of the protected keyword I'm still keeping that variable safe I'm still keeping it encapsulated so I like that aspect of it but the other aspect that I like is that if I have code written in this superclass method I don't need to duplicate that now because this this um superclass withdraw method doesn't just subtract the balance like I like I was doing it also checks to make sure that you're able to do that so I don't want to have to rewrite all of that code and you can imagine well what if this also does some other stuff maybe it goes into a database and it updates some stuff there or it goes to a website and it checks things um you know pulls the balance out of that website I don't want to have to maintain all that code I I don't want to have to have that code duplicated so if I can just call that method then that makes it uh easier for me to maintain my code and to make sure that I'm um I'm following all the rules that withdrawing money requires now a couple things to keep in mind just about the basic idea of inheritance so uh bank account is my super class and checking account is my subass uh I can still make other sub classes of bank account right so I can make uh the other example in my uh in my previous video was a savings account okay so I can make a savings account. Java and I can say that that extends bank account and this is going to have you know stuff that a uh that a savings account has so it's it might have private double um interest rate okay and then it might have a um method that is uh private void add interest or calculate interest or something like that and um I won't actually I won't actually Implement that on this video um but the idea is now I can have uh multiple subclasses of the bank account class and in my test program I can make a savings account in exactly the same way now this I'm not going to be able to demonstrate anything with it since I didn't really write any code for it but um but it's going to have uh just like when we started the video this is going to have access to deposit and to get the balance and all of that stuff that comes with the bank account class so it's not a it's not a one-time deal I can make as many sub classes of bank account I can make another way of saying that is I can make as many different types of bank accounts as I need and often we do have many many different sub classes in fact one of the very common things we do just as a design approach is that we make your bank account might not even be really an account that we really use it's just sort of a base class that we use um to put all of the code that checking accounts and savings accounts have and we never actually use a bank account so all of that is perfectly fine so this really is just the basics of how we use inheritance uh we've learned how to use the extends keyword to create a subass and we've added some fields and methods and we've kind of demonstrated that works in our test program uh with all of that as a foundation we can now start learning some other inheritance Concepts and we'll do those in separate videos in the meantime you can reach out if you have any questions

# Transcript 3

okay so we understand how inheritance works on a basic level we've gone into Java and we've extended a class we've tested that to make sure it's working on a very basic level for the next part we need to understand how method calls work in an object-oriented programming environment we're going to consider a situation in which we have two classes that we're going to start with we're going to have a friend class and a family member class and these classes will encapsulate all the things that we keep track of about these two different uh sets of people and all the behaviors that they have we then are going to add a close friend class now a close friend is probably a lot like any other friend but maybe has some special behaviors um that a regular friend doesn't have and so we can consider a an inheritance arrangement here we would say a close friend extends friend and this setup does pass our is a test right this is where we say well a close friend is a friend that works and so that suggests that this is probably a good arrangement for our inheritance if we add another class to the mix that is um best friends and turns out that our best friends are our cousins so now we have this class that causes a little bit of a problem is a best friend cousin a friend or is a best friend cousin a family member The Logical solution here would be that it can extend both right because a a best friend cousin is a friend and a best friend cousin is a family member but as the Beatles might say you can't do that at least you can't do that in Java this would be called multiple inheritance and it's where a subass has more than one super class that it extends some programming languages do allow this and the problem with it is that you can run into situations where there's a an AM ambiguity concern right we want to run a method in one of the super classes and we don't have any way to be sure which one gets priority here so the developers of java just avoided it and said well we're just not going to support that we're not going to let you do it Java does have something called interfaces and interfaces give us kind of a workaround they give us a way that we can have something like multiple inheritance but that avoids that ambiguity problem that's beyond the scope of what we're talking about here but it is an option so in Java we can't do multiple inheritance in that sense but what we can do is what we might call multi level inheritance and that is to say that um one class can have a super class and can have only one super class but that super class can also have a super class and that super class can have a super class and we can go as far up there as we need to go this idea ends up resulting in what we can call an inheritance hierarchy which is the kind of a visualization of how all of these inheritance relationships fit and there probably would be other branches to this like for example we might have a work friend for someone who you don't really like but there's nobody else at work so you have to hang out with them there or like a friend of me that you are nice to each other's faces but then talk bad behind each other's backs those would be whole other sections of this hierarchy we're just going to figure um on talking about this one kind of branch that happens to fit nicely on my screen right now now these three classes all have methods they all have behaviors that they offer right so a friend can give us a hug a close friend can be honest with us if we ask them do these shoes make my feet look big they'll give us a true answer but since a close friend is also a friend they inherit the ability to give us a hug that means of course that a best friend inherits the ability ility to be honest about our shoes and to give us a hug all the way up the inheritance tree we get to inherit all that stuff and then a best friend adds on the ability to lend money now because these inheritance hierarchies can get complicated and big it's important for us to understand how method calls work so let's imagine that we have an instance of the best friend class and we're going to call it BFF and we want to call the honest opinion method for that BFF we need to ask them about a pair of shoes we're thinking about buying so here's what the compiler does it's going to look and see that okay BFF is an instance of best friend so it goes to the best friend class and it looks for this honest opinion method of course it's going to look around here and not see that but what it is going to see is that oh best friend doesn't have an honest opinion method but it does extend close friend so I better go look there and so it's going to go up the inheritance hierarchy one level and it's going to look in the close close friend class and it's going to look and see oh here's an honest opinion method so I'm going to use that if we try and call a method that doesn't exist like we try and ask our we want to ask our BFF to go on a date which is not a behavior that a best friend has we probably shouldn't be doing that that but it's going to come to the best friend class it's going to look around say hey I'm looking for a method to to go on a date nope but this extends close friends so let's go look there looks around no go on a date method there but that extends friend so we better go to friend and look no go on a date method here either and so now we have a crash right once it gets to the top of the inheritance hierarchy if there's no method there where our code is not going to compile now we're going to see in a little bit that there's a little bit more to this there's kind of one more step in the process but for now that's our understanding of how this works a method call is going to start out in the subass and it's going to work its way up through super classes until it finds an implementation it's looking for now what happens if we have a situation where a subclass inherits a behavior but it does that behavior in a little bit of a different way so for example a close friend inherits the ability to give a hug but a close friend hugs in a different way than a friend does right this type of situation is called overriding right not to be confused with overloading which is a a topic that you probably learned about previously in method overriding it means that a subass is going to uh customize an implementation for a method and because we understand the way that a method call works up the inheritance hierarchy we know that a best friend if we call the the give hug method it's going to move up until it sees an implementation for that essentially it means that a best friend hugs the way that a close friend does which is maybe a little different than the way a friend hugs all right so this is all great with these examples of friends and best friends and things like that but let's go ahead and connect this a little more to some code that we're going to write so um here's an example uh that we have used in previous videos it's a bank account super class and then we have implemented a checking account we also implemented a savings account that's not shown here uh and what we want to add right now is a VIP checking account so that we have that idea of multi-level uh inheritance right a VIP checking account is a certain type of checking account and I I don't know it's going to you get a little bonus when you make a deposit if you're a VIP what I want to point out here is that the VIP checking account has a deposit method just like a bank account does so this is an example of overwriting it's a little different than our previous example because we're skipping one class in the middle it doesn't matter the idea of an override is that we're going to implement a method to replace something somewhere up the inheritance hierarchy right so um a a deposit for a VIP checking account Works a little bit differently than a deposit for a bank account which is inherited by a checking account if we look at that checking account I do want to point out we have a right check method that really is almost the same as a withdraw method right they both withdraw money from the account uh writing a check involves a little bit of other information this is not actually an example of overwriting for it to be overwriting you have to use the exact same method name all right so the deposit methods this uh VIP checking account deposit overrides the bank account deposit the right check method happens to do something very similar to the bank account withdraw but it's not really overriding let's go ahead and jump into Visual Studio code and see some examples of this and you can get the the files that I'm using from canvas I've spent all this time talking about friends and close friends and best friends we should at least see a little example of this in code I I don't even know if I'll post these files because they're just really simple classes the first one is a friend class and it has a give hug method so if I come over here to my test program which just makes um one instance of each so I can say friend. give hug and if I run that it says this is a friendly hug now if I have a close friend so I'm going to delete that and I have my close friend dot um what can a close friend do oh the close friend can give an honest opinion so I'll run that so our inheritance seems to be working let's take a little bit of a look at the overriding so my close friend class extends friend it has an honest opinion method and then it has a give hug method so this is the over ride when I have my best F my close friend or best friend because it inherits from that um give a hug it shouldn't say this is a friendly hug this time it says this is a close friends hug and if I change that to BFF BFF inherits give hug from best uh from close friend so we should get that same output so far so good U there's not much to this syntax here but you got to be careful that you are naming and including case sensitivity uh your method the same as the super class method you overwriting and to help you with that um there's something we call a compiler directive and so this is the um at sign and I'm going to say at override and this has no impact on the execution of the program this is just a message to the compiler that says hey this next method is supposed to be overriding something in a super class again it doesn't change oops that one's not going to run let me switch over to my test uh it doesn't change the way anything runs still says this is a close friend's hug what it does is it helps catch any mistake you might make when you're overriding because it's really easy for example to have a typo without that override directive there that doesn't give an error because this is not bad syntax it's called give hug without the e in there well Java does doesn't know that's a misspelling it's just another method but when I run my code now save that come over here and run it I should be getting a close friends hug but instead I'm getting giving a friendly hug I called give hug it came I came to best friend didn't find any of those it came to close friend it didn't find the correct give hug it found this other one but it didn't use it and it went up to friend and it ran that one so the override directive is just just a way to um help catch the errors that you might make um I often forget to use it but I I encourage young programmers beginning programmers to use that because it's it is easy to to mess those kinds of things up so that's just a really basic example of multi-level inheritance and Method overriding let's switch gears and look at that bank account example okay so this is my bank account class I think it's pretty much where I left it off after that last video um nothing particularly interesting to us here at the moment um then I've got my checking account which extends a bank account and it adds the ability to write a check and the one I'm creating is going to be called a VIP checking account and so now when I go to extend this I'm going to say extends checking account because this VIP account is a form of a checking account and I misspelled extends there we go and the idea here is I want to override the deposit method and we're going to make it so that if you have a VIP checking account because you're rich and fancy and have one of those little monal things in your eye um whenever you make a deposit you're going to get like a 05% um bonus or something like that so I have that override compiler directive and now I'm going to say public and I've forgotten I don't remember what my deposit method looks like so I'm going to go check um a deposit method um it's a void method it takes an amount and by the way it wouldn't have to be exactly uh this I'm just I can't remember what it was and so I want to get it right the first time U but okay so it's G to be public void deposit and if I was messing it up the compiler override would help me anyway double amount all right so first thing I'm going to do is I'm going to calculate a um a bonus here double bonus equals amount times I'm never good at math but if I want 0.5% half a percent I believe that's 0.05 am I doing that right 0.5 point I think I went one Z too far we'll find out um and then we're going to say that this dot balance equals or we'll just say plus [Music] equals bonus now we've run into this error that we have seen previously balance is a private variable in bank account so I don't have access to it remember that we had two solutions I could uh I could change that private keyword to protected and that would give me access to it and that's not a bad solution I I certainly don't want to change it to public I think I made that clear in the other video but I can change it to um I can change it to protected and that would give me access and I said I also could just call a method uh in the previous in the super class excuse me so if I say deposit and I'm going to deposit the bonus plus the amount let's see what happens if we run this so I'm going to come to my test account I'm going to um not going to write the check I'm going to try and make this as simple as possible so I'm going to put in $100 right because a nice even number and then we're going to print out that balance oh and I I almost made a mistake here I need to change my checking account to a VIP checking account and I probably should rename that so I don't confuse myself again um I like to use a little rename um function in vs code because then it's sort of like a global find and replace there let's go and run this and see what happens I've got this odd error okay I've got this message that has printed out a whole bunch of times and I'm not sure what's going on here well I am but but you're not sure so let's take a look we're going to use the um the debugging feature in uh in VSS code for this I'm going to set a break point and let's run this again and see what happens oh and I've made a mistake I I ran it instead of saying debug it so I got to change that to say debug Java there we go now we've stopped and here I am at the um VIP checking account equals new VIP checking I can step over that I can step over that but this is the one I'm interested in I want something's going wrong when I'm doing my deposit so I'm going to step into that method and I've got some variables over here amount is equal to 100 that's good I'm going to step over that and my bonus is now 0.5 okay so um 50 cents I think that's right again if my math um is is uh serving me correctly and now I've got this call to deposit remember I want to call the deposit method in my super class so that it just does that deposit for me so I'm going to step in here oh and it didn't go to my super class it stayed right here in VIP checking and it's just going to keep staying here it's calling itself so we call this recursion and there are there is a way to use recursion uh intentionally where it essentially is a loop but I haven't done that here this is just an error and what happens is every time you call a method the Java runtime copies a bunch of stuff related to that method and puts it in a place on the memory called the stack and eventually that stack runs out of uh runs out of room and we say that the stack overflows right so I'm getting a stack Overflow error you probably know that phrase stack Overflow as the site where you go to cheat when you're stuck on one of your homework problems but that's where it gets its name it's from this idea of a stack Overflow error when the U when that area of memory runs out so we've got to solve this problem I again I could just change it to protected but this is a good chance to teach a little concept here I don't want to call this deposit method I want to call the deposit method for my super class so what I can do in Java is say super. deposit and the super keyword um this is essentially like you know I've always do things with like this okay this do deposit would refer to this object super refers to the super class so now let's try let's save that and try running it see if that solves our problem turn off my breakpoint and be optimistic that this is going to work okay $100 50 so it did work the concept here is that when we have a need to call the super class version of a method specifically we can sort of prefix that with super Dot and it will jump to that super it and what it actually does is it just starts moving up the inheritance tree from here until it gets to another uh implementation of deposit because remember the super class here is checking account checking account doesn't have a deposit method so then that extends bank account it comes to bank account and it finds it there um this is a concept that we're going to come back to in an upcoming video because it has a really important role this uh this idea of calling a super class uh when we get to it but for now this is just a nice little example of how we can have multi-level inheritance and how method calls are going to move up that inheritance hierarchy uh and allow us to do things like method over overwriting which is giving a custom implementation of a method within a super class now a few minutes ago I talked about how a method call moves up the inheritance hierarchy in this case BFF go on date which doesn't exist and when it gets to the highest level super class it then essentially won't compile from there but I had that little comment that that's actually not quite it there is another step to the process so there is one more thing that plays into this um idea of method calls and this hierarchy and it's probably easiest to just go into Visual Studio code and show you all right there's a method that has kind of a special behavior in uh Java it gets treated a little bit differently than most other methods and so I'm going to demonstrate this by using a print line statement and what I'm going to print out is just my instance of the VIP checking account object so that's VIP one now VIP P1 is not a string it's an object if I run this I'm going to turn off debugging here and just run it regular what I get when I print out an object it looks sort of nonsensical it actually is it does mean something so you can read some of it it's the name of the class that this object is an instance of and then it's got this thing that we call a hash code it's you can think of it as kind of an address uh in memory where this where this object is can be found so it's it helps um helps the runtime find this object when it needs to use it well what happened here is that the print line method really just takes a string and a vip1 is not a string it's a VIP checking account so what the compiler has done essentially is it it's added on to that a little method called called two string as in we're going to convert that object to a string and if I run this it should work exactly the same way because essentially I believe the compiler actually is just adding that call on when it when it compiles I think I'm accurate when I say that uh but certainly conceptually that's what's happening so um where does that come from it turns out that there is one class to rule them all in Java and that is a class that is called object and since I don't have a fancy PowerPoint up there for this uh object a class called object and every class in Java extends object if it doesn't extend something else so in other words VIP checking extends checking account and it can't extend anything else because there's no such is multiple uh inheritance in Java but checking account then extends bank account bank account finally we get to one that doesn't extend anything so again I'm not sure if I'm 100% accurate but I think we can think of it as the compiler is inserting a line here that says extends object all right and the object class provides a handful of functionality that every object in Java needs to have there aren't a lot there's not a lot to it you can look it up online if you're curious but the main one that we interact with is this two-ring method right because often we want to print out an object it is a method just like any other method so I can overwrite it just like any other method so I'm going to come to my VIP checking account and I'm going to use the override co uh compiler directive again so the two string I happen to know this you would have to look it up if you didn't but public string two string and I um I've misspelled it that was not intentional but it does reinforce the idea of this compiler directive because if I didn't have it um it would think that oh well I'm getting a different error now sorry that because I'm not returning anything yet so let's just uh return VIP for now but now I'm getting an error because uh because I misspelled that and if I have that override directive turned off it thinks that's just fine again it doesn't understand that it's a that it's a misspelling so I'm overwriting two string and if I come and run my code again now instead of printing that uh class name and the hash code it prints out whatever the two string method provides in general what we use this for is I mean the way I explained it is we want a text representation of the object and so um I might say VIP checking account and then a new line and I'm going to say balance equals and remember this. balance is private so I can't use that but I can use the getter and I'm going to return that once I fix my typo and this is typically the kind of thing you use a two string method for um I often uh will just have it sort of dump out all of the all of the fields in my class just so that I can run my code I can print it out I can print out an object and I can just see what all the instance very variables are so this is another example of overwriting right in this case we're overwriting the two- string method even though we didn't write it and up until a few minutes ago we maybe didn't even know that it existed I think this is a good place to stop as is always the case just reach out if you have any questions

# Transcript 4

I want to talk about the way Constructors work in an inheritance situation because there are some special behaviors that we have to be aware of otherwise our code will not compile you might have noticed that in my other inheritance videos I've avoided using any Constructors at all that's because we hadn't gotten to this video yet in order to make sense of this you do need to understand how to extend a class and you really need to understand how overwriting works and that I idea of a kind of an inheritance hierarchy that's used for method calls but if you've watched those let's go ahead and take a look at Constructors we'll use the bank account and checking account classes that we have been using up until now and we're just going to ignore the other methods here and we're just going to talk about the constructors recall that when you create a class if you don't Define a Constructor then the compiler is going to create one for you it's just going to take all the fields and set them to some initial values like zero or a null reference or something like that so I've got those Constructors shown on my diagram here even though we didn't write the code for them now consider just the sort of generic line of code we use to create an object to instantiate an object uh so checking account is the class that we're dealing with uh we're going to call it account one and then we're going to use this new keyword with checking account Constructor call and again I haven't created The Constructor but the compiler is doing that for me now when we talk about overwriting we had just gotten used to the idea that um overwriting method calls take place from the bottom of the hierarchy and work their way up but with a Constructor it's different a Constructor starts at the top it starts with a superclass Constructor and it has to run that Constructor no matter what after it's run the superclass Constructor then it's going to go ahead and go on to any subclass Constructors it finds all the way down the inheritance tree let's go ahe and take a look at and see how this looks in Visual Studio code same old bank account that we've been working with and a checking account that extends it the only changes I've made is I added a new test file for it a new test program because my old one was getting a little bit cluttered and so this is just called test Constructors and it's got a main method with nothing in it uh let's go ahead and create a checking account object here checking account account one equals new checking account and again that's that default Constructor call because I haven't defined a Constructor of my own and I will just print out the uh we'll get the uh account number from this so so system out print line count one.get account number just so we have something that I can test and make sure things are working so far so good again uh that's that null reference that I mentioned a moment ago uh since my default Constructor doesn't know what else to put in that uh in that in these fields it just puts a null reference there for that string and let's go ahead and go over to the bank account and create a Constructor so public bank account and we won't have any arguments here um people a lot of people refer to this now as a default Constructor and this is where I get nitpicky it's no longer a default Constructor because I wrote it so a default Constructor really is what the compiler makes but everybody calls this a default Constructor since it uh doesn't have any parameters or arguments excuse me uh I'm going to not do Constructor things with this here I'm just going to have it print something out so I'm going to say have it print out bank account Constructor and then I am going to go to the checking account and I'm going to make basically the same thing public checking account another parameter list Constructor here and I'm going to say checking account Constructor save both of those come over to my uh test Constructor program here and give it a shot sure enough it says bank account Constructor first then it says checking account Constructor then it says this no pointer that's from part from our account number here so this is just confirming what we saw a moment ago that unlike regular method calls a Constructor call uh works from the top down now where the potential problems come in is when we talk about Constructors that do have arguments and so let's say I get rid of that that parameterless Constructor that default Constructor and now I've got one that takes an account number which is a string when I go to instantiate an object at first this looks okay because the checking account uh class does have that parameterless Constructor still but what happens is it comes in looking for this Constructor in the bank account first and it says well here's the only Constructor I see and it requires some information I don't have that information so I can't do anything here the solution here is that our subclass Constructor needs to specifically or explicitly call that bank account Constructor and pass in the information that's needed so let's go take a look at how to do that okay so let's go over to our bank account and make this new Constructor so I'm going to say public bank account and this time it's going to take a string I'll call it account number like that and all it's going to do is assign oops assign account num equals that parameter now I I in the interest of speed here I'm not doing all the other variables but I really should assigning values to all those and now at this point we're still okay my program should still run just fine because this quote unquote default Constructor is still available the problem is often we don't want or don't need to create that that sort of default Constructor there and so now when I try and run this I have a problem it might be hard to see there but it's an uh it says implicit super Constructor bank account is undefined uh it must explicitly invoke another Constructor so if the only Constructor I have requires one or more um parameters then this checking account class doesn't compile because it doesn't deal with that piece of information that's missing so the idea here is if your super class requires parameters in order to create an object the very first thing you have to do in your subclass objects are call is call that superclass Constructor so remember we had that keyword super that we used in our uh in our overwriting video if we just put super with empty parentheses that now refers to the Constructor and in that parentheses I can put um I'll put in just 1 two 3 four five as my account number that should now compile and run it so so we're going to kind of clarify or add some Nuance to my description before it's not that the bank that it starts at the top and begins calling Constructors from there it's that the default Constructor of the subass is really calling that default Constructor of the super class first and it does have to be the first thing so if I take um if I take that line of code and I put it at the bottom that doesn't work it's going to say that it must be the very first statement in the Constructor okay now I don't have to use that sort of literal value that constant value 1 2 3 4 5 I can for example make a new Constructor that also takes a parameter and I can just say super and pass in that account number and that should work as well and so now I can create a bank account by specifying that account number I'll just say AB c d e and if I run that that works ABCDE e now to be clear I've got two Constructors here I don't need to have both of them necessarily it's just a question of which way you need to call and create your objects but the takeaway here is that if you have any requirement that the super class Constructor has parameters has gets some data and you don't also provide a parameter list Constructor then your subclasses have to address that they have to use this super keyword to call back and run that Constructor in the super class before it can go on and do anything else it's a somewhat confusing concept at first you get the hang of it pretty quickly the error messages once you start seeing them you'll go oh yeah I forgot that I got to take care of that um so it's it's not hard once you get the hang of it but if you have any trouble with it reach out and let me know

# Transcript 5

when I talk about object-oriented programming I often talk about these four pillars of object-oriented programming and it's not that I think it's important that you know what these four pillars are I just think they are uh instructive about what we consider to be important in object-oriented programming one that I haven't talked about yet is polymorphism and in this video we're going to talk about polymorphism um in the context of arrays and we're not really going to go any deeper than that we're not going to go into any other uses of polymorphism uh but in order to understand what we're doing you are going to need to know how to use arrays and you are going to need to be um fairly familiar with inheritance at this point you need to know how to extend classes and need to understand how uh method overwriting works so polymorphism gives us the ability to treat instances of different classes as if they were objects of a shared super class what the heck does that mean what it means is that if we have a checking account and a savings account object we can treat both of those the same as long as we're treating them as bank accounts because they both uh inherit or they both extend bank accounts so that means that um regardless of whether it is specifically a checking account or specifically a savings account we can make withdrawals deposits we can get the balance all those shared um elements of those two uh subclasses now the advantages of being able to do that may not be obvious right away so let's take a look at how this applies to something like an array if we consider that we have made an array of checking account objects that hold six uh instances I can put obviously checking account objects in that but if I try and put a savings account object into that array it doesn't work right and this goes back to our basic understanding of arrays which is they can hold um a collection of things as long as those things are all of the same data type what polymorphism says is that if we change this array and instead of making it an array of checking accounts we make it an array of bank accounts now that savings account can go in that same array and we can have an array that holds a mix of any subass of the bank account class and we can interact with those in an array and we've already seen that when you have an array first of all you can hold big amounts of data but just as importantly we can run Loops through that data and we can process it really efficiently so this gives us a way to handle all of our accounts in one array let's take a look at how this would work in code here I've got this array of checking account objects and I've attempted to put six instances into that array and the ones that are using the checking account Constructor are working just fine but anytime I try and use that savings account Constructor I'm getting a compiler error as we discussed just a moment ago we can change this from a check-in account to a bank account and I'll have to change it in both places here for the moment um now those will at least compile I'm then going to deposit money into them I'm going to run it right uh real quick just to make sure it really is working and it is there's no output because this program doesn't produce any output yet and let's just start by doing a traversal Loop to go through and get the balances of these uh these objects so for in I equals zero this is just our good old fashion uh traversal array and so now I'm going to print out the balance for each of these in order balance equals and we'll just say accounts I do get balance let's go ahead and run that see if that works so we are off and running the idea here is that I we've already seen it working because in the uh in the lines of code here where I'm making deposits I am successfully count uh calling um super class methods for instances of each of these subclasses so it is working but the the more important thing is that in my array I can have this array of as many accounts as I want and each element in that array can be from either subass and it will still work there are some implications here to be aware of um one of them for example is that I can't call any methods that are specific to one of the sub classes so if I say accounts and let's pick one of my savings account account five well I'll just put I we're in the loop so I'll put I um and I'll say add interest which is one of my subass methods that's a savings account method that won't compile right because it's looking at this as a bank account remember in polymorphism we're treating the subass as the super class the subclass instance I should say and so um it's looking at it as a bank account it's essentially starting that um that method call up the inheritance tree it's starting at bank account because I've said that that's what the class is um and that's not um that's not specific to the loop so if I take this up here because that maybe the that I makes it confusing right because it could we never know what it's going to be but even if I pick something that is explicitly a savings account that still doesn't work now we have some tricks that that we're not going to get into here because it's kind of beyond the scope of this short little video um where we can maybe make that work um but instead of going into depth I just want to show you a little um example of that or or a hint of that I should say and that is that we do have a way we have a couple ways we can test uh an object from this array and see which class it is and so if I say accounts i. get class so there is a method called get class and uh you can see in the output here it's saying checking account or sa's account um well where did that get Class come from we recently have learned where that probably comes from from when we call get class it's going to look at bank account first and doesn't see a get class method there so every class in Java inherits from a class called object remember it's one class to rule them all well get class is a method in that object class so any object in Java we can call get class and find out what it is an instance of speaking of instance of we can also make that uh an if statement by we can there's a Boolean um it's not really a method it's an operator but I can say um if um accounts I instance of savings account and I misspelled savings account there I put an S on the end um so this instance of is a um Boolean operation that compares whatever's on the left to whatever's on the right uh or well sorry it looks at what's on the left and says is this an instance of the class that I refer to on the right and it's going to return a true or false um now it's not quite as simp Le is then um saying well okay we know it's a we know it's a savings account now so let me just add the interest it's not quite that simple um that we have to do some downcasting and some things that again are beyond the scope of what we're doing here but I just wanted to point out that we're not stuck we do have some options there the other thing that's a little more relevant to us now with what we know so far is that um regardless of of what the instance uh instance of returns without us having to worry about it our method calls will move up the inheritance tree um in terms of overriding so if I go to my savings account I have an overridden deposit here but if I do that if I say public void deposit and we'll say double amount and I'm just going to say system out print line this is a worthless override but it shows the point this is a savings deposit and so uh now I can just do my super class deposit with the amount and I don't have to add any other code um so now if I come back to my uh my test method here my test program and run this again and we're really looking at these lines of code up here so uh anytime that we call a deposit method it is moving up that inheritance tree I think I kind of misspoke a few minutes ago we are still going to move up that inheritance Tree in terms of overriding right and um it's not clear what's happening because they're only they were sort of mixed in here but every time we deposited to a savings account it ran ran this overridden uh this override version of that deposit method and so it spits out that little line of text the point here uh overall is that we are still able to create an array of bank accounts put the subclasses in it we can't call the methods that are specific just to that subass but if we call a method that's overridden it will respect that and it will call the correct version of our overridden method so this really has been just the very briefest look at what polymorphism is it has a lot of implications Beyond just arrays but arrays of subclasses is something that we can put to use right away and in fact I'm going to ask you to put it to use right away and it builds very logically on what we just learned about in arrays as is always the case if there's been anything in this video that you're not sure about please reach out