ADDITIONAL MATERIAL FOR INFSEN01-2

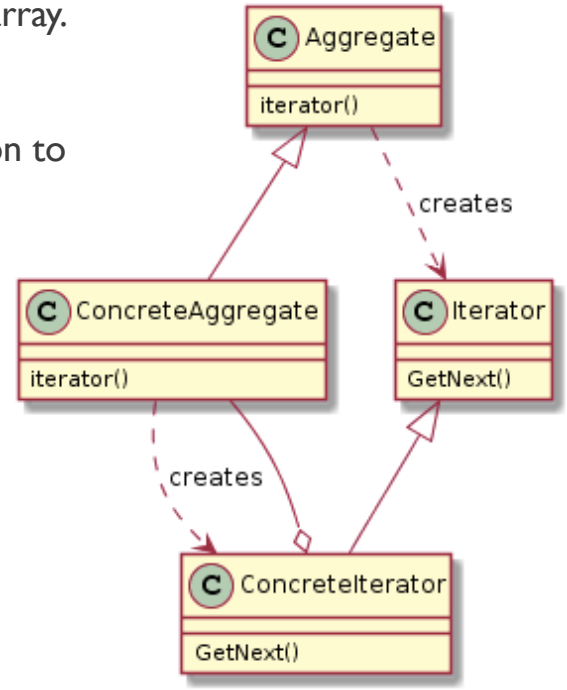
DR ANNA V. KONONOVA

LECTURE 4. ADAPTER PATTERN. MAY 23, 2017

- Reminder: **Read all official slides as well** (your exam is prepared based on them)
- Photo opportunity!
- Last week, we have revisited the shopping cart example, studied Iterator pattern and applied it to ”Apples and bears” example
- **Today** we will: quickly revise last lesson’s example, study Adaptor design pattern and apply it to a new example problem.

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



ITERATOR: EXAMPLE

- Write code here

ADAPTER: INTENT, APPLICABILITY

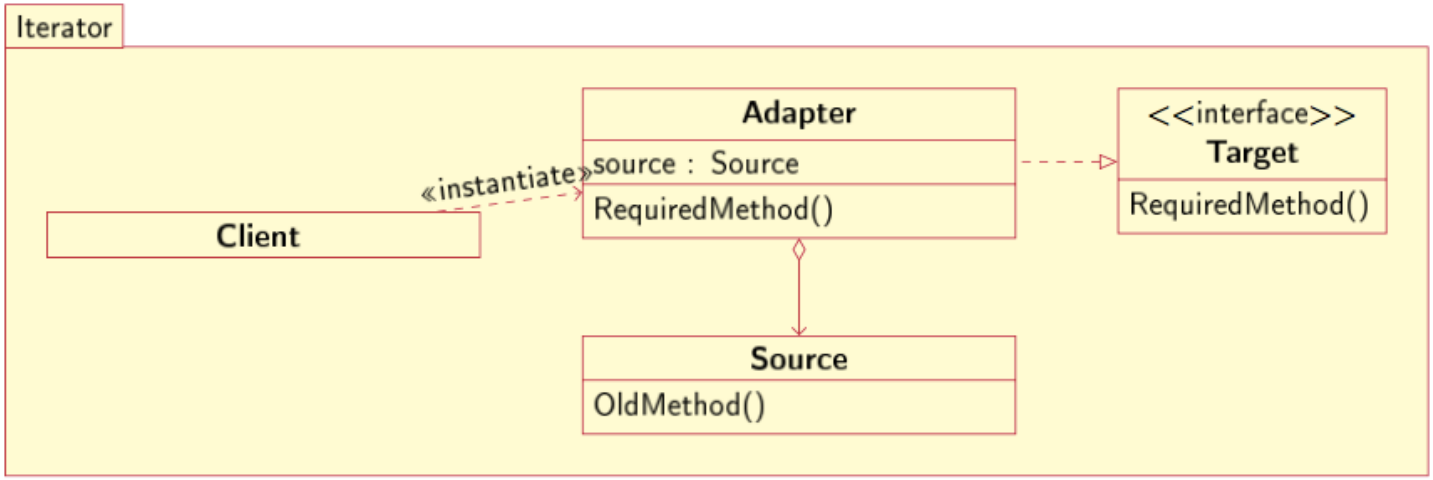
- Sometimes code reuse does not seem possible because interfaces does not match. For example, an "off the shelf" component offers functionality that you'd like to reuse, but its "view of the world" is not compatible with architecture of your system.
- **Intent**: convert the interface of a class into another interface that is expected by a client. Adapter is also known as Wrapper.
- **Applicability:**

- Use an existing class but its interface does not match the one you need
- Creating a reusable class that cooperates with unforeseen classes that do not necessarily have compatible interfaces
- Use several existing subclasses but it is impractical to adapt their interfaces by subclassing every one (object adapter)

ADAPTER: STRUCTURE (OFFICIAL SLIDES)

- Given two different interfaces *Source* and *Target*
- An *Adapter* is built to adapt *Source* to *Target*
- The *Adapter* implements *Target* by means of a reference to a *Source*
- A *Client* interacts with the *Adapter*
- *Adapter* deals with *Source* internally

(typo in the diagram below: title should read Adapter)



ADAPTER: CONSEQUENCES AND REMARKS

- Adapter is a structural pattern: concerned with how classes and objects are composed to form larger structures
- Object composition: Adapter keeps a reference to Source(s)
- Client is bound to an interface not implementation
- Adapters map behaviours across the domains ("bridge two worlds");
- Adapting may not change or add behaviours; it should preserve full behaviour
- Two-way adapters (TraditionalIterator and Iterator in the official slides)

- Think of examples where you would use an Adapter
- ➢ legacy systems, different frameworks, closed libraries, etc.

ADAPTER: EXAMPLE

"**Heating element**". Our department makes software for remote house climate control. It uses *commercial* modelling package which calculates the *current expected time* (in seconds) required for heating up the room up to a given temperature (based on model parameters, i.e. some physical properties of the house and the heating system).

Our software is so streamlined that the investors decided to reuse it in two other departments: one dealing with the same systems for American market and another working with temperature control units for science labs. American customers are used to the Fahrenheit scale and scientific labs use the Kelvin scale (we are used to the Celsius scale).

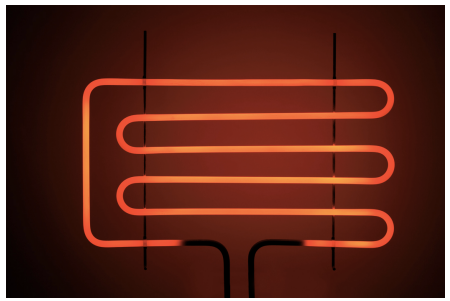
- *Assume some modelling package that for input tempC returns the value tempC*10 seconds (black box)*
- Write remote house climate control application, simulate some calls to the modelling interface (0°C, 10°C, 23°C)
- Extend the interface of the heating element controller for easy reuse (via Adapter pattern)
- Remember to use only Celsius inside the interface, and only equipment scale outside the interface
- Write a short program simulating calls to new interface from all three types of equipment and prinint model output (try inputting 36.8°C, 98.24°F, 309.95K)

(If you have problems imagining different scales think of how height can be given in meters, feet, etc.)

tC = **tK** - 273.15 **tF** = (**tC** * 9/5 + 32) **tK** = (**tF** + 459.67) * 5/9

tK = **tC** + 273.15 **tC** = (**tF** - 32) * 5/9 **tF** = **tK** * 9/5 - 459.67

Water freezes: 0°C = 32°F = 273.15K
Weather today: 18°C = 64.4°F = 291.15K
Normal human: 36.8°C = 98.24°F = 309.95K
Water boils: 99.98°C = 211.96°F = 373.13K



ADAPTER: EXAMPLE

Design:

- Our new applications operates in terms of Fahrenheit or Kelvin scales; we do not want to change the modelling package we use
- ➢ create two *adapters* (1) converting Fahrenheit to Celsius or Kelvin to Celsius, (2) calling the existing modelling software with values in Celsius
- Source: to do with Celsius
- Target: to do with Fahrenheit/Kelvin
- Adapter(s): converter(s)

tC = **tK** - 273.15

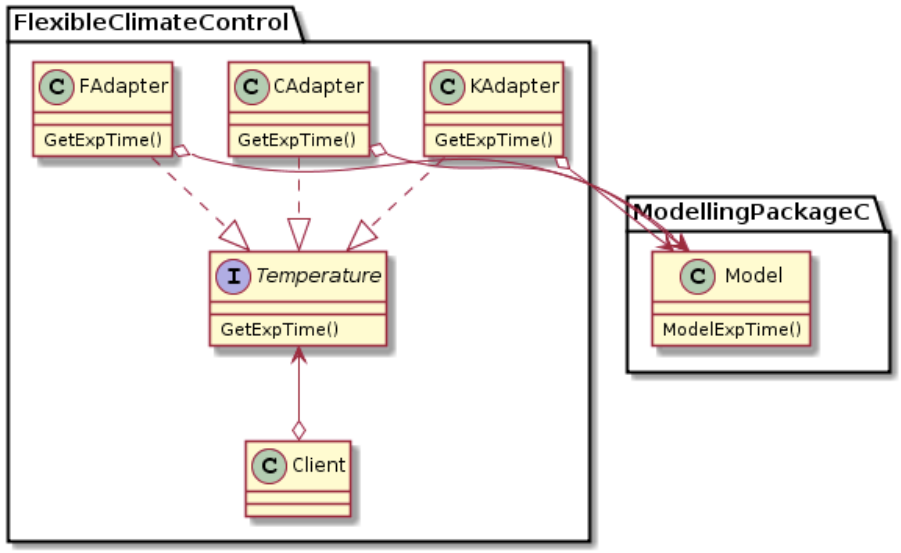
tF = (**tC** * 9/5 + 32)

tK = (**tF** + 459.67) * 5/9

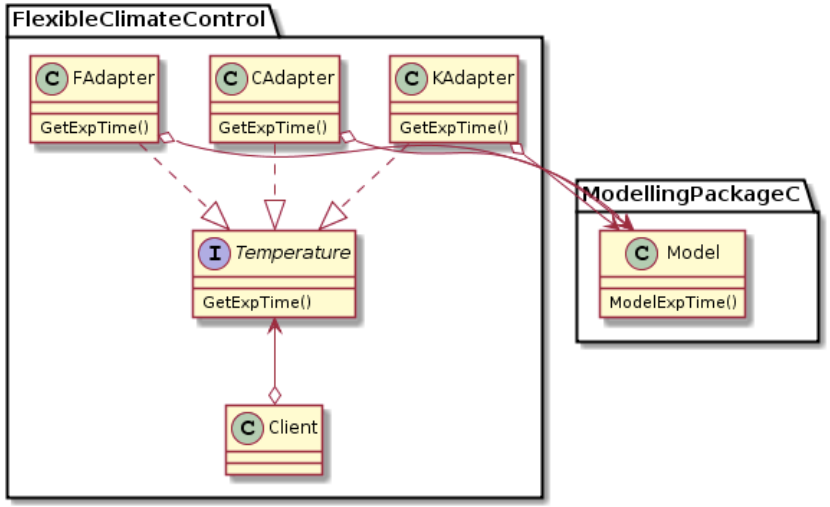
tK = **tC** + 273.15

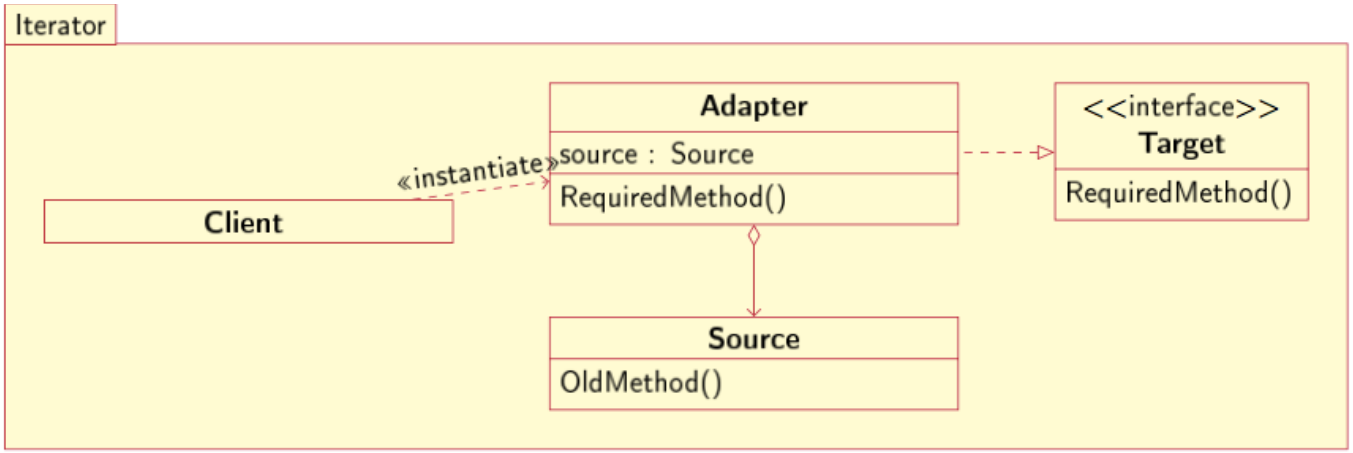
tC = (**tF** - 32) * 5/9

tF = **tK** * 9/5 - 459.67



ADAPTER: COMPARE EXAMPLE DESIGN





EXAMPLES FROM OFFICIAL SLIDES

Making Option "iterable": a naive approach

- Consider the *Option* data type that is some kind of collection
- Without adapter:

```
Interface Iterator<T>{
  public Option<T> GenNext();
}
Interface Option<T> : Iterator<T> {
  …
}
class None<T> : Option<T>{
  Option<T> GetNext(){
    return new None<T>();
  }
  …
}
```

```
Class Some<T> : Option<T>{
  private T value;
  private bool visited = false;
  public Some(T value){
    this.value = value;
  }
  Option<T> GetNext(){
    if(visited){
      return new None<T>;
    }
    else{
      visited = true;
      return new Some<T>(value);
    }
  }
  …
}
```

Is it always needed for the option to be iterable?
No! Adapter is better as it allows to extend option to any additional services required without changing the option data structure

EXAMPLES FROM OFFICIAL SLIDES

Iterating an Option<T> with adapters

- *Iterator<T>* is a *Target*, *Option<T>* is *Source*, *IOptionIterator<T>* is *Adapter*
- *GetNext*() returns *Some* only in the first iteration; iterating *None* returns *None*

```
Class IOptionIterator<T> : Iterator<T>{
  private Option<T> option;
  private bool visited = false;
  public IOptionIterator(Option<T> option){
    this.option = option;
  }
  Option<T> GetNext(){
    if(visited){
      return new None<T>();
    }
    else{
      visited = true;
      if(option.IsSome()){
        return new Some<T>(option getValue());
      }
      else{
        return new None<T>();
      }
    }
  }
}
```

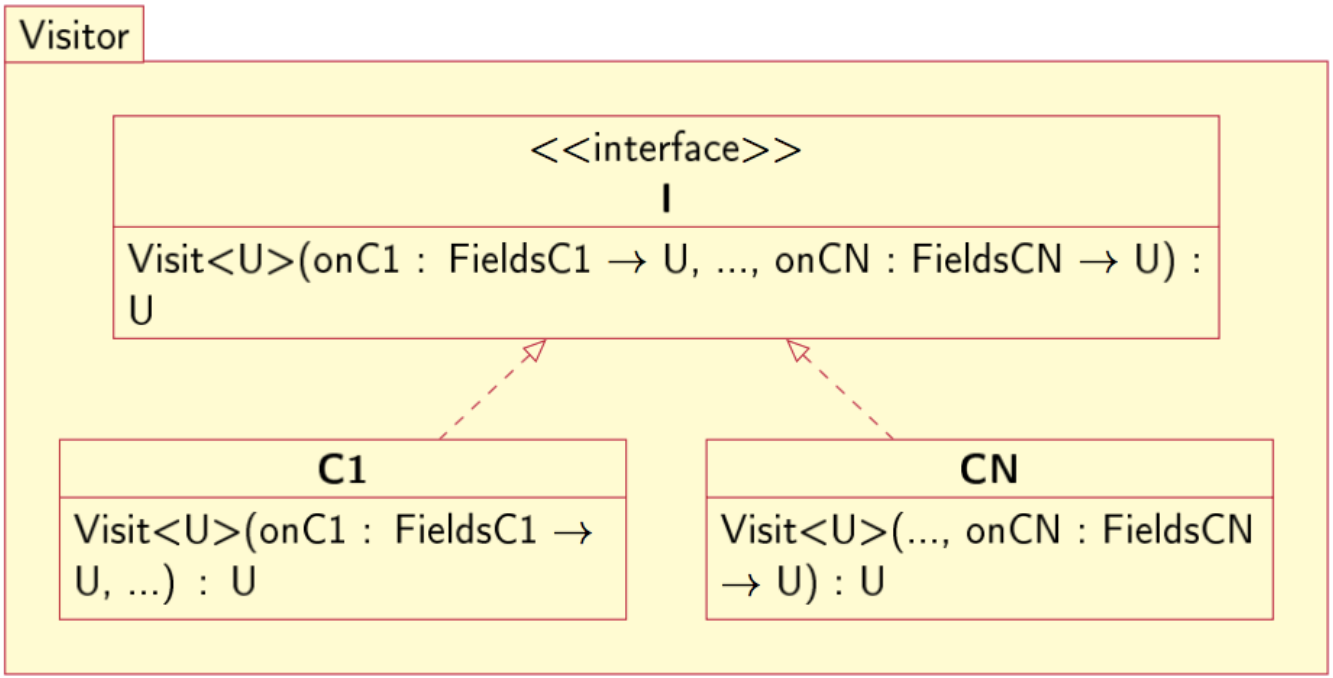
EXAMPLES FROM OFFICIAL SLIDES

Iterating an Option<T> with Visitor

```
Class IOptionIterator<T> : Iterator<T>{
  private Option<T> option;
  private bool visited;
  public IOptionIterator(Option<T> option){
    this.option = option;
    this.visited = false;
  }
  Option<T> GetNext(){
    if(visited){
      return new None<T>();
    }
    else{
      visited = true;
      return option.Visit<Option<T>>(() => new None<T>(), t => new
Some<T>(t)
    }
  }
}
```

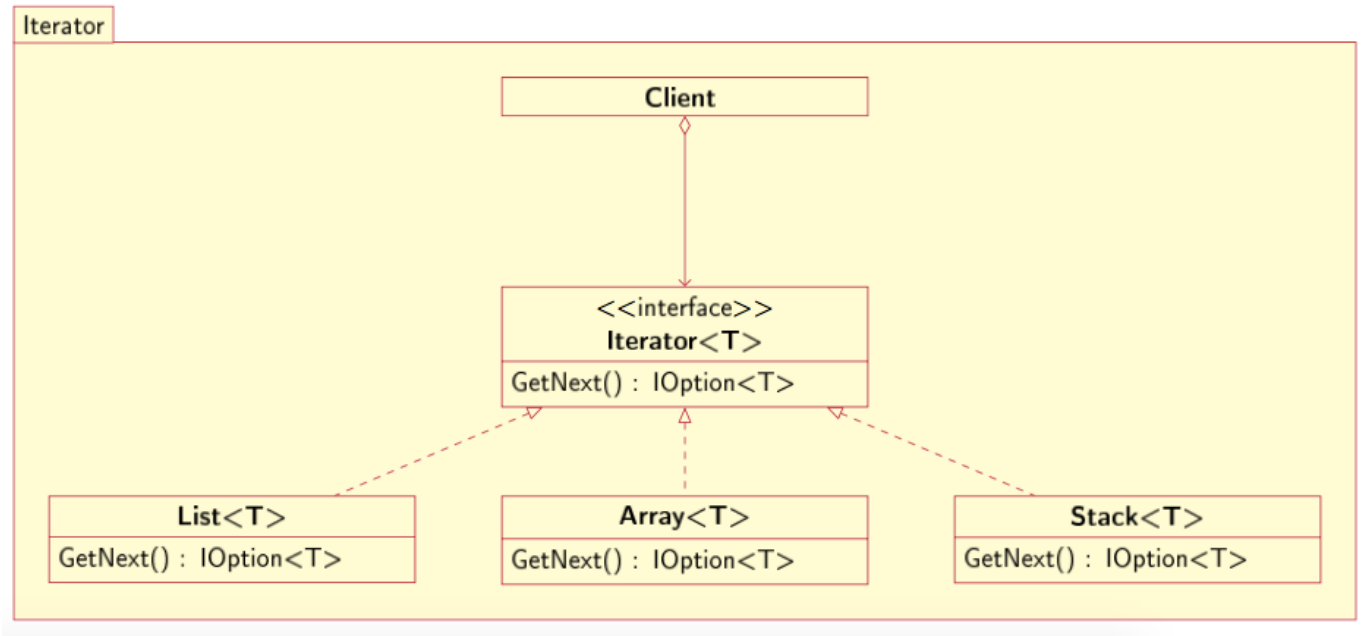
λ version of the Visitor

Intent: represent an operator to be performed on the elements of an object structure. Visitor allows a definition of a new operation without changing the classes of the elements on which it operates



λ version of the Iterator

Intent: a way to access elements of an aggregate object without exposing its underlying representation



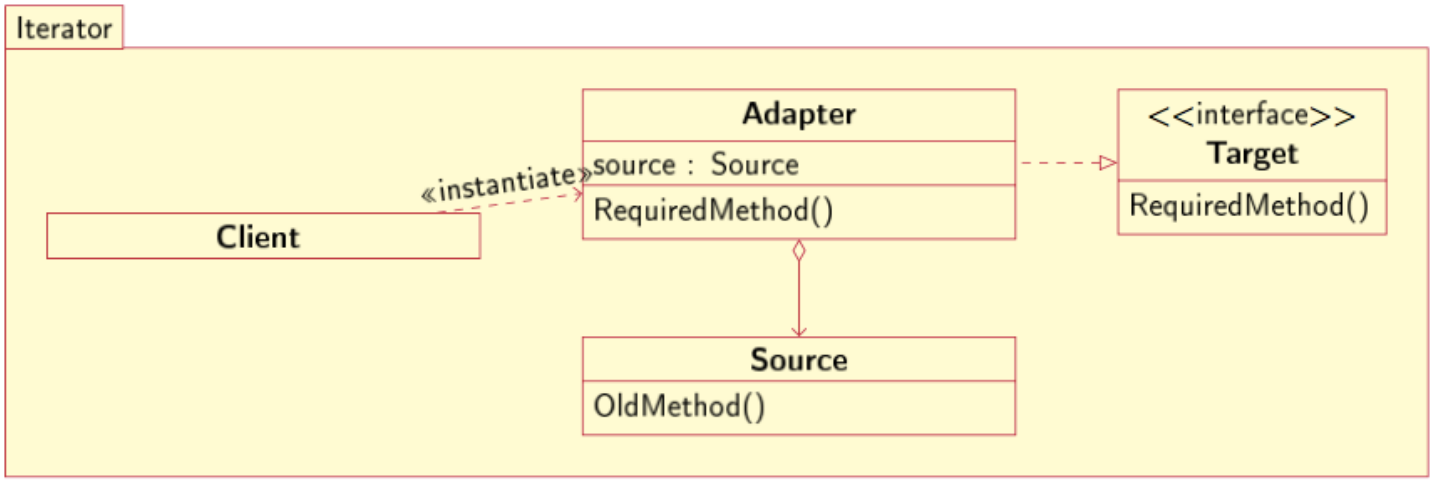
REVISING Λ NOTATION
ITERATOR: OFFICIAL SLIDES

REVISING Λ NOTATION
ADAPTER: OFFICIAL SLIDES

λ version of the Adapter

Intent: to convert the interface of a class into another interface that is expected by a client

(typo in the diagram below: title should read Adapter)



CONCLUSION

- **Today** we have revised last lesson's example, studied a new design pattern and applied it to a new example problem
- **You should have already started 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!