REPORT ZYNY\_CLASS\_08.  
\*----------------------------------------------------------------------\*  
\*       CLASS account DEFINITION  
\*----------------------------------------------------------------------\*  
\*  
\*----------------------------------------------------------------------\*  
CLASS account DEFINITION ABSTRACT.  
  PUBLIC SECTION.  
    METHODS:  constructor IMPORTING  
                value(i\_account\_holder) TYPE string  
                value(i\_amount)       TYPE f,  
  
              withdraw ABSTRACT  
                IMPORTING i\_money                TYPE f  
                          i\_within\_notice\_period TYPE string  
                EXPORTING e\_money                TYPE f,  
  
              deposit ABSTRACT  
                IMPORTING i\_money TYPE f  
                EXPORTING e\_money TYPE f,  
  
              get\_account\_holder  
                RETURNING value(r\_account\_holder)  TYPE string.  
  PROTECTED SECTION.  
    DATA: account\_holder TYPE string,  
          balance      TYPE f.  
ENDCLASS.                    "account DEFINITION  
  
  
\*----------------------------------------------------------------------\*  
\*       CLASS current DEFINITION  
\*----------------------------------------------------------------------\*  
\*  
\*----------------------------------------------------------------------\*  
CLASS current DEFINITION INHERITING FROM account.  
  PUBLIC SECTION.  
    METHODS:  withdraw REDEFINITION,  
              deposit REDEFINITION.  
ENDCLASS.                    "current DEFINITION  
  
\*----------------------------------------------------------------------\*  
\*       CLASS notice30 DEFINITION  
\*----------------------------------------------------------------------\*  
\*  
\*----------------------------------------------------------------------\*  
CLASS notice30 DEFINITION INHERITING FROM account.  
  PUBLIC SECTION.  
    METHODS:  withdraw REDEFINITION,  
              deposit REDEFINITION.  
  
  PROTECTED SECTION.  
    DATA within\_notice\_period TYPE c.  
ENDCLASS.                    "notice30 DEFINITION  
  
\*----------------------------------------------------------------------\*  
\*       CLASS account IMPLEMENTATION  
\*----------------------------------------------------------------------\*  
\*  
\*----------------------------------------------------------------------\*  
CLASS account IMPLEMENTATION.  
  METHOD constructor.  
    account\_holder       = i\_account\_holder.  
    balance              = i\_amount.  
  ENDMETHOD.                    "constructor  
  
  METHOD get\_account\_holder.  
    r\_account\_holder = account\_holder.  
  ENDMETHOD.                    "get\_account\_holder  
ENDCLASS.                    "account IMPLEMENTATION  
  
  
\*----------------------------------------------------------------------\*  
\*       CLASS current IMPLEMENTATION  
\*----------------------------------------------------------------------\*  
\*  
\*----------------------------------------------------------------------\*  
CLASS current IMPLEMENTATION.  
  METHOD withdraw.  
    WRITE: / 'Current Account Openning Balance: ', balance EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
    IF i\_money <= balance.  
      balance = balance - i\_money.  
      e\_money = i\_money.  
    ELSE.  
      WRITE / 'You do not have sufficient funds for a Withdrawal in your account'.  
    ENDIF.  
    WRITE: / 'Current Account Closing Balance: ', balance EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
  ENDMETHOD.                    "withdraw  
  
  METHOD deposit.  
    WRITE: / 'Current Account Openning Balance: ', balance EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
    balance = balance + i\_money.  
    e\_money = i\_money.  
    WRITE: / 'Current Account Closing Balance: ', balance EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
  ENDMETHOD.                    "deposit  
  
ENDCLASS.                    "current IMPLEMENTATION  
  
  
\*----------------------------------------------------------------------\*  
\*       CLASS notice30 IMPLEMENTATION  
\*----------------------------------------------------------------------\*  
\*  
\*----------------------------------------------------------------------\*  
CLASS notice30 IMPLEMENTATION.  
  
  METHOD withdraw.  
    DATA: zbalance TYPE f.  
  
    WRITE: / 'Notice30 Account Openning Balance before : ', balance EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
    WRITE: / 'Notice30 Account Openning ZBalance before : ', zbalance EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
  
    IF i\_within\_notice\_period = 'Y'.  
      zbalance = balance \* '0.95'.  
    ELSE.  
      zbalance = balance.  
    ENDIF.  
  
    WRITE: / 'Notice30 Account Openning Balance after : ', balance EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
    WRITE: / 'Notice30 Account Openning ZBalance after : ', zbalance EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
    IF i\_money <= zbalance.  
      balance = balance - ( i\_money + ( balance \* '0.05' ) ).  
      e\_money = i\_money.  
      IF i\_within\_notice\_period = 'Y'.  
        WRITE '   - PENALTY APPLIED'.  
      ENDIF.  
    ELSE.  
      WRITE / 'You do not have sufficient funds for a Withdrawal in your account'.  
    ENDIF.  
    WRITE: / 'Notice30 Account Closing Balance: ', balance EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
  ENDMETHOD.                    "withdraw  
  
  METHOD deposit.  
    WRITE: / 'Notice30 Account Openning Balance: ', balance EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
    balance = balance + ( i\_money \* '1.001' ).  
    e\_money = i\_money \* '1.001'.  
    WRITE: / 'Notice30 Account Closing Balance: ', balance EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
  ENDMETHOD.                    "deposit  
  
ENDCLASS.                    "notice30 IMPLEMENTATION  
  
DATA: o\_account   TYPE REF TO account,  
      account\_tab TYPE TABLE OF REF TO account,  
      holder      TYPE string,  
      amount      TYPE f.  
  
START-OF-SELECTION.  
  
  CREATE OBJECT o\_account  
    TYPE  
      current  
    EXPORTING  
      i\_account\_holder = 'Mr A'  
      i\_amount         = 1000.  
  APPEND o\_account TO account\_tab.  
  
  CREATE OBJECT o\_account  
    TYPE  
      notice30  
    EXPORTING  
      i\_account\_holder = 'Mr B'  
      i\_amount         = 2500.  
  APPEND o\_account TO account\_tab.  
  
\*  CREATE OBJECT o\_account  
\*    TYPE  
\*      current  
\*    EXPORTING  
\*      i\_account\_holder = 'Mr C'  
\*      i\_amount         = 1000.  
\*  APPEND o\_account TO account\_tab.  
\*  
\*  CREATE OBJECT o\_account  
\*    TYPE  
\*      notice30  
\*    EXPORTING  
\*      i\_account\_holder = 'Mr D'  
\*      i\_amount         = 2500.  
\*  APPEND o\_account TO account\_tab.  
  
  LOOP AT account\_tab INTO o\_account.  
    holder = o\_account->get\_account\_holder( ).  
    o\_account->deposit(  EXPORTING i\_money = 225 IMPORTING e\_money = amount ).  
    WRITE: / 'Deposit transaction for', holder, 'to the sum of ', amount EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
    SKIP.  
  
    write : / 'without penalty'.  
    o\_account->withdraw(  EXPORTING i\_money = 225  
                                    i\_within\_notice\_period = 'N'  
                          IMPORTING e\_money = amount ).  
    WRITE: / 'Withdrawal transaction for', holder, 'to the sum of ', amount EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
    SKIP.  
  
    holder = o\_account->get\_account\_holder( ).  
    o\_account->deposit(  EXPORTING i\_money = 225 IMPORTING e\_money = amount ).  
    WRITE: / 'Deposit transaction for', holder, 'to the sum of ', amount EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
    SKIP.  
  
    write : / 'with penalty'.  
    o\_account->withdraw(  EXPORTING i\_money = 225  
                                    i\_within\_notice\_period = 'Y'  
                          IMPORTING e\_money = amount ).  
    WRITE: / 'Withdrawal transaction for', holder, 'to the sum of ', amount EXPONENT 0 DECIMALS 2 LEFT-JUSTIFIED.  
    ULINE.  
  
  ENDLOOP.

In this lesson, we're going to go over an example program that I've created already based on the bank

account classes from the presentation.

So let me load up the program and then we'll step through it line by line.

So I'll post it in.

Here we go.

So what I've done is I've created some basic classes and some classes, and we'll go through them one

step at a time.

The first one, as you can see, I've created an abstract class called account.

Now, if you remember back to our lessons on abstract classes, this means I can actually instantiate

any objects from an abstract class.

It's used as a base class that we have to inherit to our subclasses to be able to create objects.

So here's the count class public section.

I've got a constructor where I'm importing the account holder and it's of time string within amount

of time float.

Then we have the withdraw abstract method with this withdrawal method, we're going to be importing

money and within notice period.

So this is going to represent 30 for a 30 day notice period account.

We're going to export another variable called money for our deposit method.

We have one important variable and one export embeddable, both a money parameter of time float and

then we have a get account holder method which will return some account holder details of time string.

Then we come down to the protected section and this is quite minimal.

We're just holding the account holder attribute there with the balance attribute.

So that's the base class.

Now, let's have a look at the current class, which inherits from our account class and as you can

see, all we're doing in the implementation, the definition is redefining the withdrawal method and

the deposit method, because if you remember the withdrawal method and deposit method abstract.

Then we move down to our notice 30 class, and again, this class is inheriting from the accounting

class and once again, we're redefining the withdraw and deposit methods.

But this one has a protected section with its own within notice period data object.

This is the data attribute, and it's a type C.

So now we come down to the implementation method and for the account, this abstract account, we're

declaring the constructor and it's some simple code just where we have the count older, less likely

to be equal, the important parameter account holder, and the balance will equal the imported amount.

When we look at the get a hold, a method that's simple as well, they are account holder is going to

equal the account holder.

So if you remember, back up on the account class.

It gets account holder method is the one that's returning the value to the calling program.

Let's continue on down.

Then we come to the implementation for the current account.

So if you remember, we're redefining the method of withdrawal and deposit, so let's have a look at

the logic here, was simply writing out a line to the screen and we're going to put down your opening

balance.

Then we have a little bit of logic saying if the imported money parameter is less than or equal to the

balance, then we need to make our balance equal to balance minus the imported monument.

So don't forget, this is a withdrawal method.

So we're taking money out of the accounts.

So it's basically saying if the money we are trying to withdraw is less than the amount we have in our

bank account, that's fine.

We can withdraw it.

And that's exactly what it does here.

And then it passes back.

The amount of money we have withdrawn.

Now, if we are trying to withdraw more money than we actually have in our bank account, in our current

account.

Then we get this message, you do not have sufficient funds for a withdrawal in your account, easy

stuff, and then we write out just a message saying closing balance is and then what the balance is.

Then for the deposit where we want to put money into our account.

Simple message to the screen and then we add the amount that we are adding into the bank account to

our existing balance.

And all we do is send back count the amount of money that we have added.

And then print out the closing balances and the natural balance of the account.

So it's pretty simple, isn't it?

Well, the withdrawal method, that'll take money out of the account if it can.

If there's enough there.

If not, it gives the error a message.

And then the deposit method allows us to put money back into our account.

So let's close these down.

And then we'll go to this 30 day notice period account.

Now, again, we're redefining the withdrawal method within this method, we're using a local data object

called Redbones as a temporary storage.

And if you remember back to the slide, we said we might have some special rules for this 30 day notice

period account.

So I've said the code to represent those rules.

So first of all, we start with this logic statement saying if the within notice period important parameter

is set to Y.

So I've tried to keep it simple, as simple as possible and know we're going to do is in the calling

program actually passing an indicator here to say whether the withdrawal is within the 30 day notice

period or not.

So if you want to look at that and just Double-Click for forward navigation.

And we can see for our account class we are importing this field.

Parameters, should I say, sort of step back.

So back to this logic, if we are withdrawing money within the notice period, that's a bad thing for

our customer because that means we're going to apply a penalty.

So what I've done here, I'm storing a temporary balance called z. balance that we've declared here.

And I'm saying this balance is going to be the actual balance of our account right now, basically minus

five percent.

That's what that calculation will do.

If the withdrawal is not within the notice period, not within 30 days, hey, that's fine, let's just

update that balance with the actual balance.

That's no problem at all.

Then we move down just right the balance to the screen.

Then we have some more logic.

If the import parameter money is less than or equal to our balance that we've just calculated.

Then we can perform these calculations, so that basically says, just like the previous current account

that we just looked at it saying if we're trying to withdraw money, there is less than our balance,

no problem, we can work it out.

So our calculation will be ballons, take away.

This little calculation to give it a final balance and then we export back to our calling program the

amount of money that we have withdrawn.

And a simple bit of logic saying if you're within the notice period, we're going to right out to our

screen our little statement that yet we've applied a penalty and we come down to this statement and

that's basically saying you don't have enough money in your account to make the withdrawal.

And then we write out the closing balance, so let me close that said.

It really doesn't matter too much what's inside, she's just, you know, pay attention to their current

account, has one way of withdrawing and depositing money and then notice 30 has a different way.

Did we look at the deposit?

I don't think we did, did we?

Let's just go through it.

OK, we're reaching out to the screen and then simply we're adding money into the account, we're just

going to work out the brand new balance.

But also this look, each up is going to get a bonus payment from making a deposit.

Now, notice it's nowhere near the penalty that he gets if he takes it out within the notice period.

But that's how this bank works.

And then we're exporting the amount of money that has gone into his account.

And so I've coded it like this just to show you that it is a different way of calculating the new balance,

which means our deposit method is different and that's what we want for this polymorphism example.

So let me collapse that cost down.

And then we come to our calling program, you can see I'm using a number of local variables in Holder

and amount and then we have zero account, which is a reference variable of the account class.

I'm also creating an internal table to actually store multiple account type objects.

And then we have the start of selection where we can actually start going through the code.

Does logic for our calling program.

So first of all, we're going to create an object of account.

The basic account note is here, I'm specifying the actual type is based on the subclass current and

I'm going to be exporting from my calling program into the importing parameters of the class, the account

name and the amount.

And then once that's done, once the object has been instantiated, I am storing the object reference

and the account tab in turn will table next.

I'm doing exactly the same thing.

I'm creating a brand new object again based on the superclass.

But specifying the type notes is 30.

So let me rephrase it.

I'm not creating the objects on the superclass and creating an object based on a reference variable

of type superclass, but making this a type notice 30 subclass.

OK, so create Object Zero Account Time Nottie 30.

And I'm exporting these parameters again so I can fill in the data.

And then a third example, creating another current account object.

And then as we step through and creating another nose's 30 account object.

Now we come down to the actual processing of these object reference variables that we have stored in

the internal table.

So I'm going to loop through the internal table and then process this code inside.

So let me just position properly.

So the first thing I'm going to do is say, look at the accounts of the internal table into zero accounts

and then I'm going to fill this variable.

If you remember, I declare that here within the calling program is just a string.

So we're going to fill this string with a method call gets account holder referring to our zero accounts

object.

But bear in mind what's actually happening here.

I created all the objects with specific subclass types.

So I'm performing a method on one of the subclasses.

I actually don't know at this point what type of subclass it is.

And this is polymorphism.

This is the code executing exactly the same command on a different object and getting a different result.

Now that gets account holder.

Actually, I think let's expand this.

Yeah, it's very basic.

So I think it's the same all the way through.

But it's this type of thing when we go down to the withdrawal method.

Exactly the same type of thing is happening when we call this method, we're calling it without knowing

what type of subclass it is, we are calling we are calling the withdrawal method.

And depending on what subclass it is, it's going to perform a different method.

So if I refer back to the slide here, let me see if I can bring it back up.

Here we go, this is this statement here, polymorphism gives us the flexibility, it allows the right

action to be carried out at the right time without the need for complex control structures to determine

the type of object it is that we're dealing with.

If you remember in the last video, I said it automatically allows us to carry out the same action on

objects of different times, resulting in a different result.

That's exactly what we have got going on in this code.

So back to the code.

So as we move down, once we've done this method, we then do the deposit method and we don't know what

object we're using, but the correct method for the correct object is going to get executed.

Same thing here as we come down here, exactly the same thing and the same thing here.

So let's see this in action in debug mode.

Let me make sure it's activated.

Here we go, I tell you what, we'll just run it through, but make sure it does work it again go.

So we're getting lots of output here.

But I think it is a lot better if we if we run it through debug, then we can see it all in action in

all its glory.

So let me put a break point here at the start and let's get going.

OK, so first of all, we'll create the first object.

And we can perform in the constructor.

And that's being passed in fine, we look at Belen's.

Supah, remember, this is a float.

So that object has been created and then it gets appended to the internal table.

And we perform the next one.

And again, that gets added to the internal table.

And we'll just step.

Or execute this one and it to the table, execute this and add it to the table, so we now have four

objects stored in the table.

Object references, should I say, we have to current account objects and we have to notice 30 accounts.

So now we come to our Loopt.

So we're going to loop at our internal table.

Into Zairo Account.

And the first thing we're going to do is call the get accountable the method.

And he's going to perform.

The method from our superclass, our account.

Because we haven't redefined this in any way.

And that returns the holder.

Then we're going to perform the deposit method, and normally we've no idea which method is going to

be executed.

Well, we don't do, don't we?

Because we know we can just look into the actual accounts and we can see this is of time current.

So it's going to perform the current deposit method.

But as we step, it goes into the deposit method.

Just scroll up.

You can see he's performing for the current account class.

It's working out the calculations and then we right out to the line and skip next, we're going to perform

a withdrawal method.

So just shoot in exactly the same code for every different object in the internal table is going to

determine which withdrawal method to execute using polymorphism.

And he knows it's the current let me step out of this.

Right to the screen.

And then.

We fill the holder again.

Perform the deposit method right to the screen.

And withdraw some more money.

And that's it.

Now, if we want to check the output of the screen, there's a little trick in this version of the debugger

done.

If you notice when you have programming, you can actually see the output in this version.

You normally have to watch a switch, should I say, to the classic debugger.

But if we go

sjp m.

S s y zero.

Percent on this call list.

That's the one, yeah, that would go.

This is not your little internal table here that holds the whole outputs that we have going to the screen.

And if we go to this column here, we can see the output.

So we can see the individual lines here represent the different lines were written out to the screen,

you can see we have an imbalance of 1000.

We deposited two hundred twenty five given as a closing balance.

Now, probably not in the right order, are they?

But not to not to worry.

They're then the next one imbalance.

Twelve, twenty five.

We took out two to five giving us a balance of 1000.

Nice and easy.

Next transaction we deposit 225 again giving us twelve twenty five.

Then we take the money away.

All simple stuff.

Why have we bothered doing it twice.

Well you'll see in the next examples.

So let me go back.

And we'll continue.

So now we're on to our second iteration through the table.

So we have a look at zero account when we step through.

It's now gone and changed, so we have a different object reference being loaded into zero account,

and if we take a look at it, we can see this is a notice to account.

So now we know choice to notice 30 account deposit and withdrawal methods that are going to get executed.

So when we're depositing two hundred twenty five and then, oh, withdrawing 225, actually our figures

are going to be different because we're actually putting in the parameter to say whether it's within

the 30 day notice period or not, which means we will have interest taken away or interest added, but

then penalty fees taken away from our accounts.

So let's see that go on so that the deposit I'll jump into when we got to the deposit method.

And if you screw up, you can see when we get there where in the notice 30 account now, which means

we're going to armed one percent on top of the money we're adding in.

Very nice.

And we'll write that out to the screen.

If we go and have a look at the output.

Here we go now you see there's a difference.

So we've deposited 225 and got this 22 pence extra.

Go into the next.

This is a withdrawal and let's have a look, we are saying it's not in the notice period, so it's going

to withdraw without applying a penalty.

Go have a look at the output.

Here we go.

So you know what, I don't think we've finished at your Kotobuki let me go back here.

Yeah, we haven't written out the final bit.

Now we can.

Go back and see the output.

Yeah, we go.

So we took out 225 Levinas now that high a balance of 23, 63, 96.

And so the story goes so I can just execute these, the main thing to take away, as I've already mentioned,

we don't know really what object methods are going to get executed by our culling program, but we are

making use of polymorphism here through inheritance to make it.

So we're performing the exact same method, but we're getting a different outcome.

Let's say we're calling the same method, but it's executing a different method based on the actual

object that we have in our reference variable, so we can just zoom through the rest of this code.

Let me do the fakie, see the output, and it goes through the process of calling the withdrawal method

and deposit method many times applying the penalties, adding interest and so on for the various objects

that we have stored in the internal table.

I think there was four, wasn't there?

So that said, that's an example of polymorphism within our program that uses inheritance.

I really strongly suggest you play around with this program, make changes to it yourself and see if

you can maybe add some more methods and add some more code to create more objects in the calling program.

So you thoroughly understand exactly what polymorphism means in the context of our other programs and

how you can make use of it.

Let's move on.