



## Lecture 4

# OBJECT ORIENTED ANALYSIS

PB007 Software Engineering I  
Faculty of Informatics, Masaryk University  
Fall 2019

# Outline



- ✧ UML Objects and classes [Lecture 3]
- ✧ Finding analysis classes [Lecture 3]
- ✧ Relationships between objects and classes
  - Links
  - Associations
  - Dependencies
- ✧ Inheritance and polymorphism
- ✧ UML State diagram



---

# Relationships Between Objects and Classes

## Lecture 4/Part 1

# What is a link?



- ✧ Links are connections between objects
  - Think of a link as a **telephone line** connecting you and a friend. You can send messages back and forth using this link
- ✧ Links are the way that objects communicate
  - Objects **send messages to each other** via links
  - Messages **invoke operations**
- ✧ OO programming languages implement links as object references or pointers
  - When an object has a stored reference to another object, we say that there is a **link** between the objects

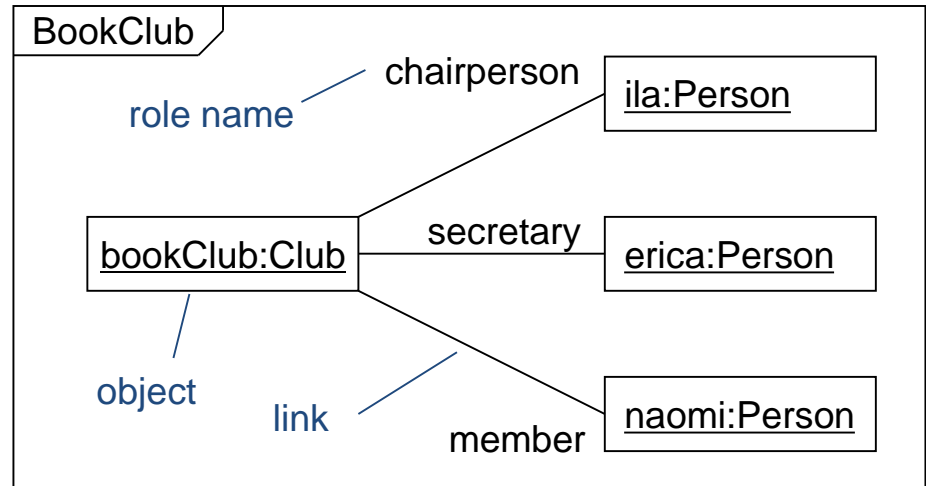
# Object diagrams



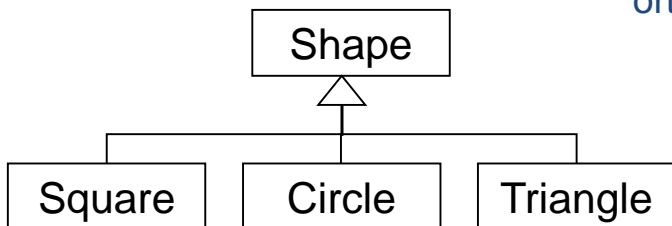
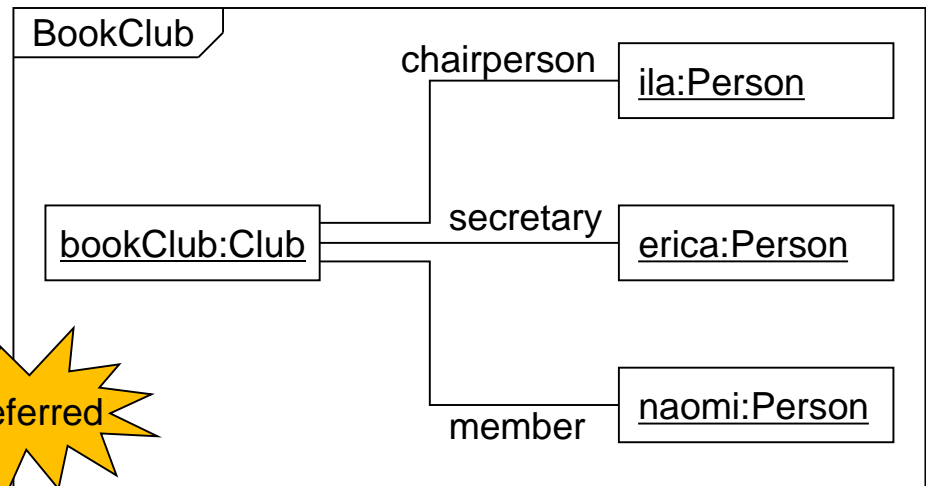
❖ Paths in UML diagrams can be drawn as orthogonal, oblique or curved lines

❖ We can combine paths into a tree if each path has **the same properties**

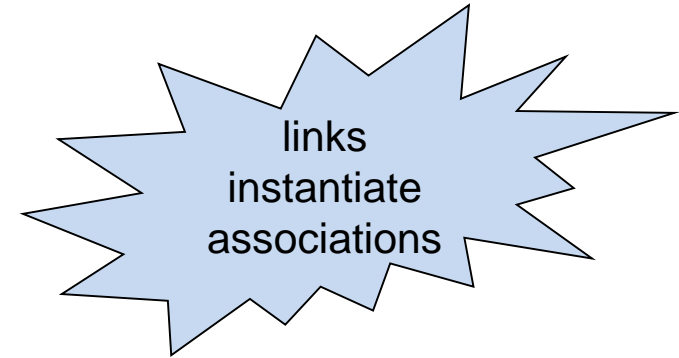
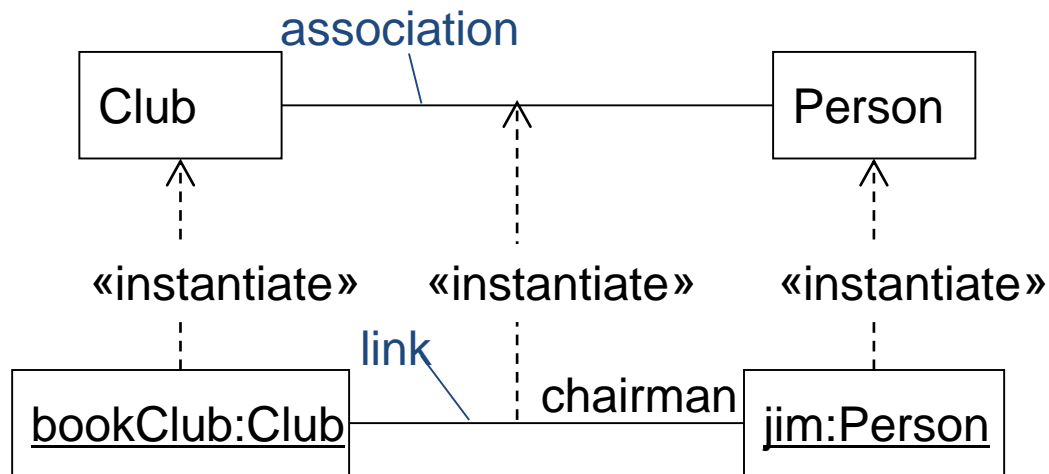
oblique path style



orthogonal path style

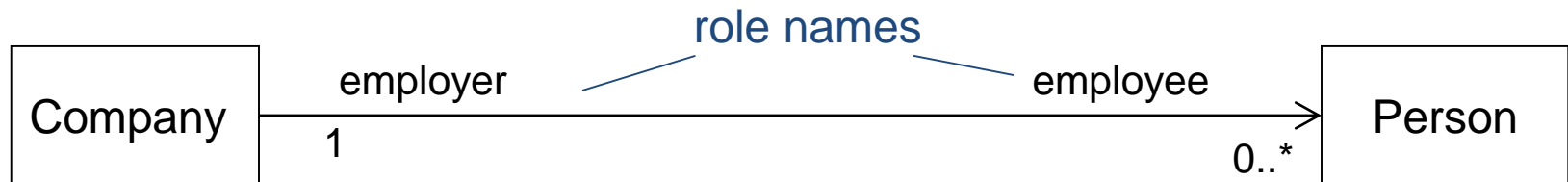


# What is an association?



- ✧ Associations are relationships between classes
- ✧ Associations between classes indicate that there **may be** links between objects of those classes, while links indicates that there **must be** associations
- ✧ Can there be a communication between objects of two classes that have no association between them?

# Association syntax



- ✧ An association can have **role names** OR an **association name**
- ✧ **Multiplicity** is a constraint that specifies the number of objects that can participate in a relationship at any point in time
  - If multiplicity is not explicitly stated in the model then it is undecided – *there is no default multiplicity*

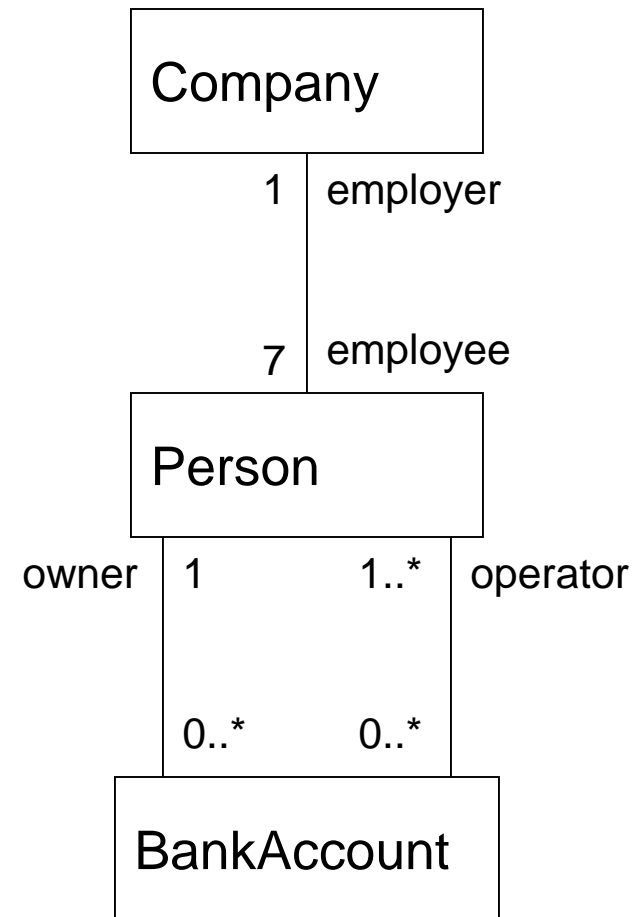
multiplicity: min..max	
0..1	zero or 1
1	exactly 1
0..*	zero or more
1..*	1 or more
1..6	1 to 6

# Multiplicity exercise



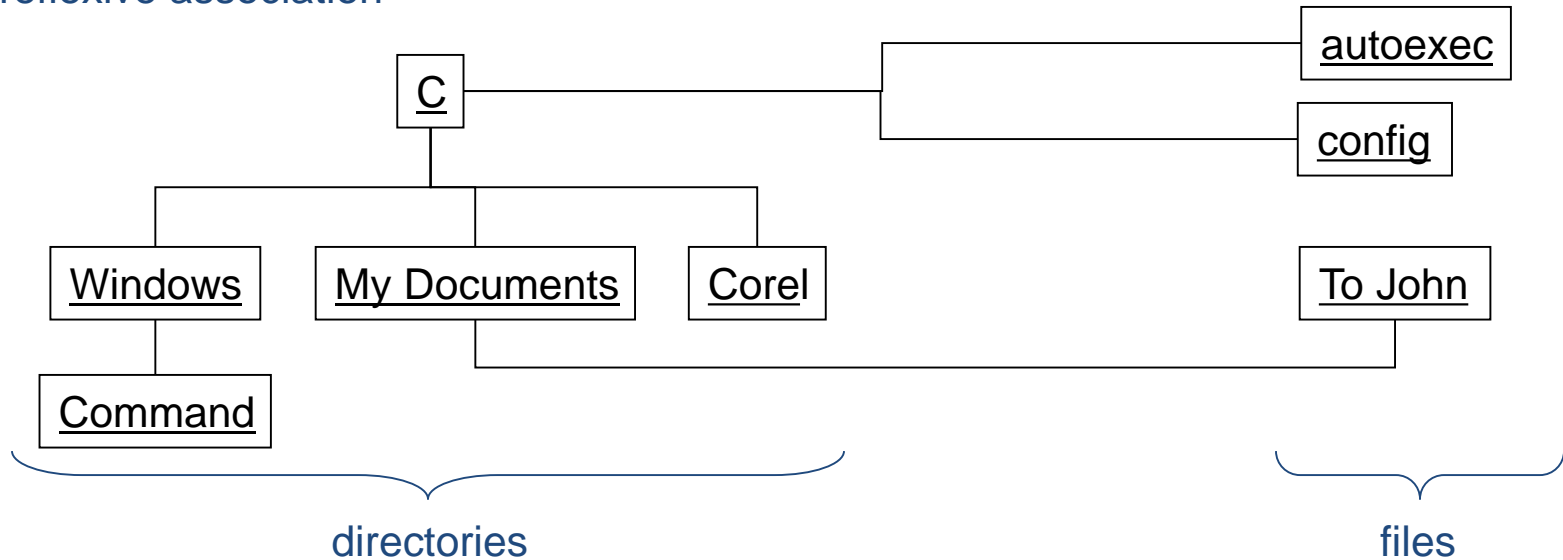
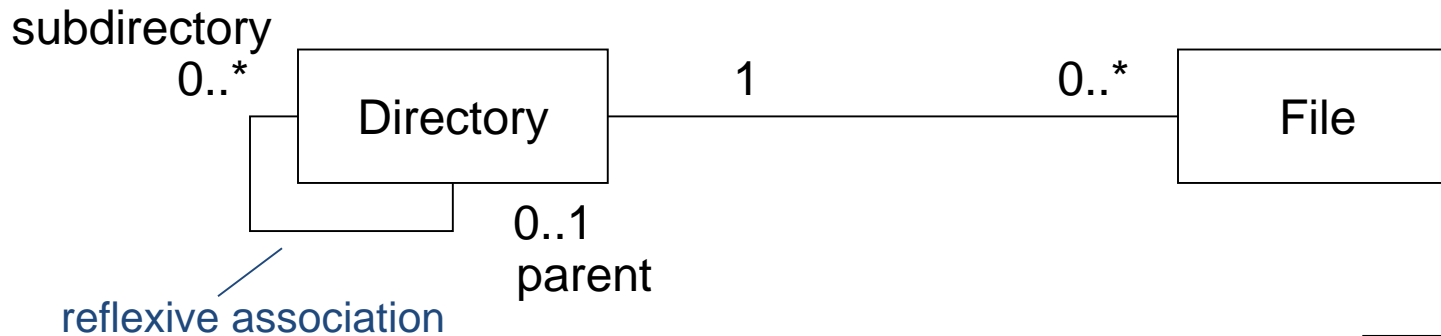
## ✧ How many

- Employees can a Company have?
- Employers can a Person have?
- Owners can a BankAccount have?
- Operators can a BankAccount have?
- BankAccounts can a Person have?
- BankAccounts can a Person operate?





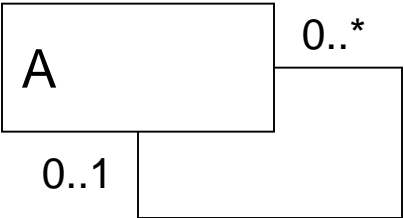
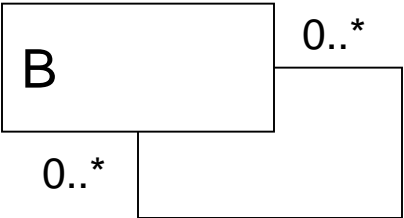
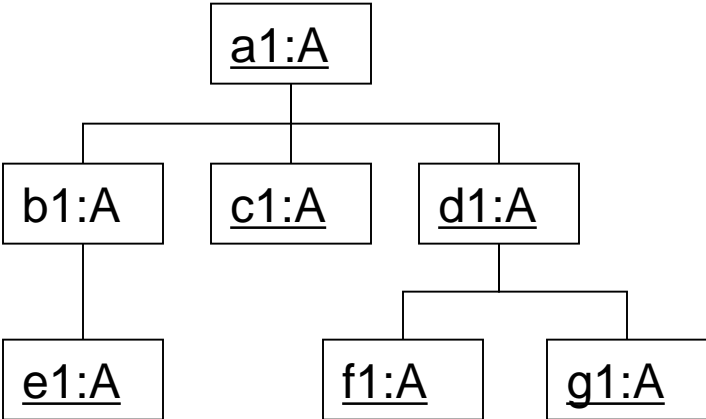
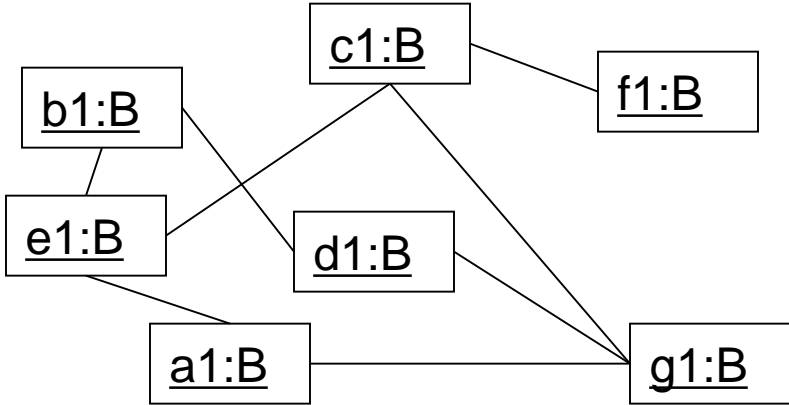
# Reflexive associations: file system example



✧ If ToJohn was a directory, would it still conform to the class diagram?

# Hierarchies and networks



hierarchy	network
	
	
<p>In an association hierarchy, each object has <b>zero or one</b> object directly above it.</p>	<p>In an association network, each object has <b>zero or many</b> objects directly above it.</p>

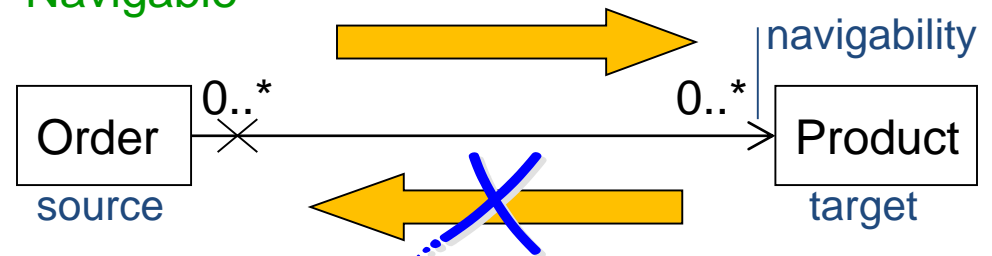
# Navigability



- ✧ Navigability indicates that it is possible to traverse from an object of the **source** class to objects of the **target** class

An Order object stores a list of Products

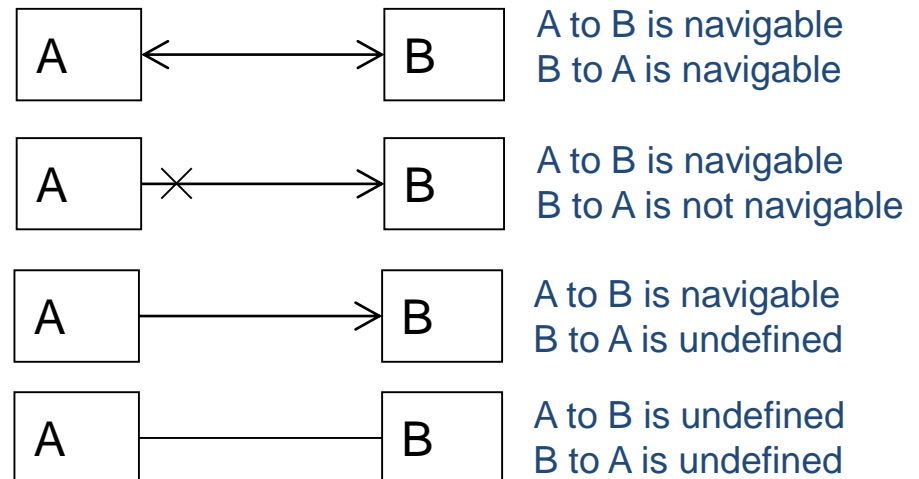
Navigable



Not navigable

A Product object does not store a list of Orders

- ✧ Can there be a communication in a direction not supported by the navigability?
- ✧ Are some of the cases on the right equivalent?

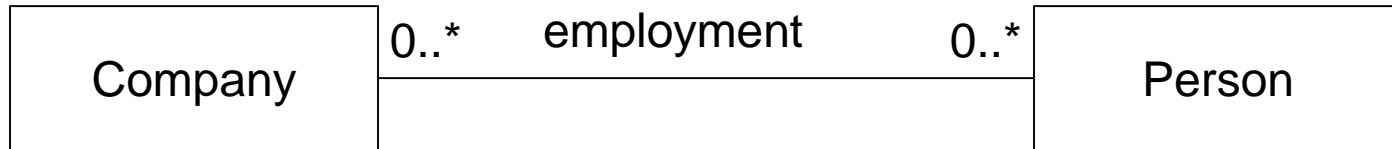


# Associations and attributes

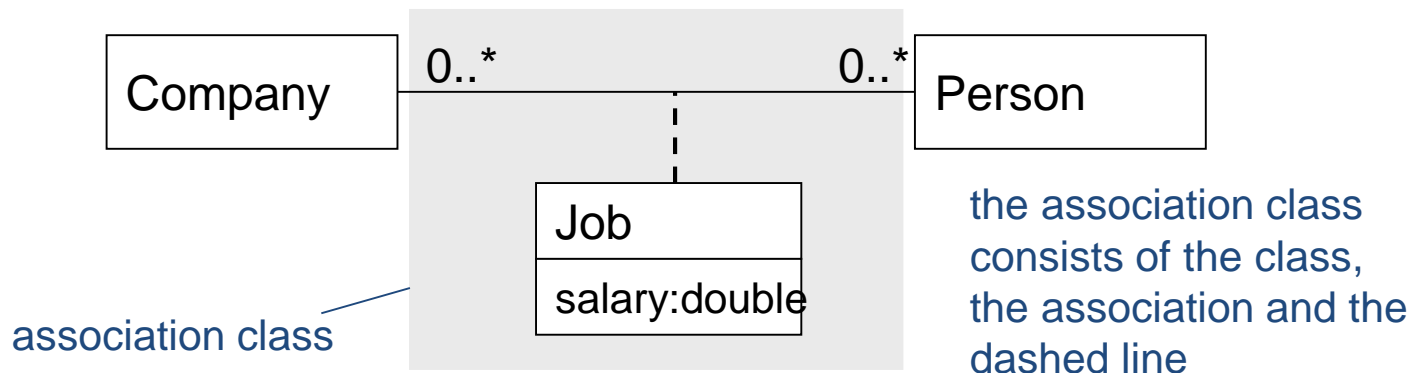


- ✧ An association is (through its role name) a **representation of an attribute**
- ✧ **Use associations when:**
  - The target class is an important part of the model
  - The target class is a class that you have designed yourself
- ✧ **Use attributes when:**
  - The target class is not important, e.g. a primitive type such as number, string
  - The target class is just an implementation detail such as a bought-in component or a library component e.g. `Java.util.Vector` (from the Java standard libraries)

# Association classes



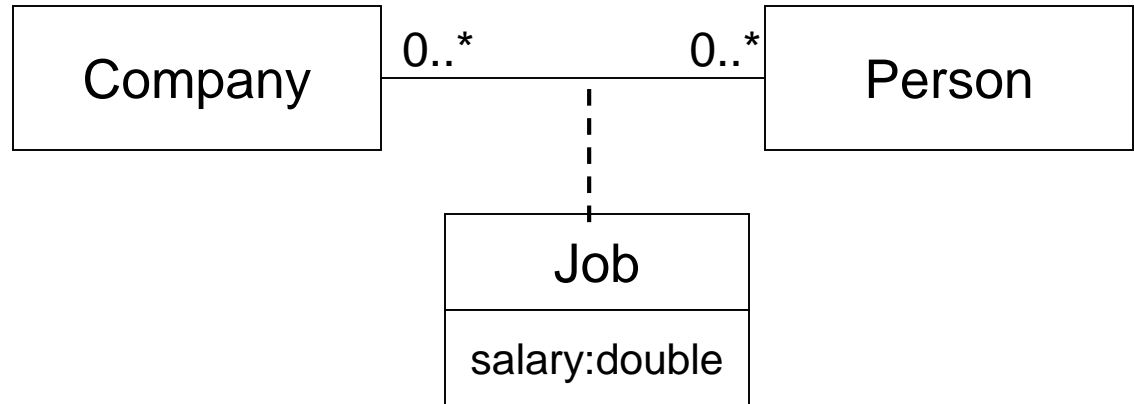
- ✧ Where do we record the Person's salary?
- ✧ We model the association itself as an association class. Exactly one instance of this class exists for each link between a Person and a Company.
- ✧ We can place the salary and any other attributes or operations which are really features of the association into this class



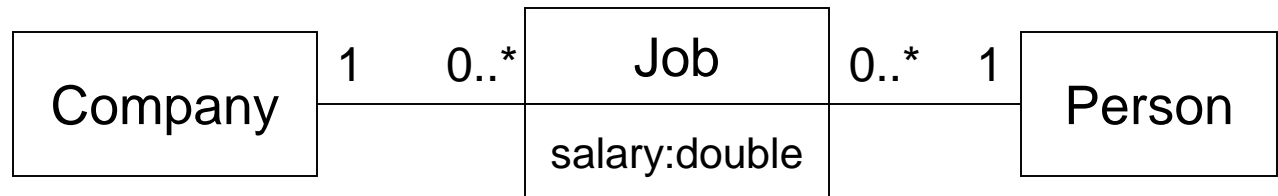
# Using association classes



If we use an association class, then a particular Person can have only **one** Job with a particular Company



If, however a particular Person can have **multiple** jobs with the same Company, then we must use a reified association



# Dependencies



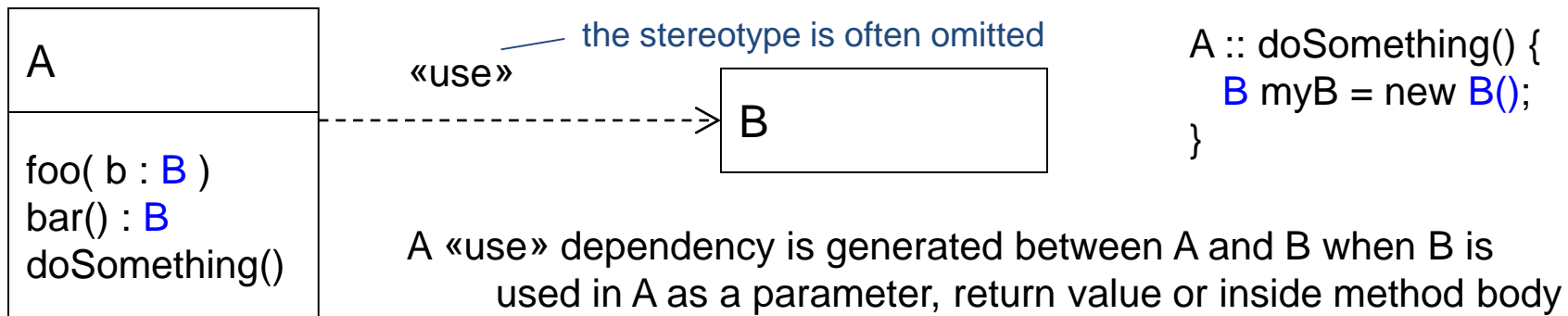
- ✧ "A dependency is a relationship between two elements where a change to one element (the supplier) may affect or supply information needed by the other element (the client)".
  - In other words, the client **depends** in some way on the supplier
  - Weaker type of relationship than **association**
  - Can there be both association and dependency between two classes?
- ✧ Three types of dependency:
  - **Usage** - the client uses some of the services made available by the supplier to implement its own behavior – this is the most commonly used type of dependency
  - **Abstraction** - a shift in the level of abstraction. The supplier is more abstract than the client
  - **Permission** - the supplier grants some sort of permission for the client to access its contents – this is a way for the supplier to control and limit access to its contents

# Usage dependencies



## ❖ Stereotypes

- «**use**» - the client makes use of the supplier to implement its behaviour
- «**call**» - the client operation invokes the supplier operation
- «**parameter**» - the supplier is a parameter of the client operation
- «**send**» - the client (an operation) sends the supplier (a signal) to some unspecified target
- «**instantiate**» - the client is an instance of the supplier





# Abstraction and permission dependencies



## ✧ Abstraction dependencies

- «**trace**» - the client and the supplier represent the same concept but at different points in development
- «**substitute**» - the client may be substituted for the supplier at runtime. The client and supplier must realize a common contract. Use in environments that *don't* support specialization/generalization
- «**refine**» - the client represents a fuller specification of the supplier
- «**derive**» - the client may be derived from the supplier. The client is logically redundant, but may appear for implementation reasons

## ✧ Permission dependencies

- «**access**» the public contents of the supplier package are added as private elements to the namespace of the client package
- «**import**» the public contents of the supplier package are added as public elements to the namespace of the client package
- «**permit**» the client element has access to the supplier element despite the declared visibility of the supplier

# Key points



- ✧ Links – relationships between objects
- ✧ Associations – relationships between classes
  - role names
  - multiplicity
  - navigability
  - association classes
- ✧ Dependencies – relationships between model elements
  - usage
  - abstraction
  - permission



---

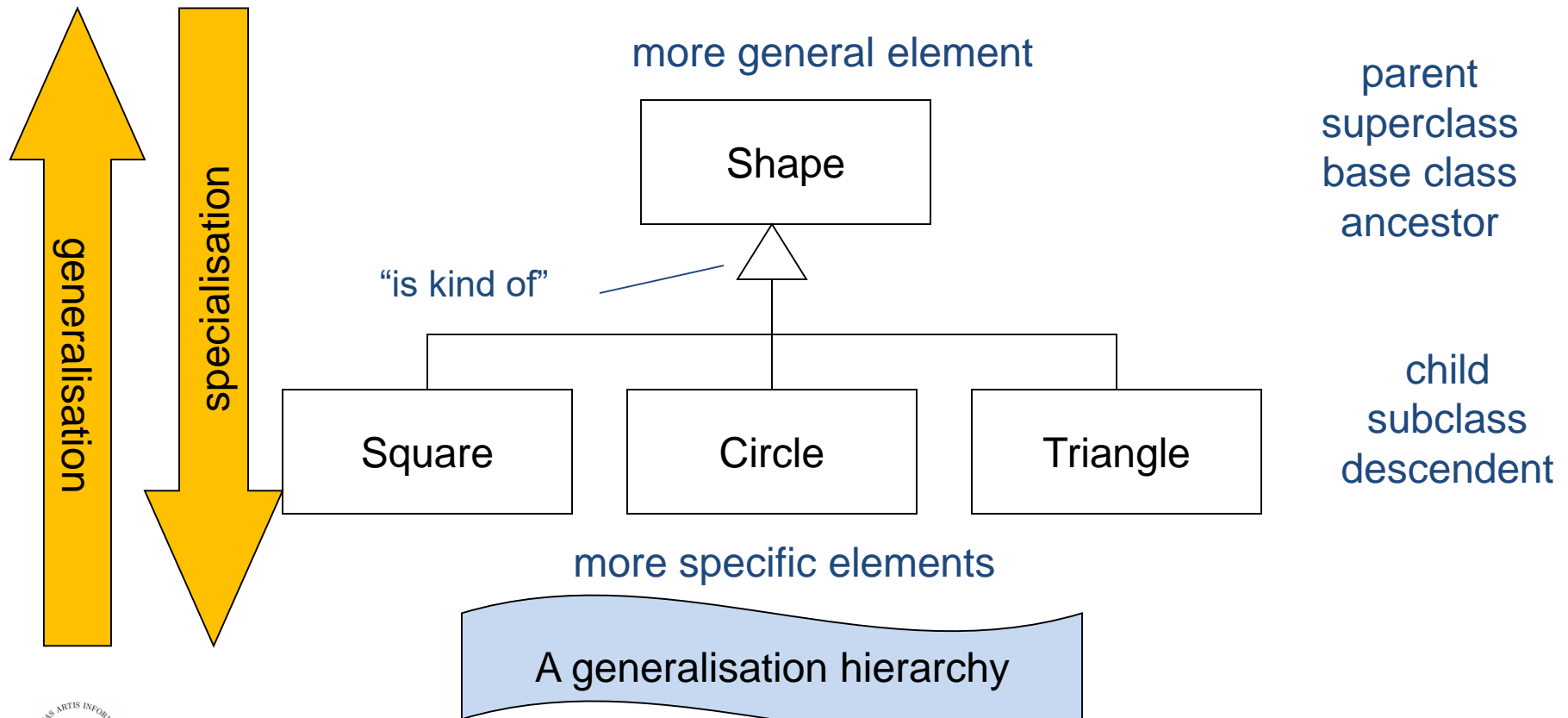
# Inheritance and polymorphism

## Lecture 4/Part 2

# Generalisation



A relationship between a more general element and a more specific element (with more information)

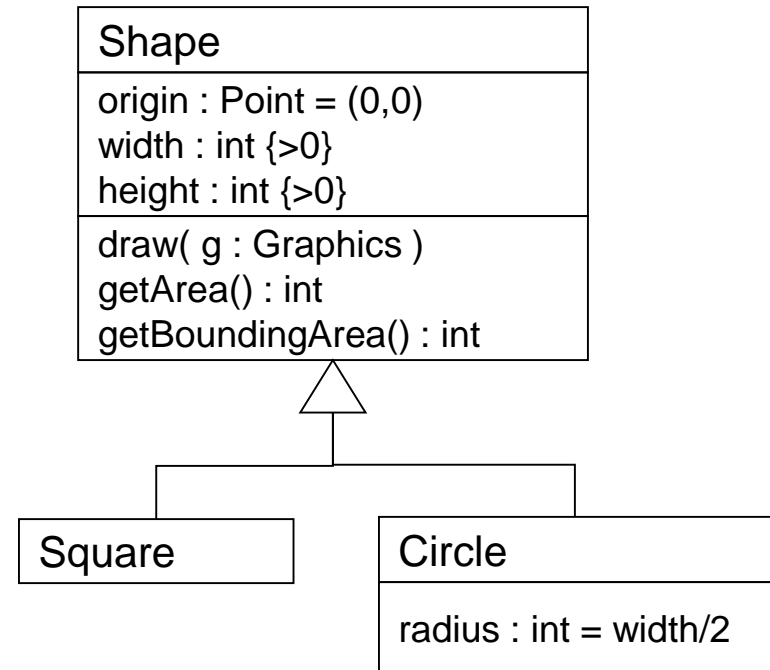


# Class inheritance



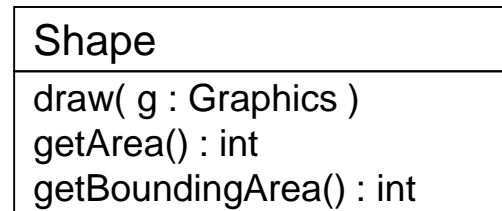
- ❖ Subclasses inherit **all** features of their superclasses:
  - attributes
  - operations
  - relationships
  - stereotypes, tags, constraints
- ❖ Subclasses can add new features
- ❖ Subclasses can override superclass operations
- ❖ We can use a subclass instance **anywhere** a superclass instance is expected

Substitutability Principle

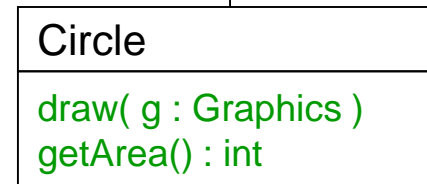
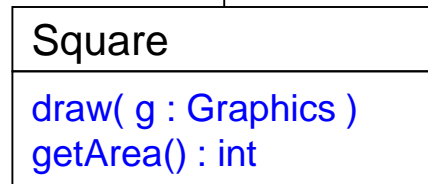
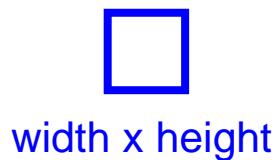


What's wrong with these subclasses?

# Overriding



What's wrong with the superclass?

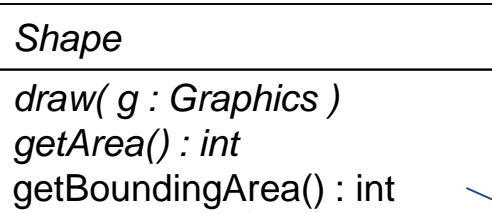


- ✧ Subclasses often need to **override** superclass behaviour
- ✧ To override a superclass operation, a subclass must provide an operation with the same signature
  - The operation signature is the operation name, return type and types of all the parameters

# Abstract operations & classes



abstract class

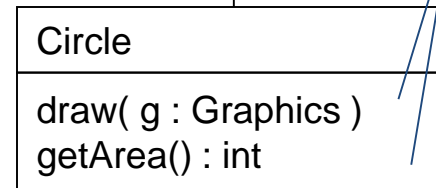
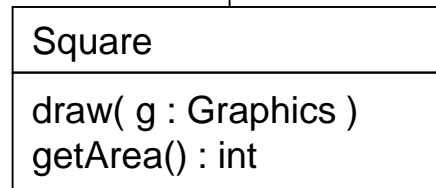


abstract operations

Abstract class and operation names must be in italics

concrete operations

concrete classes

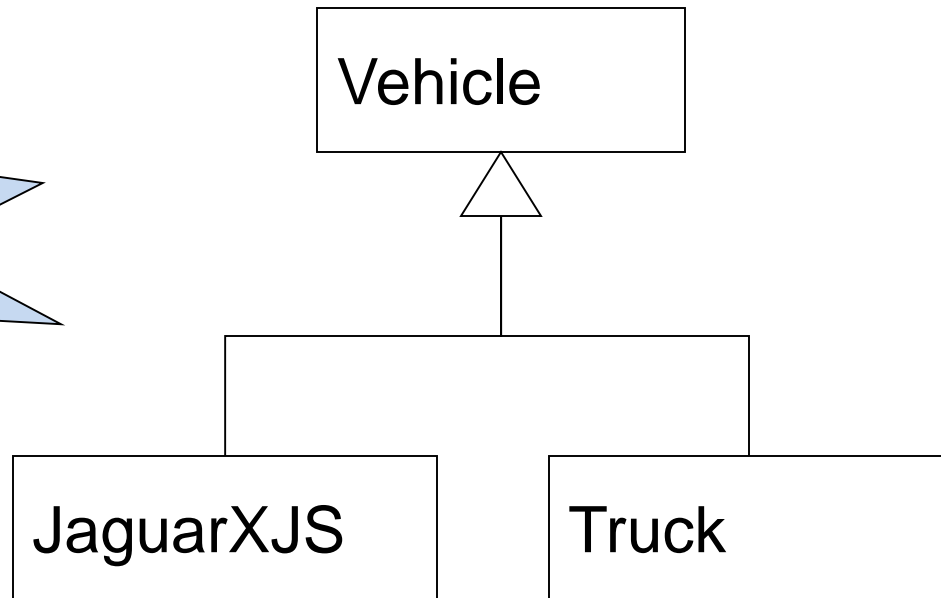


- ✧ We can't provide an implementation for *Shape :: draw(g : Graphics)* or for *Shape :: getArea() : int* because we don't know how to draw or calculate the area for a "shape"!
- ✧ Operations that lack an implementation are **abstract operations**
- ✧ A class with any abstract operations **can't be instantiated** and is therefore an **abstract class**

# Exercise



what's wrong with this model?





# Polymorphism

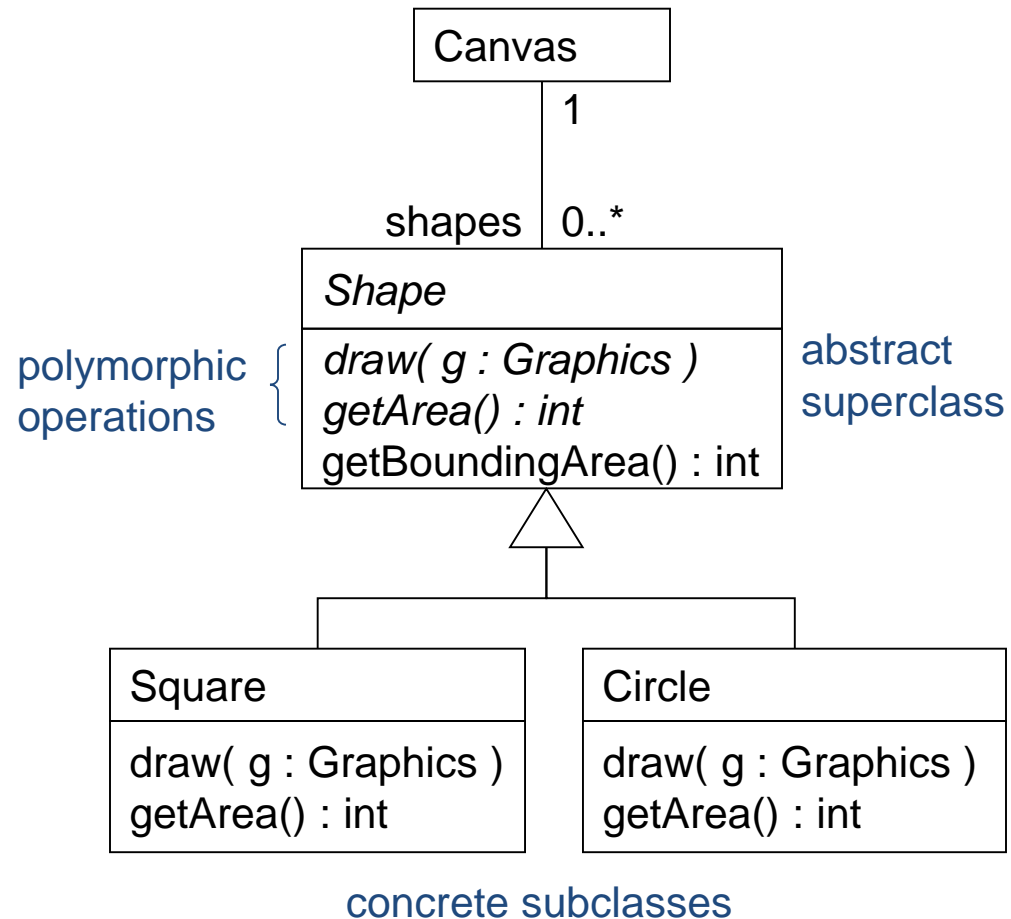


✧ Polymorphism = "many forms"

- A polymorphic operation has many implementations
- Square and Circle provide implementations for the polymorphic operations *Shape::draw()* and *Shape::getArea()*

✧ The operation in Shape superclass defines a contract for the subclasses.

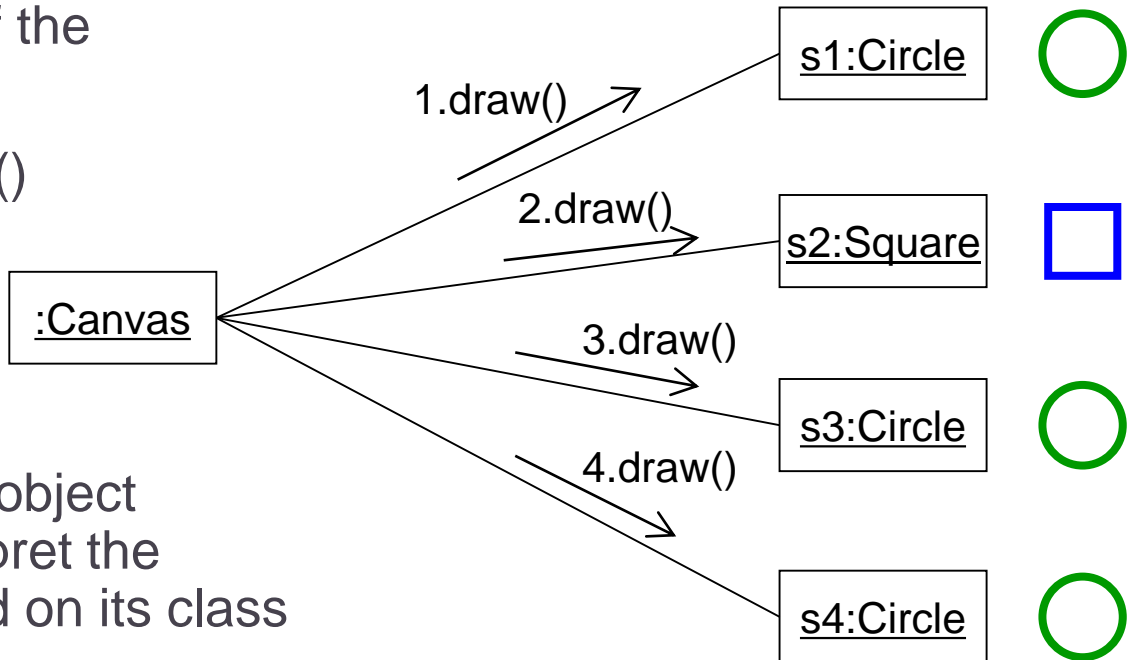
A Canvas object has a collection of *Shape* objects where each *Shape* may be a Square or a Circle



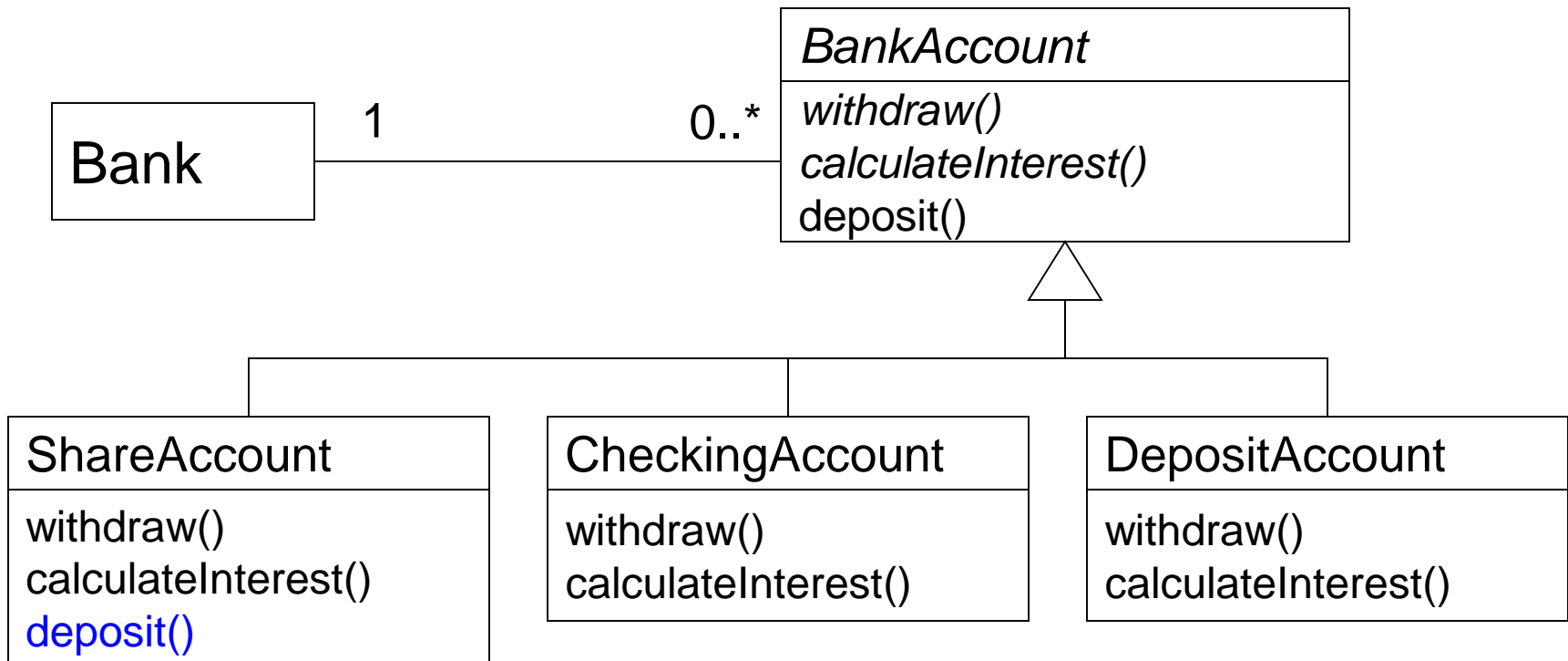
# What happens?



- ✧ Each class of object has its own implementation of the draw() operation
- ✧ On receipt of the draw() message, each object invokes the draw() operation specified by its class
- ✧ We can say that each object "decides" how to interpret the draw() message based on its class



# BankAccount example



- ✧ We have overridden the *deposit()* operation even though it is **not** abstract.

# Key points



✧ Generalisation, specialisation, inheritance

✧ Subclasses

- inherit all features from their parents including constraints and relationships
- may add **new features**, constraints and relationships
- may **override** superclass operations

✧ A class that can't be instantiated is an abstract class



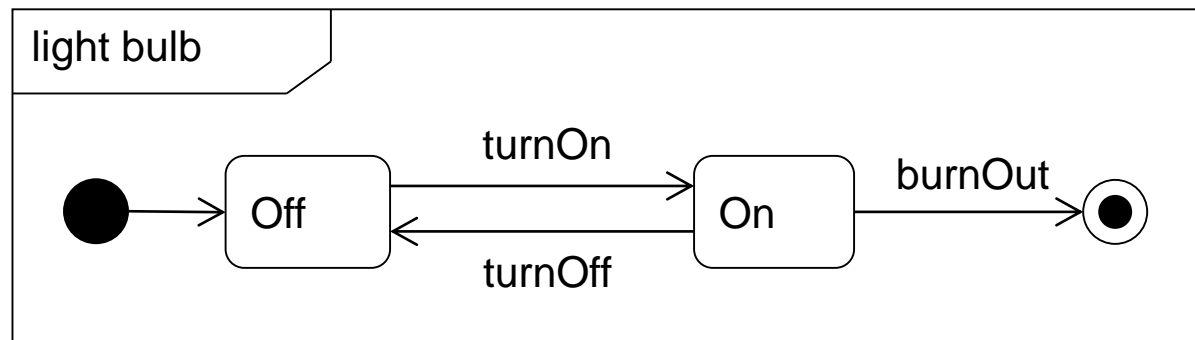
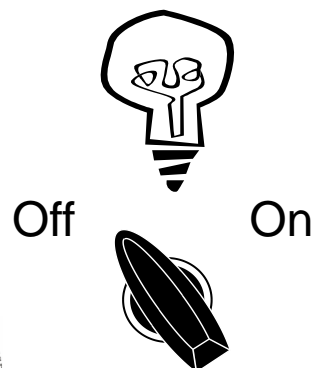
# UML State Diagram

## Lecture 4/Part 3

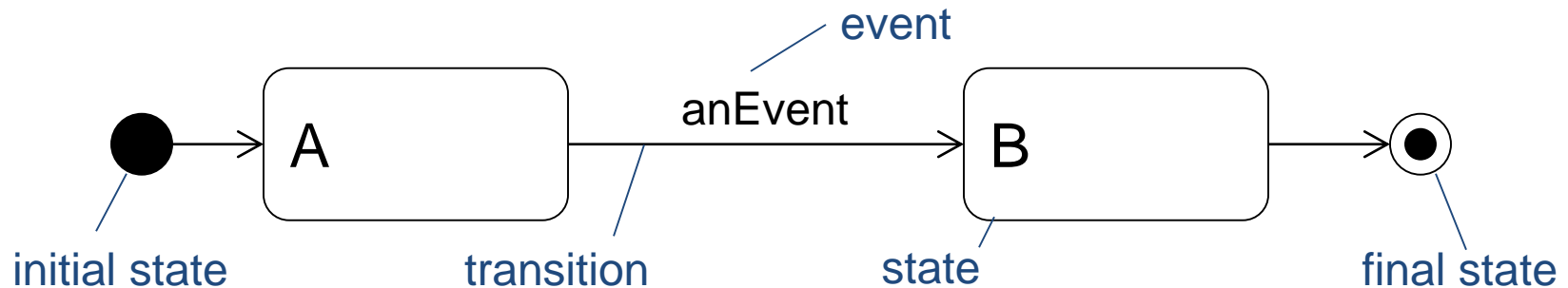
# State machines



- ✧ Models life stages of a **single** model element – e.g. object, use case, module
- ✧ Every state machine exists in the context of a particular model element that:
  - Has a clear life history modelled as a progression of **states**, **transitions** and **events**
  - Responds to events dispatched from outside of the element
- ✧ There are two types of state machines:
  - **Behavioural state machines** - define the behaviour of a model element
  - **Protocol state machines** - model the protocol of a classifier
    - E.g. call conditions and call ordering of an interface that itself has no behaviour



# Basic state machine syntax



✧ State = a situation or condition during the life of an object

- Determined at any point in time by the **values of its attributes**, the relationships to other objects, or the **activities** it is performing.

✧ Every state machine should have one initial state which indicates the first state of the sequence

✧ Unless the states cycle endlessly, state machines should have a final state which terminates its lifecycle

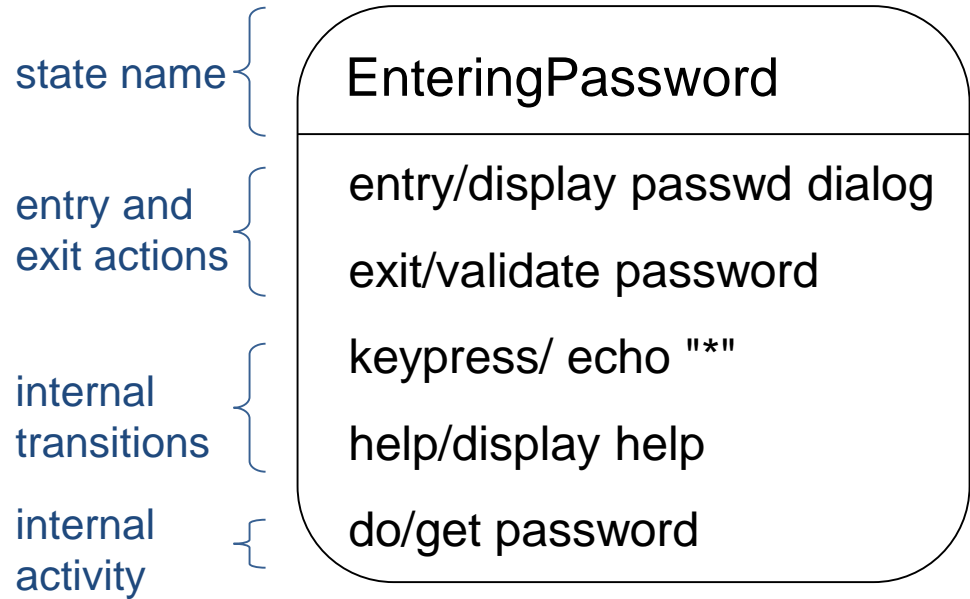
How many states?

Color
red : int green : int blue : int

# State syntax



- ✧ Actions are **instantaneous** and **uninterruptible**
  - Entry actions occur immediately on state entry
  - Exit actions occur immediately on state leaving
- ✧ Internal transitions occur **within** the state. They do not fire transition to a new state
- ✧ Activities take a finite amount of time and are interruptible



Action syntax: eventTrigger / action  
Activity syntax: do / activity

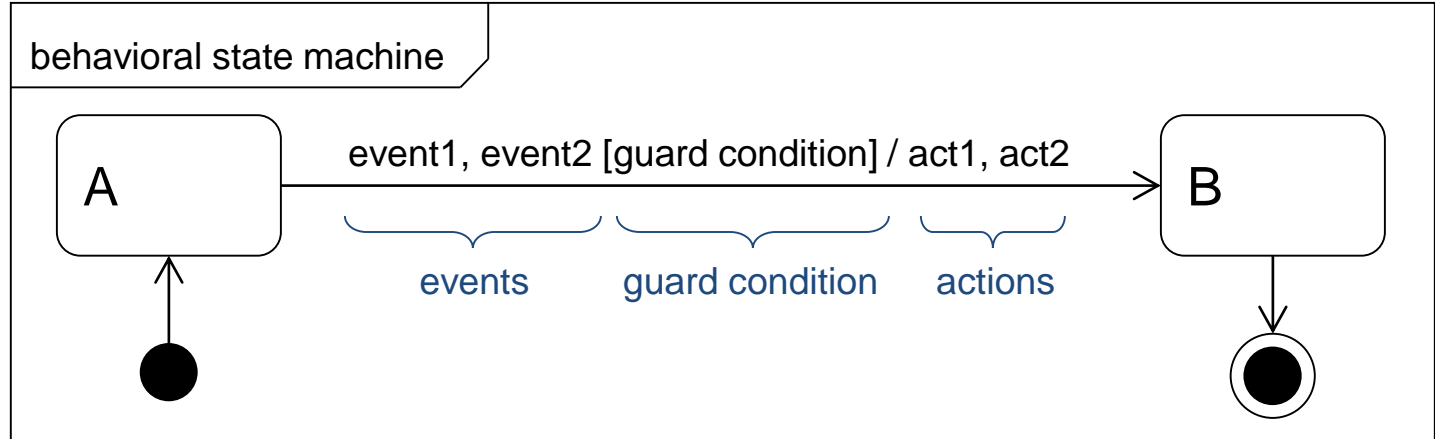


# Transitions



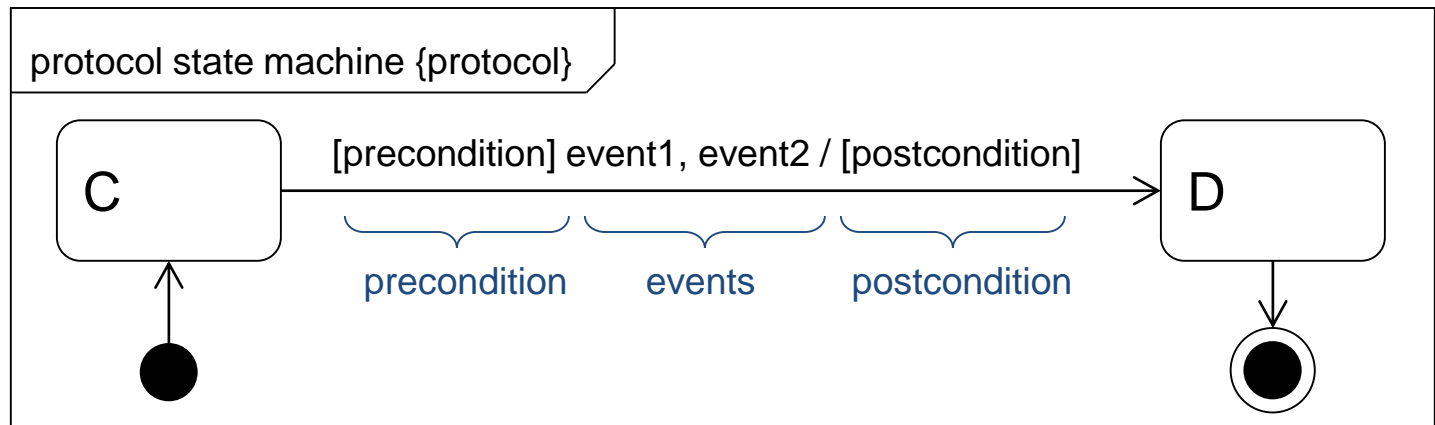
## Behavioral state machine

Specifies object's reactions to events.



## Protocol state machine

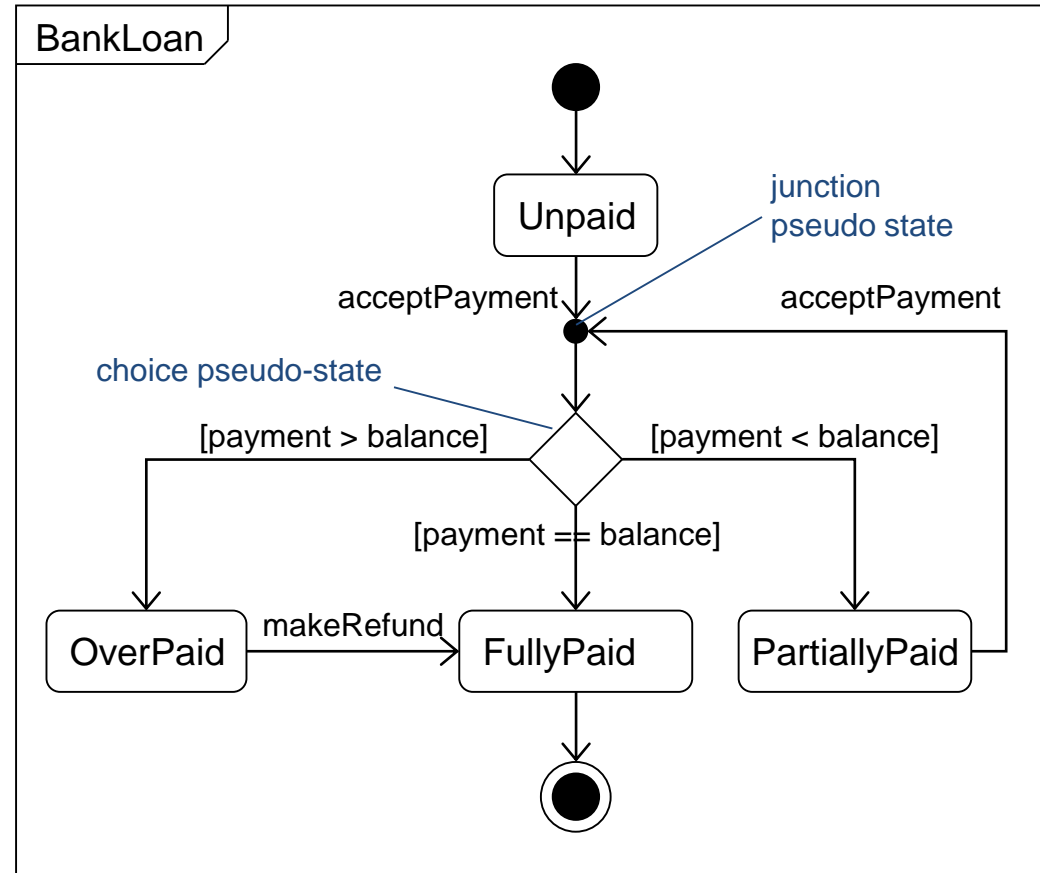
Specifies legal sequences of events.



# Choice and junction pseudo states



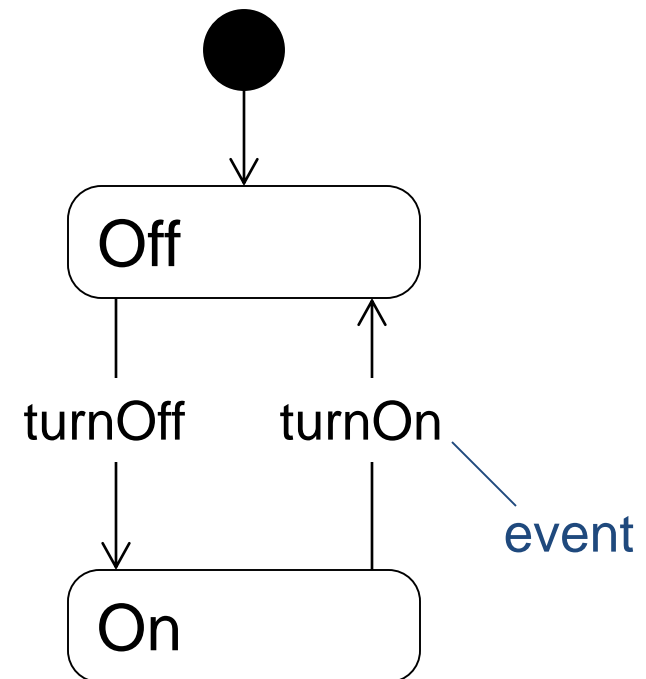
- ✧ **Choice pseudo state** directs its single incoming transition to one of its outgoing transitions
  - Each outgoing transition must have a mutually exclusive guard condition
  - Equivalent to two outgoing transitions from one state
- ✧ **Junction pseudo state** connects multiple incoming transitions into one (or more) transitions.
  - When there are more outgoing transitions, they must have guard conditions



# Events



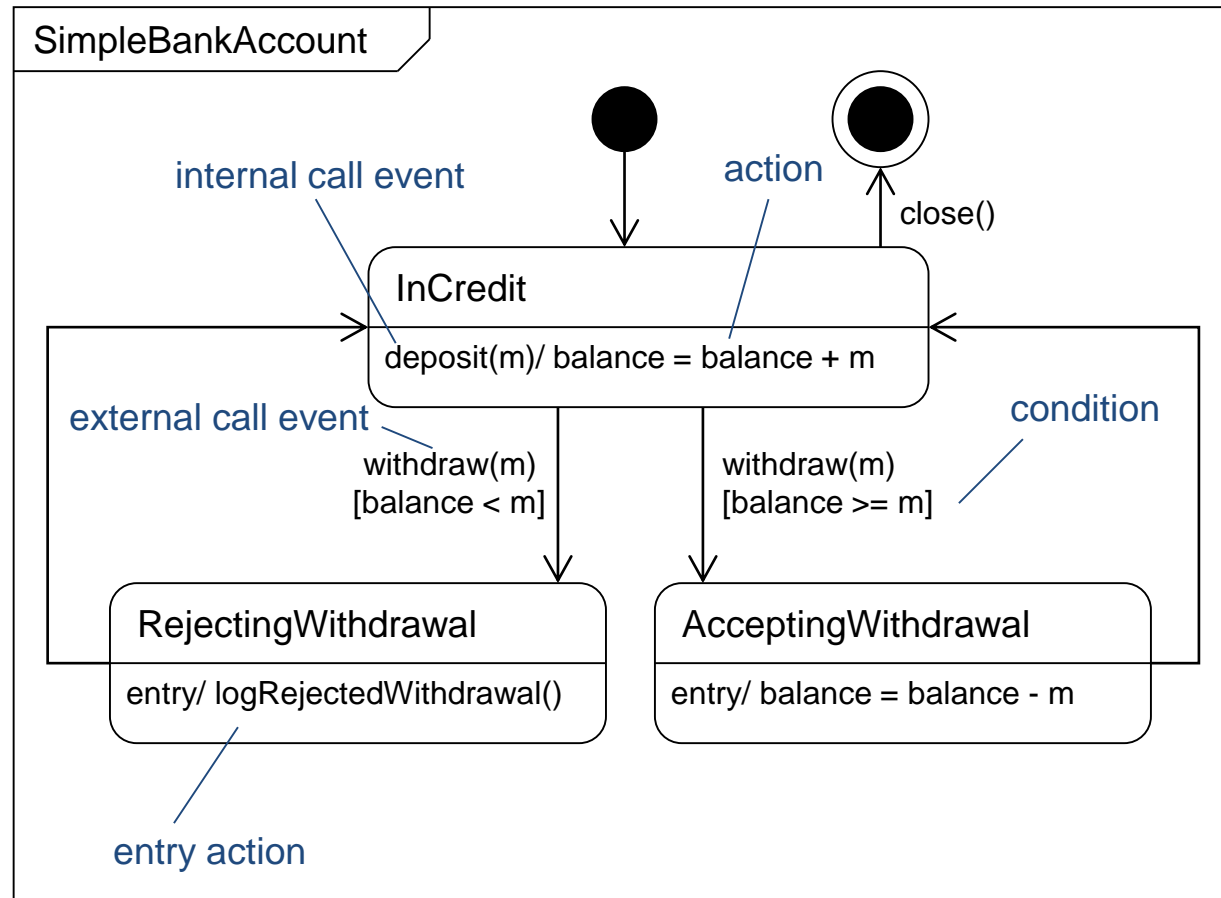
- ✧ "The specification of a noteworthy occurrence that has location in time and space"
- ✧ Events trigger transitions in state machines
- ✧ Events can be shown externally, on transitions, or internally within states (internal transitions)
- ✧ There are four types of event:
  - Call event
  - Signal event
  - Change event
  - Time event



# Call event



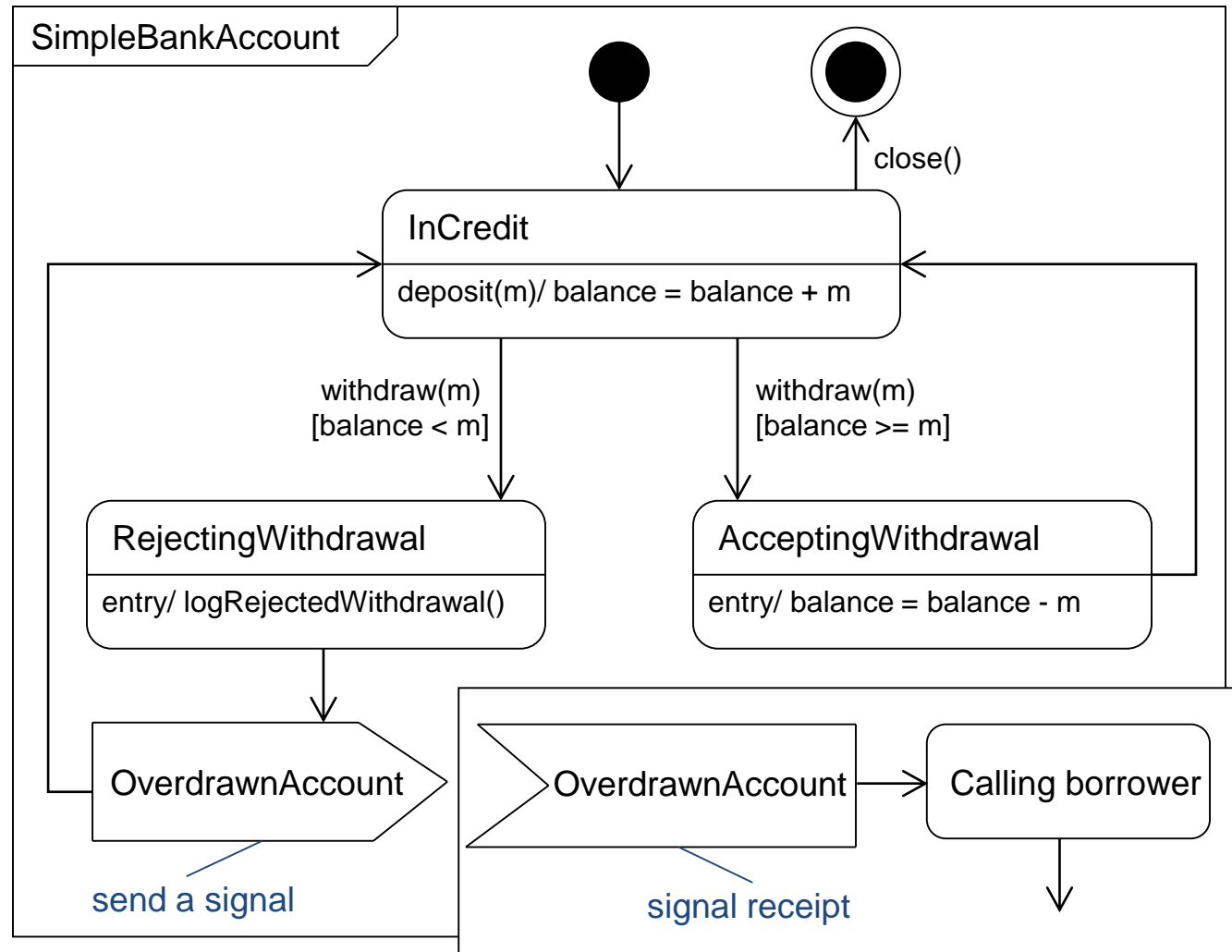
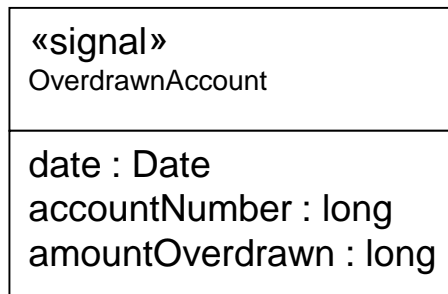
- ✧ A call for an operation execution
- ✧ The event should have the same signature as an operation of the context class
- ✧ A sequence of actions may be specified for a call event - they may use attributes and operations of the context class
- ✧ The return value must match the return type of the operation



# Signal events



✧ A signal is a package of information that is sent asynchronously between objects



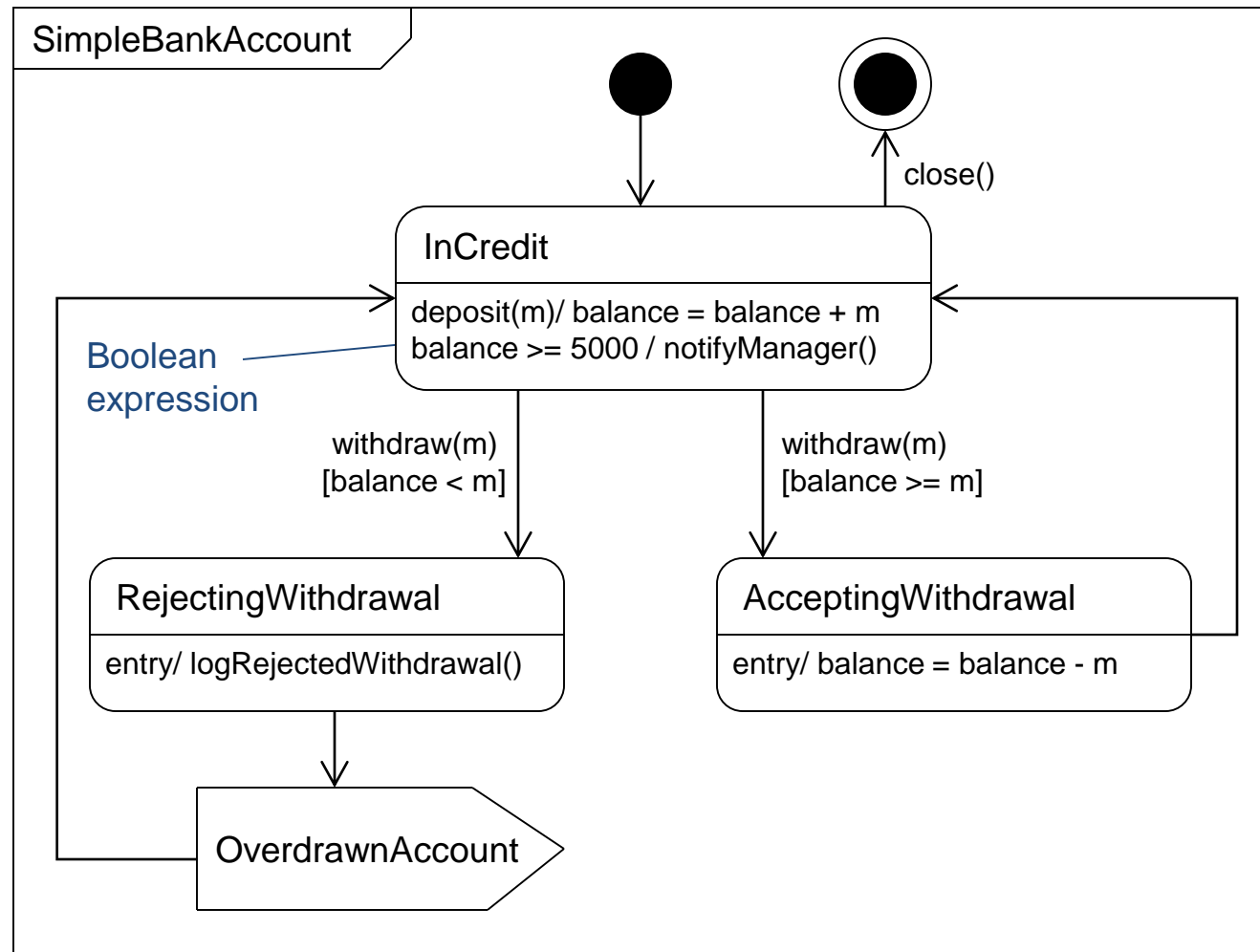
# Change events



✧ The action is performed when the Boolean expression transitions from false to true

- The event is **edge triggered** on a false to true transition
- The values in the Boolean expression must be constants, globals or attributes of the context class

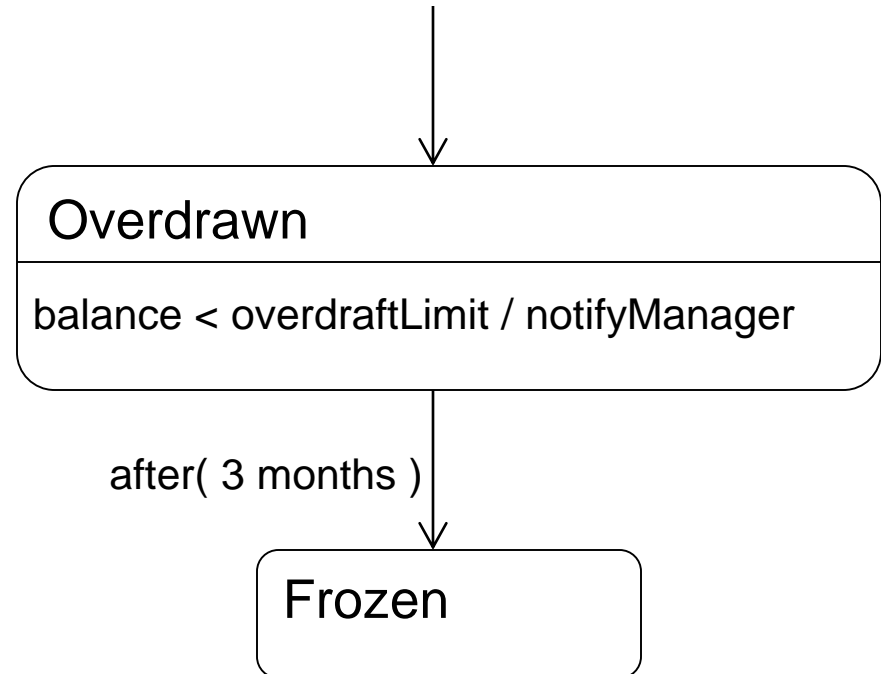
✧ A change event implies continually testing the condition whilst in the state



# Time events



- ✧ Time events occur when a time expression becomes true
- ✧ There are two keywords, **after** and **when**
- ✧ Elapsed time:
  - `after( 3 months )`
- ✧ Absolute time:
  - `when( date =20/3/2000)`



Context: CreditAccount class

# Composite states

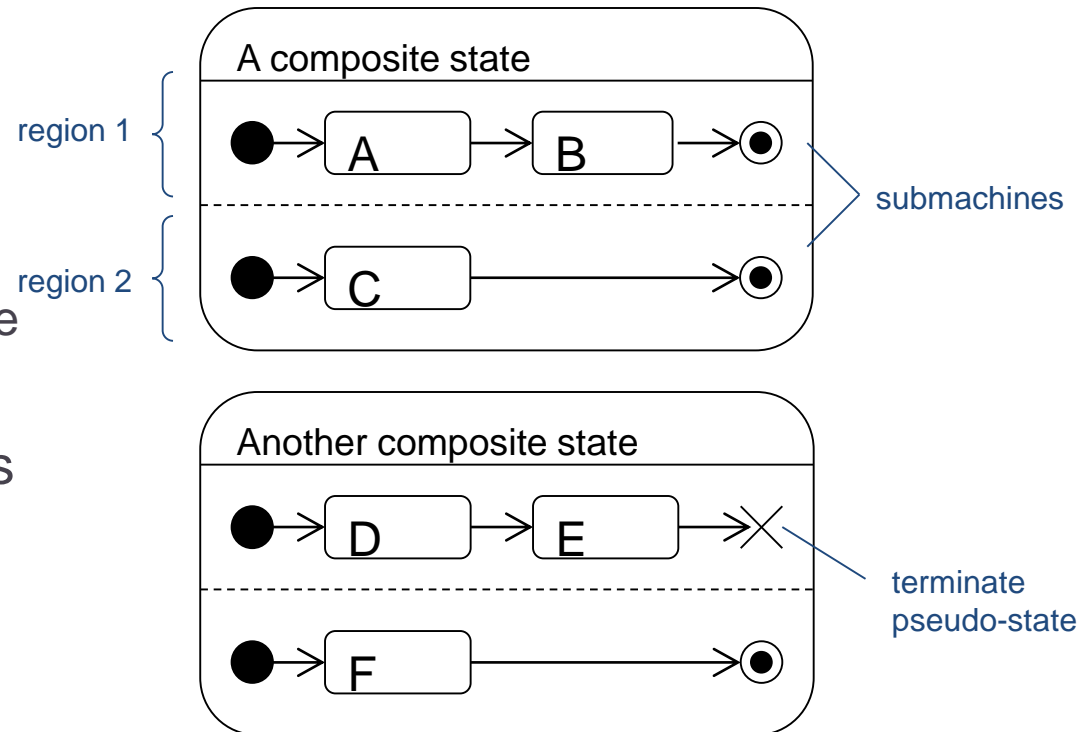


✧ Have one or more regions that each contain a nested submachine

- Simple composite state
  - exactly one region
- Orthogonal composite state
  - two or more regions

✧ The final state terminates its enclosing region – all other regions continue to execute

✧ The terminate pseudo-state terminates the whole state machine



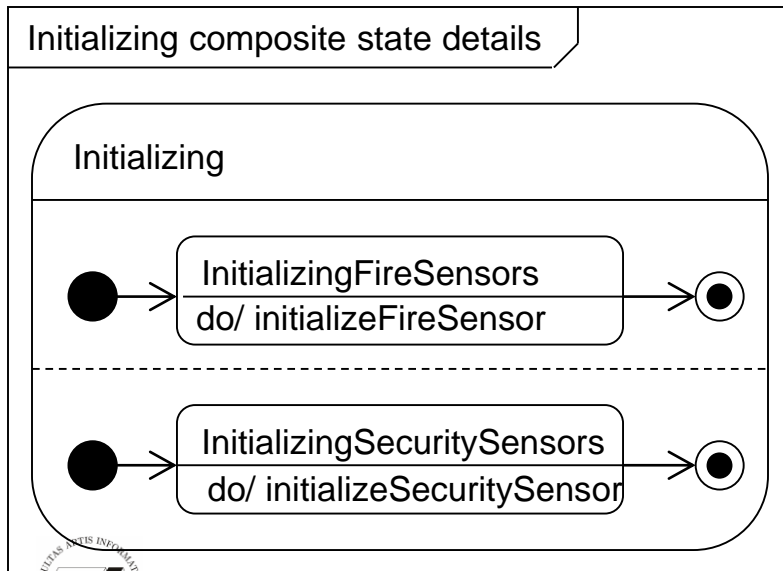


# Orthogonal composite states

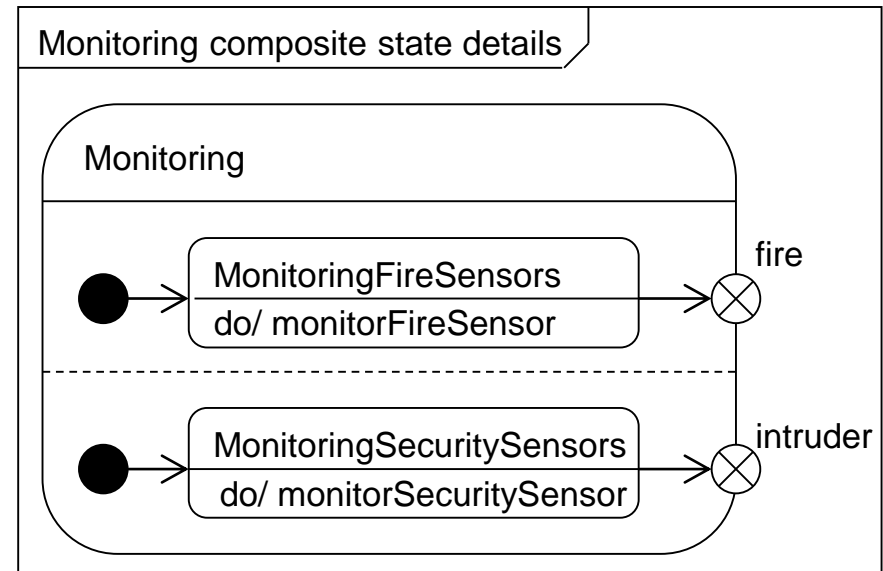


- ✧ Has two or more regions
- ✧ When we enter the superstate, both submachines start executing concurrently - this is an implicit fork

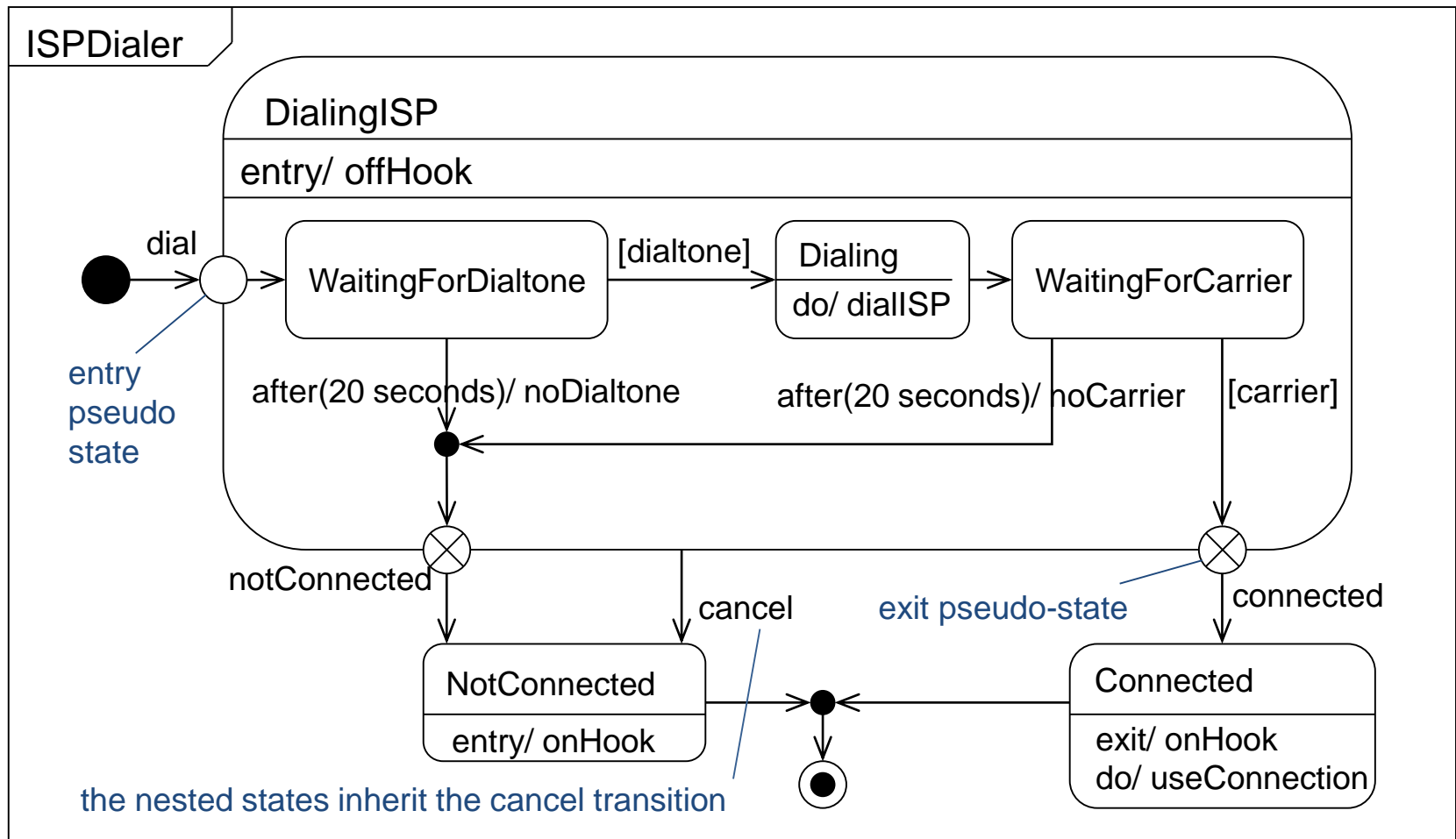
Synchronized exit - exit the superstate when *both* regions have terminated



Unsynchronized exit - exit the superstate when *either* region terminates. The other region continues



# Simple composite states



# Key points



- ✧ Behavioral and protocol state machines
- ✧ States
  - Initial and final
  - Exit and entry actions, activities
- ✧ Transitions
  - Guard conditions, actions
- ✧ Events
  - Call, signal, change and time
- ✧ Composite states
  - Simple and orthogonal composite states