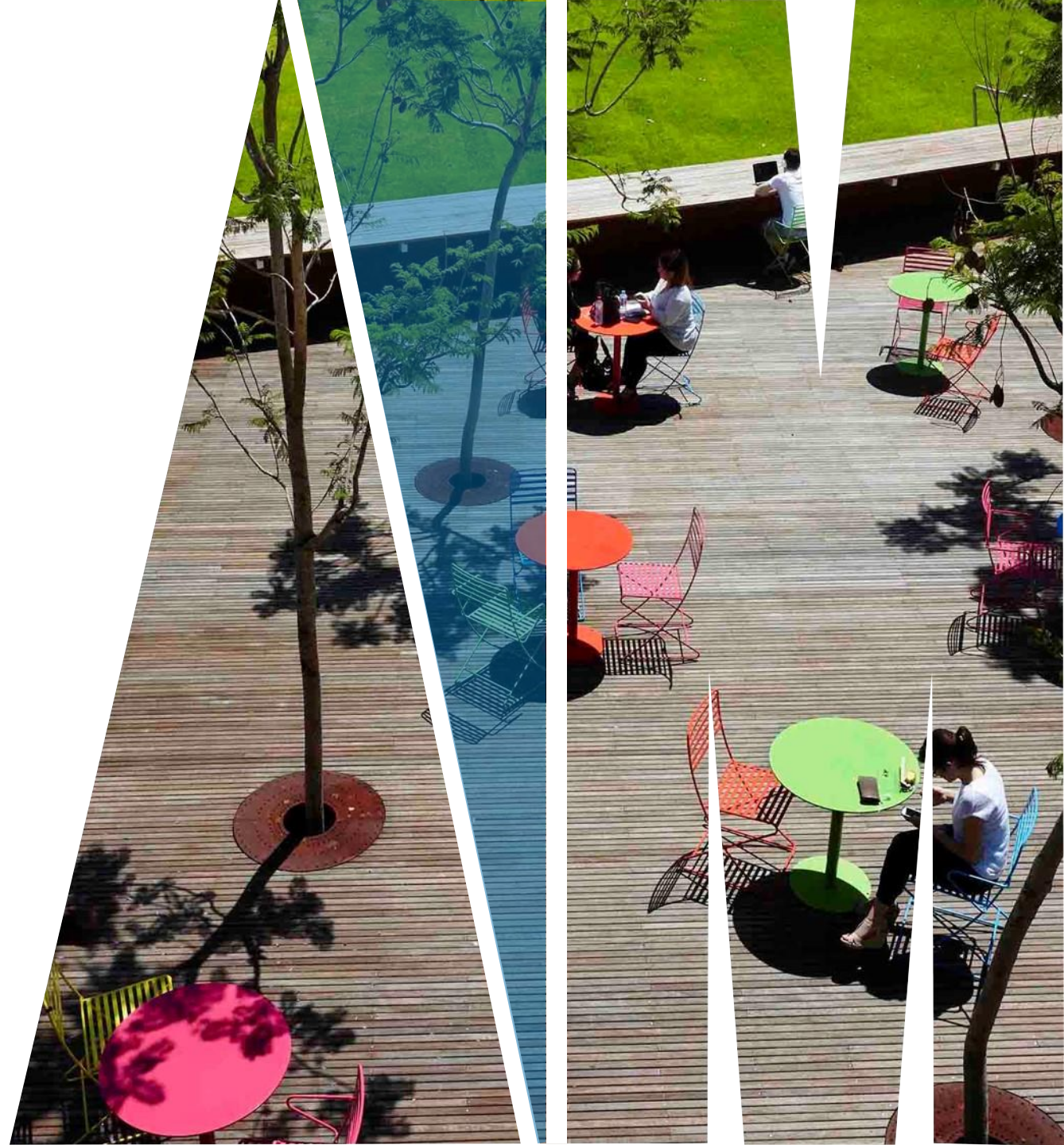


# **FIT2099 Object-Oriented Design and Implementation**

## Dynamic modelling



# Outline

Interaction diagrams

Sequence diagrams

Communication diagrams

# DYNAMIC MODELLING

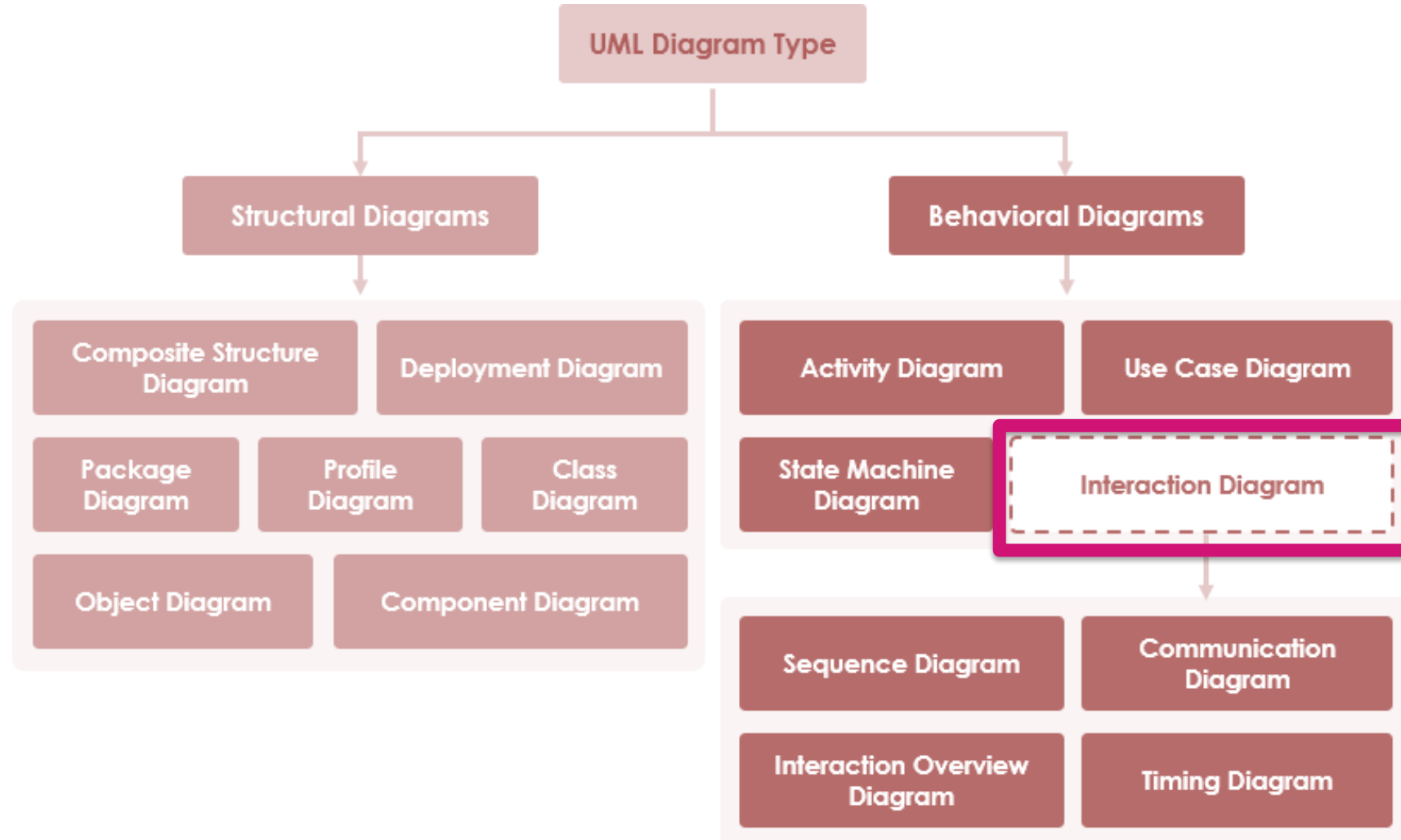
We have seen how you can design and document the **structure of a program**

- UML class diagrams
- UML package diagrams

But what about the way that classes **behave**?

- we need to understand **how classes interact**
- this is **how dependencies manifest** themselves when code is executed

# DYNAMIC MODELLING



# INTERACTION DIAGRAMS

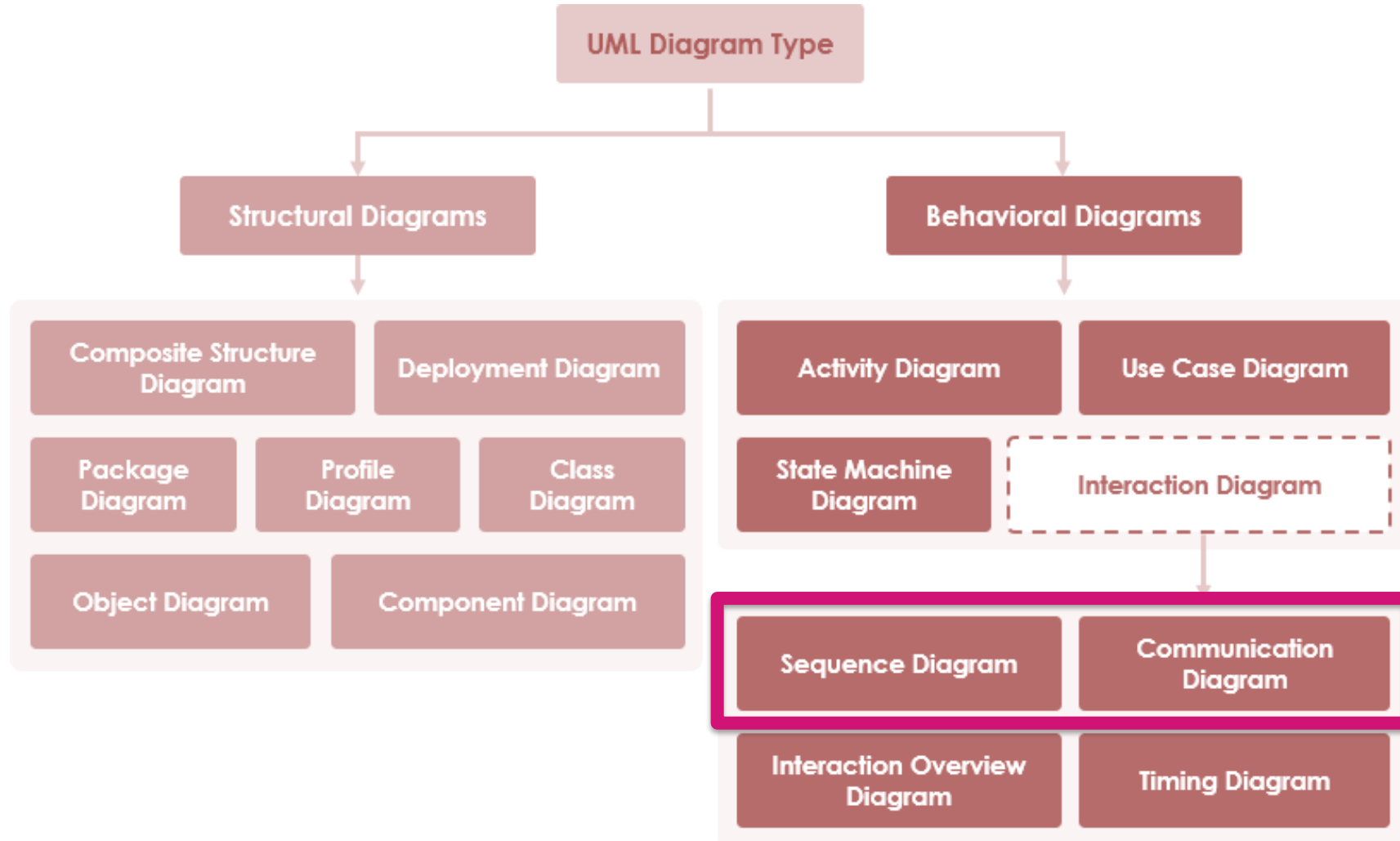
The UML supports several kinds of diagram that can be used to show the **dynamic behavior** of software

- **sequence** diagrams
- **communication** diagrams
- activity diagrams
- state diagrams

We will be looking at sequence diagrams and communication diagrams today

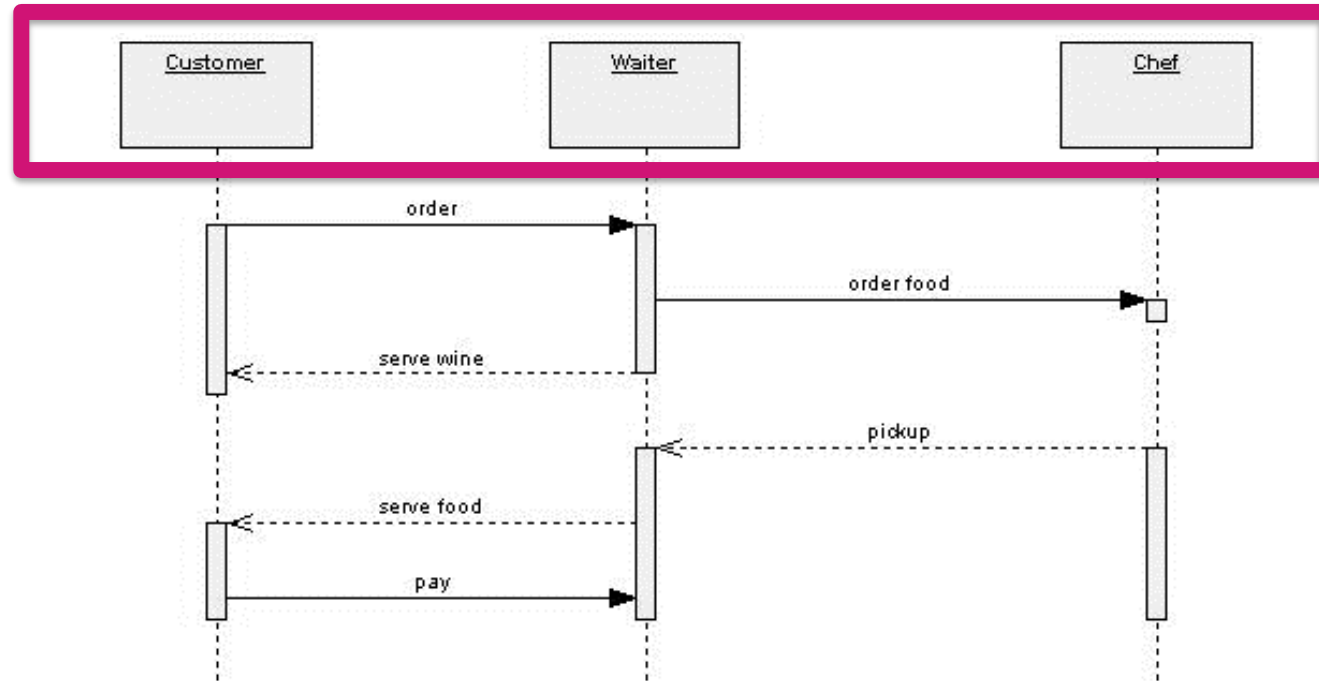
- collectively known as **interaction diagrams**

# DYNAMIC MODELLING



# SEQUENCE DIAGRAMS

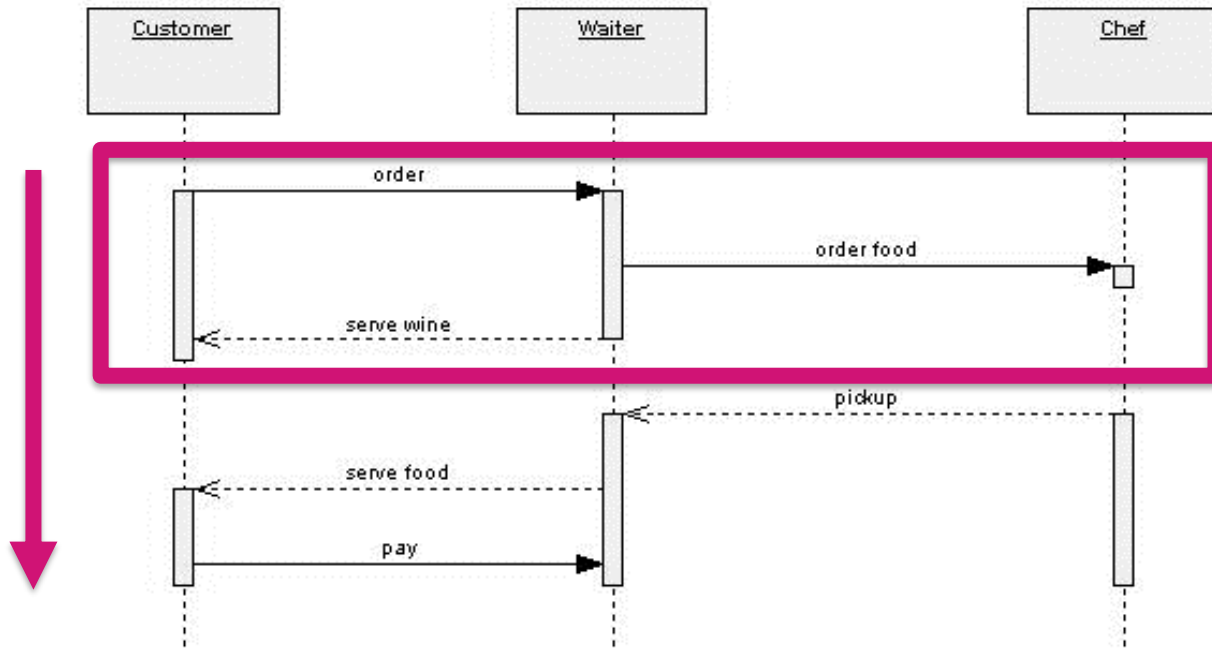
Classes





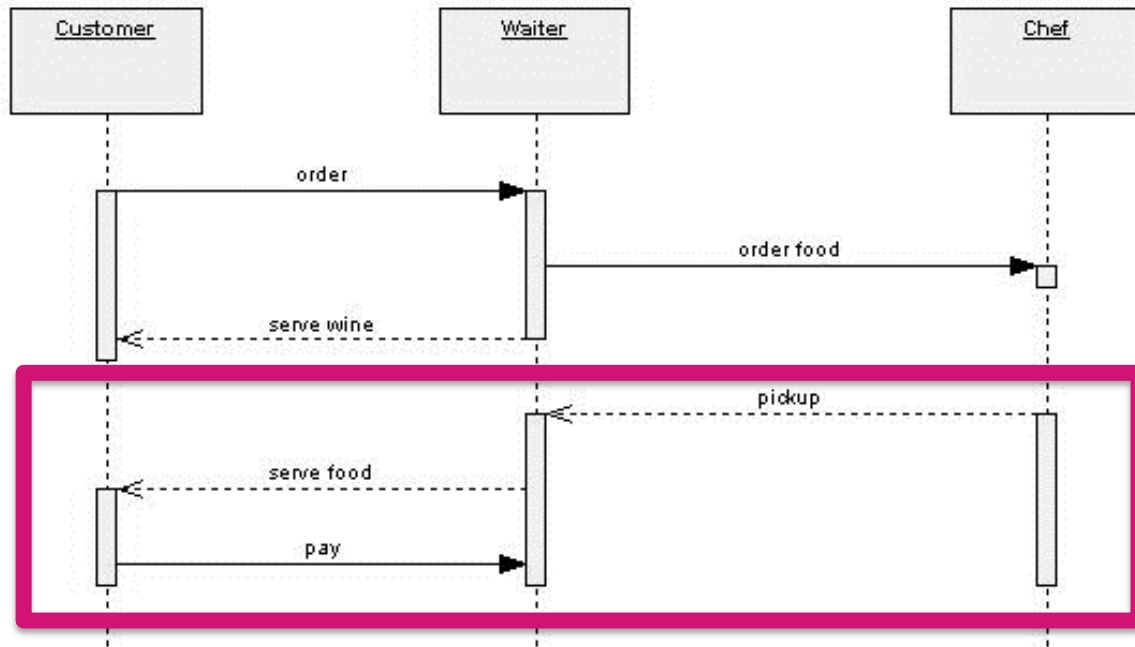
# SEQUENCE DIAGRAMS

Messages

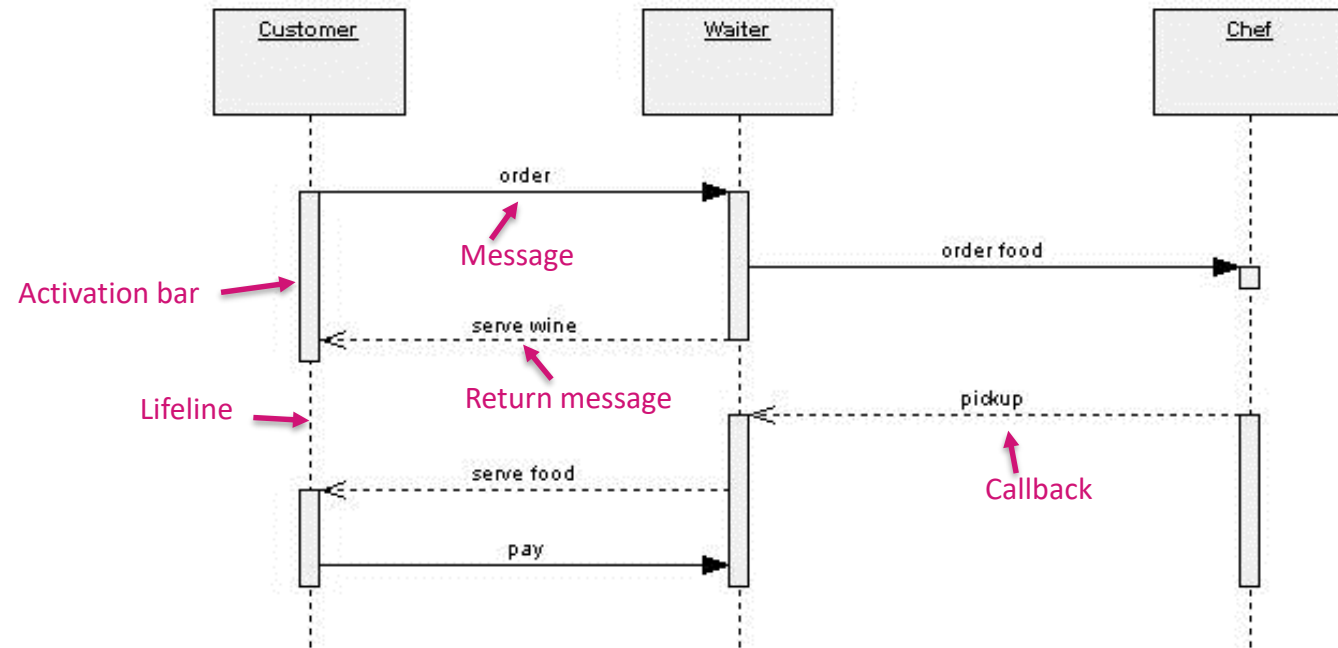




# SEQUENCE DIAGRAMS

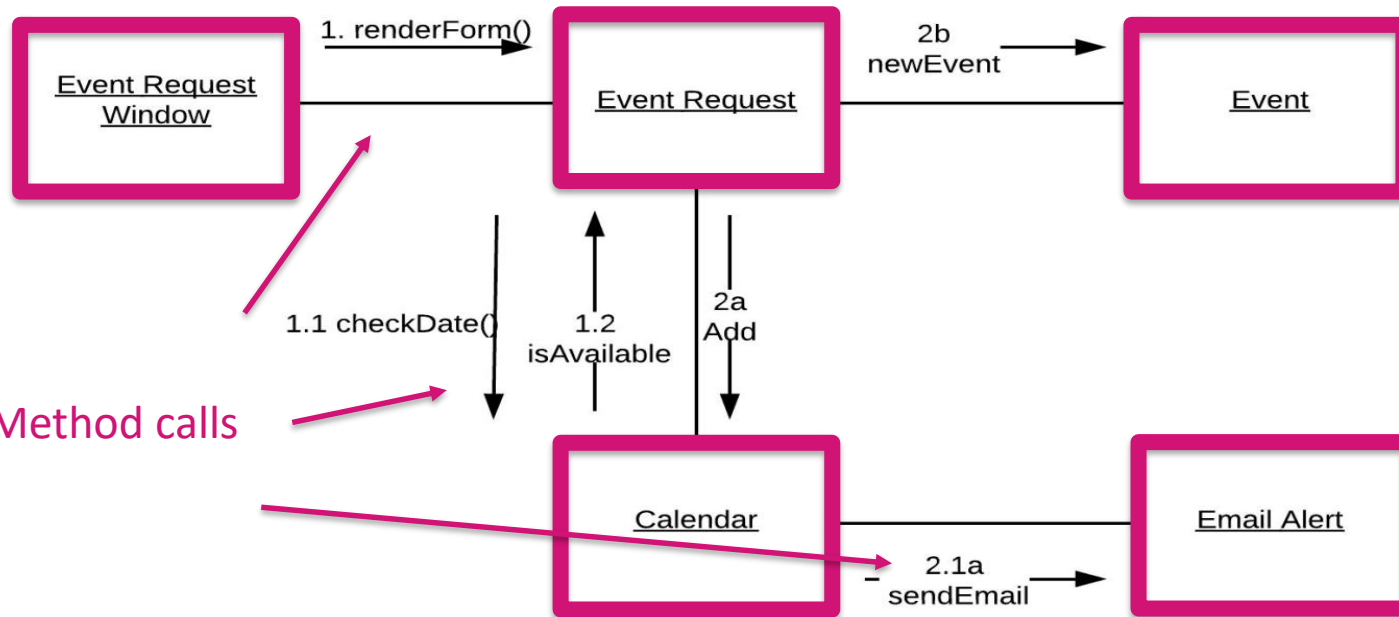


# SEQUENCE DIAGRAMS



# COMMUNICATION DIAGRAMS

## Classes



## Method calls



# SEQUENCE OR COMMUNICATION

**Sequence diagrams** are more commonly used than communication diagrams

- but they contain essentially **the same information**

Sequence diagrams are usually **easier to read**

Communication diagrams can be **easier to draw by hand**

- if you accidentally leave something out of a sequence diagram, it can be fiddly to fix
- if you're using a drawing tool, sequence diagrams usually have better support

**You may use whichever you prefer**

We will look at **sequence diagrams**

# UML SYNTAX

## CLASSES AND OBJECTS

**Objects** are represented by boxes

A label gives **object** name and/or **class**  
following the format:

**objectName : ClassName**

they are separated by a colon

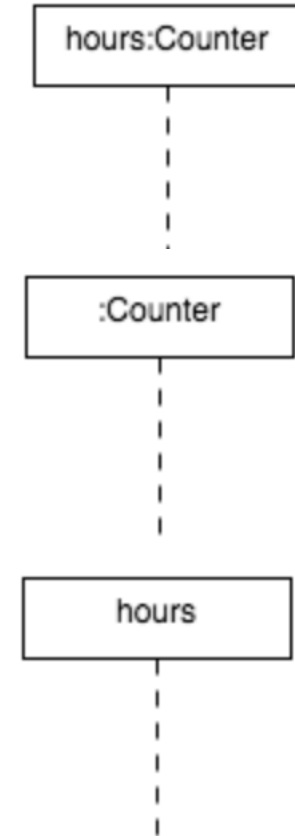
can leave the colon out if the class isn't given

Class attributes should not be listed in this shape.

**object : Class**

**: Class**

**object**



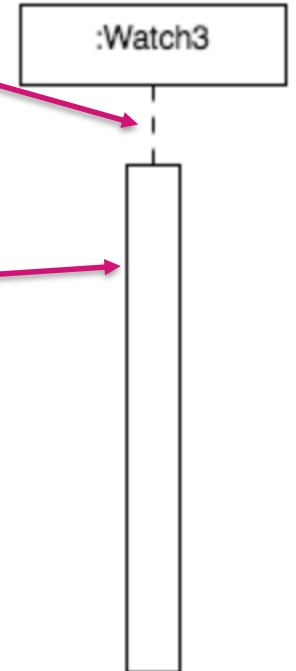
# UML SYNTAX

## LIFELINE AND ACTIVATION BARS

The dashed line dropping down from the classbox is called a *lifeline*

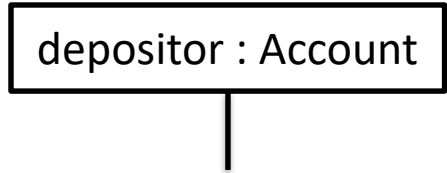
It represents the existence of the object

A thick line called an *activation bar* indicates that the object is active (i.e. a method is running in it). It also represents the time needed for an object to complete a task. The longer the task will take, the longer the activation box becomes.

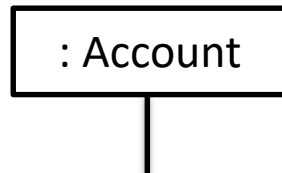


# UML SYNTAX

## CLASSES AND OBJECTS EXAMPLES

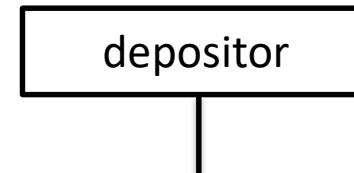


“An **object of class** Account, named ‘depositor’”



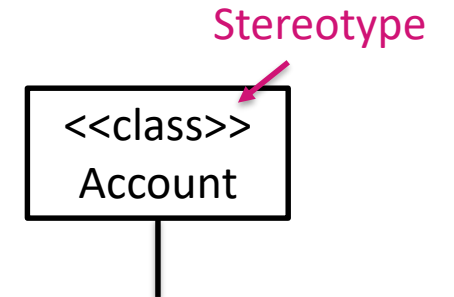
“A **nameless object** of class Account”

- ❑ can’t refer to it elsewhere in the diagram
- ❑ common ‘error’ – leaving out the colon



“An **object** named ‘depositor’, its type doesn’t matter”

- ❑ rarely used correctly



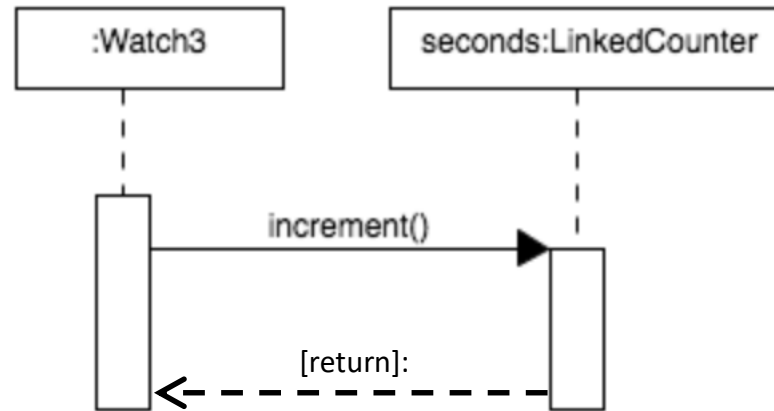
“The Account class”

- ❑ can only receive messages via **static methods**
- ❑ **Stereotype** lets you distinguish from typeless objects



# UML SYNTAX

## CLASSES AND OBJECTS EXAMPLES



Messages are indicated by arrows

- **solid arrow head** indicates that the caller will wait for a return before it will proceed
- **open arrow head** indicates that the caller won't wait for a return

The arrow is labelled with the **name of the method**

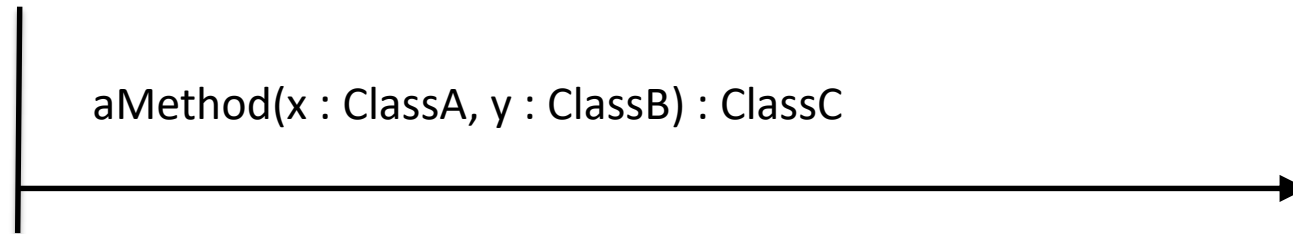
- **include parameters** if there are any

Returns are indicated by an **open headed arrow with a dashed line**

- **may** be labelled with the name of the object returned (if any)

# UML SYNTAX

## MESSAGE SYNTAX



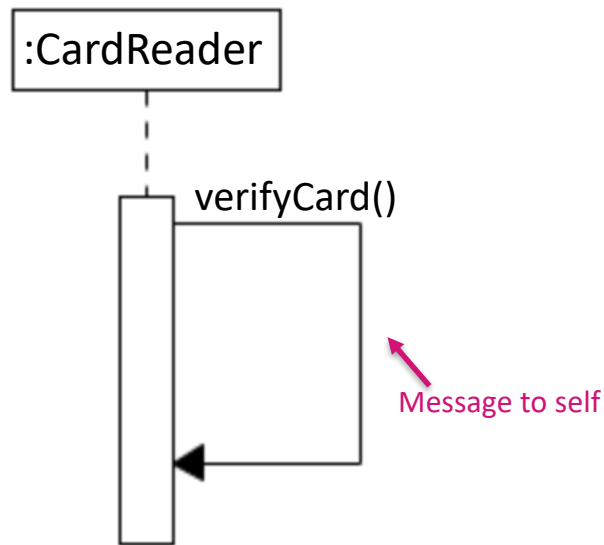
1. Client object (left) calls **aMethod** in supplier object (right) or class.
2. Parameters are a ClassA named **x** and a ClassB named **y**.
3. The method returns an **instance of ClassC**

Parameter *names* are **optional**. If present, they should match the names of the corresponding lifelines

Parameter *types* are **required**, and so is the return type

# UML SYNTAX

## REFLEXIVE MESSAGES

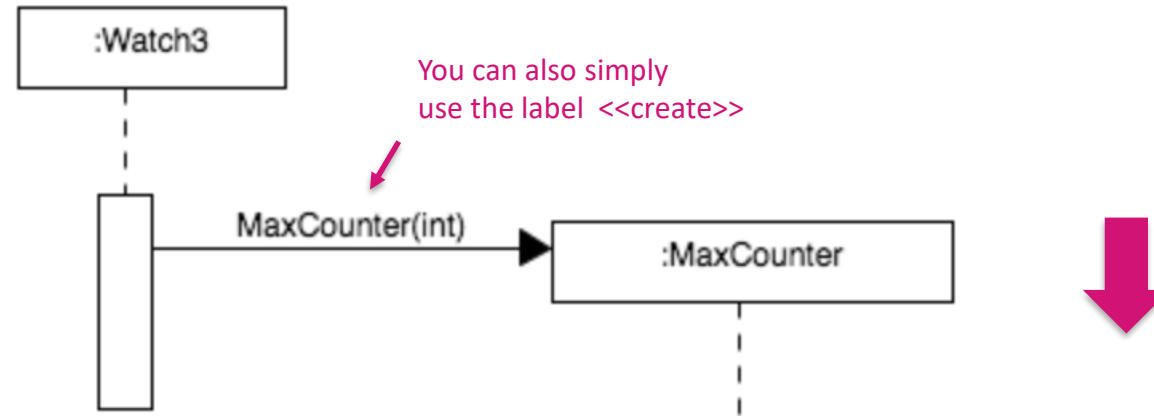


Objects can call methods on (i.e. send messages to) **themselves**

Show this as an arrow pointing back to the same object's lifeline

# UML SYNTAX

## CREATING OBJECTS



To show that a new object is created, you can drop the object symbol to show the **time of creation**

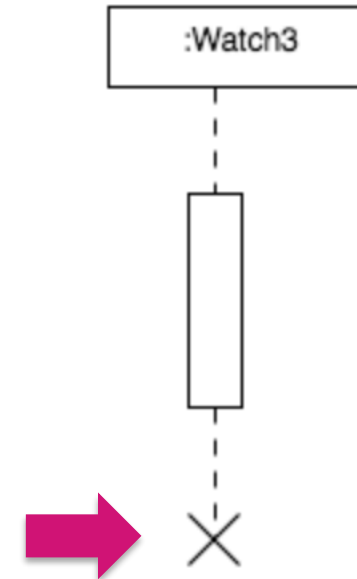
- This image shows an instance of a class called MaxCounter being created
- The message symbol is labelled with the **constructor signature**: MaxCounter(int)
- The message symbol can also be labelled with **<<create>>** instead of showing the constructor call

# UML SYNTAX

## DELETING OBJECTS

You can show the end of the life of an object by terminating its lifeline with an **X**

This image shows an instance of the class named Watch3 being terminated

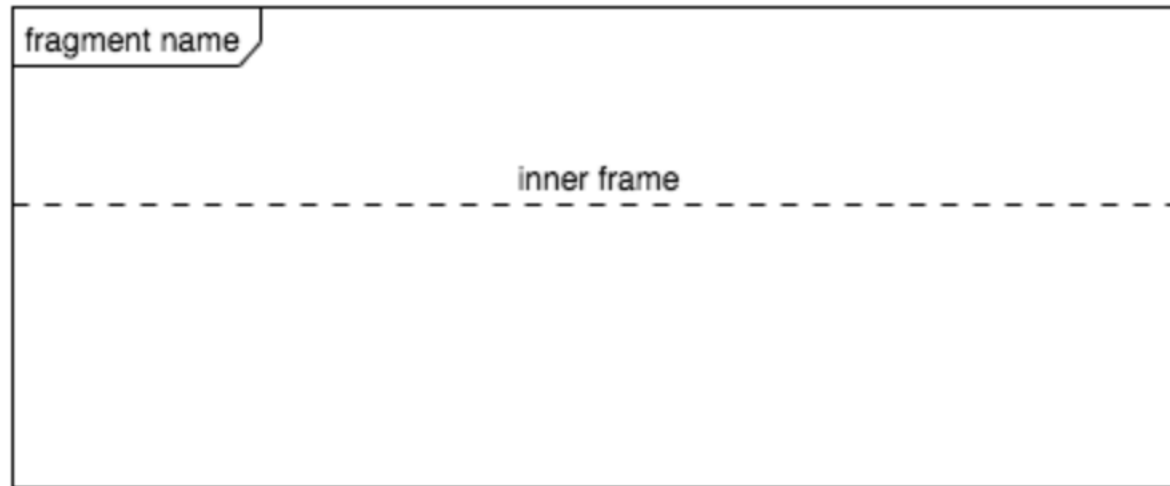


# UML SYNTAX

## FRAGMENTS

You can use a **fragment** to group a collection of messages

There are several kinds of fragments

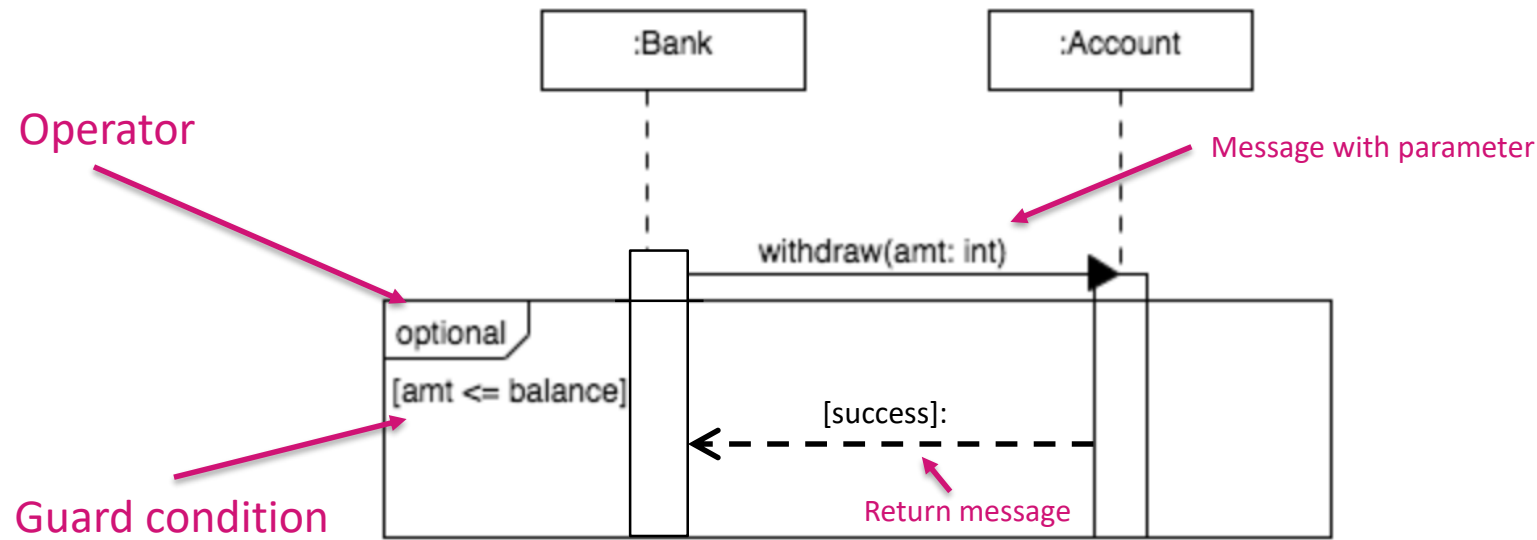


# UML SYNTAX

## OPTIONAL EXECUTION

The **opt** or **optional** sequence fragment indicates that what is inside the fragment is executed only if the supplied condition is **TRUE**.

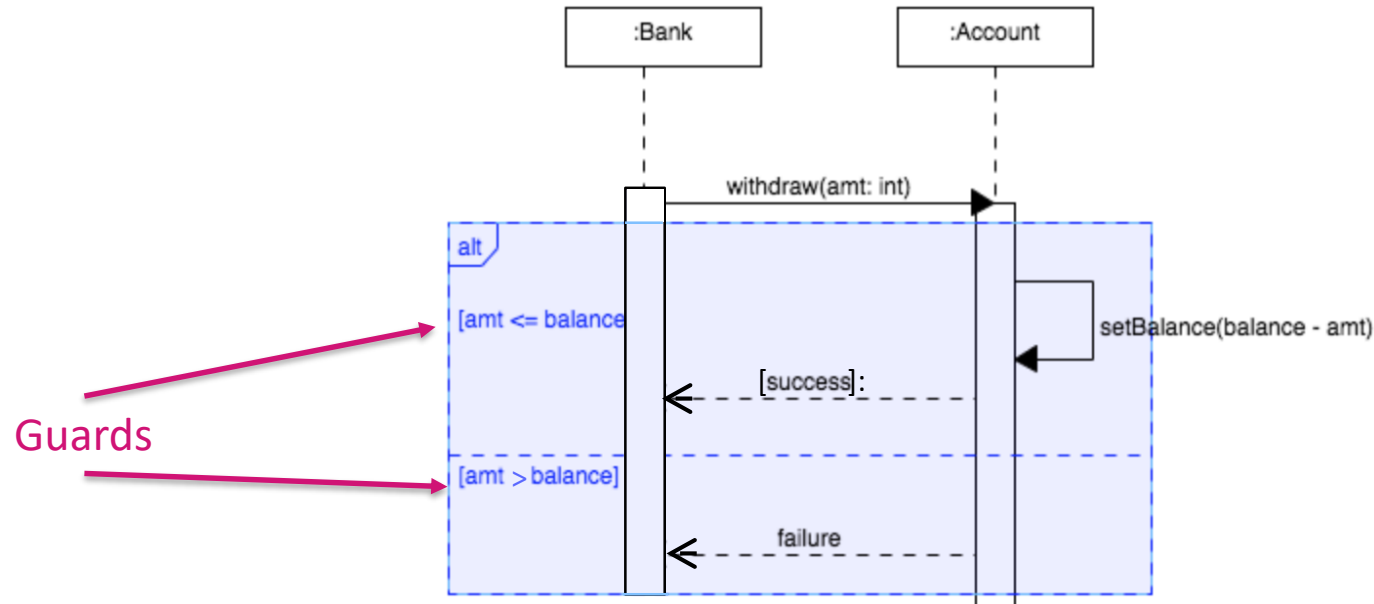
State the condition of execution in a **guard** on the left side, in square brackets





# UML SYNTAX

## CONDITIONAL EXECUTION



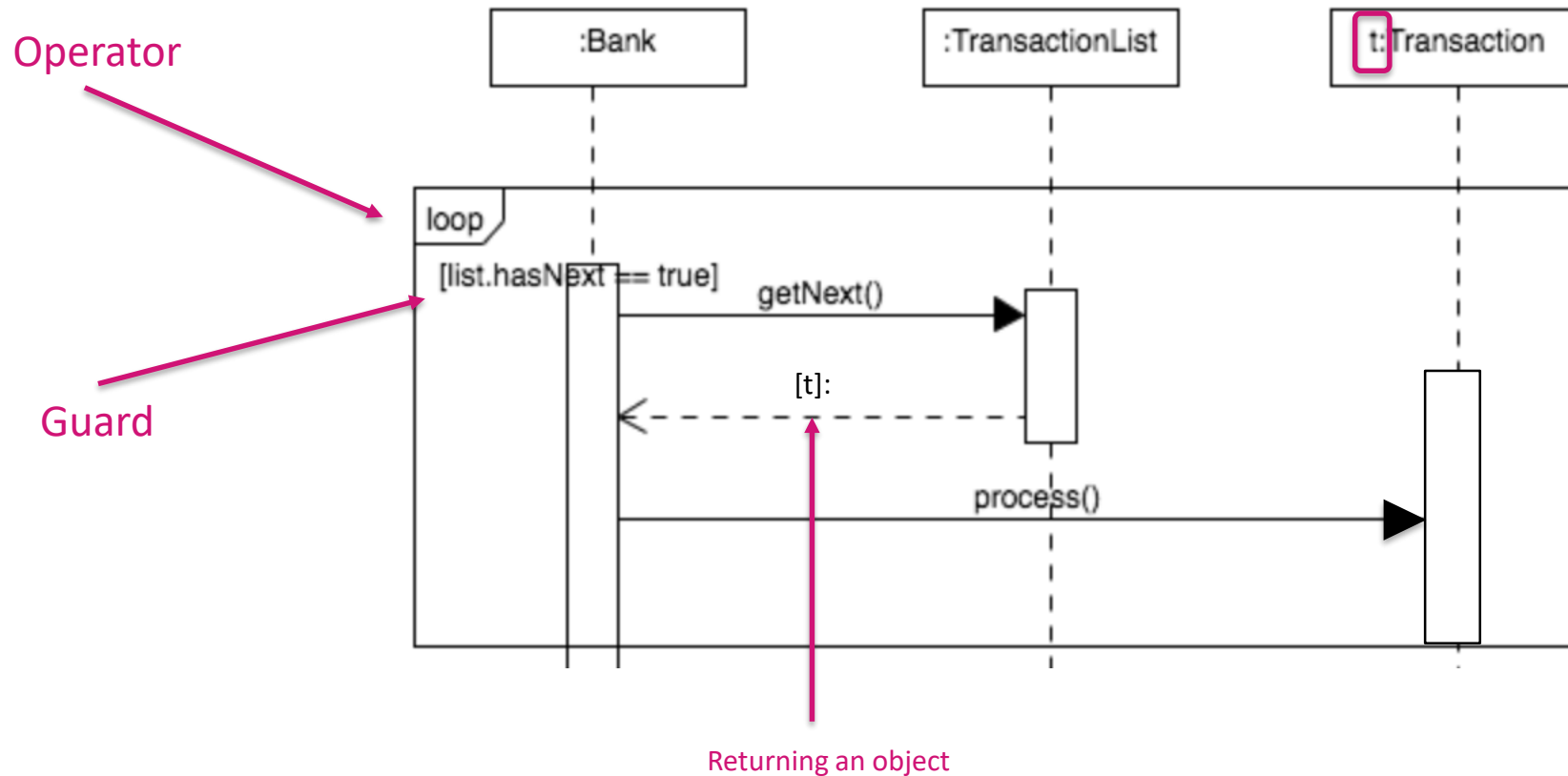
If there are **two or more alternatives** rather than a single optional component, use an **alt** or **alternative** fragment

- Syntax is **similar to the opt** fragment
- Separate alternatives with a **horizontal dashed line**

# UML SYNTAX

## ITERATION

Use a **loop** fragment for iteration



# COMPLICATED OPERATIONS

You can **nest fragments**

- so you can put an opt inside a loop inside an alt, etc.

But there are limits...

- it gets **very hard** to read this if you need to do a lot of nesting
- the syntax is **legal** UML but that doesn't mean it will be **legible**
- does not deal well with **complex** or **recursive** algorithms
- if algorithm is complicated, choose a different notation (e.g. **pseudocode**)

# EVOLUTION OF INTERACTION DIAGRAMS

Can also use interaction diagrams for **analysis purposes**

- they can be used to document any sequence of interactions, not just in software

Typically, if used for analysis or **for early design**, notation is less formal

- can't use method names and signatures if these have not yet been decided upon, or if documenting a manual process
- under these circumstances it's okay to **describe messages in ordinary English** rather than as method calls

As design evolves and becomes clearer, the interaction diagram **should become more formalised**

- for your **Assignment 1**, you should try to get as close as possible to your final method names and signatures,
- but for **Assignment 2** your sequence diagrams should match your code.

# Summary

Interaction diagrams

**Sequence** diagrams

Communication diagrams



MONASH  
University

Thanks



MONASH  
University

