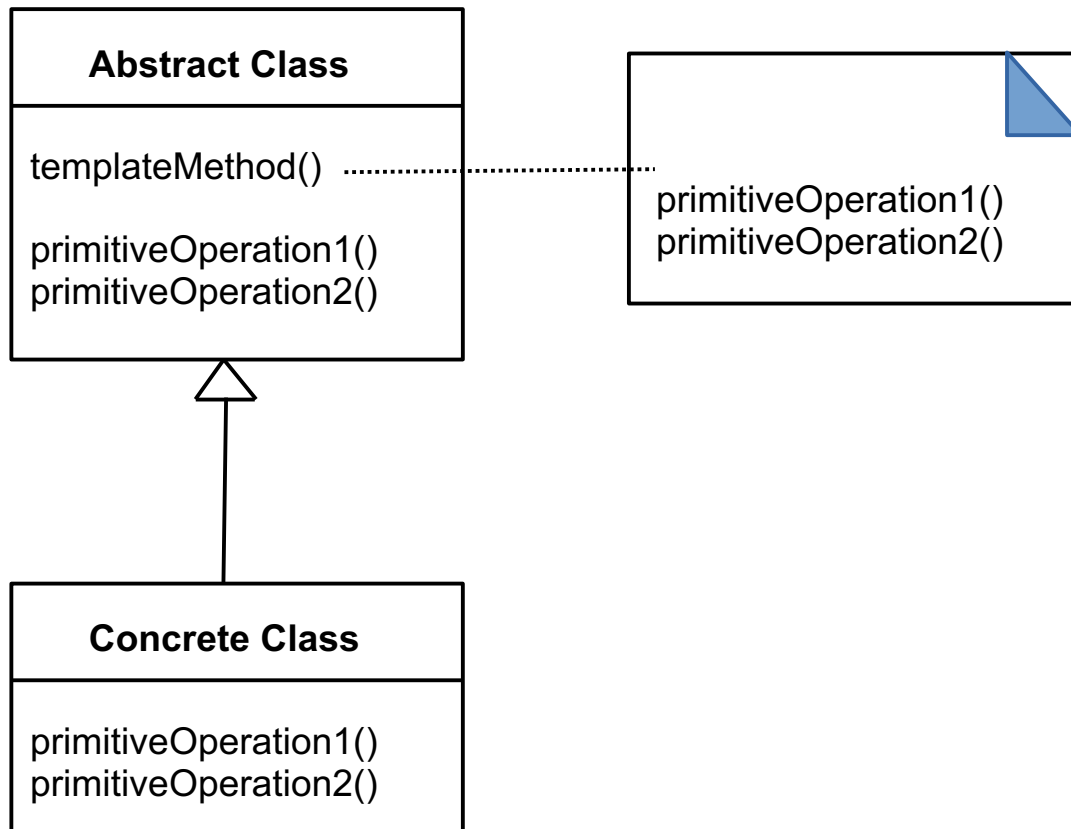


# Template Pattern

# Definition of Template Pattern

- 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.”



```
abstract class AbstractClass {
    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }

    abstract void primitiveOperation1();
    abstract void primitiveOperation2();

    void concreteOperation(){
        // implementation of it here ...
    }
}
```

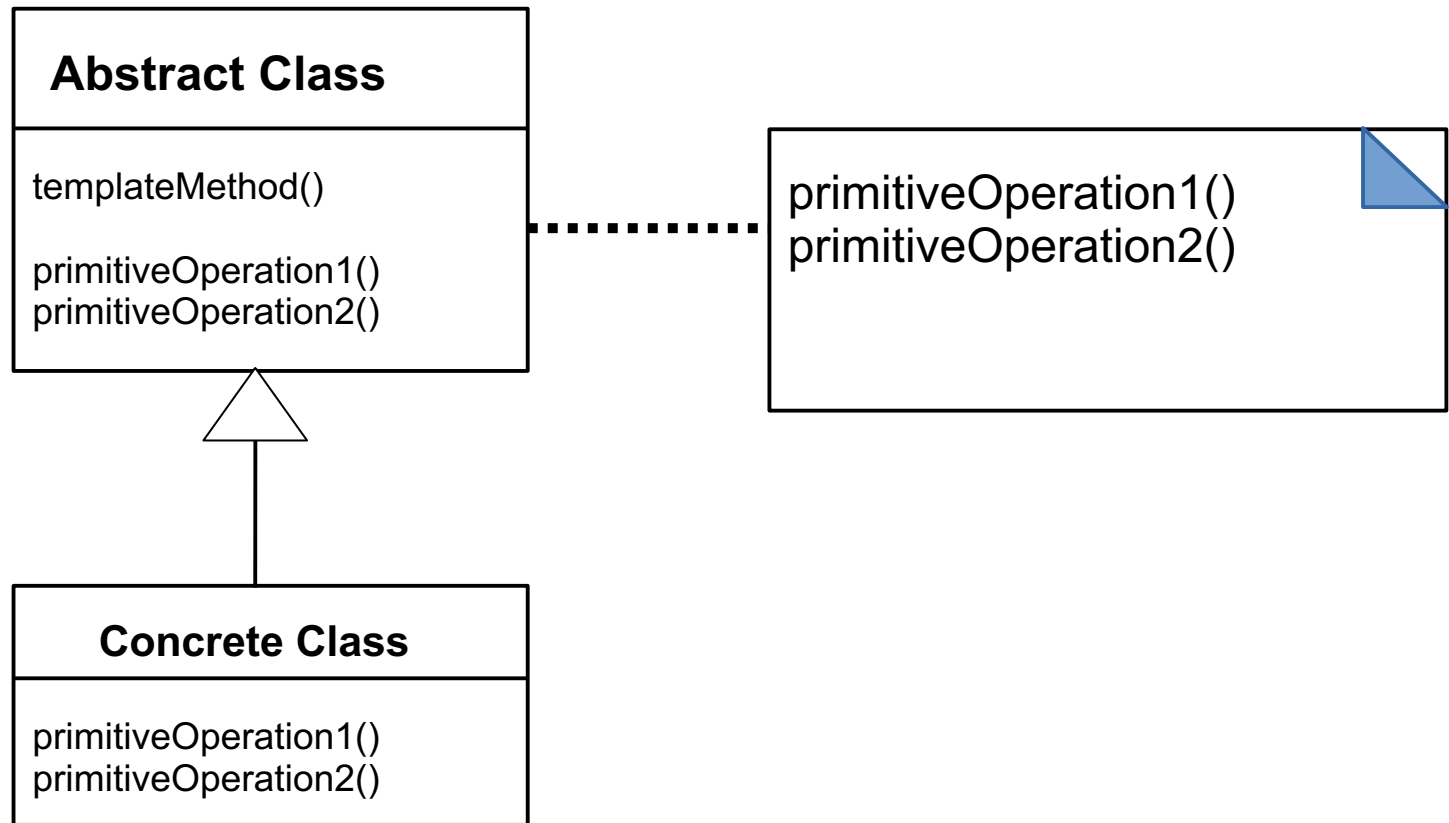
# Use Case 1 – Brew Tee and Coffee

```
public class Tea {  
    void prepareRecipe() {  
        boilWater();  
        steepTeaBag();  
        pourInCup();  
        addLemon();  
    }  
    public void boilWater() {  
        System.out.println("Boiling water");  
    }  
    public void steepTeaBag() {  
        System.out.println("Steeping the tea");  
    }  
    public void addLemon() {  
        System.out.println("Adding Lemon");  
    }  
    public void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```

```
public class Coffee {  
    void prepareRecipe() {  
        boilWater();  
        brewCoffeeGrinds();  
        pourInCup();  
        addSugarAndMilk();  
    }  
    public void boilWater() {  
        System.out.println("Boiling water");  
    }  
    public void brewCoffeeGrinds() {  
        System.out.println("Dripping Coffee through filter");  
    }  
    public void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
    public void addSugarAndMilk() {  
        System.out.println("Adding Sugar and Milk");  
    }  
}
```

```
public class Barista {  
    public static void main(String[] args) {  
        Tea tea = new Tea();  
        Coffee coffee = new Coffee();  
        System.out.println("Making tea...");  
        tea.prepareRecipe();  
        System.out.println("Making coffee...");  
        coffee.prepareRecipe();  
    }  
}
```

# Participants



- **Abstract Class**

- defines the domain-specific interface that Client uses.

- **Concrete Class**

- collaborates with objects conforming to the Target interface.

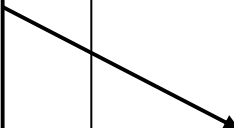
# Hook Operation

- A **Hook** is a **method** that is declared in Abstract Class but has an empty implementation or a default one.
- It gives the subclasses to **hook into the main algorithm** at different points in the case they need to do that. A subclass is also free to ignore the “hook” if there is no need for that.

**Hook operations** provide default behavior that subclasses can extend if necessary. A hook operation often does nothing by default.

A concrete method that may do nothing by default. Subclasses may override but they do not have to.

```
abstract class AbstractClass {  
    final void templateMethod() {  
        primitiveOperation1();  
        primitvieoperation2();  
        concreteOperation();  
        hook();  
    }  
    abstract void  
    primitiveOperation1();  
    abstract void  
    primitiveOperation1();  
    void concreteOperation(){  
        // implementation of it here ...  
    }  
    void hook(){}  
}
```



# Using Hook Method

```
public abstract class CaffeineBeverageWithHook {  
  
    final void prepareRecipe() {  
        boilWater();  
        brew();  
        pourInCup();  
        if (customerWantsCondiments()) {  
            addCondiments();  
        }  
    }  
  
    abstract void brew();  
    abstract void addCondiments();  
  
    void boilWater() {  
        System.out.println("Boiling water");  
    }  
  
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
    boolean customerWantsCondiments() {  
        return true;  
    }  
}
```

```
public class CoffeeWithHook extends  
CaffeineBeverageWithHook {  
    public void brew() {  
        ... // some implementation  
    }  
    public void addCondiments() {  
        ... // some implementation  
    }  
  
    public boolean customerWantsCondiments()  
    {  
        String answer = getUserInput();  
  
        if (answer=="yes") {  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    private String getUserInput() {  
        ...  
        // get the user's answer  
        return answer;  
    }  
}
```

# When to use the Template Method Pattern

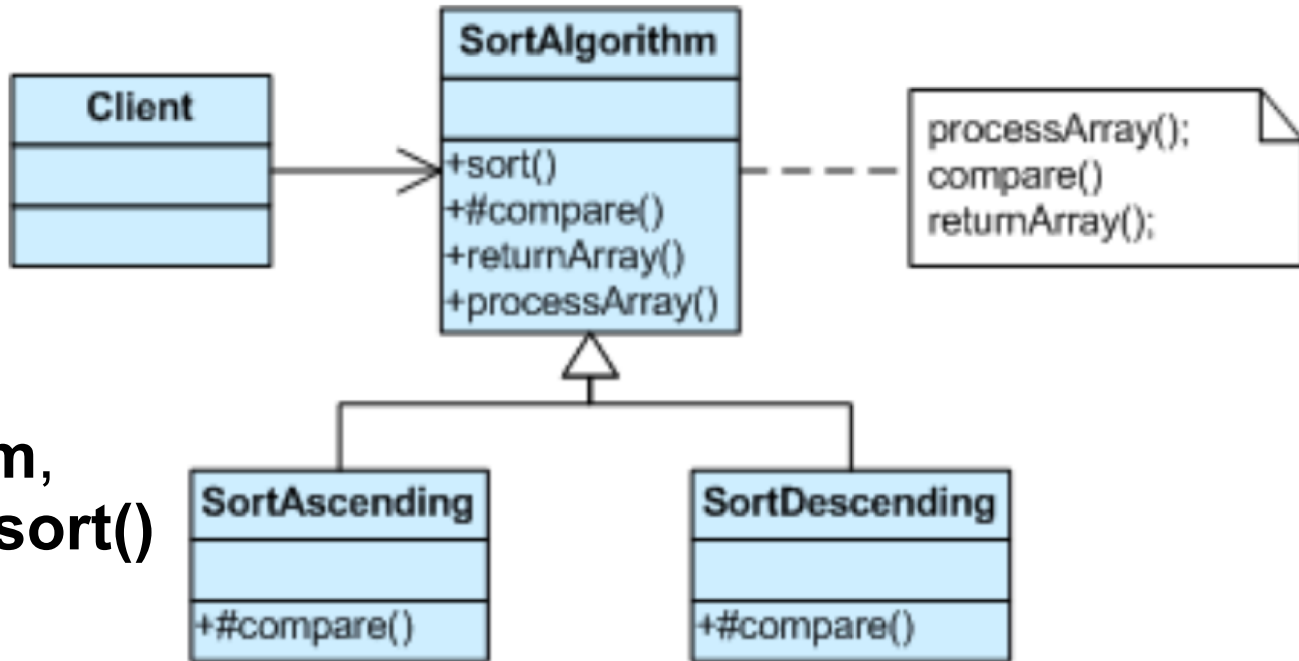
- Use Template Method pattern when
  - you want to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
  - common behavior among subclasses should be factored and localized in a common class to avoid code duplication.

# Refactoring to Generalize

1. identify the differences in the existing code
  2. separate the differences into new operations
  3. replace the differing code with a template method
- You want to control subclasses extensions.
  - You can define a template method that calls **"hook" operations** at specific points, and so you permit extensions only at those points or parts

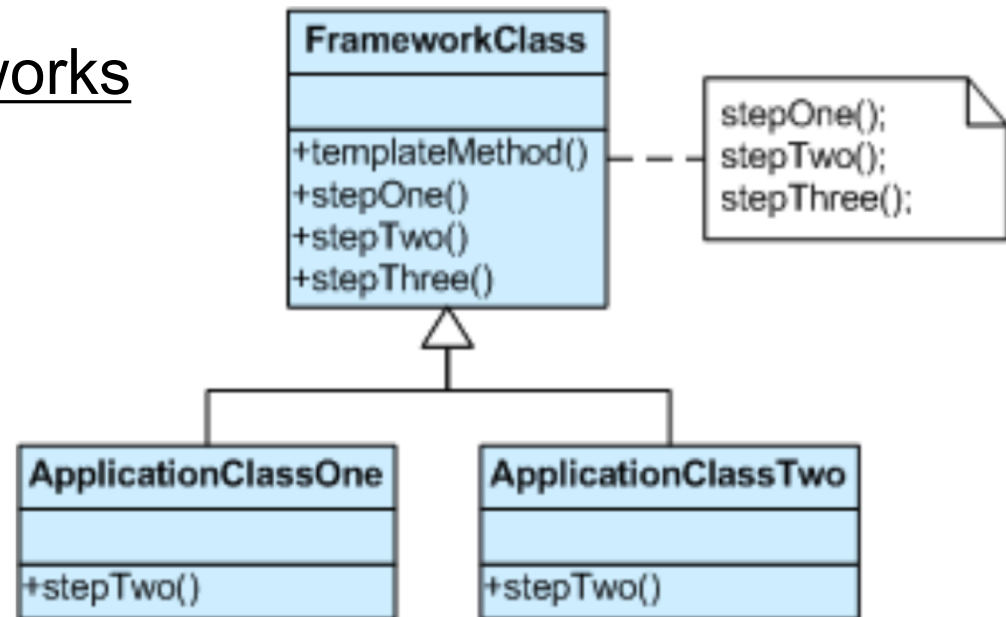


# Other Examples



For example used to implement **sortAlgorithm**,  
Like in **Java Collection.sort()**

## Used to implement Frameworks



# Consequences of using Template Pattern

- Template methods are a fundamental technique for code reuse. (Refactoring out the common behavior in library classes)
- Template methods lead to an inverted control structure that's sometimes referred to as "**the Hollywood principle**," that is, "**Don't call us, we'll call you**". This refers to how **a parent class calls the operations of a subclass and not the other way around**.
- Subclasses override **Hook Operation** to extend the behavior

# Related Patterns

- **Strategy Pattern**

Strategy is like Template Method except in its granularity.

Template Method uses inheritance to vary part of an algorithm.

Strategy uses delegation to vary the entire algorithm.

Strategy modifies the logic of individual objects. Template Method modifies the logic of an entire class.

- **Factory Method** is a specialization of **Template Method**.

# Summary of Template Method Pattern

- Template Pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses.

Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

- Template Method is of type **Behavioral Patterns**

# Use Case – Implement This Example

Home builders use the Template Method when developing a new subdivision.

A typical subdivision consists of a limited number of floor plans with different variations available for each.

Variation is introduced in the later stages of construction to produce a wider variety of models.

