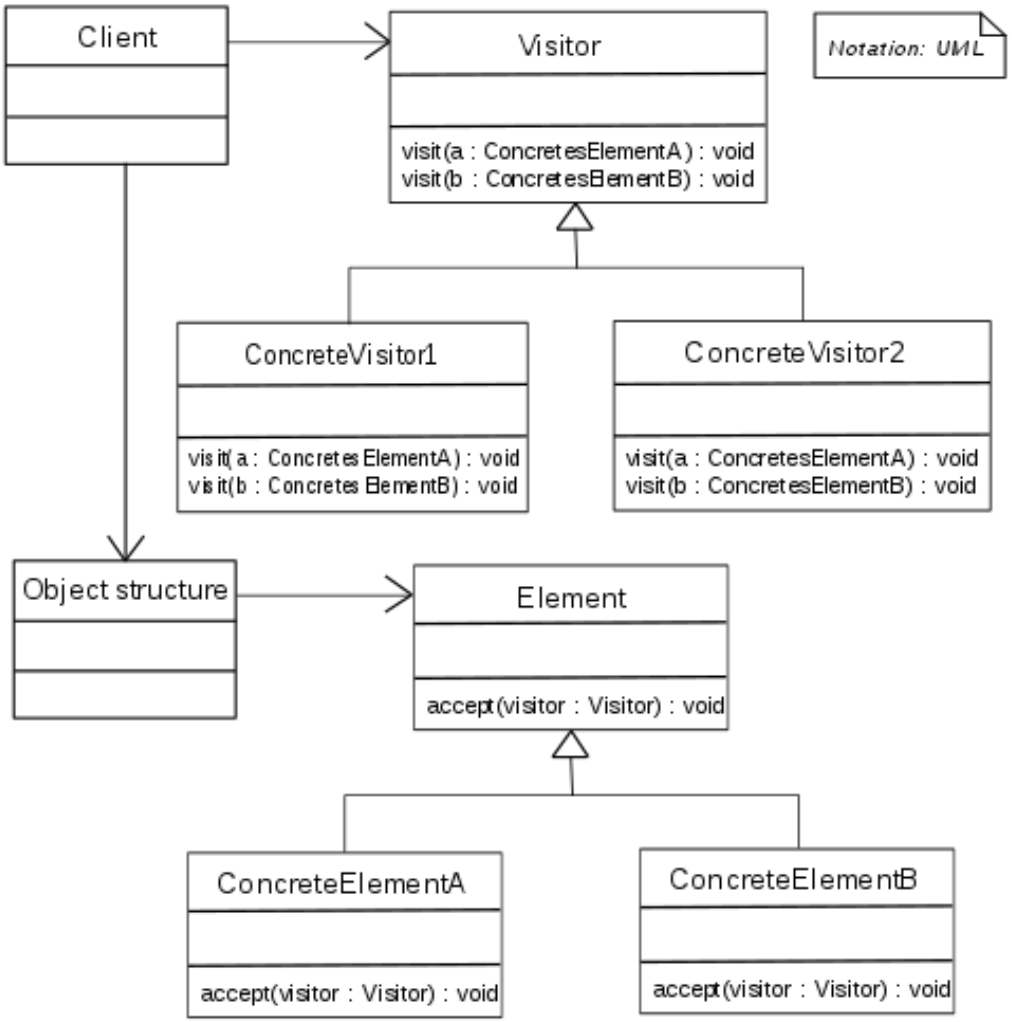
ADDITIONAL MATERIAL FOR INFSEN01-2

DR ANNA V. KONONOVA

LECTURE 3. ITERATOR PATTERN. MAY 16, 2017

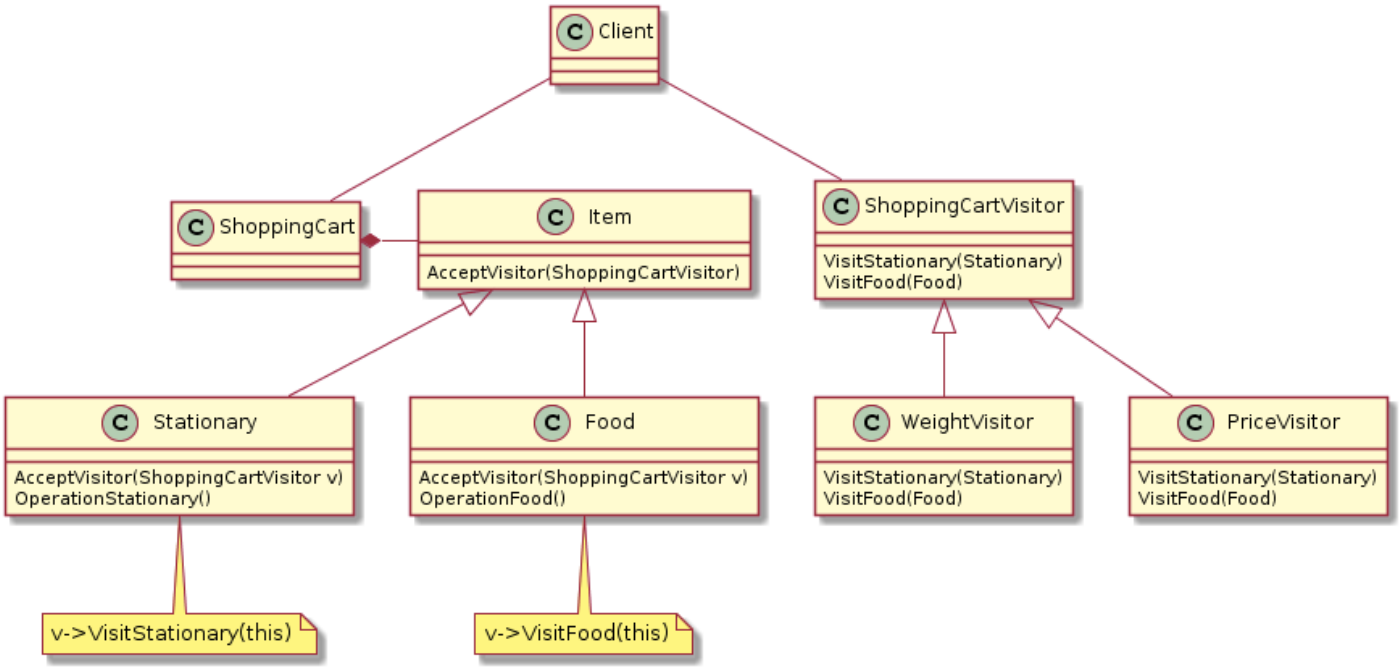
- Reminder: **Read all official slides as well** (your exam is prepared based on them)
- Photo opportunity!
- Last week, we have revisited our extensive duck pond simulator, study the first design pattern and try applying it to some new example problem
- **Today** we will: quickly revise last lesson's example, study the second behavioral design pattern and apply it to a new example problem. Plus we will deal with some small surprise

VISITOR: STRUCTURE



VISITOR: SHOPPING CART EXAMPLE

- A shopping cart in an online supermarket where different type of items can be added; clicking on checkout button calculates the total amount to be paid
- Shopping list: 1. GoF book; 2. notepad; 3. pen; 4. banana; 5. bottled water
- Each of the items has a price and a weight



VISITOR: APPLICABILITY, CONSEQUENCES

- **Applicability**:

Object structure contains many classes of objects with differing interfaces and operations that need to be performed on the depend on concrete classes

Many distinct and unrelated operations to be performed on objects and class pollution must be avoided

Object classes rarely change and operations are added often

- **Consequences**:

Easy addition of new operations: simply by adding a new visitor

Gather related operations together, separate unrelated ones

Adding new ConcreteElement classes is hard (explain to me why)

Allows visiting across class hierarchies (unlike Iterator pattern)

Visitors can accumulating state

Possible breaking encapsulation due to the need to provide public operators to access element's internals

ITERATOR: INTENT AND MOTIVATION

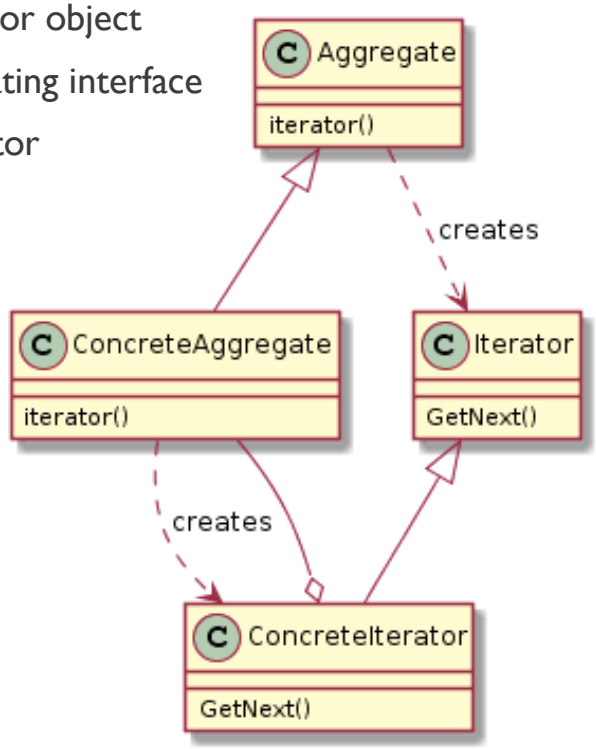
- Programs typically work on data
- Data is organised in some structures ➤ aggregate/containers = collections of objects ➤ think of examples of aggregates/containers
- Arrays (fixed length, variable length, bit, multidimensional, associative, hashes, etc)
- Lists (singly- or doubly linked, self-organising; stack, queue - normal, double-ended, priority; etc)
- Trees (binary - search, self-balancing; ternary; heap; etc)
- Others (graph, symbol table, struct)
- Traversal depends on the aggregate; different orderings are possible
- We do not wish to extend interfaces of all individual aggregates with traversal operations.
- **Intent**: a way to access elements of an aggregate object without exposing its underlying representation
- Reduce code duplication, easily extendable

ITERATOR: STRUCTURE, PARTICIPANTS, AND VERSIONS

Participants:

- Iterator – defines an interface for accessing and traversing elements
- Concrete Iterator – implements the Iterator interface
- Aggregate – defines interface for creating Iterator object
- Concrete Aggregate – implements Iterator creating interface

to return an instance of the proper ConcreteIterator



Versions:

- next(), hasNext(), getCurrent()
- next(), hasNext()
- next() only – gets the next element if it exists, and moves to the next or returns no element

Check how Iterator is defined in the official course slides!

ITERATOR: APPLICABILITY, CONSEQUENCES, EXAMPLE

Applicability and consequences:

- To access an aggregate object without exposing its internal representation
- To support multiple traversals
- To provide uniform interface for traversing aggregate structures

Example

- "Apples and bears": a recent merger between a greengrocer and toy shop revealed differences in stock inventory accounting implementations of both seller.
- Accounting of the greengrocer uses a list where each element stores item name, origin, current stock in kilos and price per kilo. Meanwhile the toy shop uses an array representation where toy name, origin, current stock and price per toy are recorded. Toys with zero stock are not removed from the array.
- 1. Draw designs previously used by both sellers
- 2. Merge both design and introduce a new function to produce a joint current stock inventory

ITERATOR: IMPLEMENTATION

- Who controls iteration?

Client is in control ➤ external iterator, iterator is in control ➤ internal iterator

- Who defines the traversal algorithm?

The aggregate can also define it and the iterator is used to store the state of iteration

If the iterator is responsible for traversal it is easy to use different iteration algorithms on the same aggregate and reuse the same algorithm on different aggregates. But traversal might need to access the private variables of the aggregate and, thus, putting traversal in the iterator violates the encapsulation of the aggregate.

- How robust is the iterator?

In other words, insertion and deletion of elements does not interfere with traversal and is done with not copying the aggregate

Register Iterator with aggregate and, on insertion or removal, aggregate will either adjust the internal state of the iterators it has produced or it maintains information internally to ensure proper traversal

- Iterators may have privileged access
- Iterators for composites

ITERATOR: EXAMPLE

- "**Apples and bears**": a recent merger between a greengrocer and toy shop revealed differences in stock inventory accounting implementations of both seller.
- Accounting of the greengrocer uses a list where each element stores item name, origin, current stock in kilos and price per kilo. Meanwhile the toy shop uses an array representation where toy name, origin, current stock and price per toy are recorded. Toys with zero stock are not removed from the array.
- 1. Draw designs previously used by both sellers
- 2. Merge both design and introduce a new function to produce a joint current stock inventory

VISITOR: Λ NOTATION (OFFICIAL SLIDES)

λ version of the Visitor

1. Given: $C_1, \ldots, C_n$ classes implementing a comon interface I; every class C_i has fields $f_1^i, \ldots, f_m^i$
2. Add method *Visit* to *I* that returns a result of type *U*. This method *Visit* is common to all the classes implementing *I*. It picks the right option based on its concrete shape. It is not known what *Visit* will return, use a generic type *U* for return type.

```
Interface I
{
   U Visit <U> (Func <FieldsC1, U> on C1,
                 …,
                Func <FieldsCn, U> on Cn);
}
```

3. The *Visit* method accepts as an input one function per concrete implementation. Each such function depends on the fields of the concrete instance and produces a result of type *U*

```
Class C1 : I
 {
   F_1 f_1;
   …
   F_m f_m;
   U Visit <U> (Func <FieldsC1, U> on C1,
                 …,
                 Func <FieldsCn, U> on Cn){
     onC1(f1, …, fm)};
 }
```

4. Every class implementing the interface *I* has the task to implement *Visit* method by selecting and calling the appropriate argument

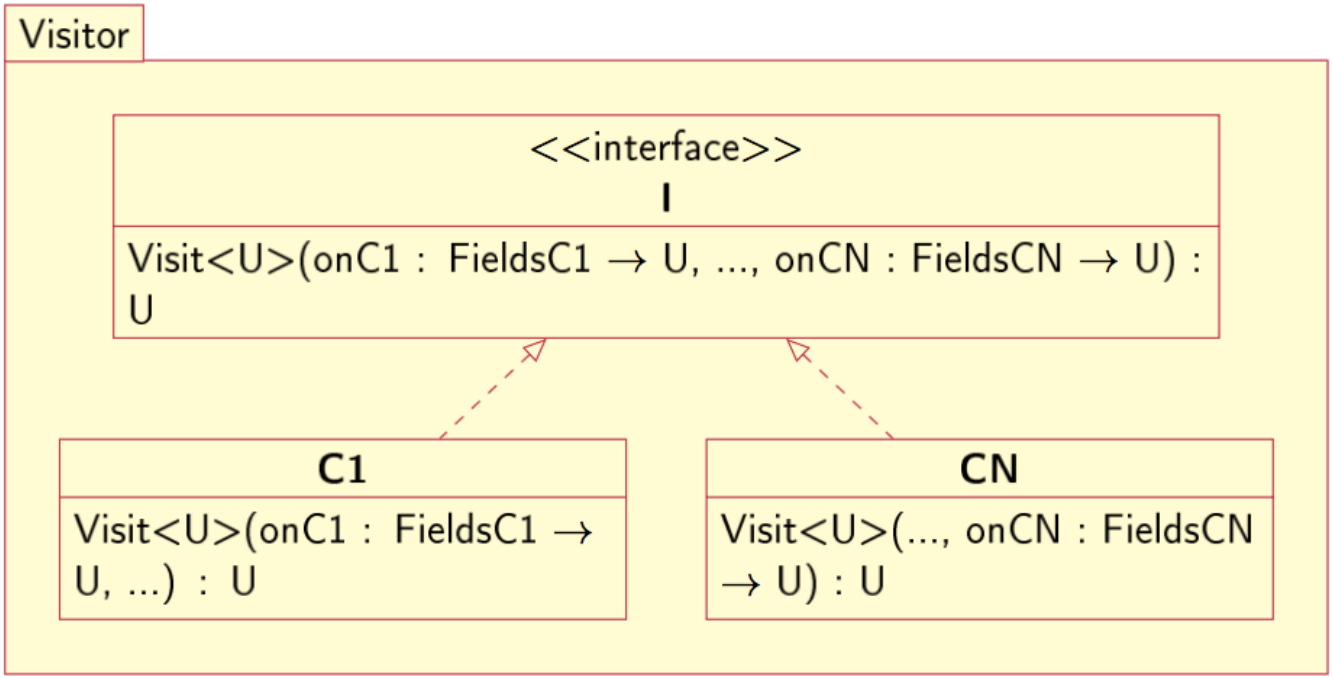
λ version of the Visitor (continued)

5. Every time an instance of type *M* is consumed it has to be *Visit*ed

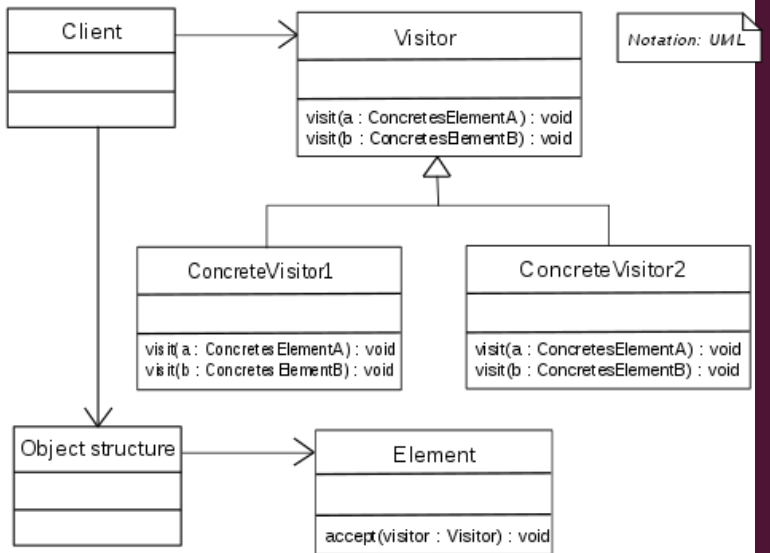
```
I i=…;
…
U result =
  m.Visit(
    i_1 => b_i;
    …,
    i_n => b_i);
```

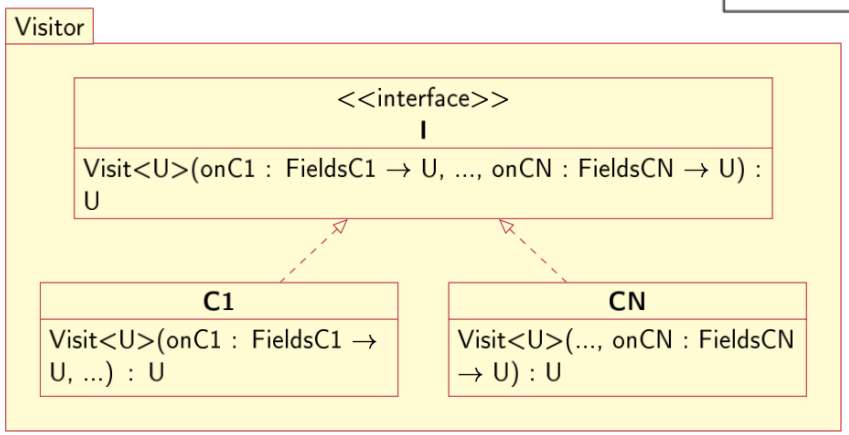
5. Every argument of the *visit* becomes a function that is triggered depending on the concrete type of *i*
6. *i_i* are the fields of the concrete instance C_i;
7. *bi* is the block of code to run when a vsit on an instance of a concrete type *Ci* is needed

λ version of the Visitor



Visitor: compare notations





VISITOR: A NOTATION

ITERATOR: Λ NOTATION (OFFICIAL SLIDES)

λ version of the Iterator

1. Interface *Iterator*<*T*> with the following signature

```
Interface Iterator <T>{
    IOption<T> GetNext();
}
```

2. Where *GetNext* returns *Some*<*T*> if there is an item to fetch and moves to the next item; it returns *None*<*T*> if there are no moreelements to fetch
3. Implementing the *Iterator* <*T*>: every collection that wants to provied a disciplined and controlled iteration mechanism has to either implement such interface or provide a way to adapt it
4. Iterating a collection with *5, 3, 2* will return : *Some(5), Some(3), Some(2), None(), None(), … None()*

ITERATOR: A NOTATION (OFFICIAL SLIDES)

Implementing the *Iterator* <*T*>:

```
class NaturalList : Iterator <int>{
  private int current = -1;
  Ioption<int> GetNext(){
    current = (current + 1);
    return new Some<int>(current);
  }
}
```

GetNext increases the counter n (starting from -1) and returnsit within a *Some*

```
class IterableList<T> : Iterator <T>{
  private List<T> list;
  public IterableList(List<T> list){
    this.list = list;
  }
  IOption<T> GetNext(){
    if List.IsNone(){
    return new None(T);
    }
    else{
      List<T> tmp = list;
      list = list GetTail();
      return new Some<T>(tmp.GetValue());
    }
  }
}
```

Iterable list takes as input an object of type list. *GetNext* returns *None* at the end of the list (when the tail is *None*), otherwise it moves to the next node and returns its value of the array at position *index* wrapped inside *Some*

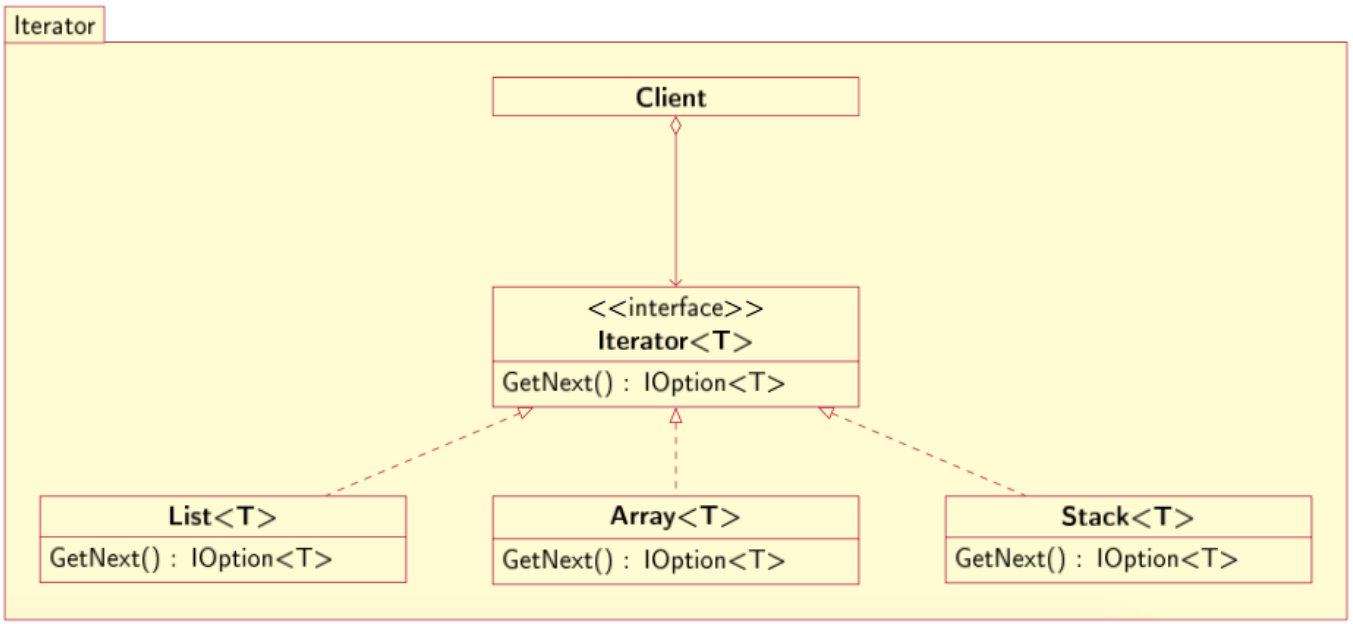
ITERATOR: A NOTATION (OFFICIAL SLIDES)

Implementing the *Iterator* <*T*>:

```
Class IterabelArray<T> : Iterator<T> {
  private T[] array;
  private int index = -1;
  public IterableArray(T[] array){
    this.array = array;
  }
  IOption<T> GetNext(){
    if ((index + 1) >= array.Length){
      return new None<T>();
    }
    else{
      index = (index + 1);
      return new Some<T>(array[index]);
    }
  }
}
```

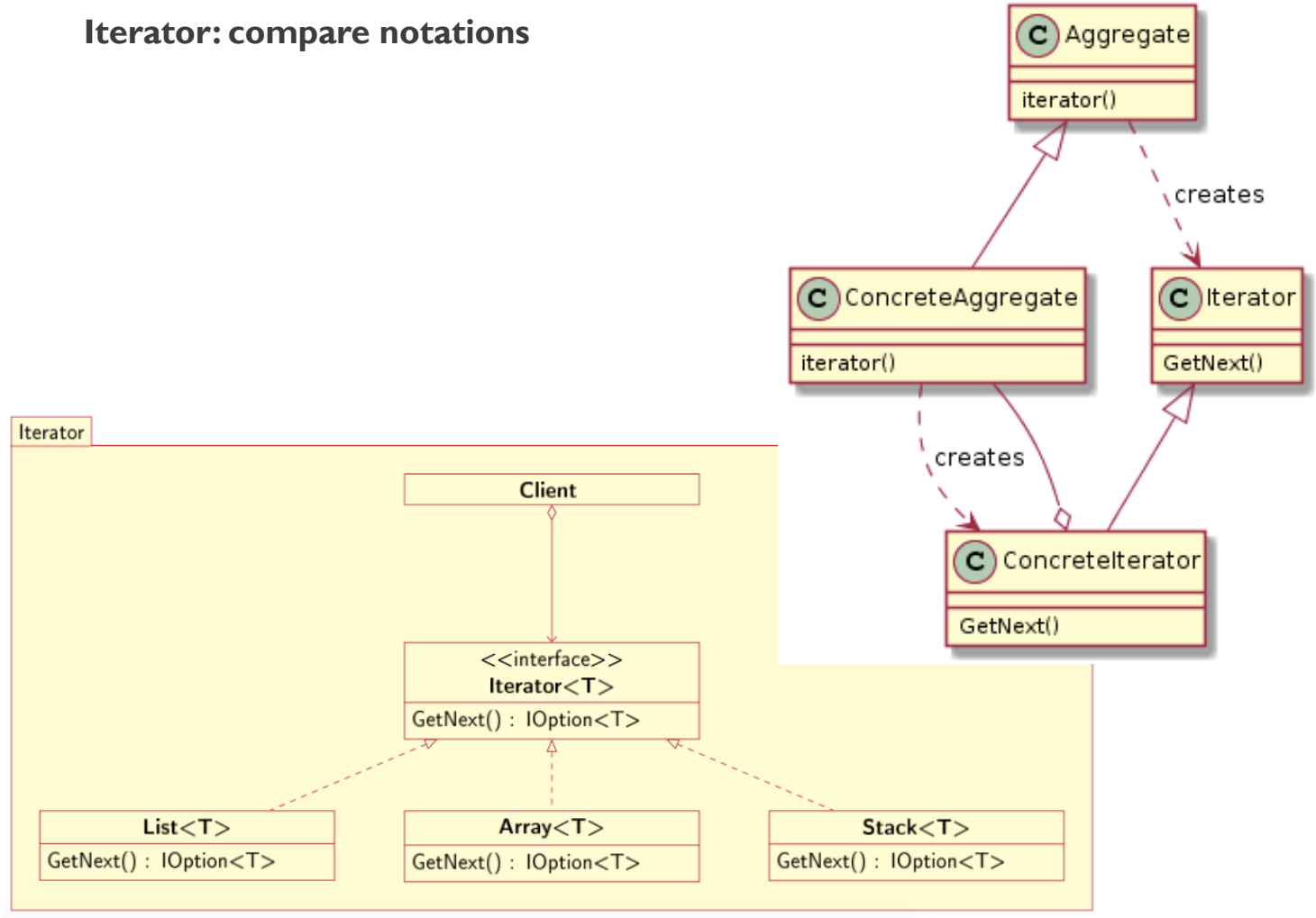
Iterable array takes as input an object of type array. *GetNext* returns *None* at the end of the array. otherwise it increases the *index* and returns the value of the array at position *index* wrapped inside *Some*

λ version of the Iterator



ITERTOR: Λ NOTATION (OFFICIAL SLIDES)

Iterator: compare notations



ITERATOR: A NOTATION

CONCLUSION

- **Today** we have quickly revised last lesson's example, studied the second behavioral design pattern and applied it to a new example problem
- **Start/continue working on assignment**

✓ Lesson 1: Introduction

✓ Lesson 2: Visitor pattern

✓ Lesson 3: Iterator pattern

⏳ Lesson 4: Adapter pattern

⏳ Lesson 5: Abstract classes

⏳ Lesson 6: Abstract factory pattern

⏳ Lesson 7: Decorator pattern

⏳ Lesson 8: Revision

- Thanks for your attention!