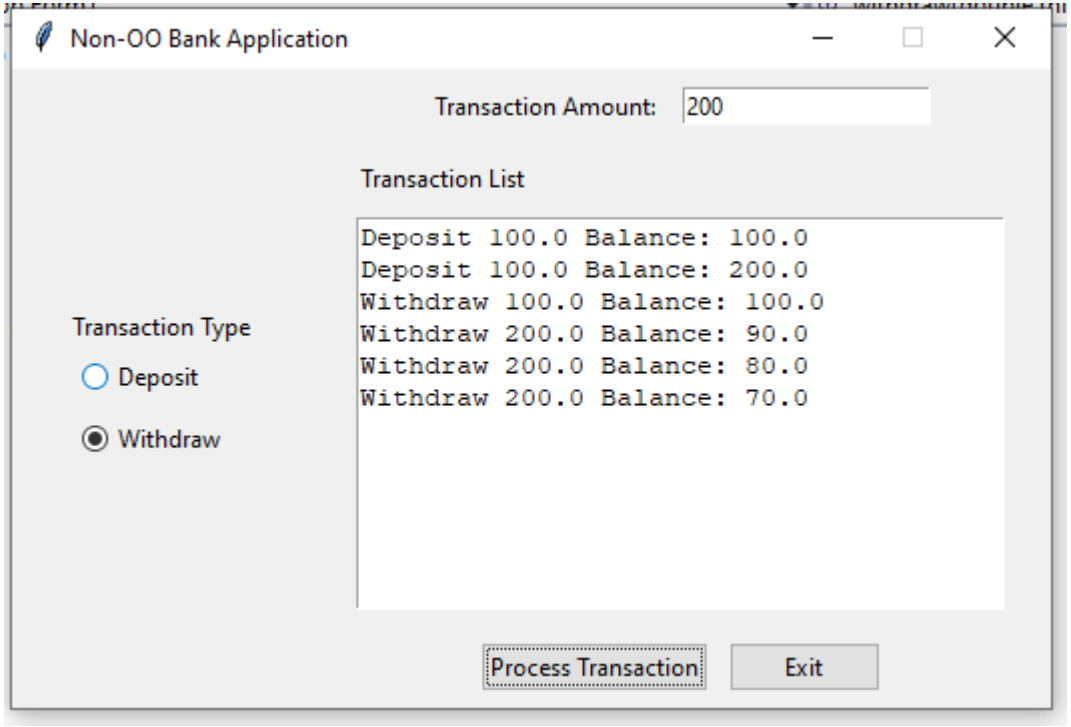
COMP642

Object Oriented Programming

Lectorial 4

Some Non-OO Code




```
balance = 0.00
transString = ""
fee = 10.00

def withdraw(anAmt):
    global balance
    global transString

    transString = "Withdraw"
    if (anAmt <= balance):
        balance = balance - anAmt
        return 1
    else:
        balance = balance - fee
        return 0

def deposit(anAmt):
    global transString
    global balance

    transString = "Deposit"
    balance = balance + anAmt
```

Some Problems

- What if different accounts have different fees?
- The balance can be changed from anywhere and at anytime (to anything!).

An Account Class

- Will this be better?

```
class BankAccount:
    def __init__(self, balance, fee):
        self.balance = balance
        self.fee = fee

    def deposit(self, amt):
        self.balance = self.balance + amt

    def withdraw(self, amt):
        if (amt <= self.balance):
            self.balance = self.balance - amt
            return 1
        else:
            self.balance = self.balance - self.fee
            return 0
```

Think??

Will this be better?

What if we want to provide overdraft facility?

How should we set the initial values of the Overdraft and Balance?

What if we wanted to prevent the overdraft ever being more than $1000 ?

Benefits of Encapsulation in the Class

- The **withdrawal** procedure can deal with all the logic of overdrafts and so on without the calling program needing to know about them.
- If the **deposit** or **withdrawal** procedure needs to be changed (e.g. a fee is charged for each transaction), then this change is made once in the class and the calling program needs no changes (the logic is separate from the interface).
- Each Account object can have its own values of fee, OD etc.

Keeping Logic and Interface Separate

- With the account class containing all the logic of withdrawals and overdrafts, etc. we can now use any interface to manage an account without having to repeat the logic.

  i.e. we can read transaction info from a file OR from text boxes on a form OR off a web page OR . . .
- It is important to make sure that the interface stays separate from the class so we can use them independently. This idea is behind the pattern Model-View-Controller. Each part should be separate so they can be interchanged.

Model-View-Controller

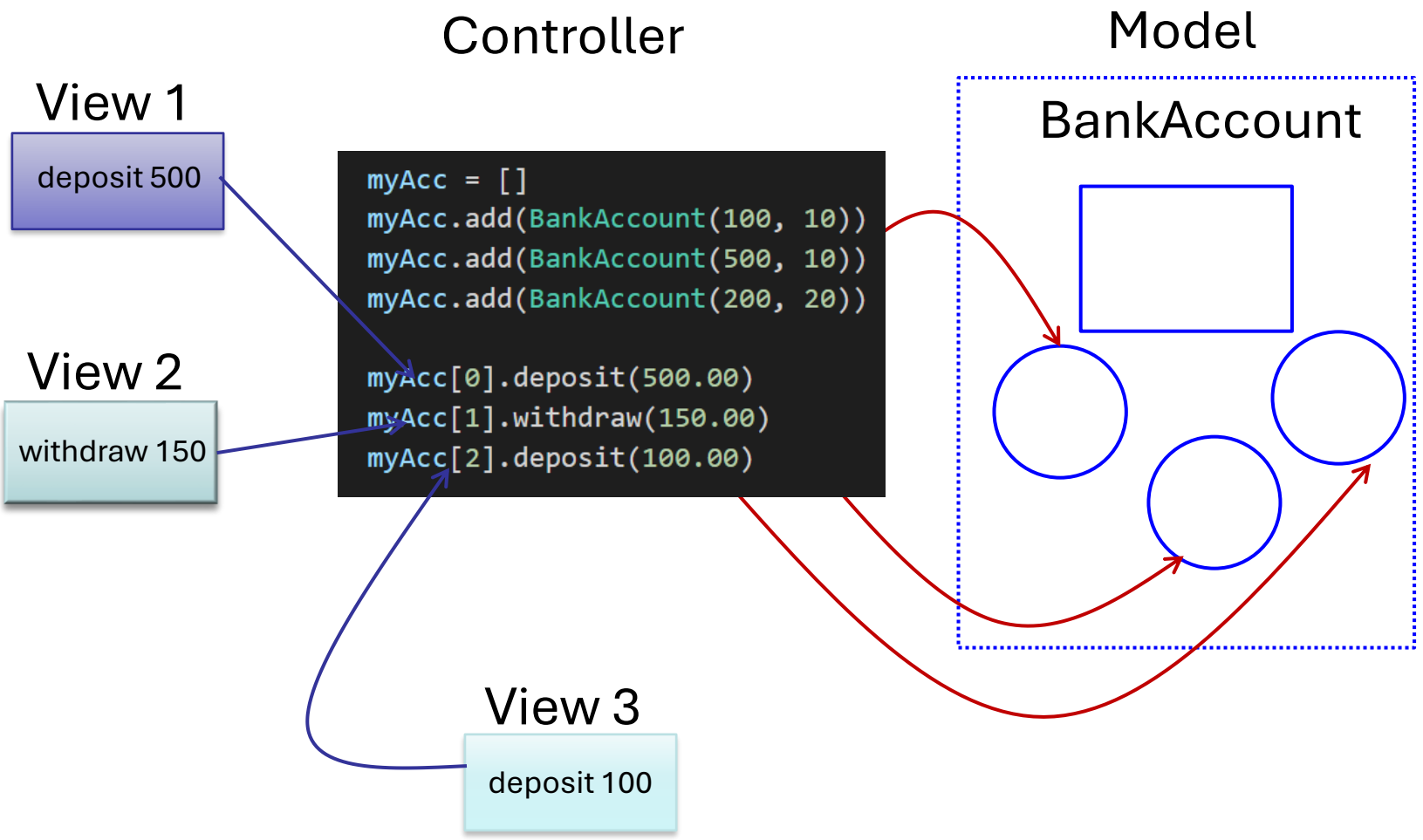
Model: contains the logic, i.e. our Account class

Controller: the code managing the class, i.e. telling what deposits and withdrawals to make (our form in this case)

View: a way of seeing the class properties, e.g. a statement or form showing transactions.

- If these are kept separate, we can mix and match them without having to alter any code. e.g.
  - read transactions from a file and display results on a screen
  - enter transactions through form and print results on a statement

Model-View-Controller



Don't Let Logic Class Talk to User (1)

```
def withdraw(self, amt):
    if (amt <= self.balance):
        self.balance = self.balance - amt
        return 1
    else:
        self.balance = self.balance - self.fee
        showinfo(title='Info', message="Withdraw Failed, Fee applied")
        return 0
```

Don’t let the BankAccount (model) class talk to the user.

If you do, then this class can be used only with an interface (view) with a screen that can display a message box!

Don't Let Logic Class Talk to User (2)

- The class should not ‘talk back’ to the user.
- The calling program (controller) should ask the class whether there was a problem and then deal with it appropriately.
  - If using a form, then the calling program might put up a message box.
  - If reading data from a file, then the calling program might write errors to a log file.

A Possible Solution

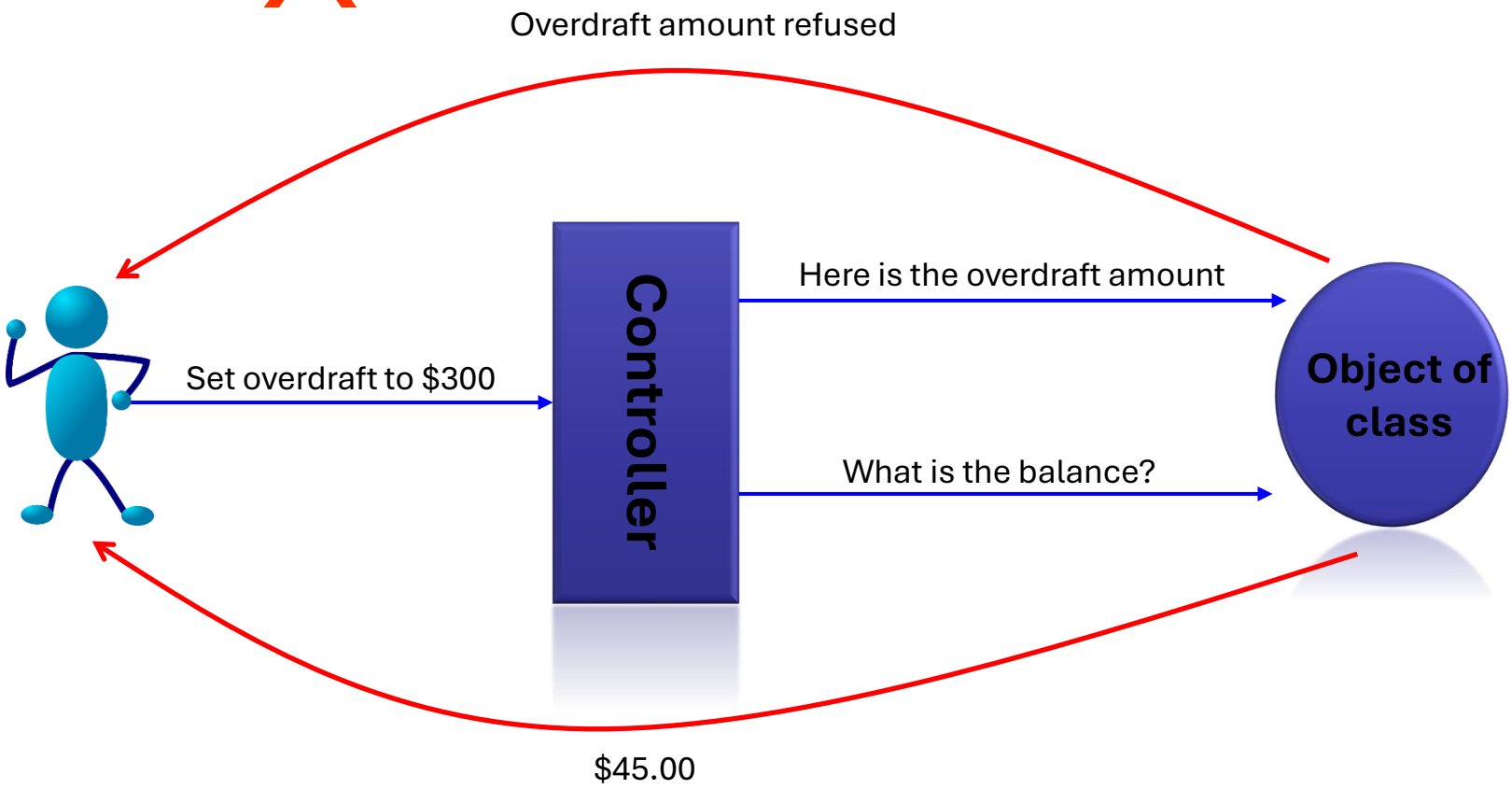
```
if (selected_size.get()=="W"):
    status = anAcct.withdraw(amount)
    if status == 1:
        showinfo(title='Info', message="Withdraw Successful")
    else:
        showinfo(title='Info', message="Withdraw Failed, Fee Applied")
    msgString = "Withdraw " + str(amount) + " Balance: " + str(anAcct.balance)
```

Controlling program can ask if transaction was OK

and if not – calling program can ask for the message which it can then display however it likes

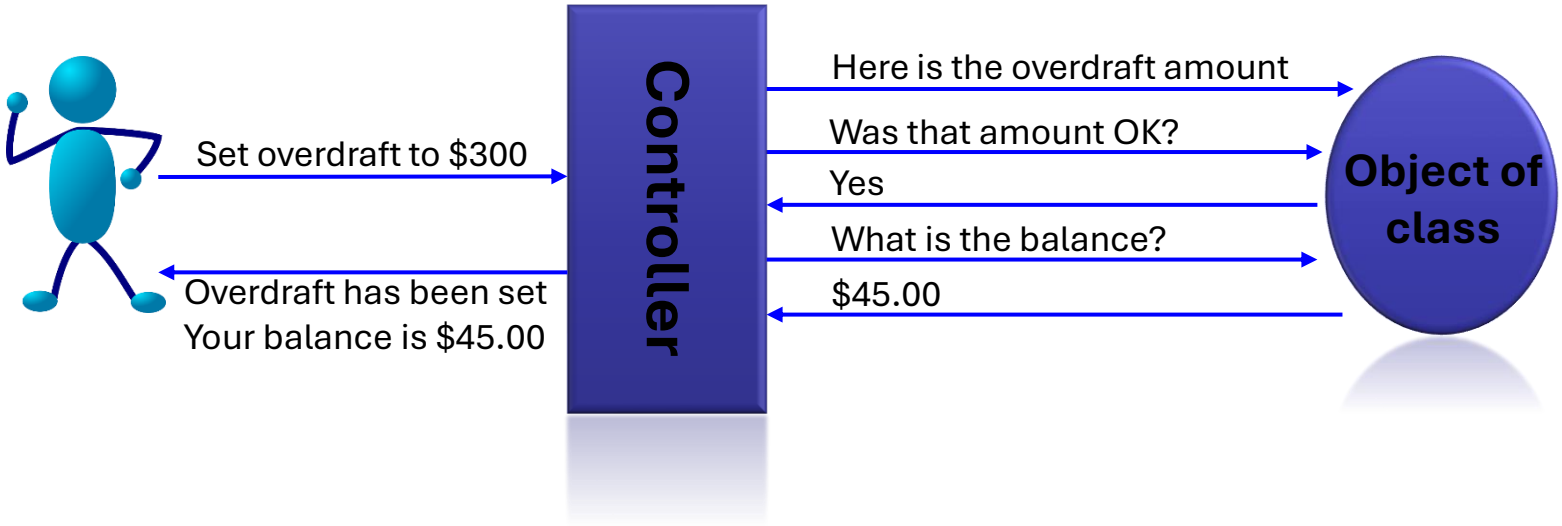
Communicating with the Class (1)

Don't do this X

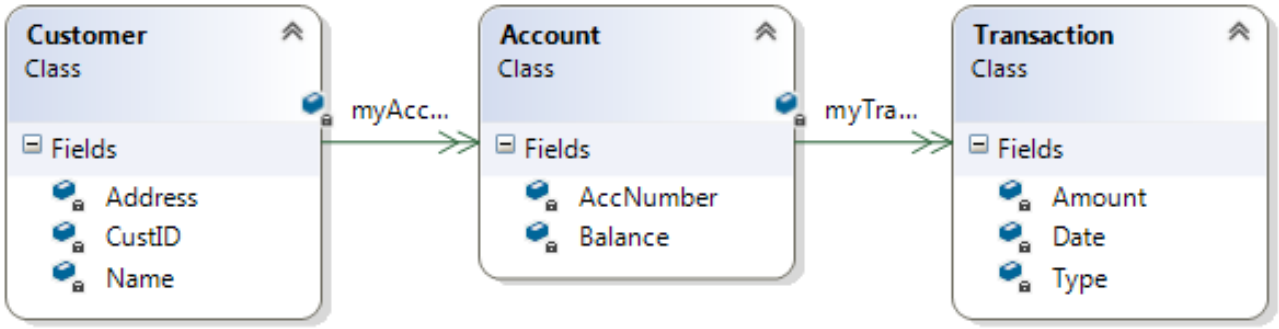


Communicating with the Class (2)

Do this ✓



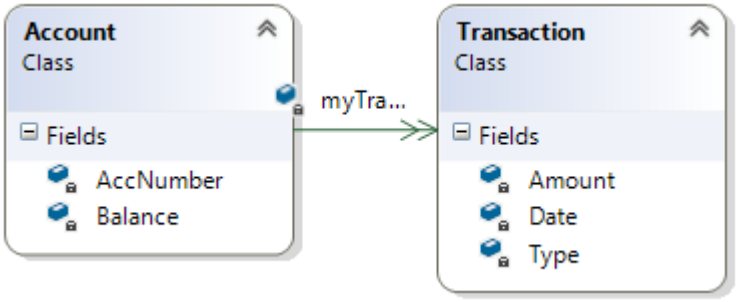
Lots of Accounts and Customers



You can read this class diagram like this:

- A customer can have many accounts (or none)
- Each account is for just one customer (no joint ones ☺)
- Each account can have many transactions (or none)
- A transaction belongs to just one account

Designing Classes to Represent Diagram



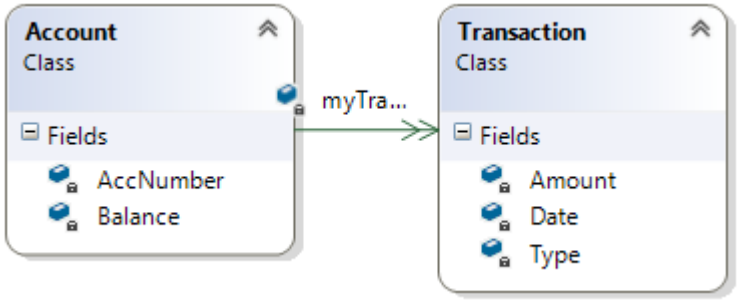
*Where there is a **MANY** end to a relationship consider a **List** to hold the references*

An **Account** class will have:

- Properties AccNumber and Balance
- It will have a List to hold references to all its transaction objects (because we want to be able to *navigate* from the account to find all transactions)

```
class Account:
    def __init__(self, accNum, bal):
        self.AccNumber = accNum
        self.Balance = bal
        self.myTrans = []
```

Designing Classes (continued)



*Where there is a **1** end to a relationship consider a **reference** to the object*

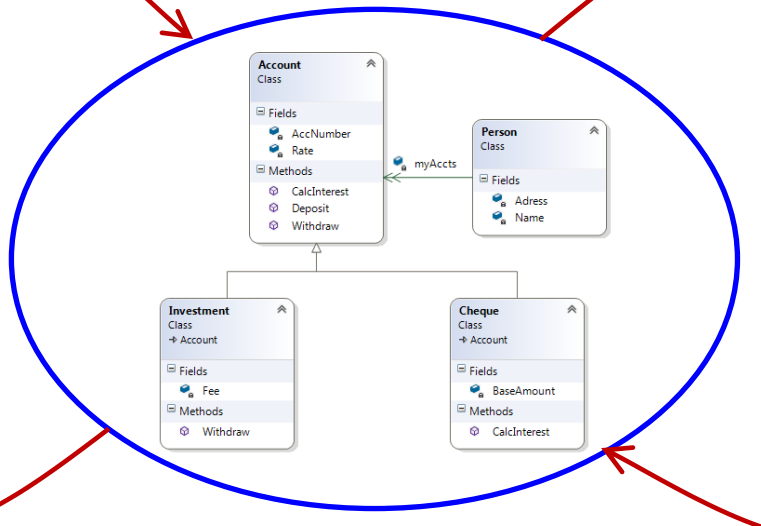
A **Transaction** class will have:

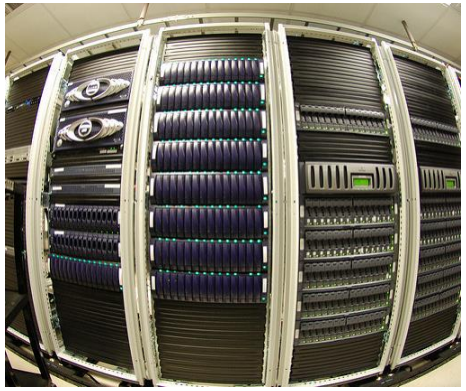
- Date, Type and Amount properties
- It could have a reference to the account object

  (But need this only if we want to *navigate* in that direction*)*

```
class Transaction:
    def __init__(self, adate, amt, type, account):
        self.Amount = amt
        self.Date = adate
        self.Type = account
```

Overview of an Application





User interface

- Forms
- Web pages
- Phones

Program

- Classes
- Methods

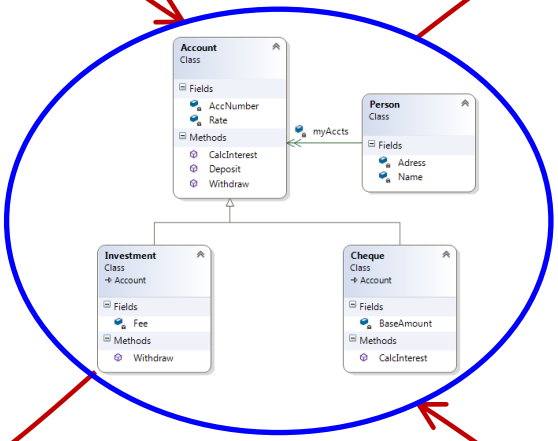
Storage

- Database
- Text file
- Spreadsheet

Business Rules

- Any application is likely to have checks and validations that need to be carried out. These are often referred to as “business rules”.
- If we had an application keeping track of students enrolling in courses at Lincoln what would some of the business rules be?

Where to Apply Business Rules



List boxes
Masks
Greying or hiding controls

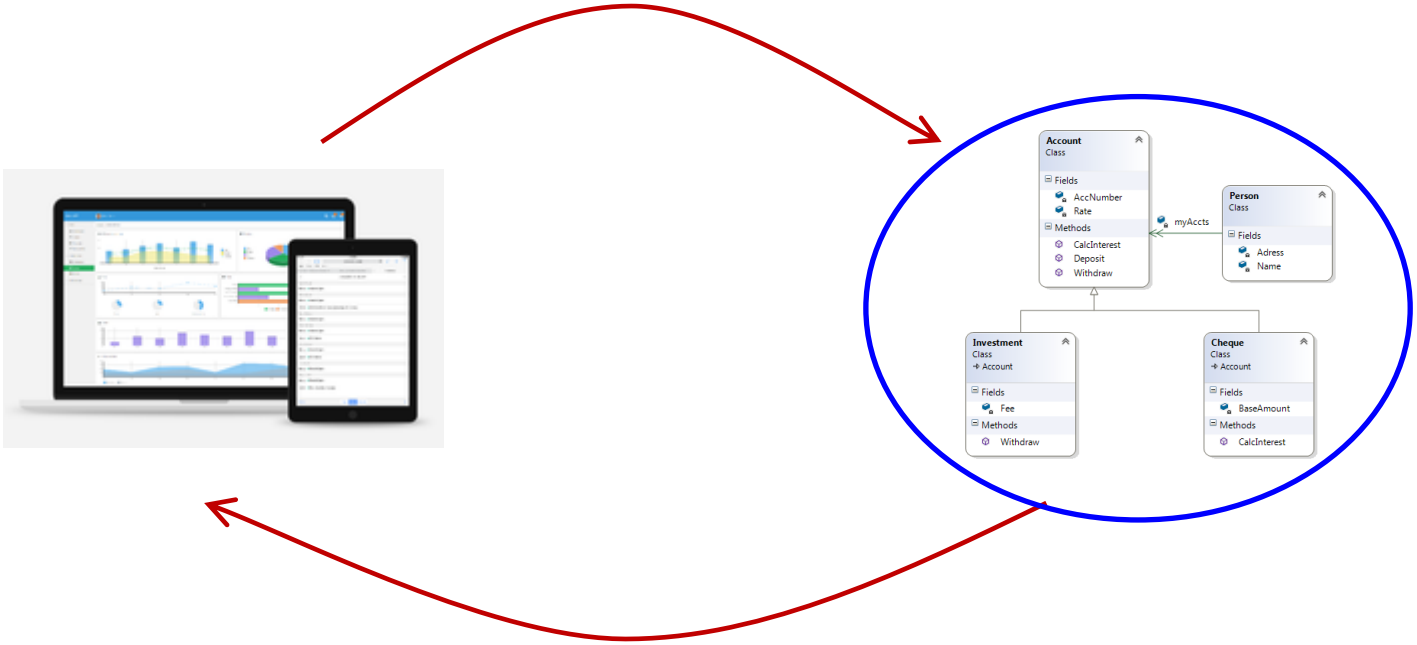
Sets and Gets
Program logic (If..)
Exceptions

Table validation
Integrity rules
Triggers

Often apps will have a mixture of all these

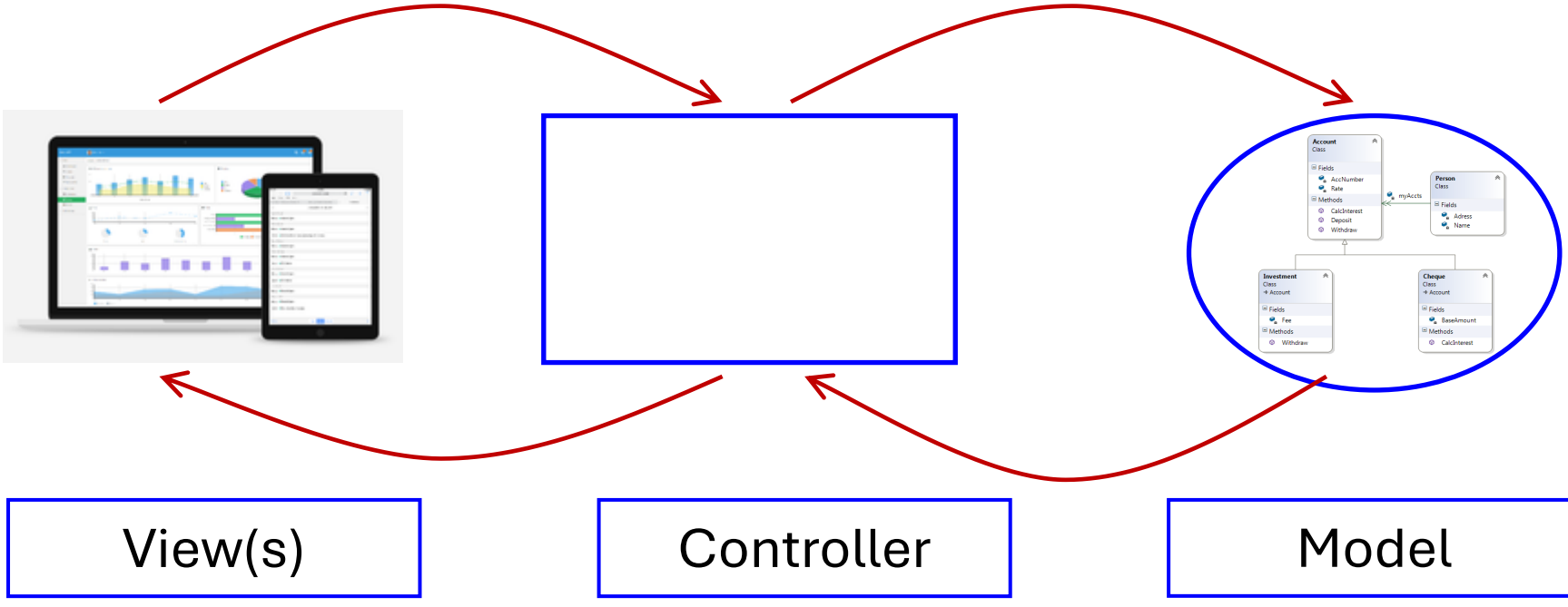
We Won’t Consider Storage End

- Applying logic and rules at the storage stage of an application is covered more fully in the advanced database course.



Managing Interaction Between User and Program

- The Model View Controller pattern (MVC) helps to keep the logic and interface of an application separate.
- That way the interface can change without upsetting the logic.



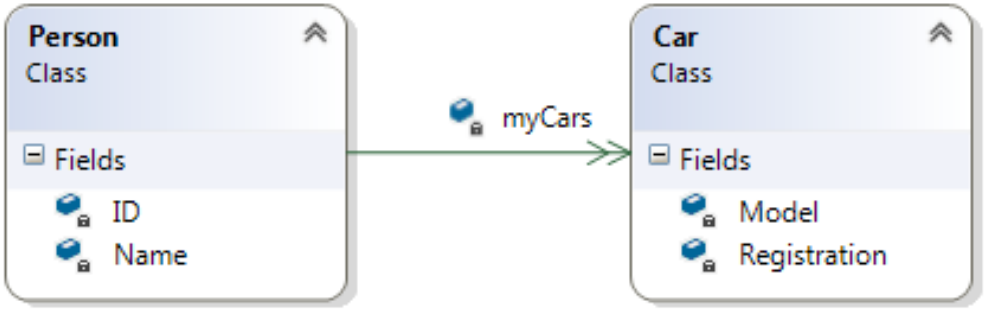
Model

- The **model** is the collection of classes in your project that keep the data about the problem domain.
- E.g.

  We might be keeping info about cars and their owners.

  We can set up classes to do that.

  These are referred to as the model classes.



Model Classes

- The model classes just maintain the data.
- They should not know where the data comes from or what happens to information it supplies.

```
class Person:
    nextID = 100
    def __init__(self, pname):
        self.__personID = Person.nextID
        self.__personName = pname
        self.__personCars = []
        Person.nextID += 1

    def addCars(self, aCar):
        self.__personCars.append(aCar)

    def removeCar(self, aCar):
        self.__personCars.remove(aCar)

    def numCars(self):
        return len(self.__personCars)
```

Doesn't know where pname comes from

Doesn't know where aCar comes from

Doesn't know who is asking about number of cars or what happens to this info

Controller

- Something has to make the car and person objects and look after them. The class responsible for this is often referred to as a **controller**.
- The controller is separate from the interface (form etc).

```
class CarController:
    def __init__(self):
        self.allCars = []
        self.allPeople = []

    def newCar(self, creg):
        aCar = Car(creg)
        self.allCars.append(aCar)

    def newPerson(self, pname):
        aPerson = Person(pname)
        self.allPeople.append(aPerson)

    def findCar(self, creg):
        for car in self.allCars:
            if car.CarReg == creg:
                return car
        return None
```

Keeps objects

Creates objects (but doesn't know how reg num is provided)

Can retrieve and manipulate objects

Who Keeps List and Owner in Sync?

- When ownership changes

  Car must get new owner

  Person must add car to list

  Car must be removed from the other list

- Could do both in `aPerson.AddCar(aCar)`

```
mycars.add(aCar);
aCar.owner = aPerson.PersonName;
```

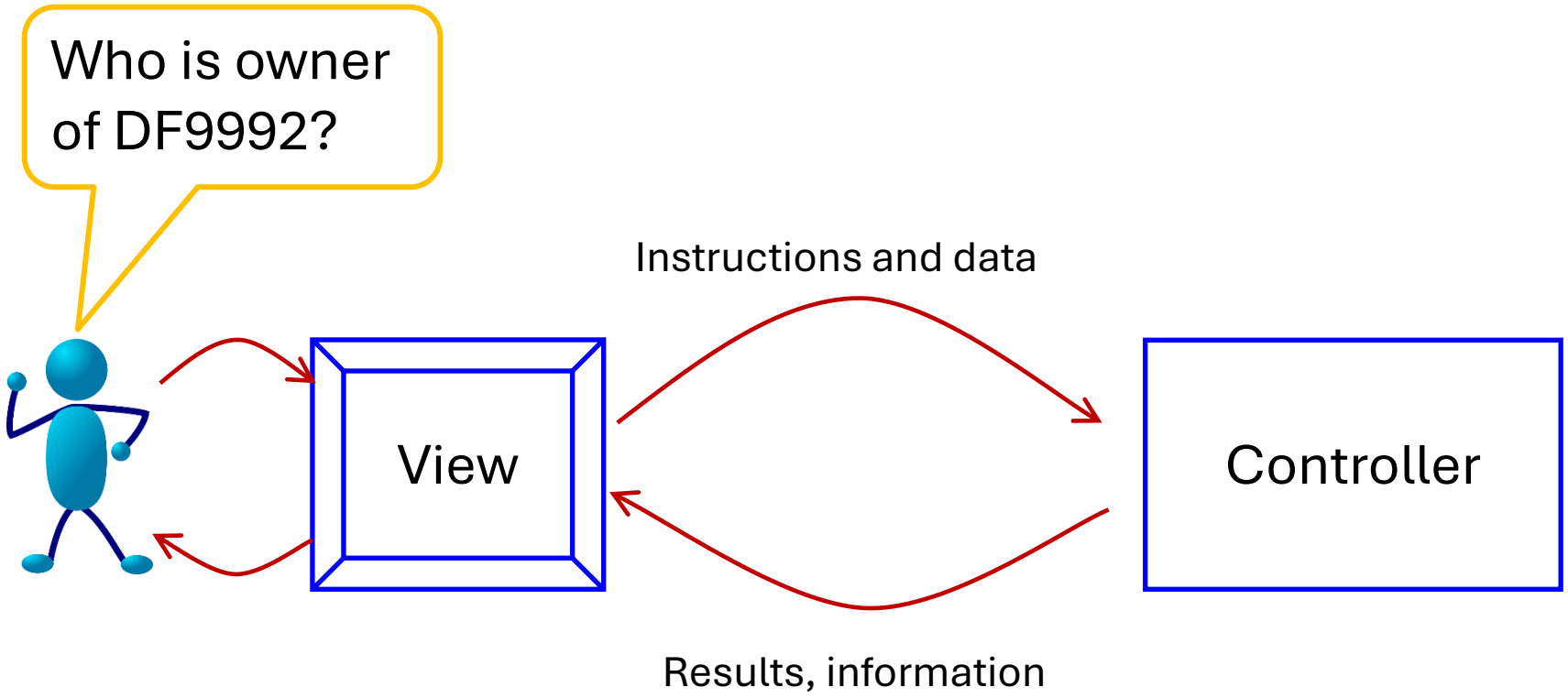
- Controller could do both

```
aPerson.AddCar(aCar);
aCar.owner = aPerson.PersonName;
```

Which?

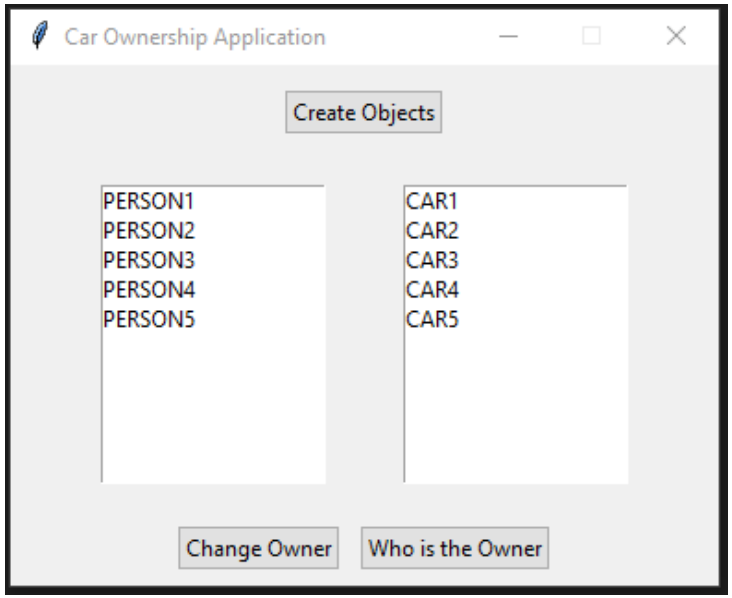
Controller and view

- The **View** classes (e.g. form) get data and instructions from user.
- View passes data to controller.



View

- The view classes interact with the user.
- They could be forms, web pages, classes that read and write to files, or a combination of all these.
- They are likely to change more often than the model and controller.



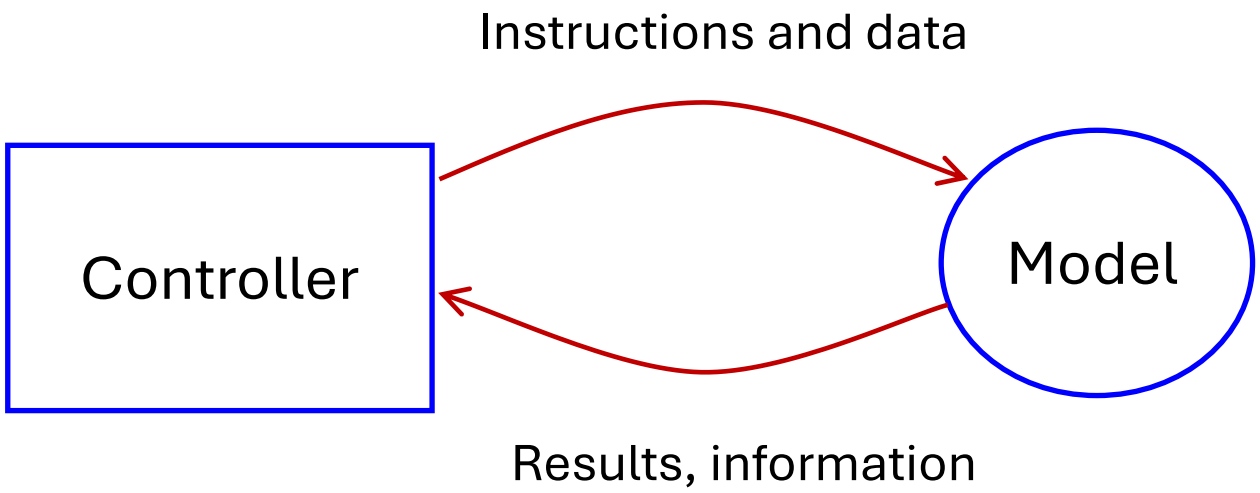

```
def btnChangeOwner():
    #get selected person
    selPersonIndex = lstbox_ppl.curselection()
    selectedPerson = lstbox_ppl.get(selPersonIndex)

    #get selected car
    selCarIndex = lstbox_car.curselection()
    selectedCar = lstbox_car.get(selCarIndex)

    #ask company to do the rest
    company.changeOwner(str(selectedPerson), str(selectedCar))
```

View obtains data from user

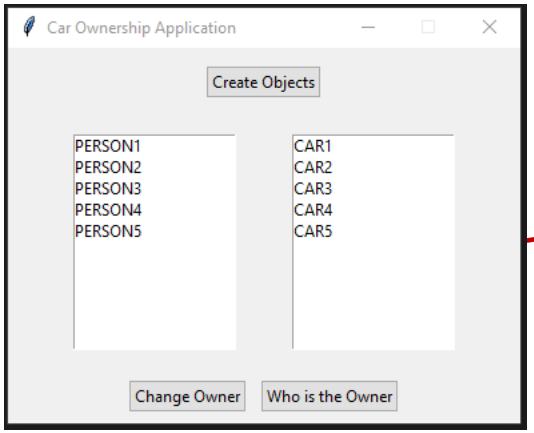
Asks Controller to do the rest

Controller and Model

- The controller sends information to the model.
- The controller class creates and looks after the model objects and provides instructions.



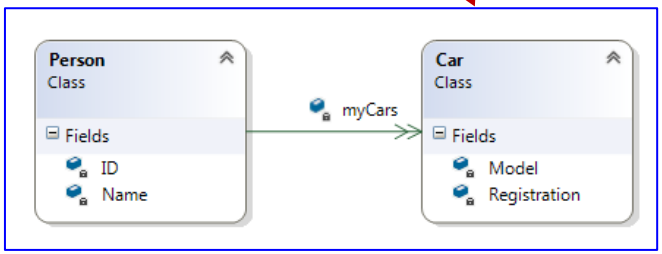
Who Knows What?



View knows about Controller Class

```
class CarController:
    def __init__(self):
        self.allCars = []
        self.allPeople = []

    def newCar(self, creg):
        aCar = Car(creg)
        self.allCars.append(aCar)
```

Controller knows about model classes



But not the other way. We can change (e.g.) the view classes without upsetting anything else.