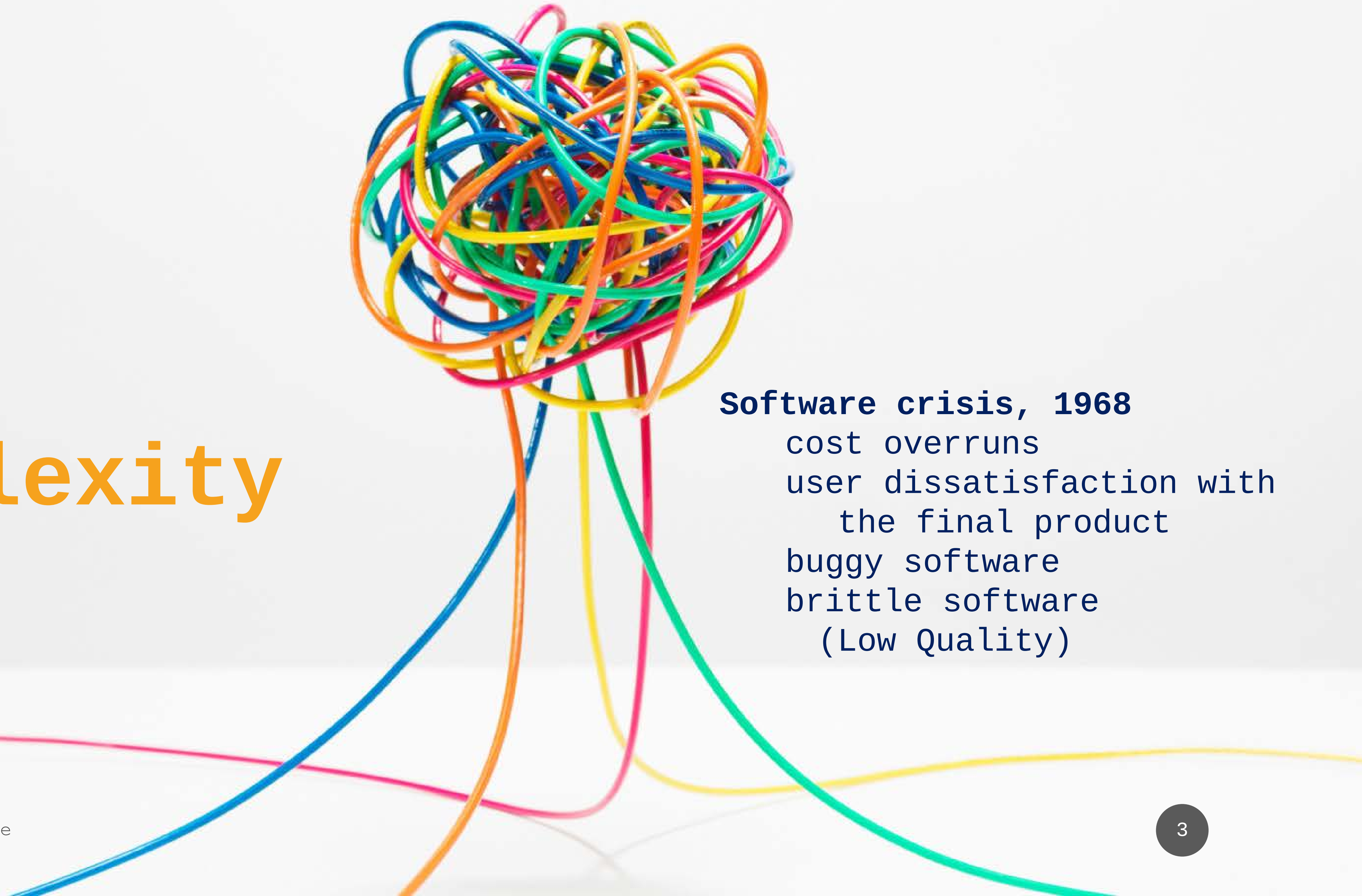
OBJECT-ORIENTATION

Dr. Issarapong Khuankrue

CONTENTS



Complexity

Software crisis, 1968

- cost overruns
- user dissatisfaction with the final product
- buggy software
- brittle software (Low Quality)

PRINCIPLE OF OBJECT-ORIENTATION

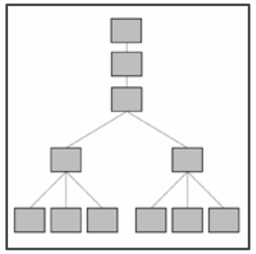
- **Major Elements** – By major, it is meant that if a model does not have any one of these elements, it ceases to be object oriented.

Abstraction

Encapsulation

Modularity

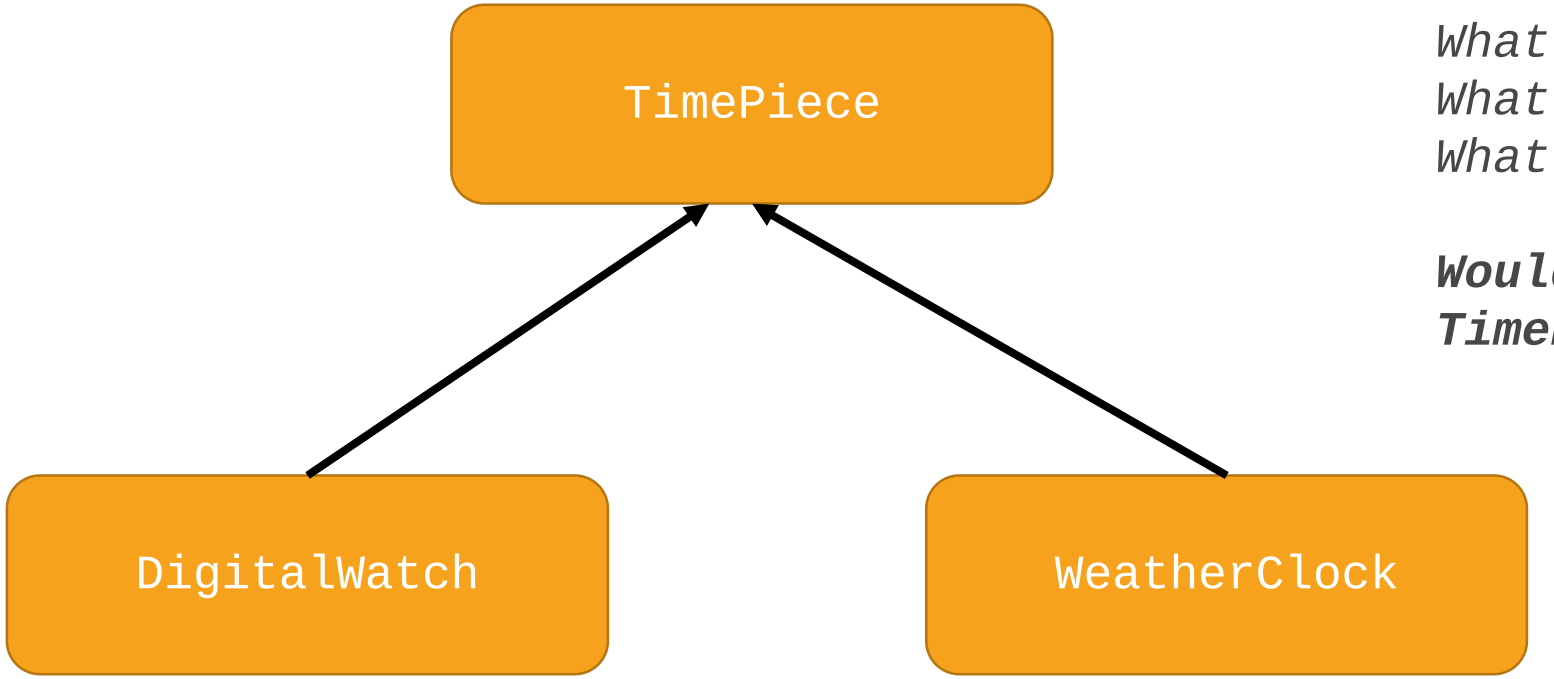
Hierarchy



EXERCISE

Write down as many of the following telephone numbers as you can

the details of the numbers away and grouping them into a new concept (telephone number)



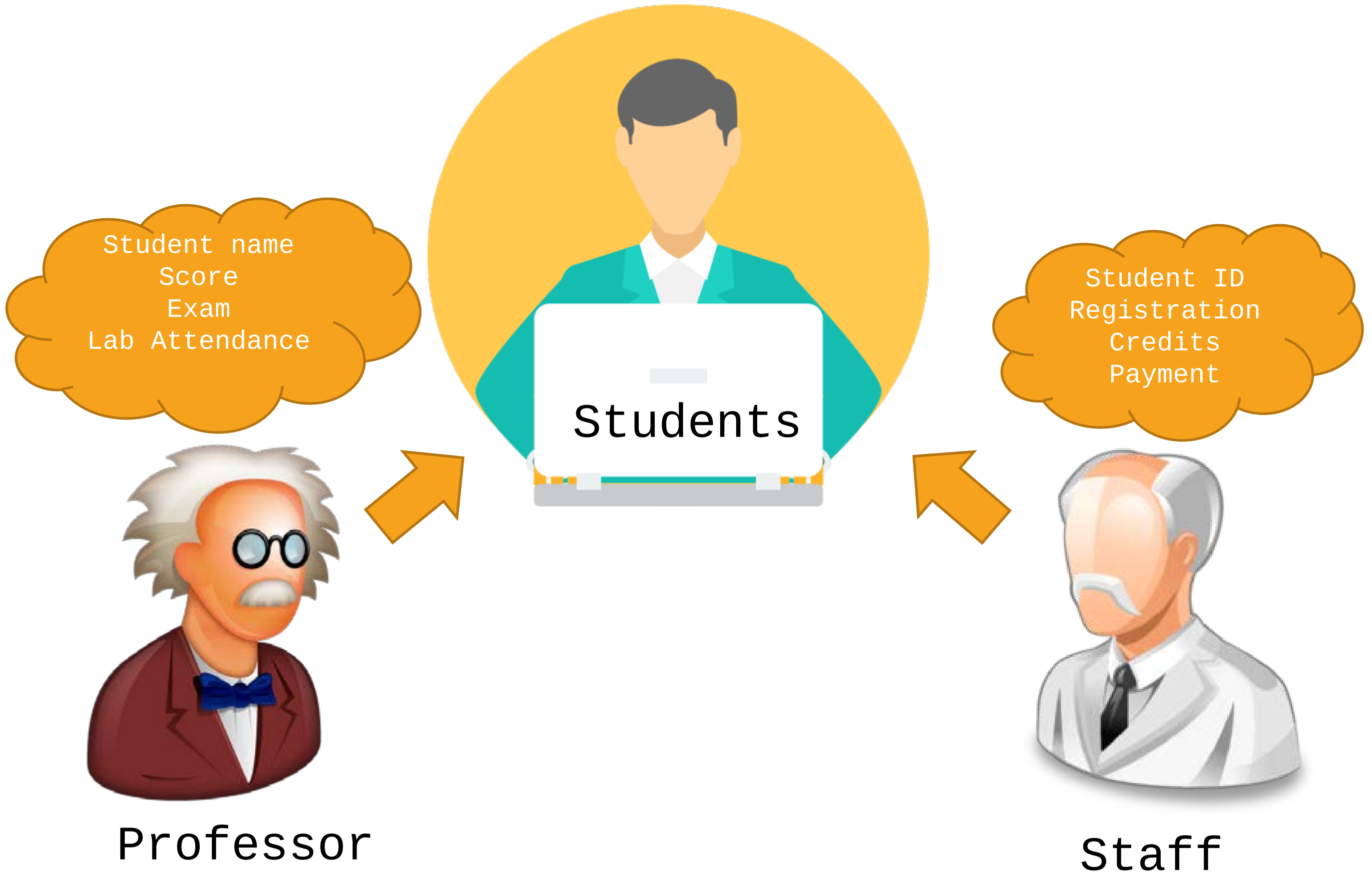
What’s in TimePiece?
What’s in DigitalWatch?
What’s in WeatherClock?

Would anybody buy a TimePiece product?

Inheritance hierarchy for TimePiece, DigitalWatch, WeatherClock

ABSTRACTION

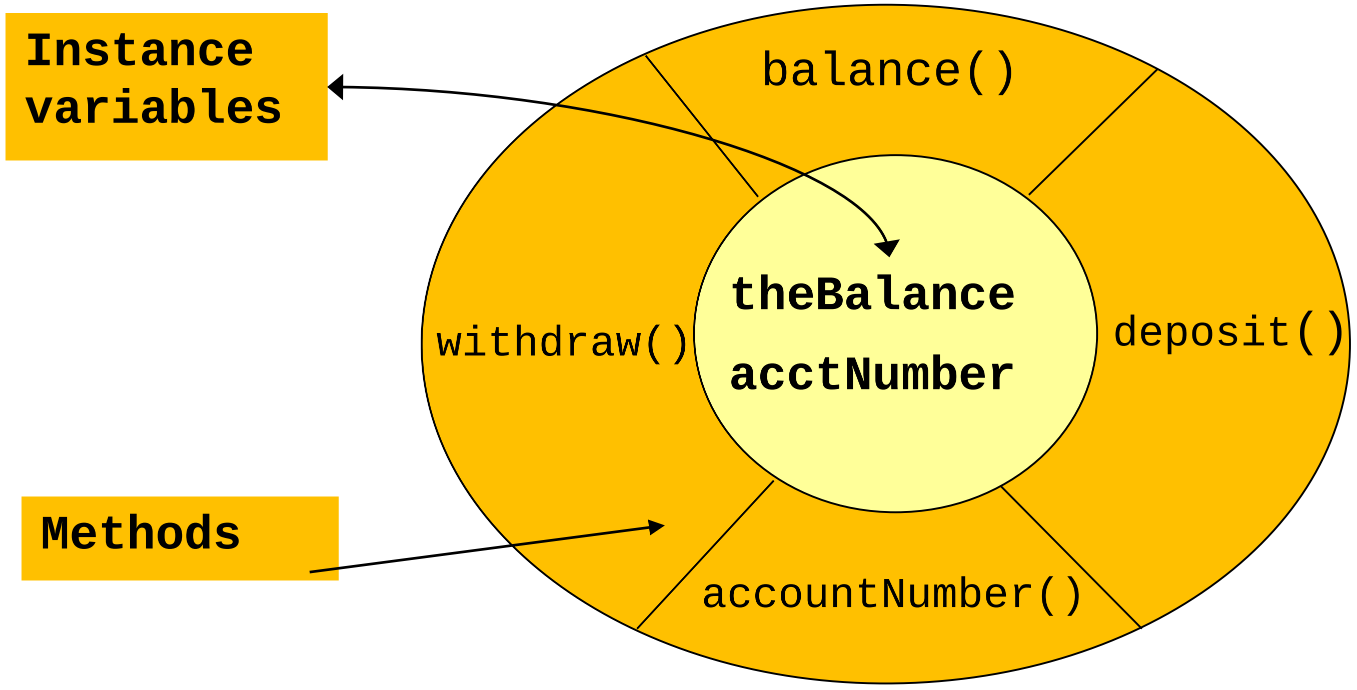
- Humans deal with complexity by abstracting details away.
- Abstraction means to focus on the **essential features of an object**.
  - The essential features are relative to the **context** in which the object is being used.



ENCAPSULATION (INFORMING HIDING)

- **Encapsulation : Hide implementation from clients**
  - Clients depend on interface – only!
  - Clients do not need to know 'how' the server operates or provides the services!

- How does an object encapsulate?
- What does it encapsulate?



Object **videoSony** of class **VideoAsia**

```
videoSony : VideoAsia
- Brand     : String
- Country   : String
- Volt      : String = 110
              or 220 Volts
- Type      : String = VDO
- Continent : String = Asia
VideoAsia (strBrand : String)
getBrand ()    : String
setCountry     : void
getCountry()   : String
getType()      : String
getContinent   : String
getVolt()      : String
player()       :void
```

External View

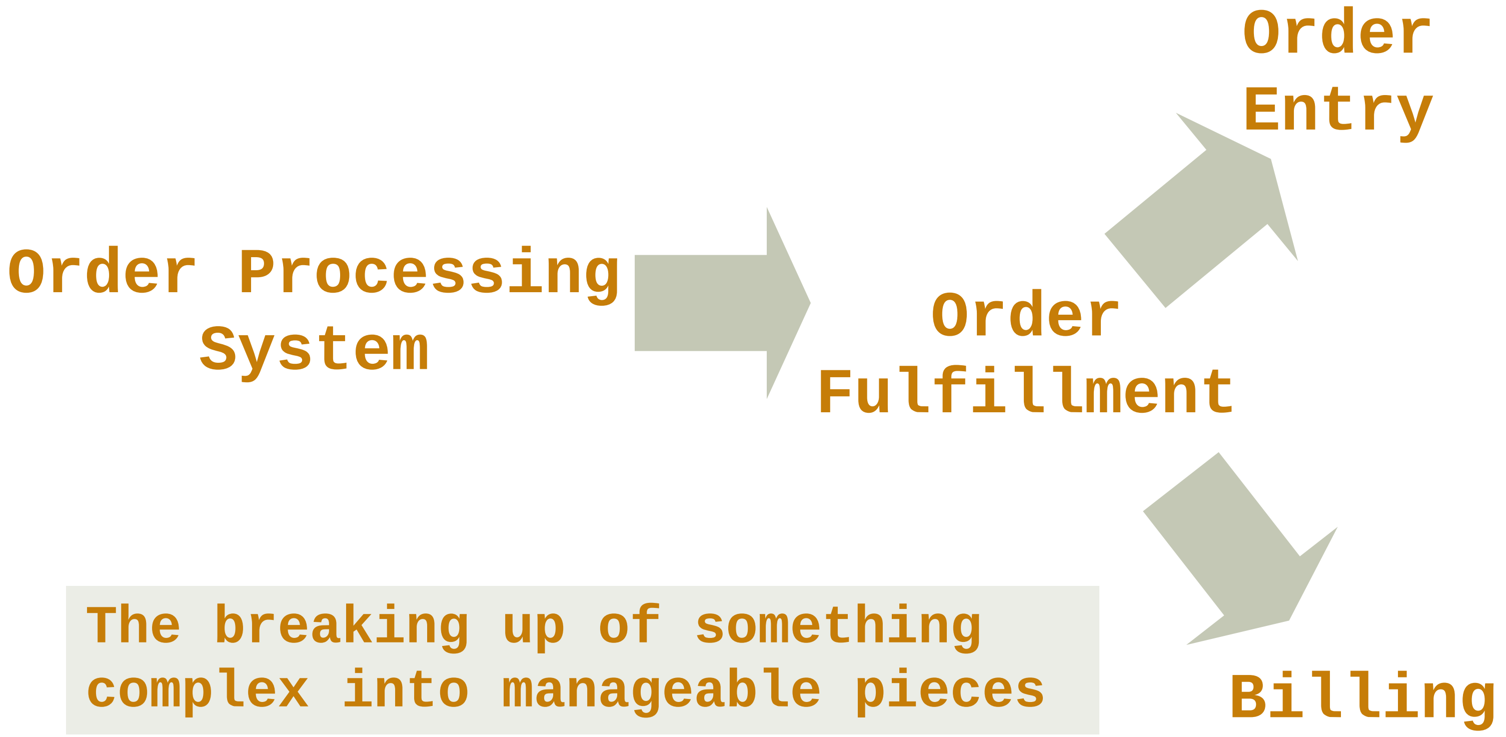
```
videoSony : VideoAsia

VideoAsia (strBrand : String)
getBrand ()    : String
setCountry     : void
getCountry()   : String
getType()      : String
getContinent   : String
getVolt()      : String
player()       :void
```

EXAMPLE

MODULARITY

- Modularity is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem.

- Modularity is intrinsically linked with encapsulation.

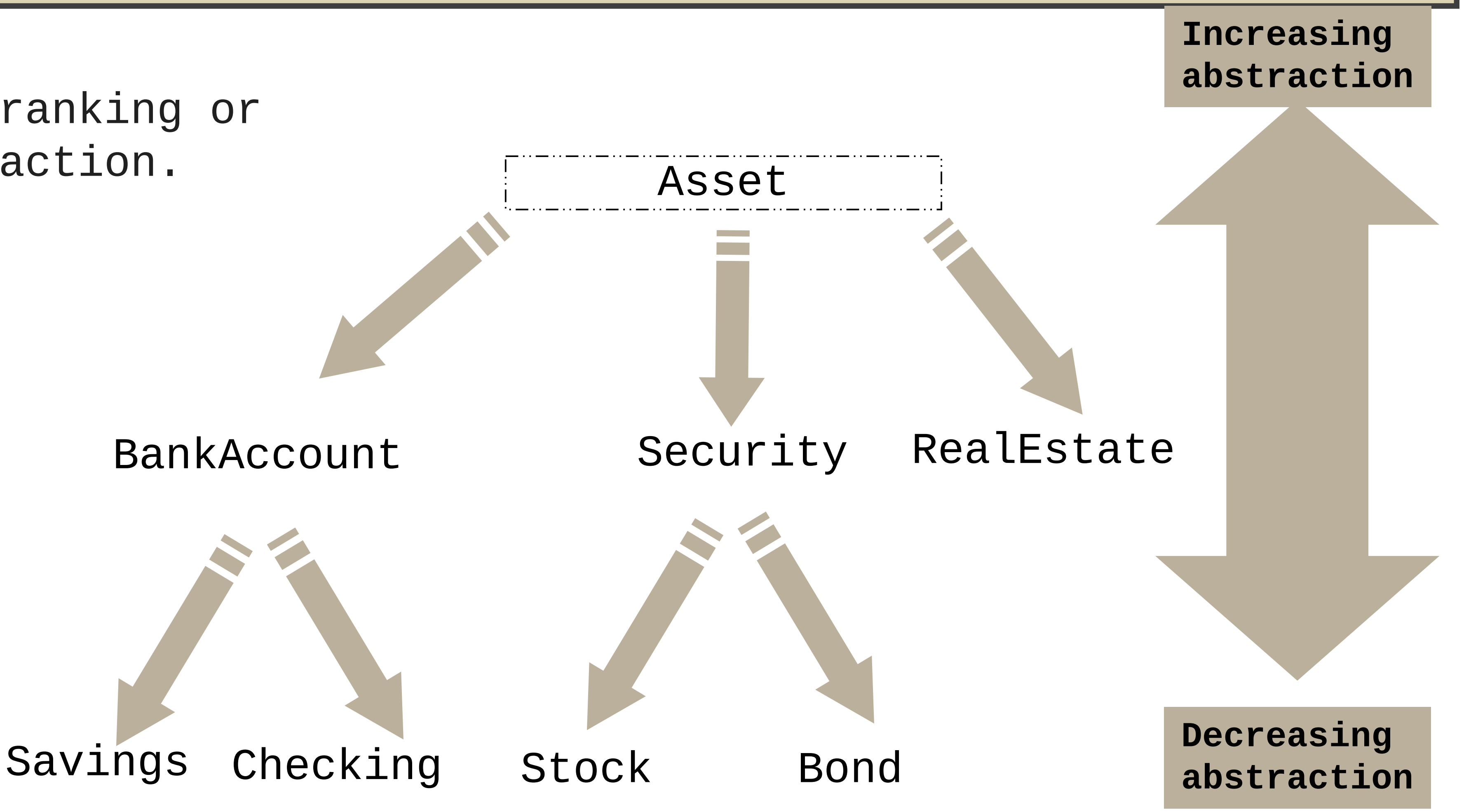


The breaking up of something complex into manageable pieces

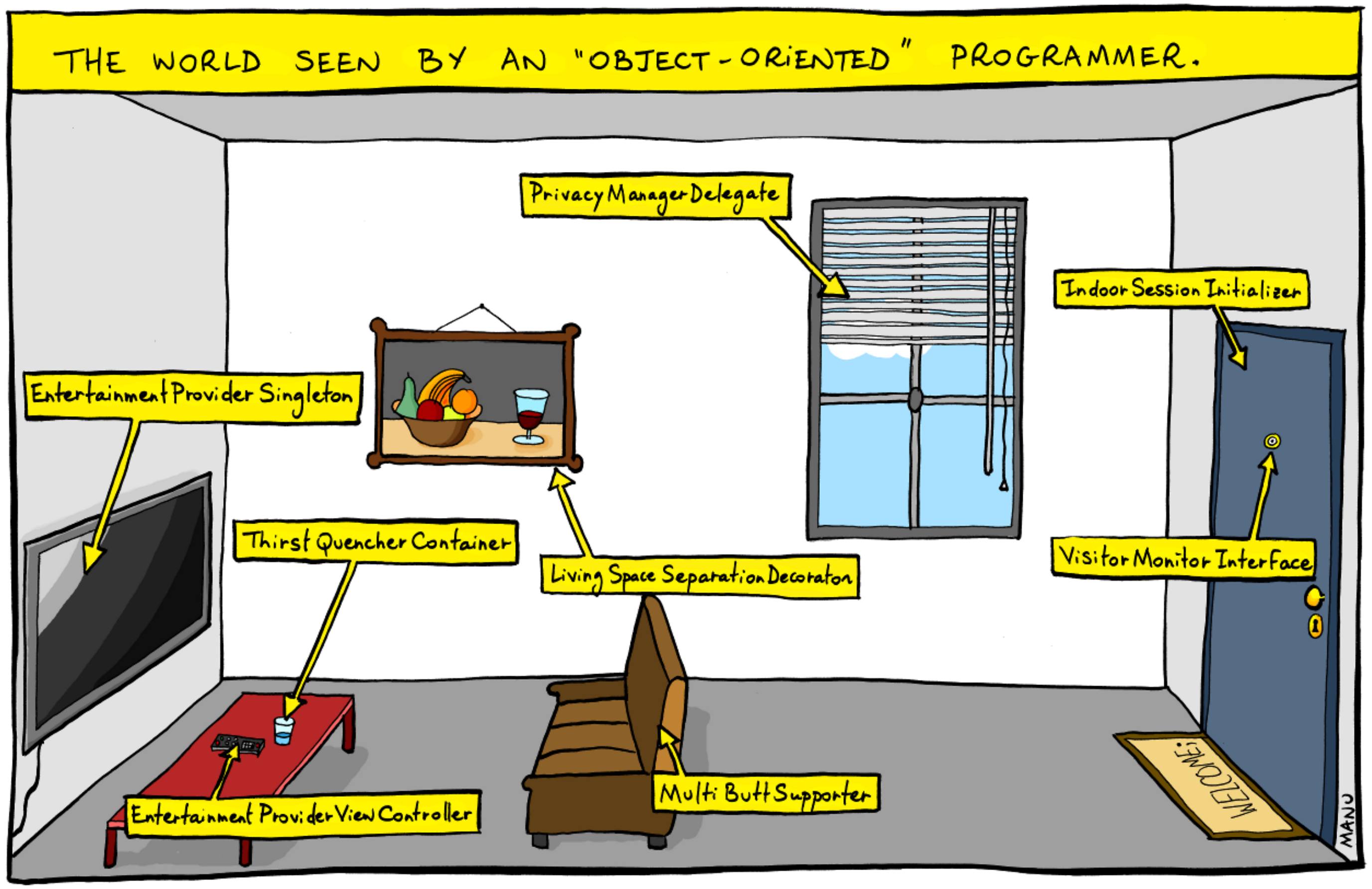
HIERARCHY

- **Hierarchy** is the ranking or ordering of abstraction.

Elements at the same level of the hierarchy should be at the same level of abstraction

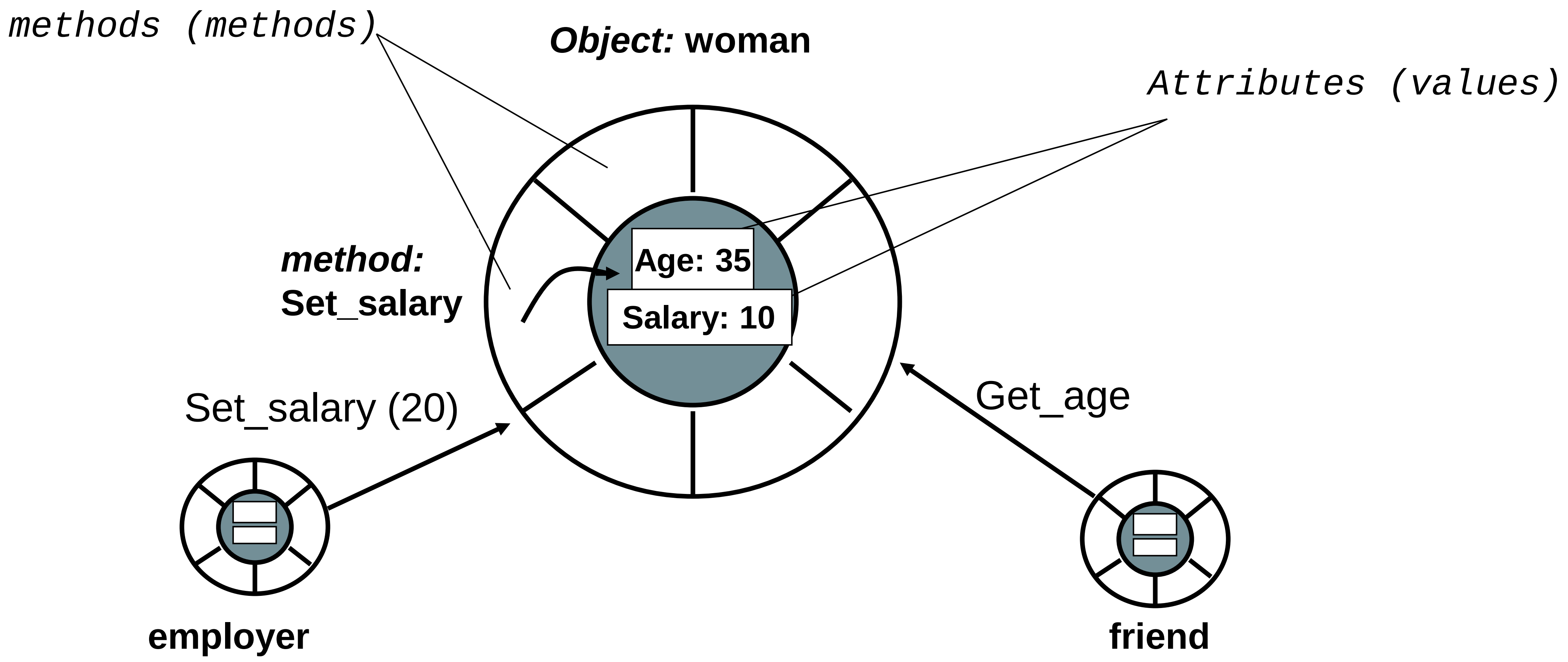


BASIC CONCEPTS OF OBJECT ORIENTATION

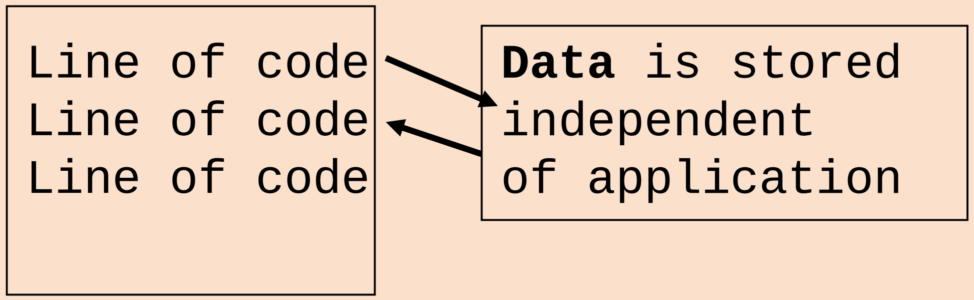
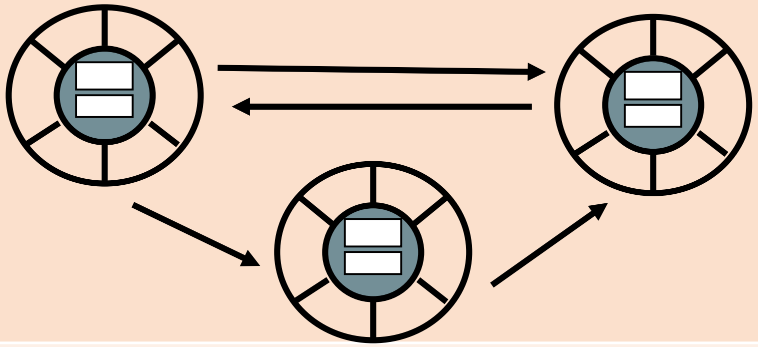

THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.
Privacy Manager Delegate
Indoor Session Initializer
Entertainment Provider Singleton
Thirst Quencher Container
Living Space Separation Decorator
Visitor Monitor Interface
Multi Butt Supporter
Entertainment Provider View Controller
WELCOME!
MANU

OBJECTS

Complex data type that has an identity, contains other data types called attributes and modules of code called operations or methods

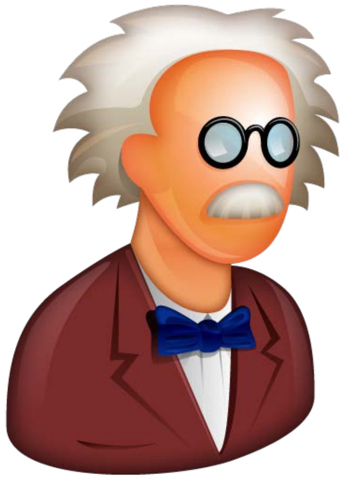


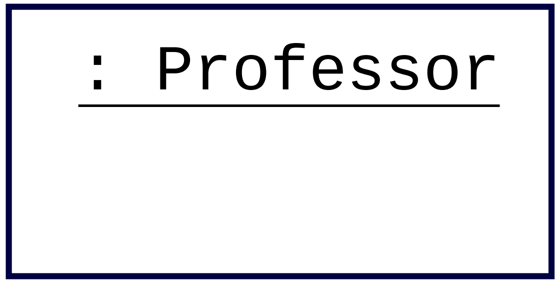
STRUCTURED APPROACH VS. OBJECT-ORIENTED APPROACH

Structured Approach	Object Oriented Approach
 Line of code Line of code Line of code **Data** is stored independent of application	
Top-down	Bottom-up
Divided into number of submodules or functions.	Organized by having number of classes and objects.
Function call is used	Message passing is used
Software reuse is not possible	Reusability
Usually left until end phases	OOD done concurrently with other phases
Clear transition from design to implementation	Not so clear transition from design to implementation
Suitable for real time system (e.g. embedded system)	Suitable for applications, which are expected to customize or extended (e.g. business/game development projects)

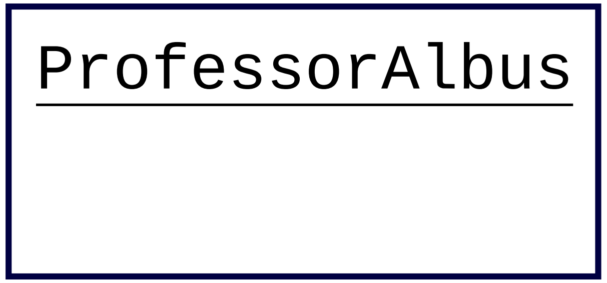
REPRESENTING OBJECTS

- An object is represented as rectangles with underlined names

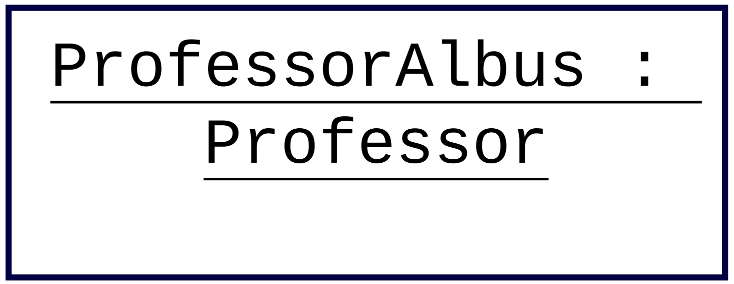

Professor Albus



Class Name Only



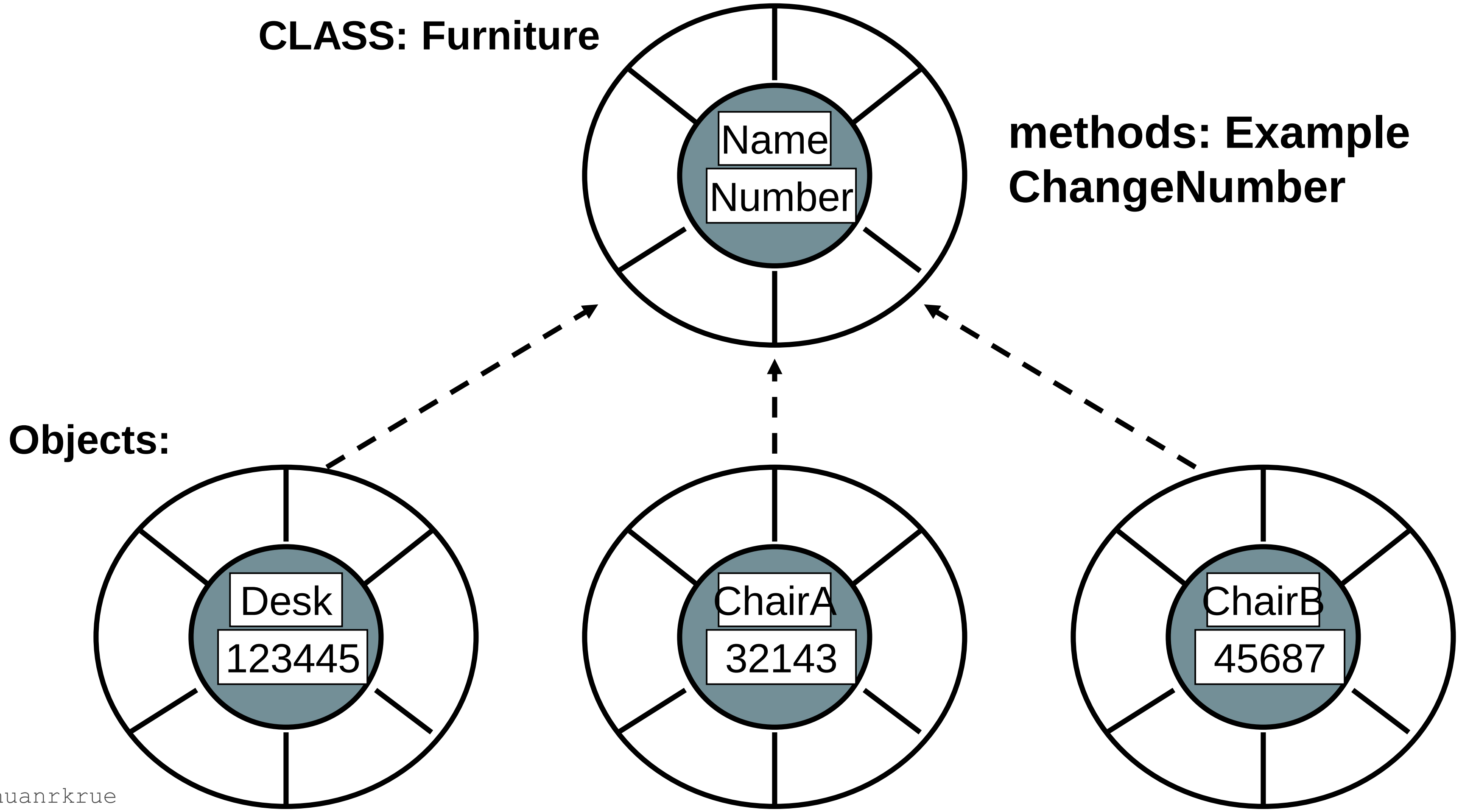
Object Name Only



Class and Object Name

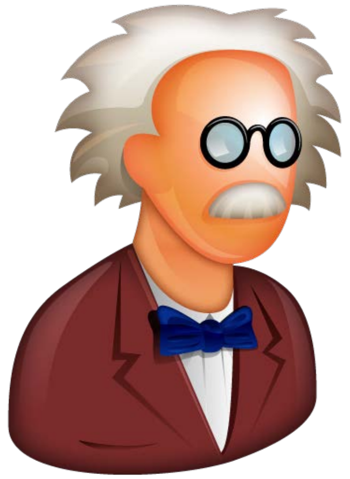
CLASSES

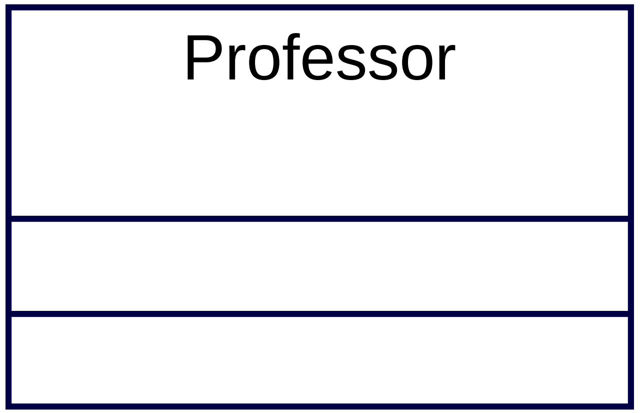
Classes are templates for objects that have methods and attribute names and type information, but no actual values!

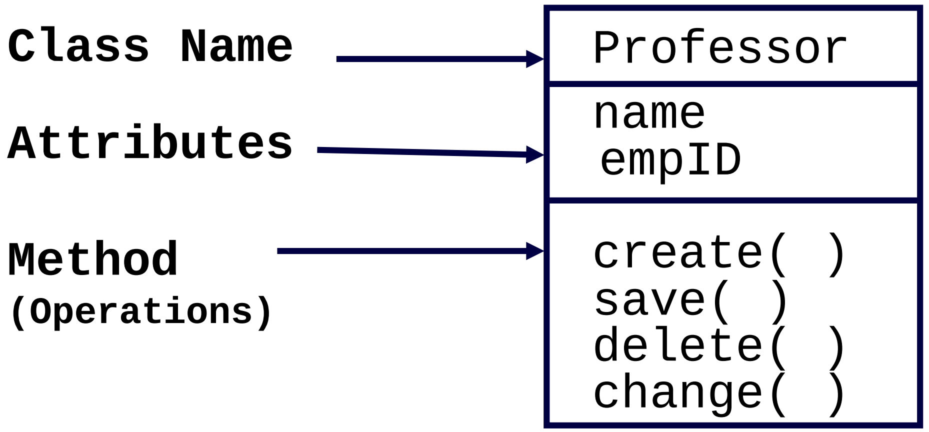


REPRESENTING CLASSES

- A class is represented using a compartmented rectangle

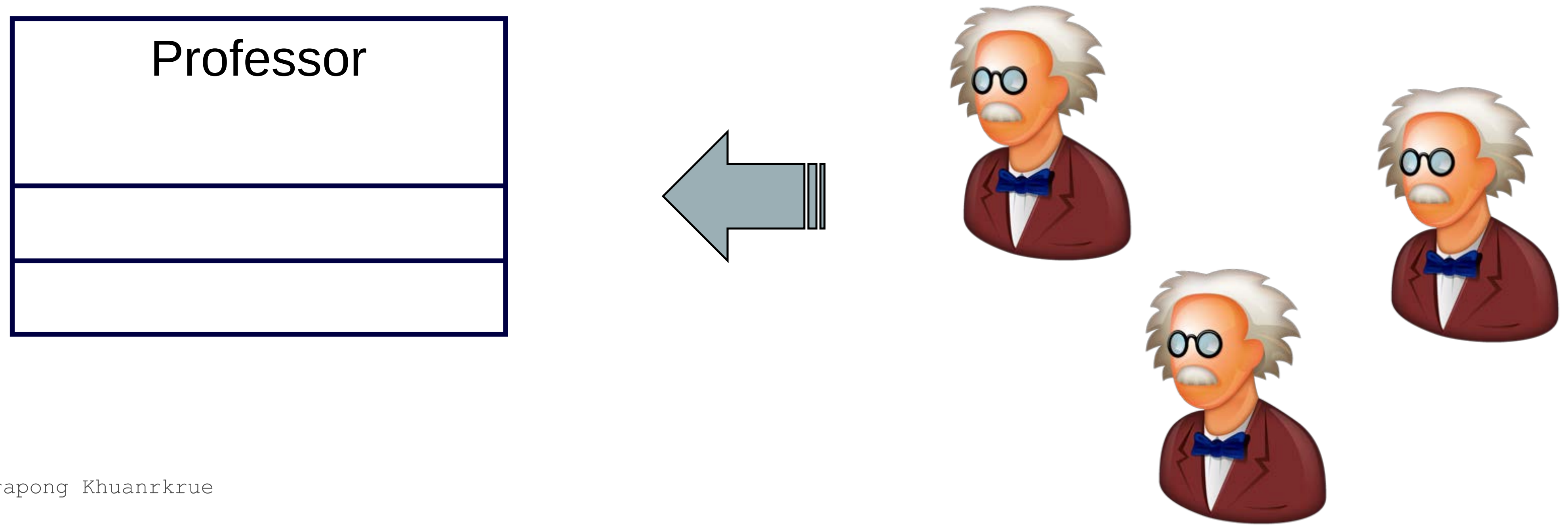

Professor Albus





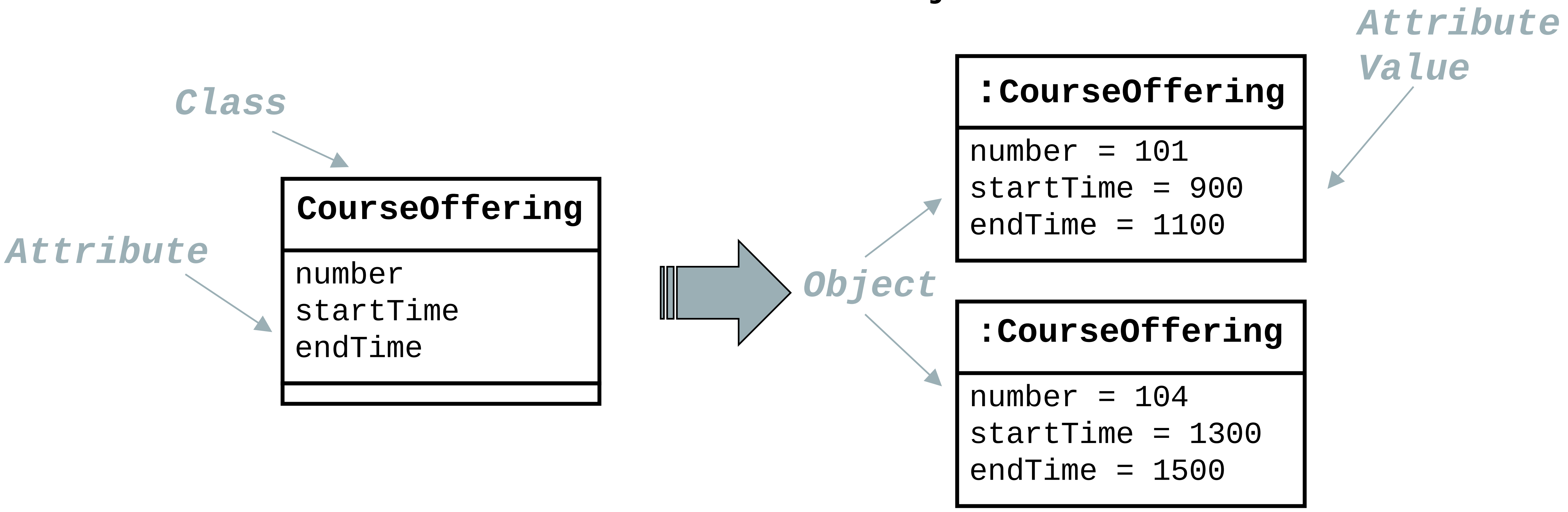
RELATIONSHIP BETWEEN CLASSES AND OBJECTS

- A class is an abstract definition of an object
  - It defines the structure and behavior of each object in the class
  - It serves as a template for creating objects



ATTRIBUTE

Attribute describe information about the object.



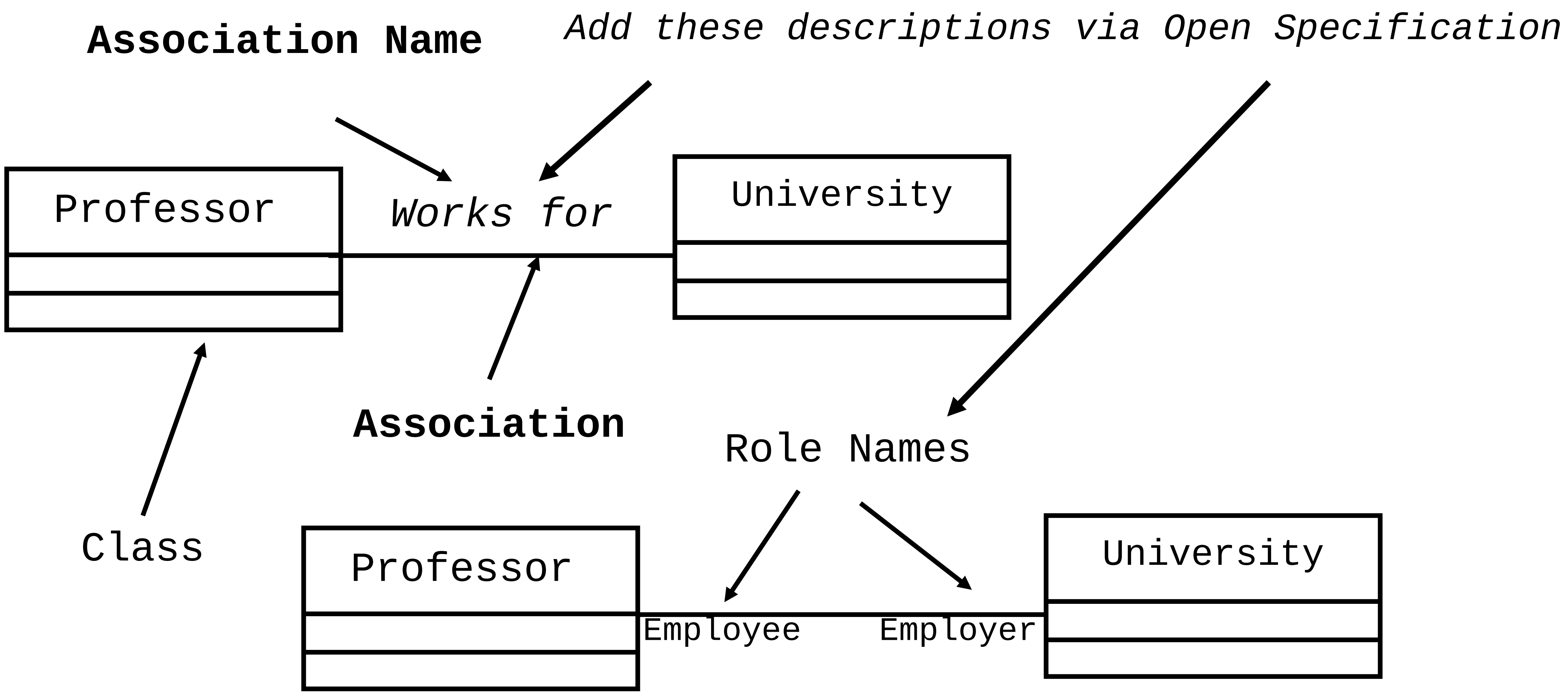
METHOD & MESSAGE

- **Methods** are associated with classes but classes don’t send messages to each other.
  - Method’s name and the parameters that must be passed with the message in order for the method to function

- Objects send **Messages**.
  - A **static diagram** (class diagram) shows classes and the logical associations between classes, it doesn´t show the movement of messages.
  - **Association** between two classes means that the objects of the two classes can send messages to each other.
  - **Aggregation**: when an object contains other objects (a part-whole relationship)

ASSOCIATION #1

Models a semantic connection among classes



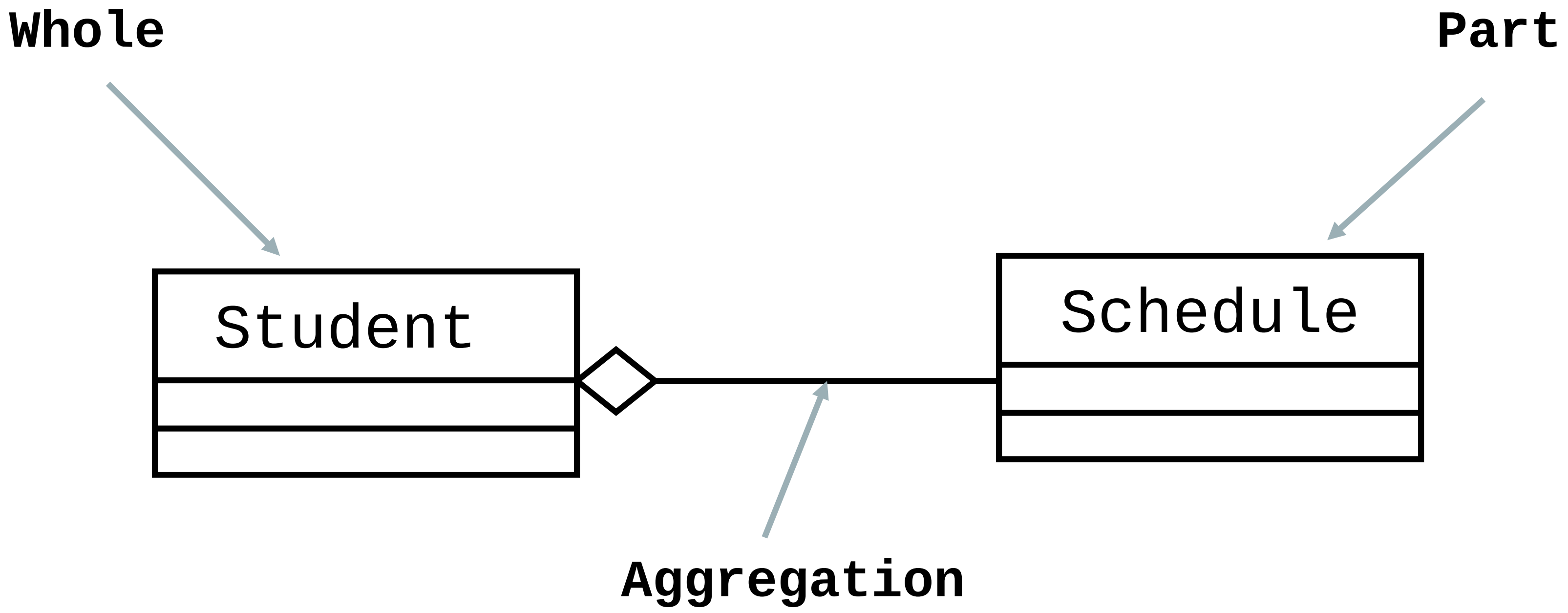
ASSOCIATION #2

Multiplicity

Unspecified	
Exactly one	1
Zero or more (many, unlimited)	0..*
	*
One or more	1..*
Zero or one	0..1
Specified range	2..4
Multiple, disjoint ranges	2, 4..6

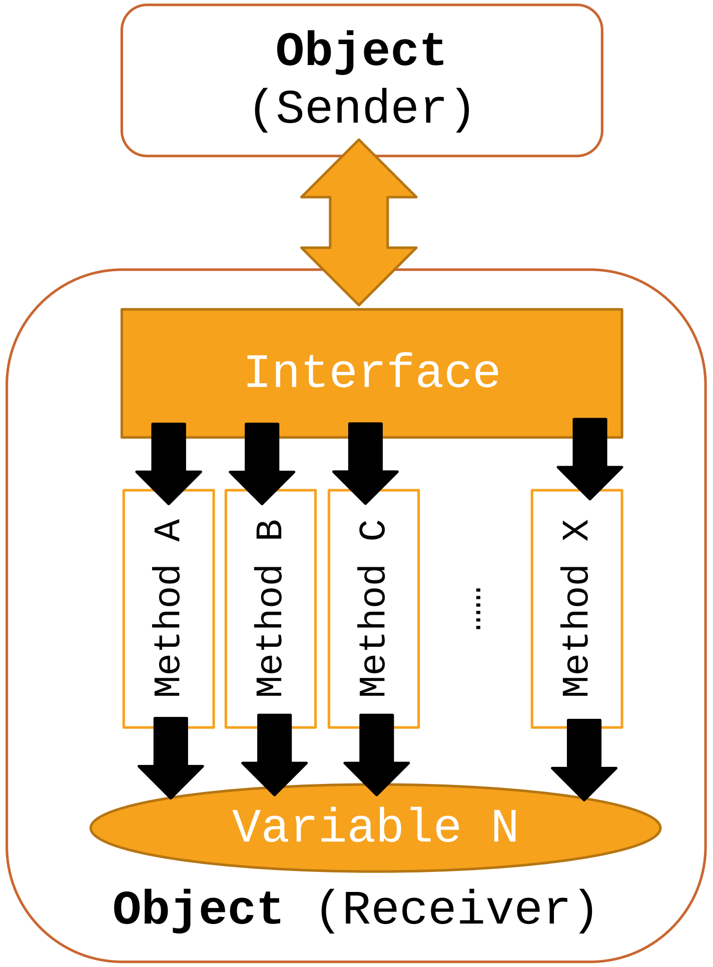
AGGREGATION

A special form of **association** that models a whole-part relationship between an aggregate (the whole) and its parts



This is sometimes called a 'has_a' relationship

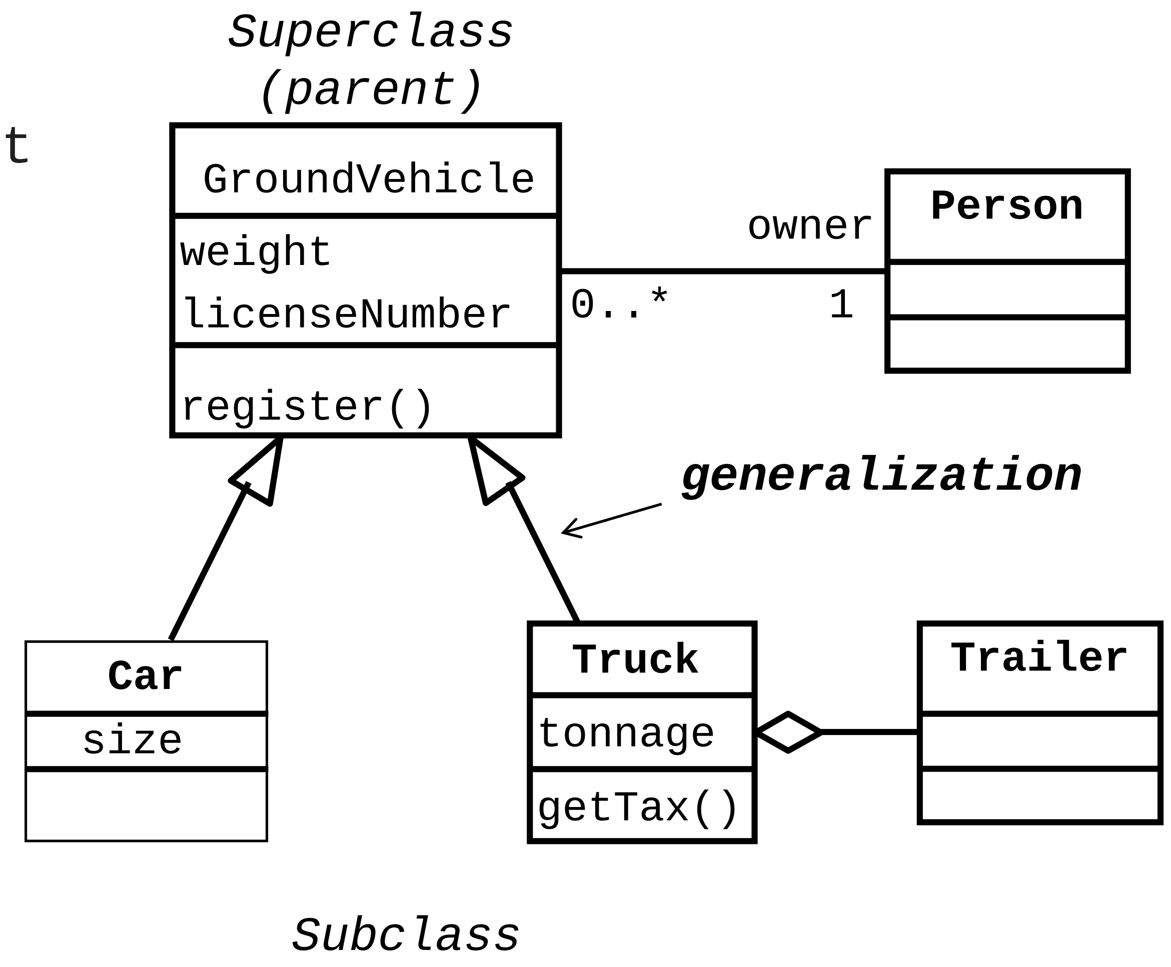
INTERFACE



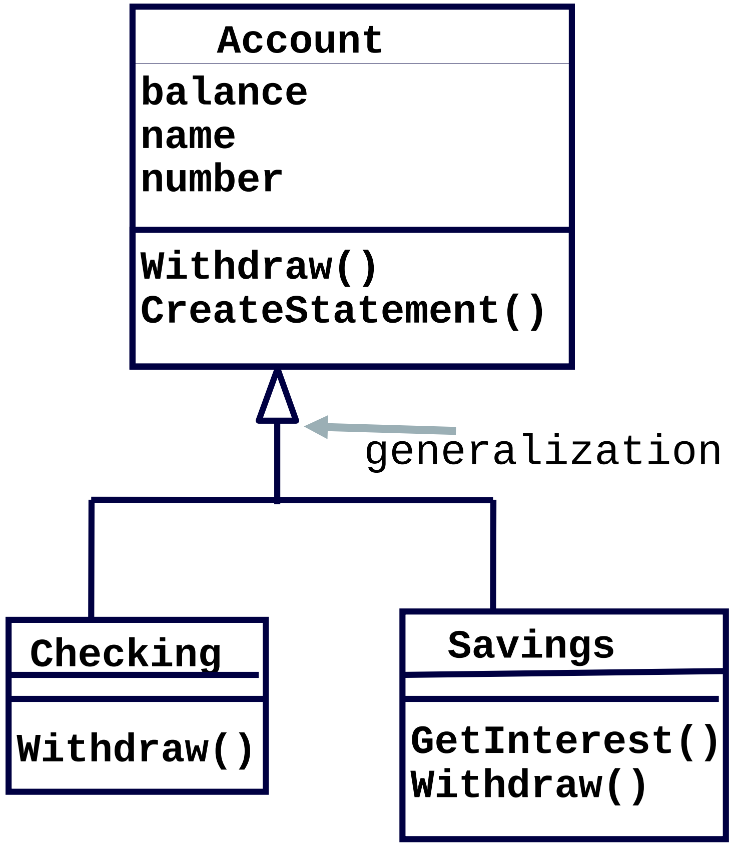
- **Attributes** can be public or private:
  - **Private**: only be accessed by its own methods
  - **Public**: can be modified by methods associated with any class (violates encapsulation)
- **Methods** can be public, private or protected:
  - **Public**: it’s name is exposed to other objects.
  - **Private**: it can’t be accessed by other objects, only internally
  - **Protected**: (special case) only subclasses that descend directly from a class that contains it, know and can use this method.

INHERITANCE (REUSE)

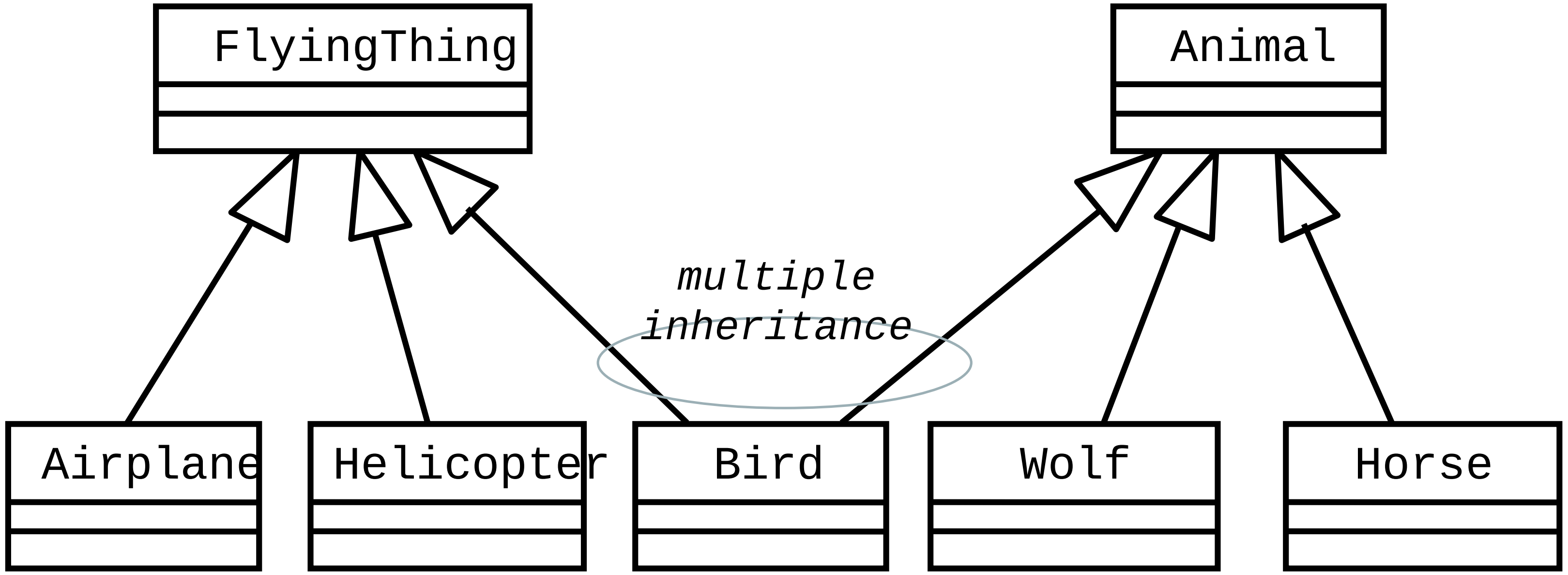
- **Inheritance** is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities.
  - The existing classes :
    - Base classes/ Parent classes/ Super-classes,
  - New classes :
    - Derived classes/ Child classes/ Subclasses.



Single Inheritance



Multiple Inheritance

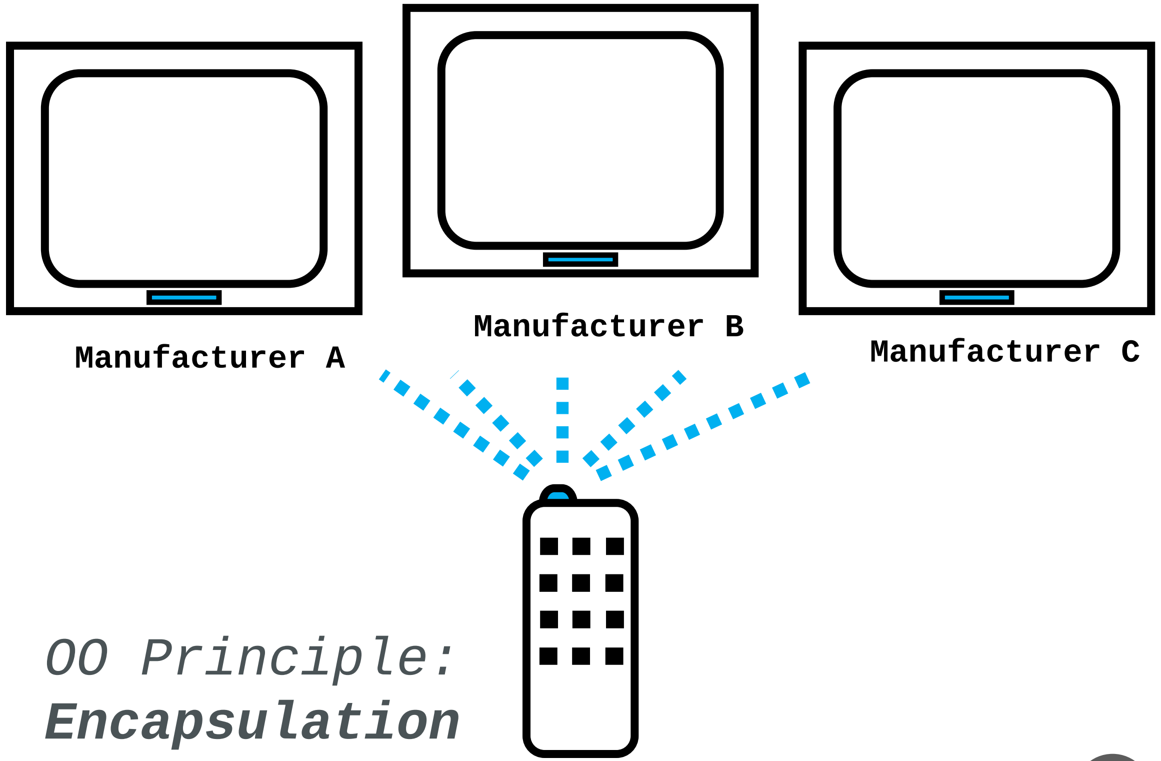


POLYMORPHISM

- The ability to hide many **different implementations** behind a single interface.

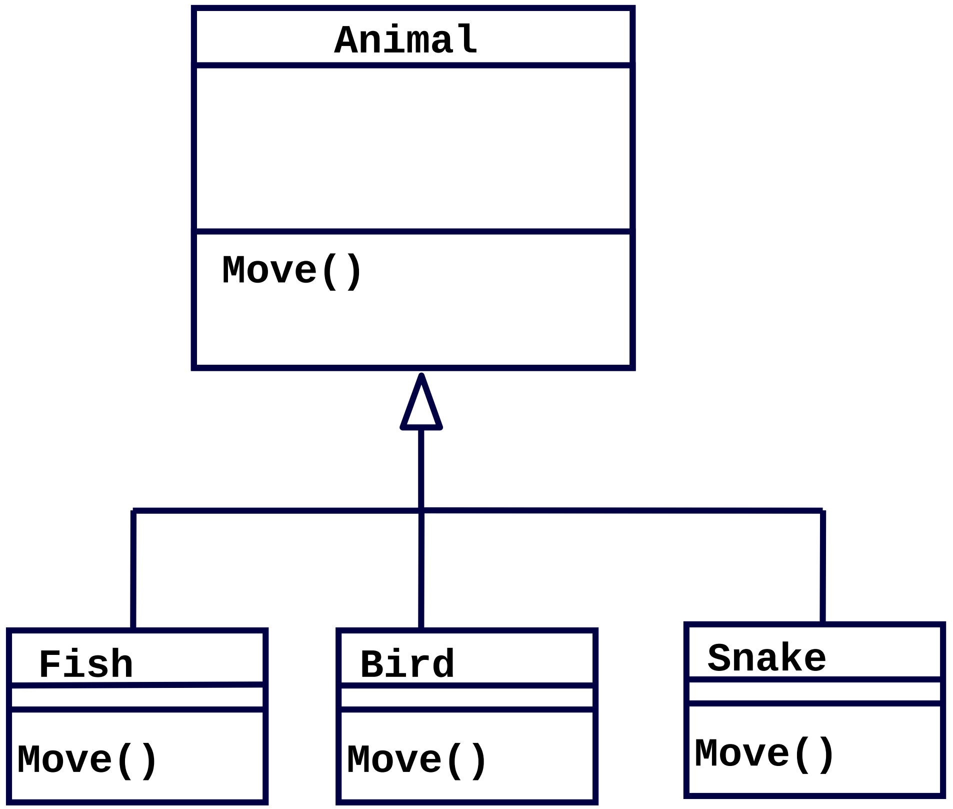
The **same method** will behave differently when it is applied to the objects of **different classes**.

In the same way, the **different methods** associated with different classes can interpret the **same message** in different ways.



OO Principle:
Encapsulation

EXAMPLE


Animal
Move()
Fish
Move()
Bird
Move()
Snake
Move()