Template

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 21

Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Encapsulation

- Last time around we talked about Encapsulation…
- Encapsulation should be thought of as “any kind of hiding” especially the hiding of “things that can change”
  - We certainly can hide data but also
    - Methods
    - Objects
    - Type
    - Behavior
    - Implementations
    - Design details
    - Derived classes
    - Instantiation rules
    - etc.
- Templates encapsulate algorithm implementations…

Example: Tea and Coffee

- Consider another Starbuzz example in which we consider the recipes for making coffee and tea in a barista's training guide

  - Coffee
    - Boil water
    - Brew coffee in boiling water
    - Pour coffee in cup
    - Add sugar and milk

  - Tea
    - Boil water
    - Steep tea in boiling water
    - Pour tea in cup
    - Add lemon

Coffee Implementation

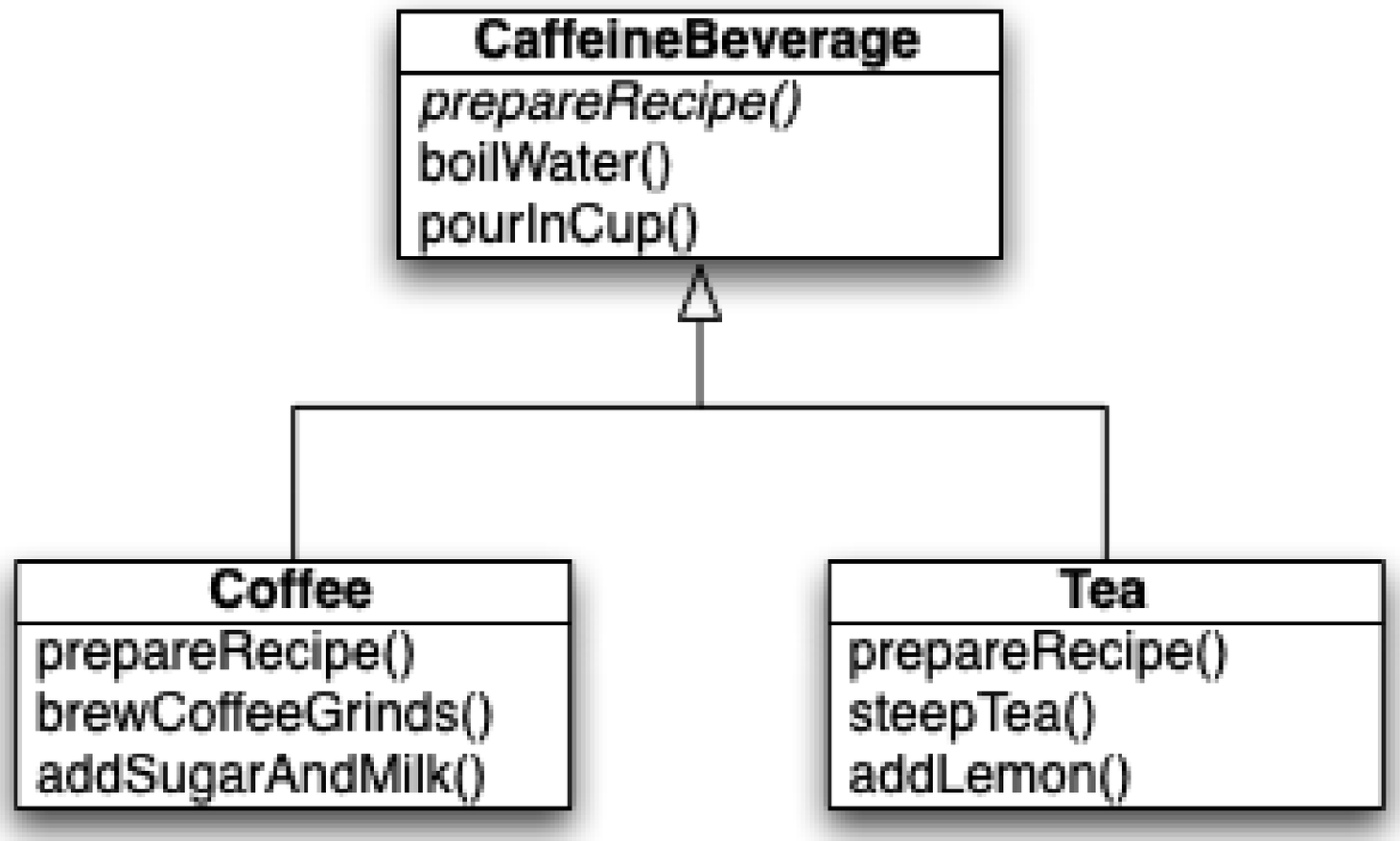
```
1  public class Coffee {
2
3      void prepareRecipe() {
4          boilWater();
5          brewCoffeeGrinds();
6          pourInCup();
7          addSugarAndMilk();
8      }
9
10     public void boilWater() {
11         System.out.println("Boiling water");
12     }
13
14     public void brewCoffeeGrinds() {
15         System.out.println("Dripping Coffee through filter");
16     }
17
18     public void pourInCup() {
19         System.out.println("Pouring into cup");
20     }
21
22     public void addSugarAndMilk() {
23         System.out.println("Adding Sugar and Milk");
24     }
25 }
26
```

Tea Implementation

```
1  public class Tea {
2
3      void prepareRecipe() {
4          boilWater();
5          steepTeaBag();
6          pourInCup();
7          addLemon();
8      }
9
10     public void boilWater() {
11         System.out.println("Boiling water");
12     }
13
14     public void steepTeaBag() {
15         System.out.println("Steeping the tea");
16     }
17
18     public void addLemon() {
19         System.out.println("Adding Lemon");
20     }
21
22     public void pourInCup() {
23         System.out.println("Pouring into cup");
24     }
25 }
26
```

Code smell: Duplication!

- We have code duplication occurring in these two classes
  - boilWater() and pourInCup() are exactly the same
- Let's get rid of the duplication, and abstract what’s common into a base class



Similar algorithms

- The structure of the algorithms in prepareRecipe() is similar for Tea and Coffee

Coffee

```
void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}
```

Tea

```
void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

Similar algorithms

- We can improve our code further by making the code in prepareRecipe() more abstract
    - brewCoffeeGrinds() and steepTea() ⇒ brew()
    - addSugarAndMilk() and addLemon() ⇒ addCondiments()

Controlling change - final

- Excellent, now all we need to do is specify this structure in CaffeineBeverage.prepareRecipe() and make it such that subclasses can't change the structure
    - How do we do that?
    - Answer: By convention OR by using a keyword like "final" in languages that support it
        - In Java – "final" can be used
            - in a variable definition to create a constant
            - with a method to prevent method overrides
            - with a class to prevent inheritance

CaffeineBeverage Implementation

```
1  public abstract class CaffeineBeverage {
2
3      final void prepareRecipe() {
4          boilWater();
5          brew();
6          pourInCup();
7          addCondiments();
8      }
9
10     abstract void brew();
11
12     abstract void addCondiments();
13
14     void boilWater() {
15         System.out.println("Boiling water");
16     }
17
18     void pourInCup() {
19         System.out.println("Pouring into cup");
20     }
21 }
22
```

Note: use of final keyword for prepareReceipe()

brew() and addCondiments() are abstract and must be supplied by subclasses

boilWater() and pourInCup() are specified and shared across all subclasses

Coffee And Tea Implementations

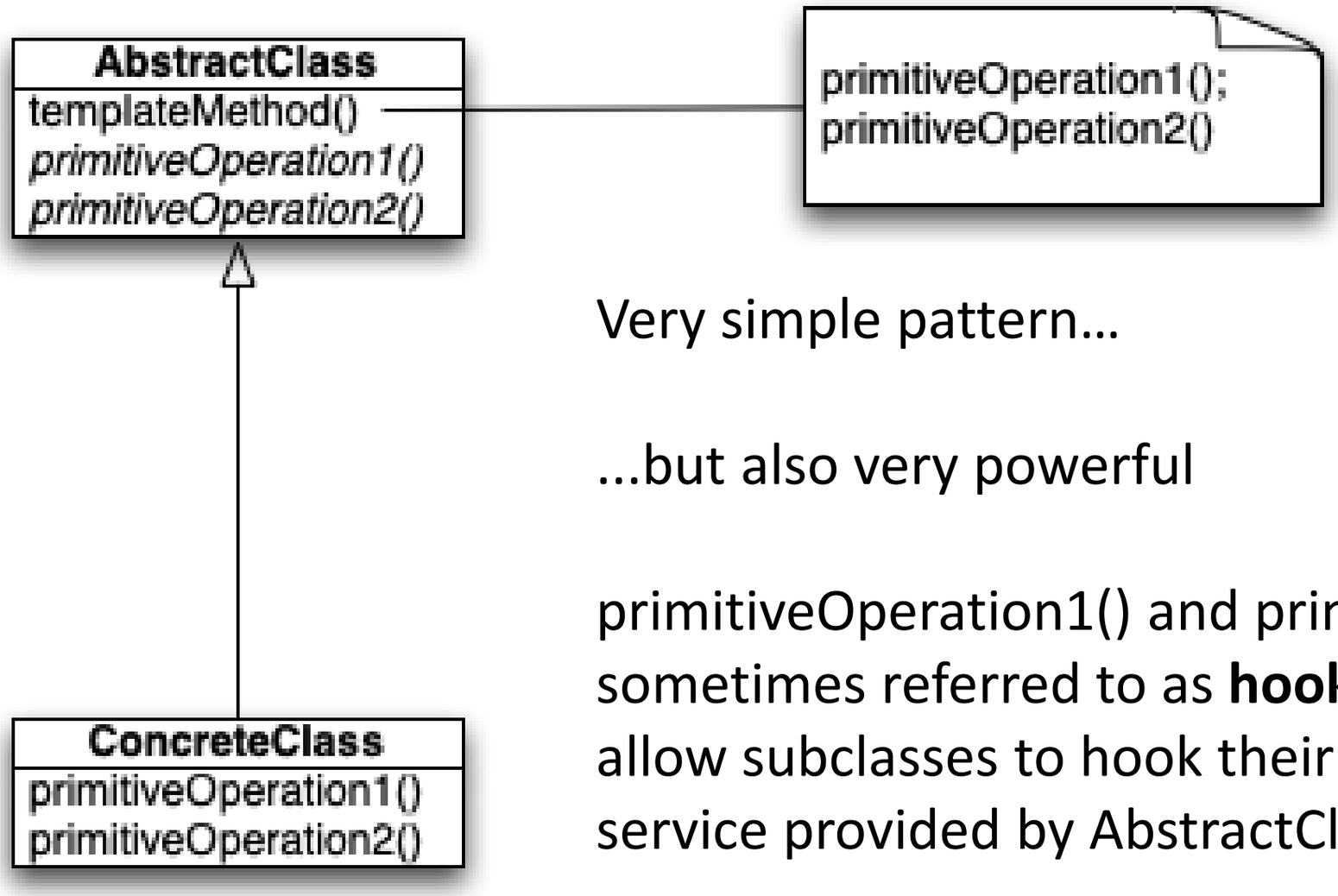
```
1  public class Coffee extends CaffeineBeverage {
2      public void brew() {
3          System.out.println("Dripping Coffee through filter");
4      }
5      public void addCondiments() {
6          System.out.println("Adding Sugar and Milk");
7      }
8  }
9
10 public class Tea extends CaffeineBeverage {
11     public void brew() {
12         System.out.println("Steeping the tea");
13     }
14     public void addCondiments() {
15         System.out.println("Adding Lemon");
16     }
17 }
18
```

Nice and Simple!

Template Method: Definition

- The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps
  - Makes the algorithm abstract
  - Each step of the algorithm is represented by a method
  - Encapsulates the details of most steps
    - Steps (methods) handled by subclasses are declared abstract
    - Shared steps (concrete methods) are placed in the same class that has the template method, allowing for code re-use among the various subclasses

Template Method: Structure



Very simple pattern…

...but also very powerful

primitiveOperation1() and primitiveOperation2() are sometimes referred to as **hook methods** as they allow subclasses to hook their behavior into the service provided by AbstractClass

Templates generally contain three types of methods: **concrete** methods, **abstract** methods (implemented by subclasses), and **hooks** (optionally implemented by subclass)

What have we done?

- Took two separate classes with separate but similar algorithms
- Noticed duplication and eliminated it by introducing a superclass
- Made steps of algorithm more abstract and specified its structure in the superclass
  - Thereby eliminating another “implicit” duplication between the two classes
- Revised subclasses to implement the abstract (unspecified) portions of the algorithm… in a way that made sense for them

Comparison: Template Method (TM) vs. No TM

- No Template Method
  - Coffee and Tea each have own copy of algorithm
  - Code is duplicated across both classes
  - A change in the algorithm would result in a change in both classes
  - Not easy to add new caffeine beverage
  - Knowledge of algorithm distributed over multiple classes
- Template Method
  - CaffeineBeverage has the algorithm and protects it
  - CaffeineBeverage shares common code with all subclasses
  - A change in the algorithm likely impacts only CaffeineBeverage
  - New caffeine beverages can easily be plugged in
  - CaffeineBeverage centralizes knowledge of the algorithm; subclasses plug in missing pieces

Adding a Hook to CaffeineBeverage

```
1  public abstract class CaffeineBeverageWithHook {
2
3      void prepareRecipe() {
4          boilWater();
5          brew();
6          pourInCup();
7          if (customerWantsCondiments()) {
8              addCondiments();
9          }
10     }
11
12     abstract void brew();
13
14     abstract void addCondiments();
15
16     void boilWater() {
17         System.out.println("Boiling water");
18     }
19
20     void pourInCup() {
21         System.out.println("Pouring into cup");
22     }
23
24     boolean customerWantsCondiments() {
25         return true;
26     }
27 }
28
```

prepareRecipe() altered to have a hook method: customerWantsCondiments()

This method provides a method body that subclasses can override

To make the distinction between hook and non-hook methods more clear, you can add the “final” keyword to all concrete methods that you don’t want subclasses to touch

```java
import java.io.*;

public class CoffeeWithHook extends CaffeineBeverageWithHook {

    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    public boolean customerWantsCondiments() {

        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}

```

Adding a Hook to Coffee

Design Principle: Hollywood Principle

- Don’t call us, we’ll call you
- Or, in OO terms, high-level components call low-level components, not the other way around
  - In the context of the template method pattern, the template method lives in a high-level class and invokes methods that live in its subclasses
- This principle is similar to the dependency inversion principle: “Depend upon abstractions. Do not depend upon concrete classes.”
  - Template method encourages clients to interact with the abstract class that defines template methods as much as possible; this discourages the client from depending on the template method subclasses

Template Methods in the Wild

- Template Method is used a lot since it's a great design tool for creating frameworks
  - the framework specifies how something should be done with a template method
  - that method invokes abstract hook methods that allow client-specific subclasses to "hook into" the framework and take advantage of its services
- Examples in the Java API:
  - Java.io has a read() method in InputStream the subclasses must implement; it's used by the template method read(byte b[], int off, int len).
  - Sorting using compareTo() method and the Comparable interface
    - We'll look at this one… Hmm… What should we sort?

Java Arrays sort

We actually have two methods here and they act together to provide the sort functionality.

The first method, sort(), is just a helper method that creates a copy of the array and passes it along as the destination array to the mergeSort() method. It also passes along the length of the array and tells the sort to start at the first element.

```
public static void sort(Object[] a) {
    Object aux[] = (Object[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

The mergeSort() method contains the sort algorithm, and relies on an implementation of the compareTo() method to complete the algorithm. If you're interested in the nitty gritty of how the sorting happens, you'll want to check out the Java source code.

Think of this as the template method.

```
private static void mergeSort(Object src[], Object dest[],
                              int low, int high, int off)
{
    // a lot of other code here
    for (int i=low; i<high; i++){
        for (int j=i; j>low &&
                 ((Comparable)dest[j-1]).compareTo((Comparable)dest[j])>0; j--)
        {
            swap(dest, j, j-1);
        }
    }
    // and a lot of other code here
}
```

This is a concrete method, already defined in the Arrays class.

compareTo() is the method we need to implement to "fill out" the template method.

Duck sorting?

Remember, we need to implement the Comparable interface since we aren't really subclassing.

```
public class Duck implements Comparable {
    String name;
    int weight;

    public Duck(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public String toString() {
        return name + " weighs " + weight;
    }
```

Our Ducks have a name and a weight

We're keepin' it simple; all Ducks do is print their name and weight!

Okay, here's what sort needs...

```
    public int compareTo(Object object) {

        Duck otherDuck = (Duck)object;

        if (this.weight < otherDuck.weight) {
            return -1;
        } else if (this.weight == otherDuck.weight) {
            return 0;
        } else { // this.weight > otherDuck.weight
            return 1;
        }
    }
}
```

compareTo() takes another Duck to compare THIS Duck to.

Here's where we specify how Ducks compare. If THIS Duck weighs less than otherDuck then we return -1; if they are equal, we return 0; and if THIS Duck weighs more, we return 1.

Duck sorting!

```
public class DuckSortTestDrive {

    public static void main(String[] args) {
        Duck[] ducks = {
                            new Duck("Daffy", 8),
                            new Duck("Dewey", 2),
                            new Duck("Howard", 7),
                            new Duck("Louie", 2),
                            new Duck("Donald", 10),
                            new Duck("Huey", 2)
         };

        System.out.println("Before sorting:");
        display(ducks);

        Arrays.sort(ducks);

        System.out.println("\nAfter sorting:");
        display(ducks);
    }

    public static void display(Duck[] ducks) {
        for (Duck d : ducks) {
            System.out.println(d);
        }
    }
}
```

We need an array of Ducks; these look good.

Let's print them to see their names and weights.

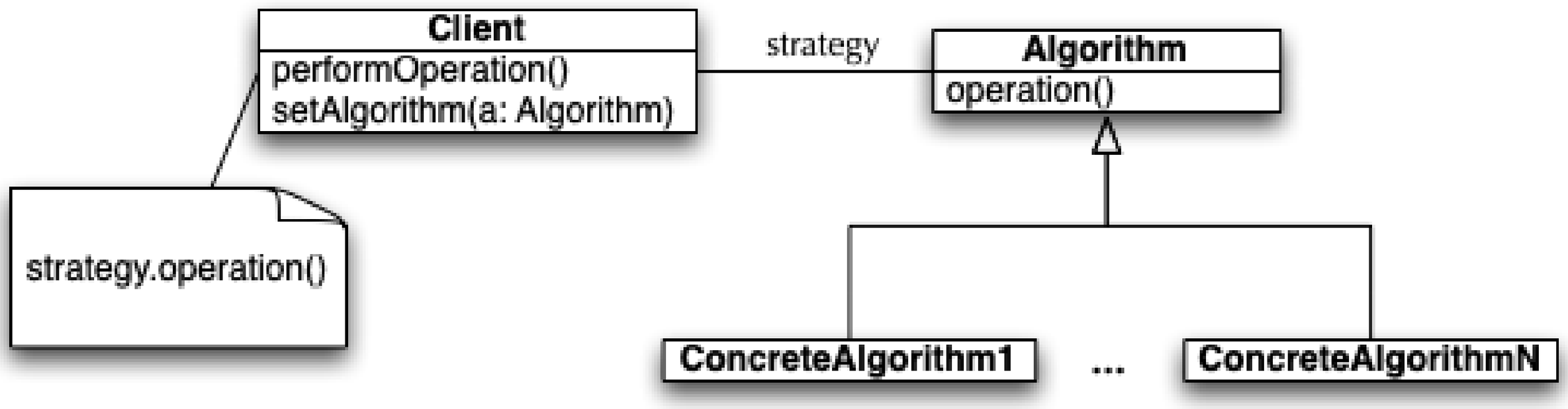
Notice that we call Arrays' static method sort, and pass it our Ducks.

It's sort time!

Let's print them (again) to see their names and weights.

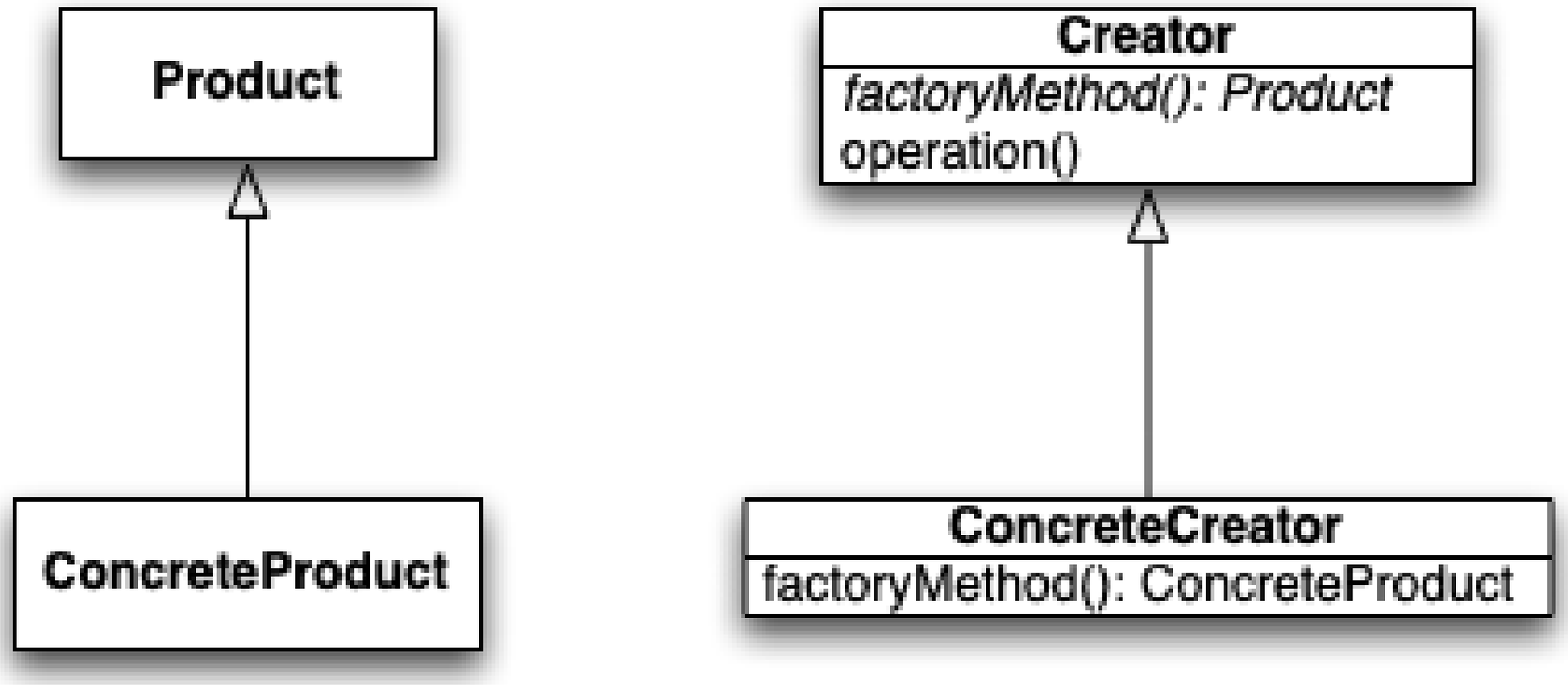
Template Method vs. Strategy

- Both Template Method and Strategy deal with the encapsulation of algorithms
  - Template Method focuses encapsulation **on the steps of the algorithm**
  - Strategy focuses on encapsulating **entire algorithms**
  - You can use both patterns at the same time if you want
- Strategy Structure



Template Method vs. Factory

- Template Method encapsulate the details of algorithms using inheritance
- Factory Method can now be seen as a specialization of the Template Method pattern



- In contrast, Strategy does a similar thing but uses composition/delegation

Template Method Comparisons

- Because it uses inheritance, Template Method offers code reuse benefits not typically seen with the Strategy pattern
- On the other hand, Strategy provides run-time flexibility because of its use of composition/delegation
  - You can switch to an entirely different algorithm when using Strategy, something that you can't do when using Template Method

Template in Python

An abstract class for the Template Method – includes the base algorithm, operations that are implemented, operations that need to be implemented by subclasses, and hook methods subclasses may override

https://refactoring.guru/design-patterns/template-method/python/example

```
from abc import ABC, abstractmethod

class AbstractClass(ABC):
    # The Abstract Class defines a template method that contains a skeleton of
    # some algorithm, composed of calls to abstract primitive operations

    def template_method(self) -> None:
        # The template method defines the skeleton of an algorithm
        self.base_operation1()
        self.required_operations1()
        self.base_operation2()
        self.hook1()
        self.required_operations2()
        self.base_operation3()
        self.hook2()

    # These operations already have implementations
    def base_operation1(self) -> None:
        print("AbstractClass says: I am doing the bulk of the work")

    def base_operation2(self) -> None:
        print("AbstractClass says: But I let subclasses override some operations")

    def base_operation3(self) -> None:
        print("AbstractClass says: But I am doing the bulk of the work anyway")

    # These operations have to be implemented in subclasses
    @abstractmethod
    def required_operations1(self) -> None:
        pass

    @abstractmethod
    def required_operations2(self) -> None:
        pass

    # These are "hooks." Subclasses may override them, but it's not mandatory
    # since the hooks already have default (but empty) implementation.
    # Hooks provide additional extension points in some crucial places of the
algorithm

    def hook1(self) -> None:
        pass

    def hook2(self) -> None:
        pass
```

Template in Python

Here we use the abstract Template to create two specialized Concrete Classes that override what's required, and possibly implement different hooks

https://refactoring.guru/design-patterns/template-method/python/example

```
class ConcreteClass1(AbstractClass):
    #   Concrete classes have to implement all abstract operations of the base class.
    #   They can also override some operations with a default implementation.

    def required_operations1(self) -> None:
        print("ConcreteClass1 says: Implemented Operation1")

    def required_operations2(self) -> None:
        print("ConcreteClass1 says: Implemented Operation2")


class ConcreteClass2(AbstractClass):

    def required_operations1(self) -> None:
        print("ConcreteClass2 says: Implemented Operation1")

    def required_operations2(self) -> None:
        print("ConcreteClass2 says: Implemented Operation2")

    def hook1(self) -> None:
        print("ConcreteClass2 says: Overridden Hook1")
```

```
def client_code(abstract_class: AbstractClass) -> None:
    # The client code calls the template method to execute the algorithm. Client
    # code does not have to know the concrete class of an object it works with, as
    # long as it works with objects through the interface of their base class.

    abstract_class.template_method()


if __name__ == "__main__":

    print("Same client code can work with different subclasses:")
    client_code(ConcreteClass1())
    print("")

    print("Same client code can work with different subclasses:")
    client_code(ConcreteClass2())
```

Summary of Template

- Template Method – defines steps of algorithm, lets subclasses implement some steps
- Useful for code reuse
- The Template abstract class contains concrete methods, abstract methods (implemented by subclasses), and hooks
- Hooks are methods that do nothing or do a default behavior in the abstract class, but may be overridden in the subclass
- Use final to prevent subclasses from changing elements
- Hollywood Principle – put decision making in higher-level modules that decide how/when to call low-level modules
- Strategy and Template both encapsulate algorithms, Template uses inheritance, Strategy uses composition
- Factory is a specialization of Template