COMP 1531
Software Engineering Fundamentals

Week 03

Domain Modelling using Object Oriented Design Techniques

Domain model

- Also referred to as a conceptual model or domain object model

- Provides a visual representation of the problem domain, through decomposing the domain into key concepts or objects in the real-world and identifying the relationships between these objects
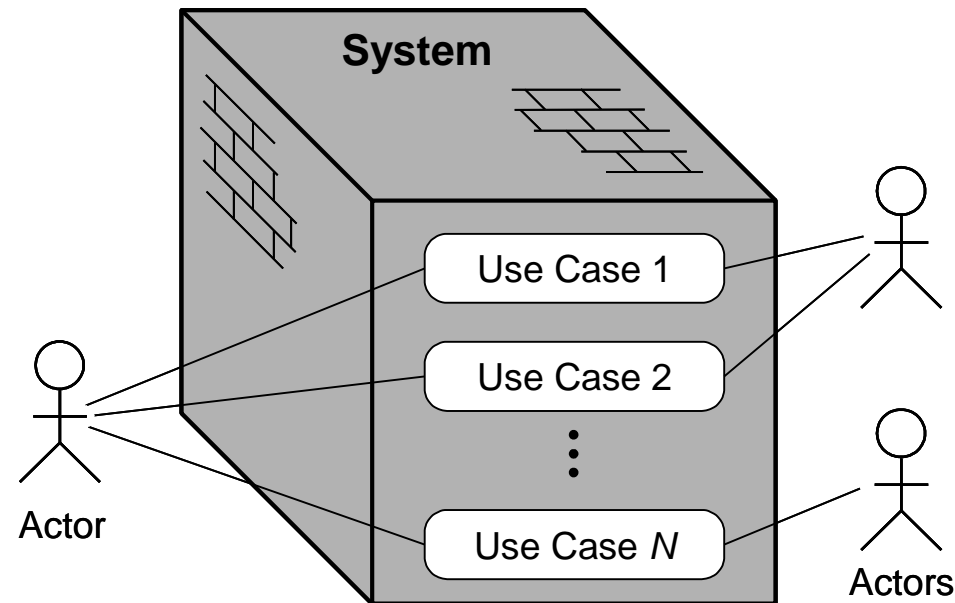
Requirements Analysis vs Domain modelling

- Requirements analysis determines " external behaviour " "What are the features of the system-to-be and who requires these features (actors) "

- Domain modelling determines "internal behavior" - "how elements of system-to-be interact to produce the external behaviour"

- Requirements analysis and domain modelling are mutually dependent - domain modelling supports clarification of requirements, whereas requirements help building up the model.
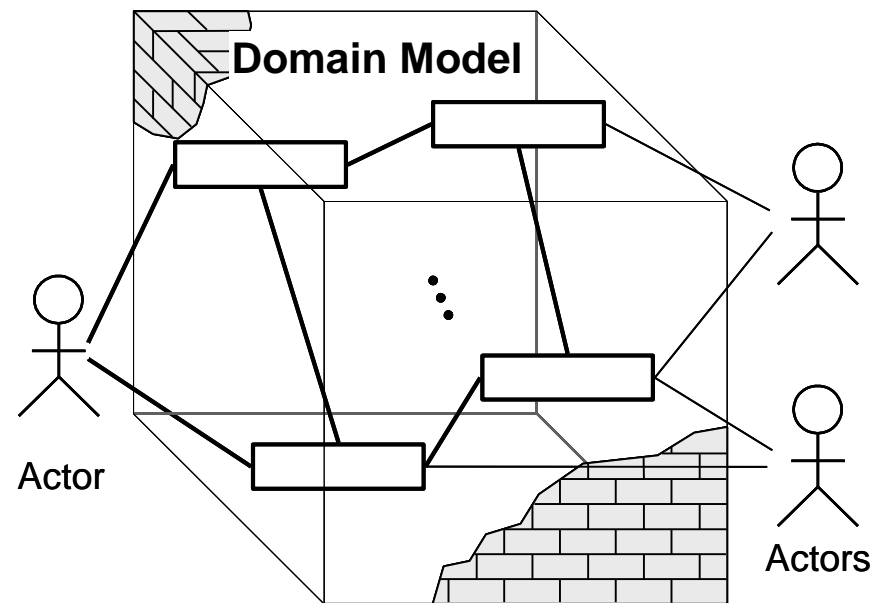
Use Cases vs. Domain Model

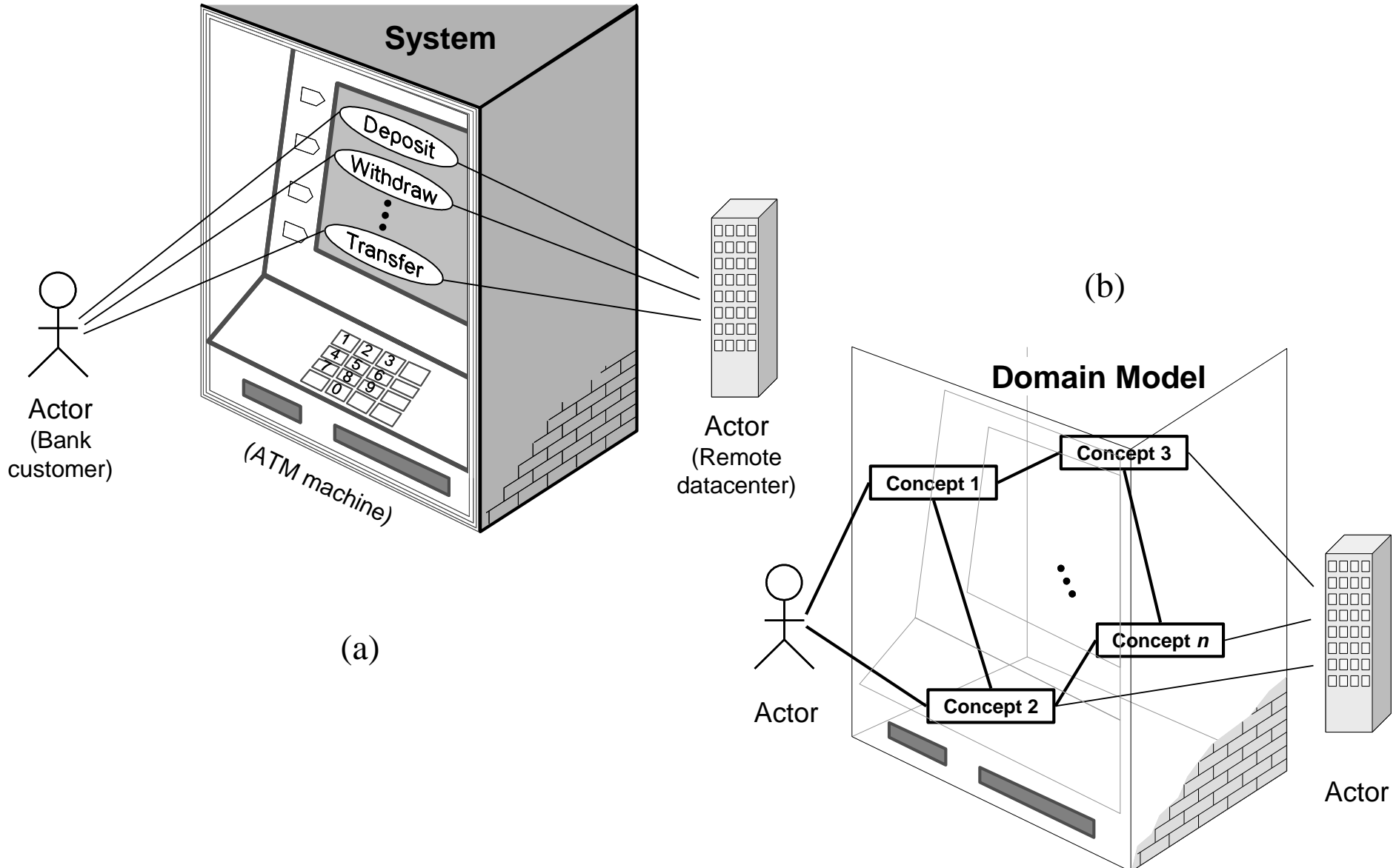In **use case analysis**, we consider the system as a "**black box**"

In **domain analysis**, we consider the system as a "**transparent box**"

(a)

Example: ATM Machine

System

Deposit

Withdraw

Transfer

Actor
(Bank customer)

(ATM machine)

Actor
(Remote datacenter)

(a)

(b)

Domain Model

Concept 3

Concept 1

Concept *n*

Concept 2

Actor

Actor

Benefits of a Domain model

- Triggers high-level discussions about what is central to the problem (the core domain) and relationships between sub-parts (sub-domains)

- Ensures that the system-to-be reflects a deep, shared understanding of the problem domain as the objects in the domain model will represent domain concepts

- Importantly, the common language resulting from the domain model, fosters unambiguous shared understanding of the problem domain and requirements among business visionaries, domain experts and developers

6

How do we create a domain model?

- One widely adopted technique is based on the **object-oriented design** paradigm

But, before we discuss OO, a brief introduction to UML

What is UML?

UML stands for **Unified Modelling Language** (http://www.uml.org/)

Programming languages not abstract enough for OO design

An open source, graphical language to model software solutions, application structures, system behaviour and business processes

Several uses:

- As a design that communicates aspects of your system

- As a software blue print

- Sometimes, used for auto code-generation

UML diagram categories

- **Structure Diagrams**
  – show the static structure of the system and its parts and how these parts relate to each other
  – they are said to be static as the elements are depicted irrespective of time (e.g. class diagram)

- **Behaviour Diagrams**
  – show the dynamic behaviour of the objects in the system i.e. a series of changes to the system over a period of time (e.g. use case diagram or sequence diagram)
  – a subset of these diagrams are referred to as interaction diagrams that emphasis interaction between objects (e.g., an activity diagram)

UML Diagram Types
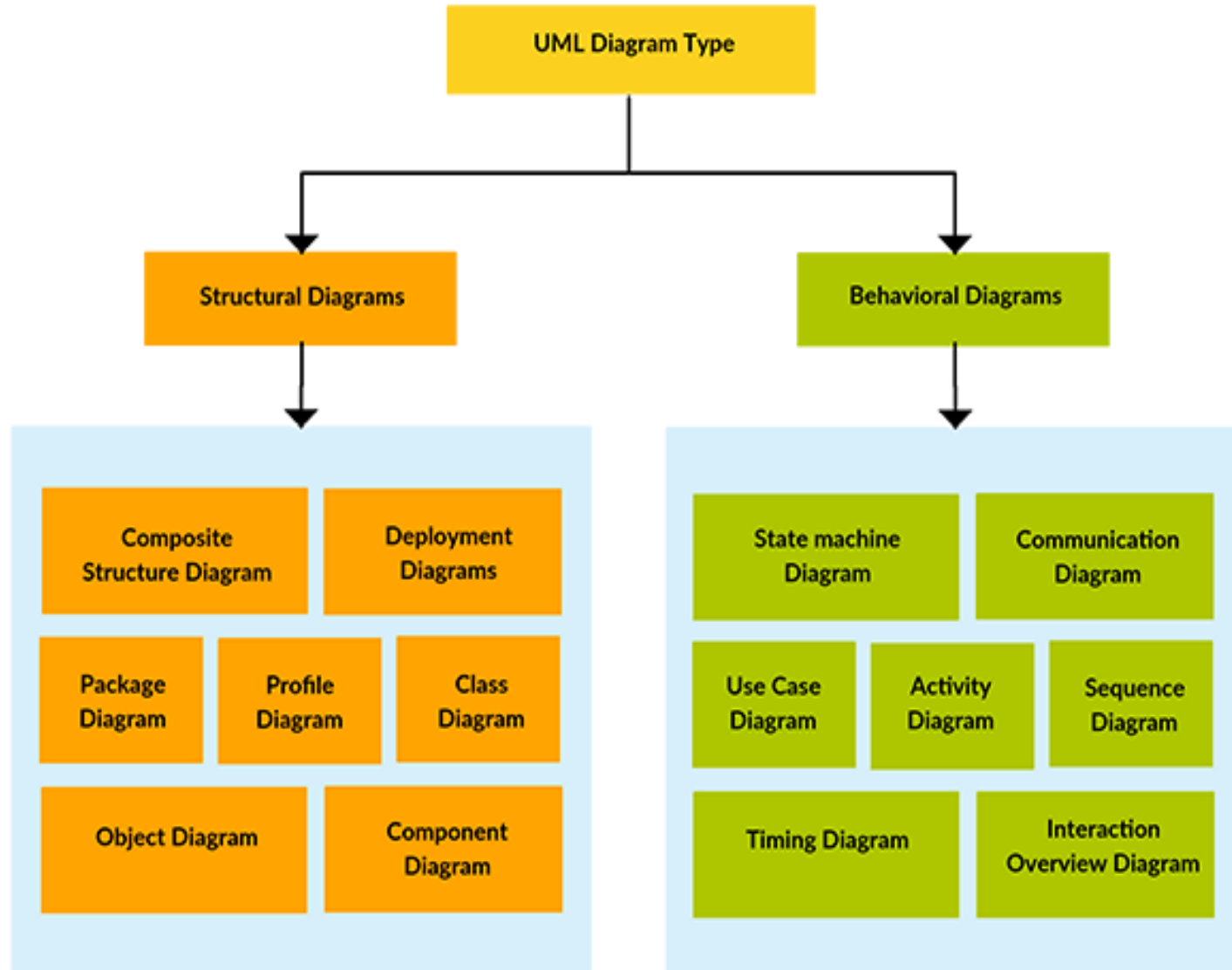
Domain Modelling using OO
Design Paradigm

Object Oriented Design

- ***Objects*** are real-world entities and could be

  - Something tangible and visible e.g., your car, phone, apple or pet.

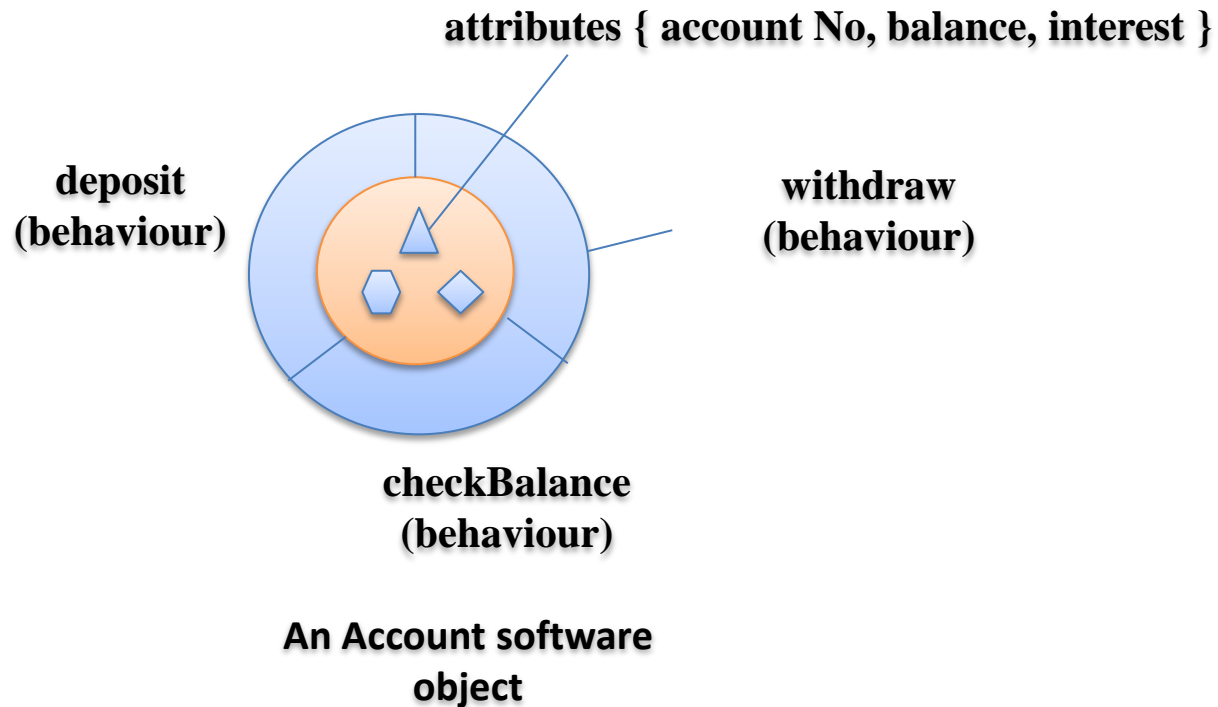 - Something intangible (you can't touch) e.g., account, time

- Every object has:

  - *attributes:* properties of the object e.g., model number, colour, registration of a car or colour, age, breed of a dog

  - *behaviour* – what the object can do (or methods) e.g., a duck can *fly*, a dog can *bark,* you can *withdraw* or *deposit* into an account

- Each object encapsulates some *state* (the *currently assigned values* for its attributes) ; gives the object its identity *(as state of one object is independent of another)*
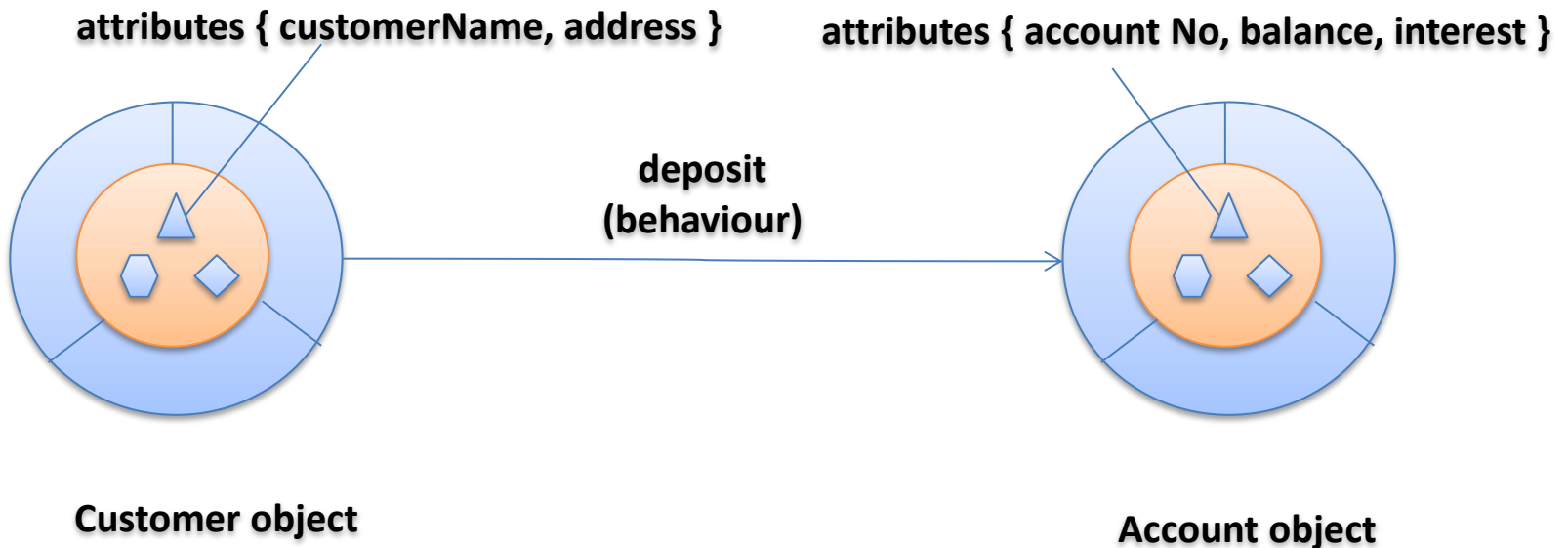
Object Oriented Design

- Identify your domain
- Identify objects

attributes { account No, balance, interest }

deposit (behaviour)

withdraw (behaviour)

checkBalance (behaviour)

An Account software object

13

Object collaboration

- Objects interact and communicate by sending *messages* to each other

- Objects play a *client* and *server* role and could be located in the same memory space or on different computers

- If *object A* wants to invoke a specific behaviour on *object B*, it sends a message to *B* requesting that behaviour

attributes { customerName, address }   **attributes { account No, balance, interest }**

**deposit
(behaviour)**

Customer object   **Account object**

Object Collaboration
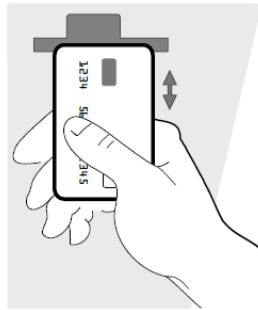
This message is typically made of three parts:

- The *object* to whom the message is addressed (e.g., John's "Account" object)

- The *method* you want to invoke on the object(e.g., deposit())

- Any *additional information* needed (e.g., amount to be deposited)

deposit(400)

Customer A   **Account**
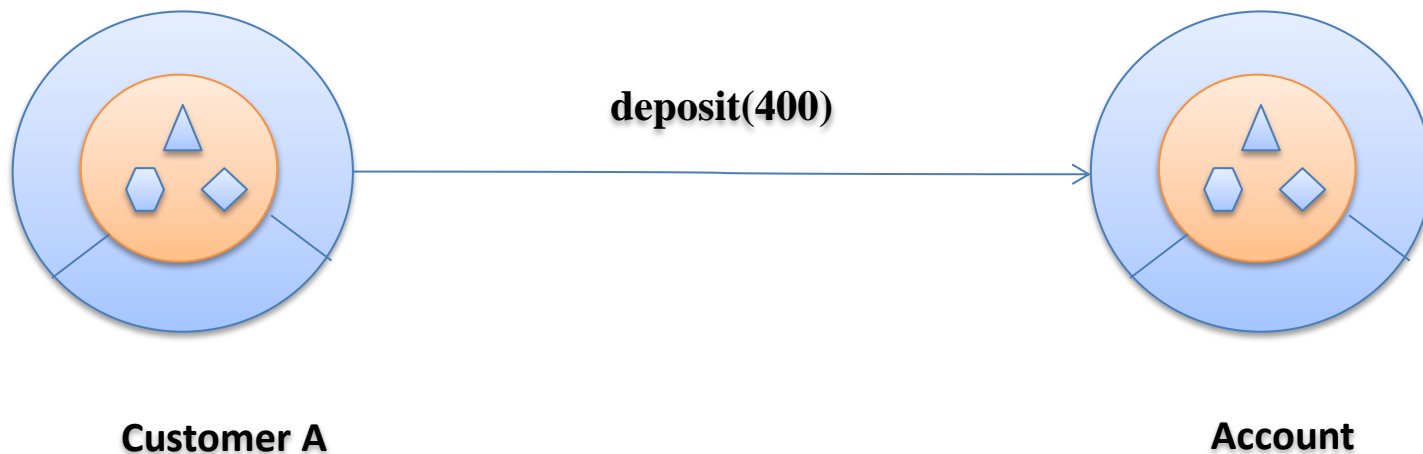
Object's Interface

Object:
ATM machine

method-1:
Accept card

method-2:
Read code

method-3:
Take selection

Interface

attributes

method-1
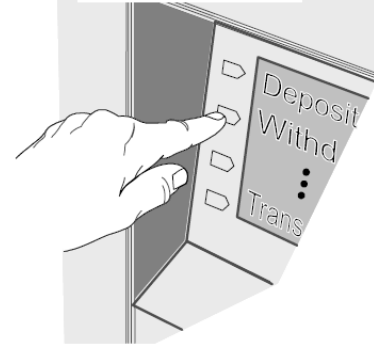
method-2
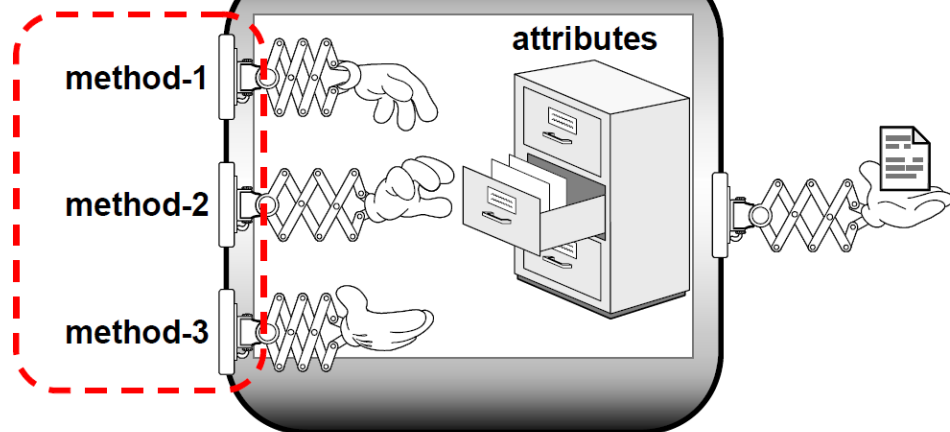
method-3

- An object's *interface* is the set of the object's methods that can be invoked on the object
- The interface is the fundamental means of communication between objects

Objects and Classes

- Many objects are of the same "**kind**" but have different identity e.g., *there are many Account objects belonging to different customers, but they all share the same attributes and methods*

- "Logically group" objects that share some common properties and behavior into a **class**

- A class serves as a *blue-print* defining the attributes and methods (behaviour) of this logical group of objects

John's account
{ accountNo = 123, balance = 100, interest=5% }

Tom's account
{ accountNo = 567, balance = 600, interest=5.2% }
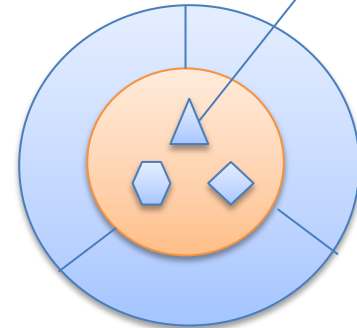
Objects and classes

– Defining a class, does not actually create an object

- An object is instantiated from a class and the object is said to be an instance of the class

  – An object instance is a specific realization of the class

- Two object instances from the same class share the same attributes and methods, but have their own object identity and are independent of each other

  – An object has state but a class doesn't

  – Two object instances from the same class share the same attributes and methods, but have their own object identity and are independent of each other
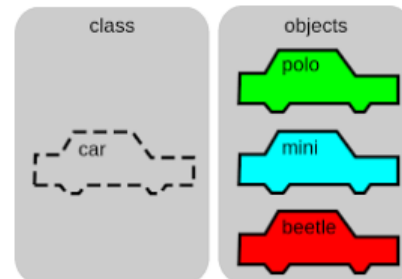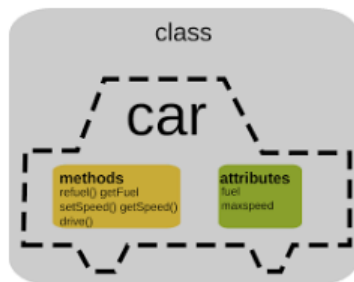
Objects and Classes

- We tend to "logically group" objects that share some common properties and behaviour. This logical group is called a **class**

- A class serves as a *blue-print* defining the attributes and methods (behaviour) of this logical group of objects

- Two object instances from the same class share the same attributes and methods, but have their own object identity and are independent of each other
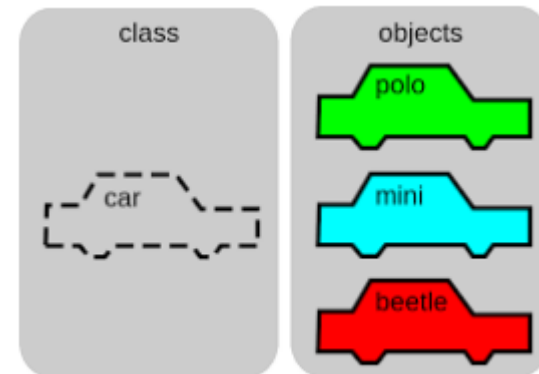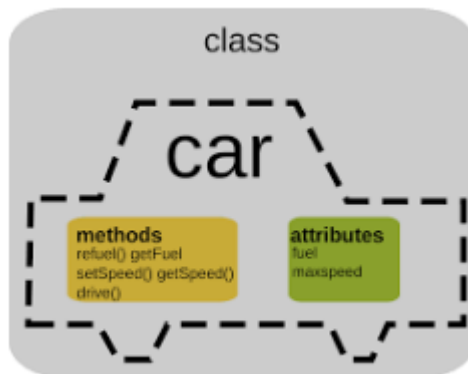
Representing classes in UML

- A class is sometimes referred to as an **object's type**.
- An object is instantiated from a class and the object is said to be an instance of the class)
- An object has state but a class doesn't

class (class diagram)

Account
-name: String -balance: float
+getBalance(): float +getName() : String +withDraw(float) +deposit(float)
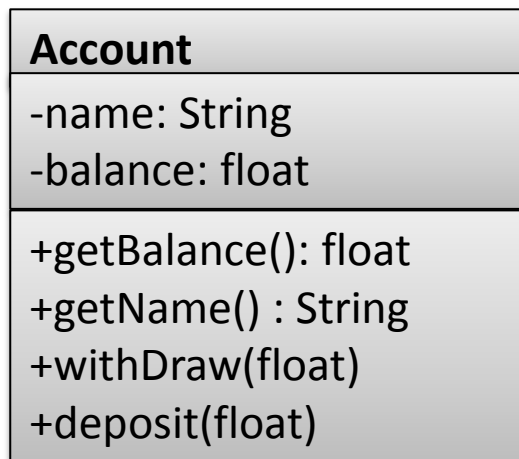
object instances (object diagram)

a1:Account
name = "John Smith" balance = 40000

a2:Account
name = "Joe Bloggs" balance = 50000
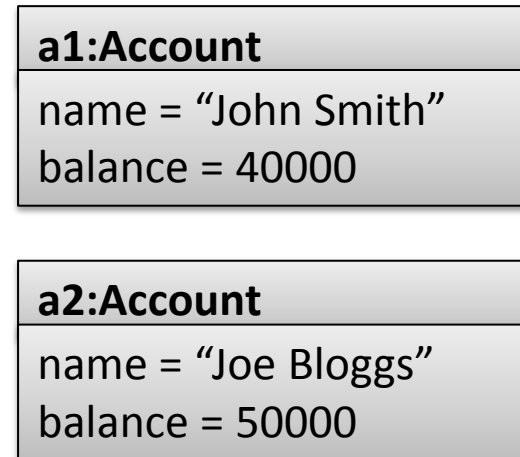
Two key principles of OO design

- Abstraction
- Encapsulation

Abstraction

- Helps you to focus on the common properties and behaviours of objects

- Good abstraction help us to accurately represent the knowledge we gather about the problem domain (discard anything unimportant or irrelevant)

- What comes to your mind when we think of a "car" ?

  Do you create a class for each brand (BMW, Audi, Chevrolet…) ?

  - write *one* class called Car

  and *abstract*;

  - focus on the common essential qualities of the object
  - focus on the current application context

- What if a specific brand had a special property or behaviour? Later on….*inheritance*

Encapsulation

- Encapsulation implies *__hiding__* the object state (attributes)
- An object's attributes represent its individual characteristics or properties, so access to the object's data must be restricted
- **Methods** provide explicit access to the object

  e.g. use of *getter* and *setter* methods to access or modify the fields

Private Attributes

Class Name
- attribute1: int - attribute2: boolean
+ operation_1(): void + operation_2(): int + operation_3(): boolean

Public Methods

Account
-name: String -min_bal: float -balance: float
+get_balance(): float +get_min_balance():float +set_balance(float) +set_min_balance(float) +deposit(float)
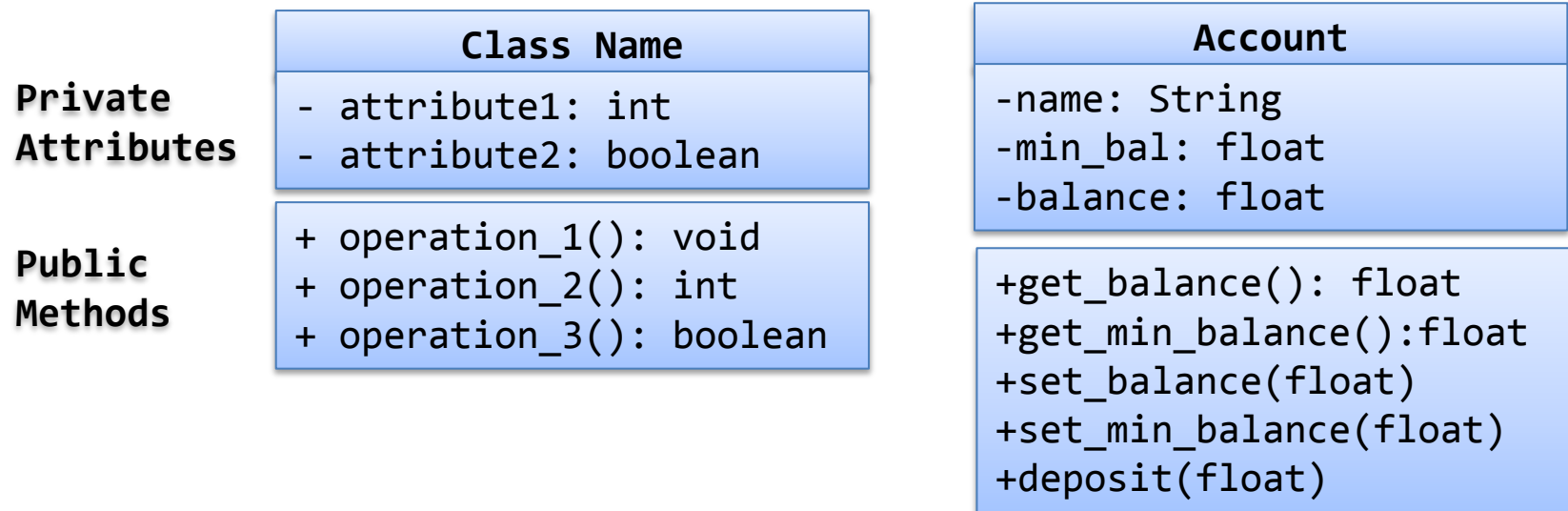
UML notation for a Class

Why is encapsulation important (2)?

1. Encapsulation ensures that an object's state is in a consistent state
2. Encapsulation increases usability
   - Keeping the data private and exposing the object only through its interface (public methods) provides a clear view of the role of the object and increases usability
   - Clear contract between the invoker and the provider, where the client agrees to invoke an object's method adhering to the method signature and provider guarantees consistent behaviour of the method invoked (if the client invoked the method correctly)
3. Encapsulation abstracts the implementation, reduces the dependencies so that a change to a class does not cause a rippling effect on the system

- So far,
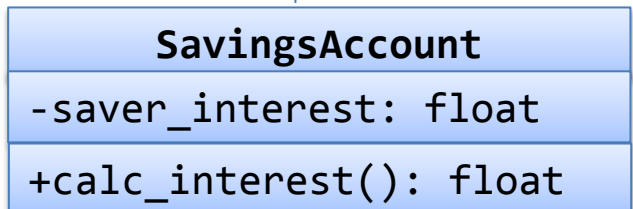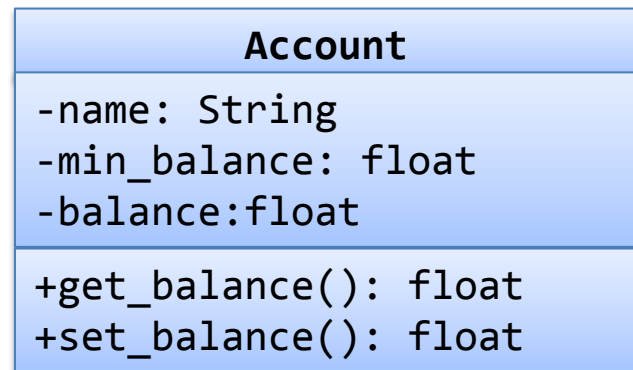 - we have defined classes and object instances
  - objects have attributes and responsibilities
  - learnt about abstraction and encapsulation
- let us now look at **relationships between objects** e.g.,
  - a dog **is-a** mammal
  - an instructor **teaches** a student
  - a university **enrols** students
- Relationships between objects can be broadly classified as:
  - Inheritance
  - Association

Relationships (1) – Inheritance

- So far, we have logically grouped objects with common characteristics into a class, but what if these objects had some special features?

    e.g., if we wanted to store that sports car has spoilers

- Answer is inheritance - models a relationship between classes in which one class represents a more general concept (parent or base class) and another a more specialised class (sub-class)

- Inheritance models a "is-a" type of relationship e.g.,

    - a savings account is a type of bank account

    - a dog **is-a** type of pet

    - a manager **is-a** type of employee

    - a rectangle **is-a** type of 2D shape

Inheritance

- To implement inheritance, we
  - create a new class (sub-class) , that inherits common properties and behaviour from a base class (parent-class or super-class)
    - We say the child class *inherits/ is-derived from* the parent class
  - sub-class can *extend* the parent class by defining additional properties and behaviour specific to the inherited group of objects
  - sub-class can *override* methods in the parent class with their own specialised behaviour

```
          Account
-name: String
-min_balance: float
-balance:float
+get_balance(): float
+set_balance(): float
```

Parent class - Account
class Account defines *name,*
min_bal, balance

```
       SavingsAccount
-saver_interest: float
+calc_interest(): float
```

Child class – SavingsAccount
extends Account class adding its own
attributes and methods e.g.,
saver_interest & calc_interest()

27

Inheritance – another example

Shape

-name: String
-area: float

+getName(): String
+setName(String)
+getArea(): float

class Rectangle *extends* Shape adding attributes *height , width*

Rectangle

-height: float
-width: float

+getArea(): float

class Rectangle *overrides* method *getArea()* to provide its own implementation
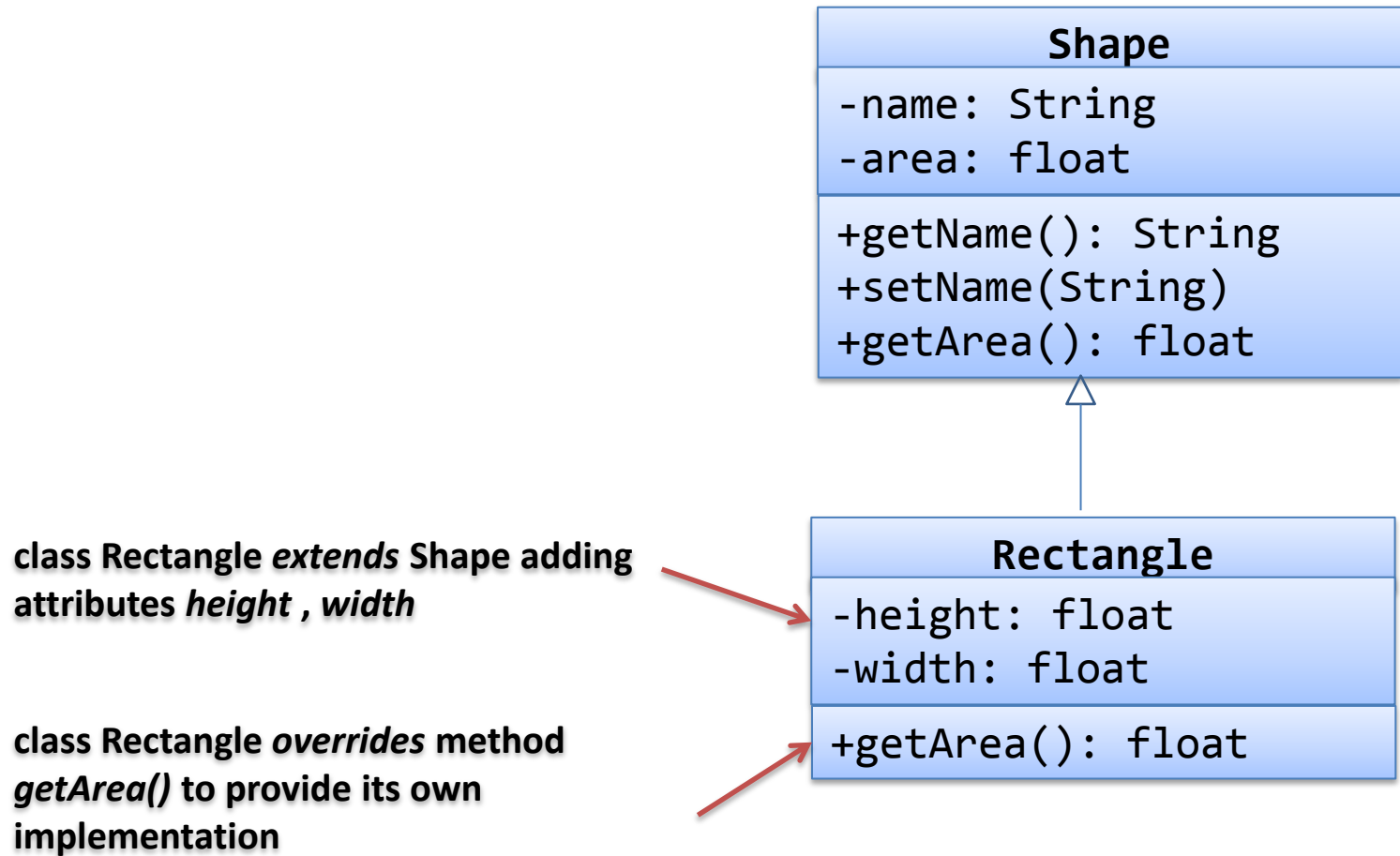
Relationships (2) – Association

- Association is a special type of relationship between two classes, that shows that the two classes are:
  - linked to each other

    e.g., a lecturer *teaches* a course-offering

  - or combined into some kind of *"has-a"* relationship, where one class "contains" another class

  e.g., a course-offering *has* students

- Modelled in UML as a line between two classes

Class 1	Association	Class 2

Dog	1 belongs 1 ▶	Owner

Professor	1 teaches 0..* ▶	Course
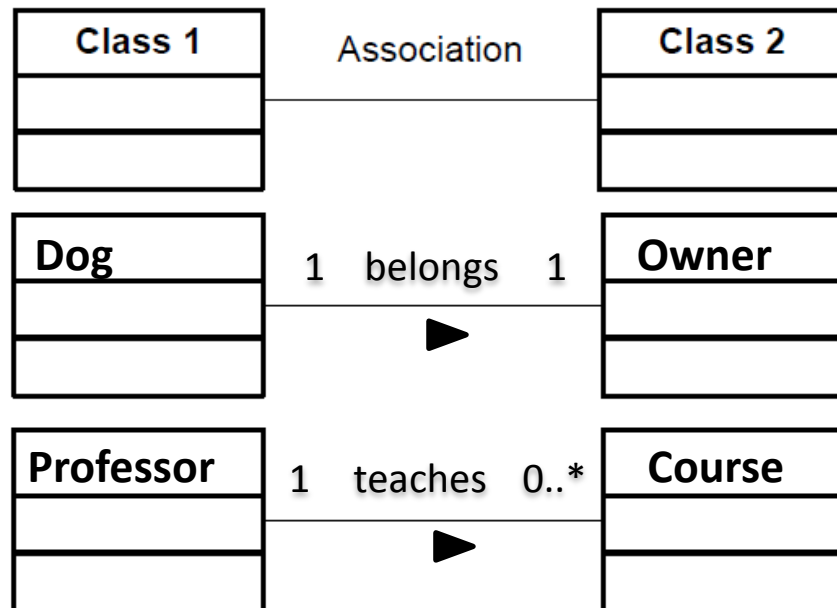
Relationships – Association

- Associations can model a *"has-a"* relationship where one class "contains" another class

- Associations can further be refined as:

  Aggregation relationship (hollow diamond symbol ◊): The contained item is an element of a collection but it can also exist on its own, e.g., a lecturer in a university or a student at a university

  Lab ◊——**contains**—— Computers

  Composition relationship (filled diamond symbol ♦ in UML diagrams): The contained item is an integral part of the containing item, such as a leg in a desk, or engine in a car

  Order ♦——**has**—— Line Items
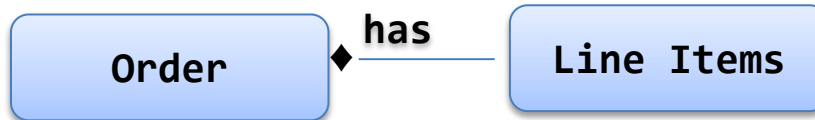
More examples of associations

Car --- 1 consists of 1 --- Engine (Aggregation diagram)
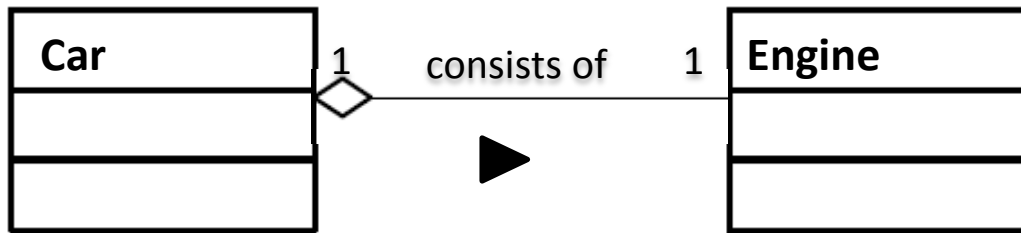
- Aggregation - "has-a" relationship where the part can exist without container

Book --- 1 ... 1..* --- Pages (Composition diagram)
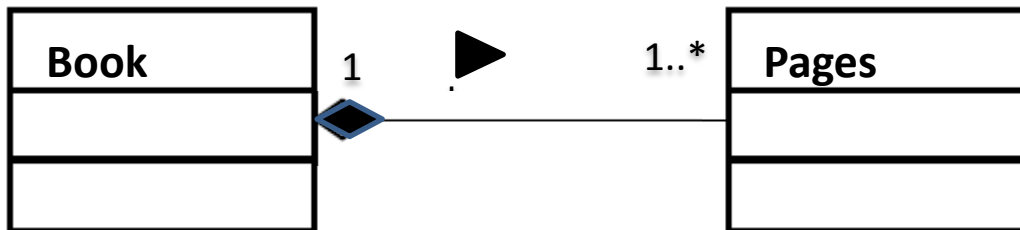
- Composition – "is-composed-of" relationship where part cannot live without container

Vehicle ◁--- Truck (Inheritance diagram)

- Inheritance – "is-a-kind-of" relationship

Re-visiting our earlier question:

<div align="center">How to create a domain model?</div>

So far, requirements engineering helped us to

- Understand the problem domain
- Establish knowledge of how system-to-be is supposed to behave (from requirements analysis, e.g., use cases and sequence diagrams)
- We now apply OO design principles to build a domain model

Next,

- Noun/Verb analysis
- CRC cards
- Domain model

Noun/Verb Phrase Analysis

- – Analyze textual description of the domain to identify **noun** phrases
- – Caveats: Textual descriptions in natural languages are ambiguous (different nouns can refer to the same thing and the same noun can mean multiple things

Consider this text about an ATM machine:

A customer arrives at an ATM machine to withdraw money. The customer enters the card into the ATM machine. Customer enters the PIN. The ATM verifies whether the customer's card number and PIN are correct. Customer withdraws money from the account. The ATM machine records and updates the transaction.

Candidate conceptual classes: ATM, Customer, Account, Card

Domain Modelling Techniques (2) – Using CRC cards

- A simple tool for system analysis and design
- Part of the OO design paradigm and features prominently as a design technique in XP programming
- Written in 4 by 6 index cards, an individual CRC card use to represent a domain object
- The output of this tool – definition of classes and their relationships
- Benefits – Easy to use, forces you to be concise and clear

Domain Modelling Techniques (2) – Using CRC cards

CRC stands for **C**lass, **R**esponsibility and **C**ollaboration

Class:

- An object-oriented class name, represents a collection of similar objects
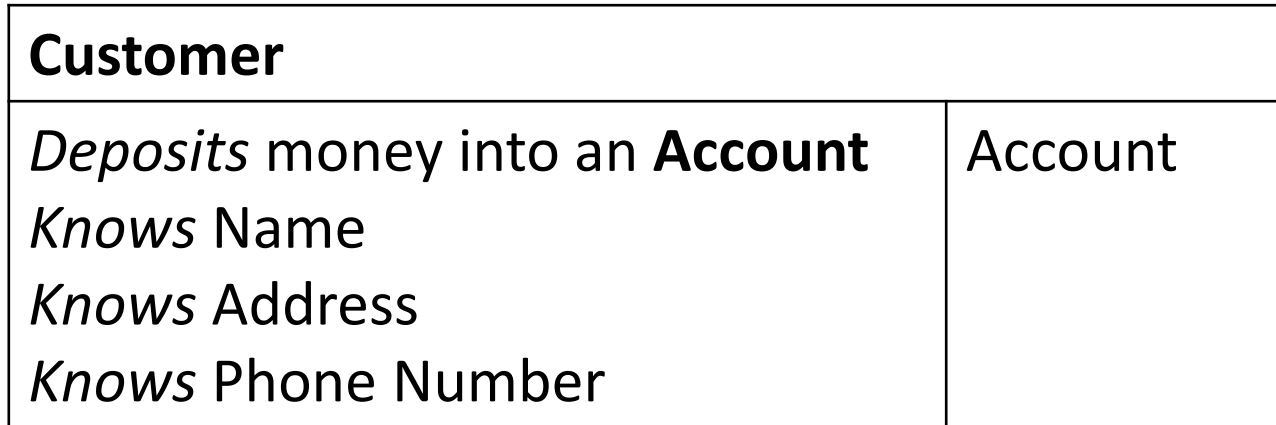
Responsibility

- What does this class know?
- What does this class do?

Collaboration:

- Relationship to other classes (What classes does this class use?

Domain Modelling Techniques (2) – Using CRC cards

Class

Customer	
Deposits money into an **Account** *Knows* Name *Knows* Address *Knows* Phone Number	Account

Responsibilities **Collaborations**

A **CRC Model** is a collection of CRC cards, that specifies the OO design of the system

How to create a CRC model?

You are given a description of the requirements of a software system

Gather around a table with a set of index cards and pen

As a team:

- Read the description
- Identify core classes (look for nouns)
- Create a card per class (start with just class names)
- Add responsibilities (look for verbs)
- Add collaborations
- Add more classes as you discover them
- Refine by identifying inheritance etc.

How can you tell it works?

A neat technique: a **Scenario Walk-through**.

- Select a set of scenarios (use cases).

- Choose a plausible set of inputs for each scenario.

- Manually "execute" each scenario.

- Start with initial input for scenario and find a class that has responsibility for responding to that input.

- Trace through the collaborations of each class that participates in satisfying that responsibility.

- Make adjustments and refinements as necessary.

Techniques for Domain Modelling (2)

Evolve CRC domain models into UML class diagrams where:

- Concepts are represented as classes

- Collaborations between the classes established as relationships

- Depending on the kind of relationship, we can use the different notations that we've used for associations – non-hierarchical, part-of (aggregation), is-a (inheritance),

Lecture Demo

- Creating a CRC Model
- Creating a class diagram

OO Programming with Python

- Creating classes

- Encapsulation

- Inheritance and abstract classes

- Implementing association, aggregation and composition

Defining a class in Python

Basic Python Syntax

- To create a class, use the keyword **class** followed by the name of the class e.g., **Account**

```
class ClassName:
    'Optional class documentation string'
    class_suite
```

```
class Account:
        'Common base class for all bank accounts'
```

- An ***object instance*** is a specific realization of the class
  - Defining a class, does not actually create an object
  - Create an instance of the Account class as follows:

Creating object instances

- A class is sometimes referred to as an object's type
- An object instance is a specific realization of the class
- create an instance of the Account class as follows

```python
class Account:
        'Common base class for all bank accounts'

# a1 and a2 are object instances
a1 = Account()
a2 = Account()
```

a1

Memory

a1:Account

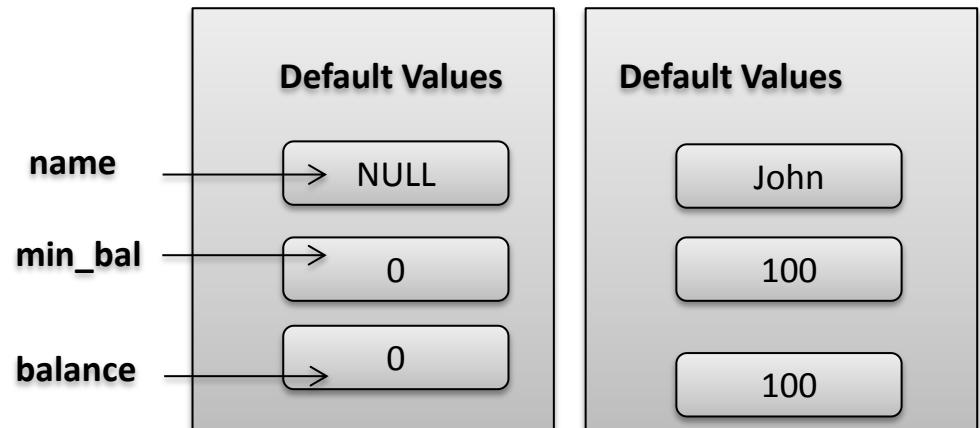**a1 == a2 -----> True or False?**

a2:Account

a2

43
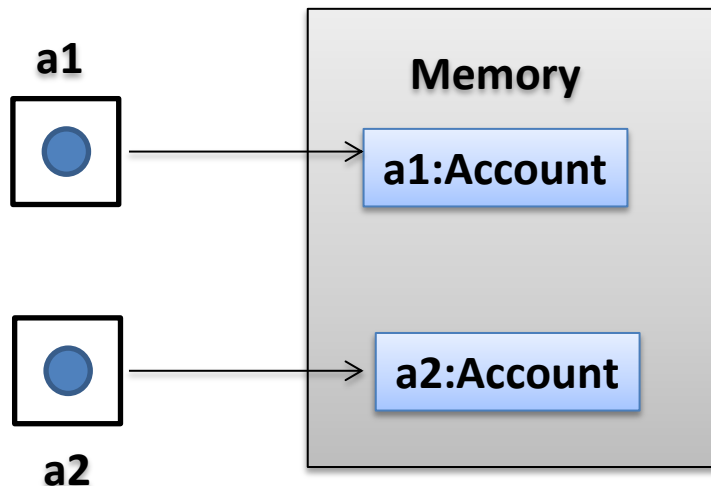
Constructor & Instance Variables

- A special method that creates an object instance and assign values (initialisation) to the attributes (instance variables)
- Constructors eliminate default values
- When you create a class without a constructor, Python automatically creates a default "no-arg" constructor for you

```python
class Account:

    def __init__(self,name,min_bal):
        self.name = name
        self.min_bal = min_bal
        self.balance = min_bal

# a1 is an object instance
a1 = Account("John",100)
```
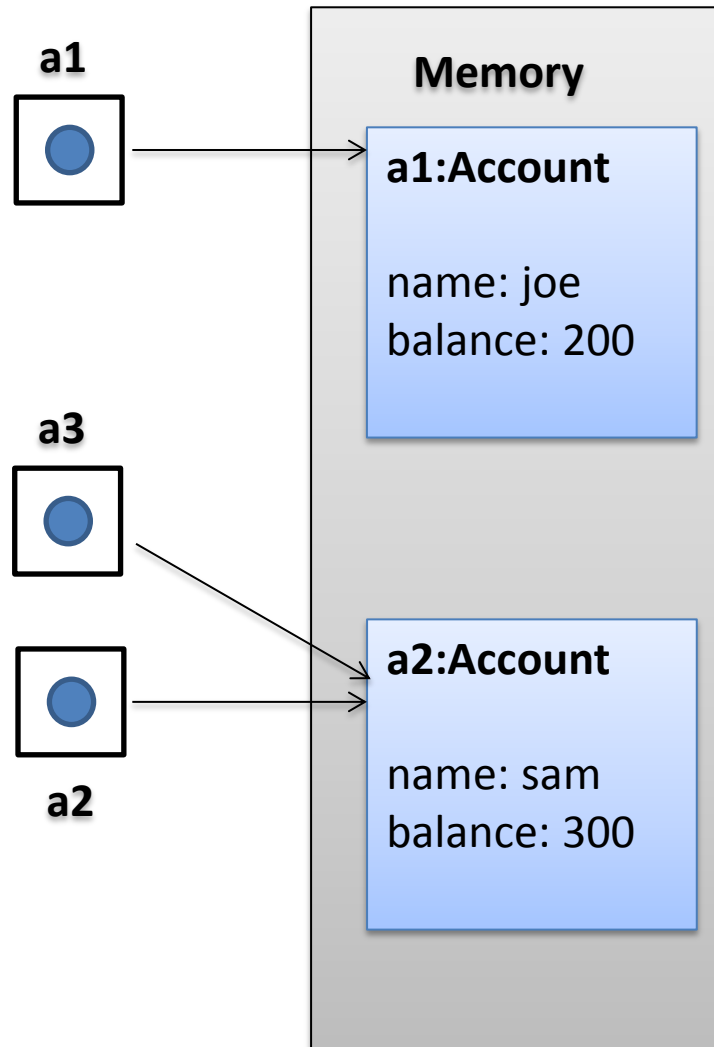
name → **Default Values** NULL / John

min_bal → **Default Values** 0 / 100

balance → 0 / 100

Object References

a1

Memory

a1:Account

name: joe
balance: 200

a3

a2

a2:Account

name: sam
balance: 300

a1 == a2 -----> True or False?

Consider,
a3 = a1
a3 == a1   -----> True or False?

Instance Methods

- Similar to instance variables, methods defined inside a class are known as **instance methods**

- Methods define what an object can do.

- In Python, every instance method, must specify **self** (the specific object instance) as an argument to the method including the constructor (**__init__()**)

```python
class Account:

    def __init__(self,name,min_bal):
        self.name = name
        self.min_bal = min_bal
        self.balance = min_bal

    def deposit(self, amount):
        self.balance += amount;

# a1 is an object instance
a1 = Account("John",100)
a1.deposit(120)
```

Encapsulation in Python

- Python does not support strong encapsulation. Attribute names are simply prefixed with a single underscore e.g., _name to signal that these attributes are **private** and must not be directly accessed by clients

```python
class Account:

    def __init__(self,name,min_bal):
        self._name = name
        self._min_bal = min_bal
        self._balance = min_bal

    def get_name(self):
        return self._name

    def get_min_bal(self):
        return self._min_bal

    def set_min_bal(self,min_bal):
        self._min_bal = min_bal

    def get_balance(self):
        return self._balance

    def deposit(self, amount):
        self.balance += amount;
```

Recall why we need encapsulation important

1. Encapsulation ensures that an object's state is in a consistent state

```python
class Account:

    def __init__(self,name,min_bal):
        self._name = name
        self._min_bal = min_bal
        self._balance = min_bal

    # define the getter and setter methods
    # ...

    def deposit(self, amount):
        self.balance += amount;

    def withdraw(self, amount):
        if self._balance - amount <= self._min_bal:
            print("Minimum balance must be maintained")
        else:
            self.balance -= amount

a1 = Account("John",100)
a1.withdraw(50)

a1._balance = 10
print("Current balance: {0}".format(str(a1._balance)))
```

breaking encapsulation and direct assignment of the *balance* attribute, potentially set the *balance* to an amount less than minimum balance, violating the business constraint

encapsulation enforces that *balance* is hidden and can only be changed through *deposit* and *withdraw* methods

Implementing Inheritance in Python

```python
class Account(object):

        def __init__(self, name=None, min_bal=0):
             self._name = name
             self._balance = min_bal

        def get_name(self):
                return self._name
        def get_balance(self):
                return self._balance
        def set_balance(self,amount):
                self._balance = amount



class SavingsAccount(Account):
        def __init__(self,name,amount):
                Account.__init__(self,name,amount)
                self._saver_interest = 0.05
        def get_interest(self):
                return self._saver_interest

a2 = SavingsAccount("joe",1000)
print("{0}'s balance is {1} building interest at {2}:"
      .format(a2.get_name(),a2.get_balance(),a2.get_interest()))
```

Abstract Classes in Inheritance

- In the example below
  - `Savings` and `Cheque` both inherit from the base class `Account`
    - But `Account` is really not a real-world object
    - `Account` is a *concept* that represents some real-world objects like a Savings Account), so `Account` is said to be an abstract class
    - It does not make sense to create an instance of an abstract class

```
         Account
        /       \
   Savings     Cheque
```
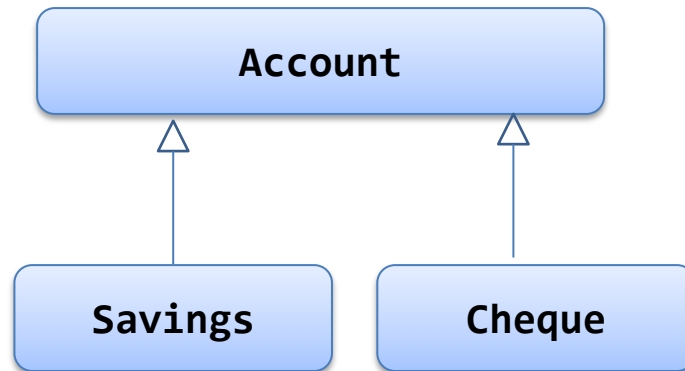
Abstract class – Vehicle
Defines common attributes e.g., *make, model, year, miles, wheels*