Template Method Pattern
Composite Pattern

514770-1
Fall 2024
11/20/2024
Kyoung Shin Park
Computer Engineering
Dankook University

Template Method Pattern

- "Define **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."
- Base class declares algorithm 'placeholders', and derived classes implement the placeholders.
- Base class has a preset structure method, called template method, which defines the steps to execute an algorithm. This steps can be an abstract method which will be implemented by its subclasses.

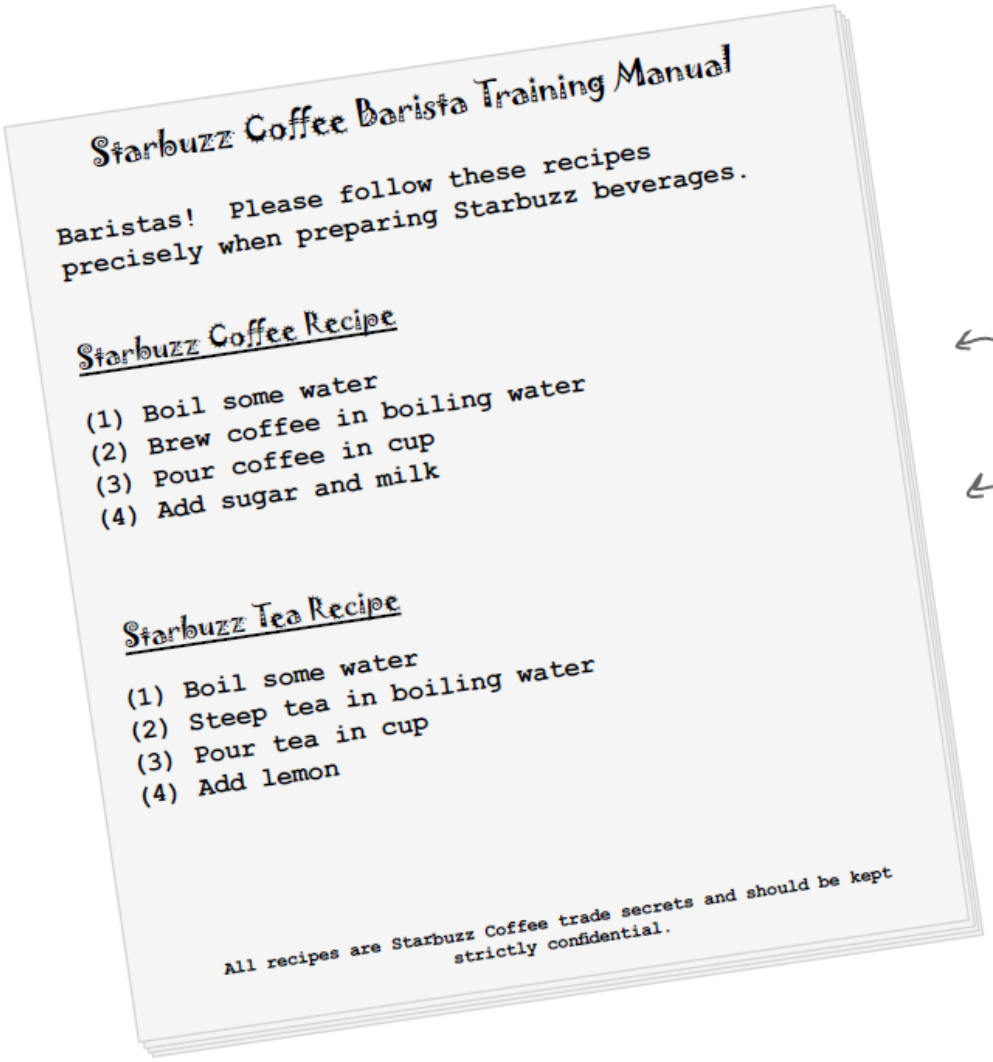
Template Method Pattern

- The template method pattern is quite common in framework, e.g. Spring framework. Developers often use it to provide framework users with a simple means of extending standard functionality using inheritance.
- All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader, java.io.Writer
- All non-abstract methods of java.util.AbstractList, java.util.AbstractSet, java.util.AbstractMap. AbstractList provides a skeletal implementation of the List interface.
- javax.servlet.http.HttpServlet, all the *doXXX()* methods (most commonly, doGet or doPost) are called from the *service()* method. A template method, *service()*, defers some of its processing to individual methods defined in subclasses.

Template Method Pattern

	Description
Pattern	Template Method
Problem	A set of object that all follow the same algorithm, but which vary the implementation of particular steps in that algorithm.
Solution	Encapsulate the same algorithm in the base class and implement only the different parts in the subclass.
Result	Avoid duplication in the code, Easy to maintain

Barista Training Manual (HFDP Ch. 8)



Starbuzz Coffee Barista Training Manual

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

(1) Boil some water
(2) Brew coffee in boiling water
(3) Pour coffee in cup
(4) Add sugar and milk

Starbuzz Tea Recipe

(1) Boil some water
(2) Steep tea in boiling water
(3) Pour tea in cup
(4) Add lemon

All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

The recipe for coffee looks a lot like the recipe for tea, doesn't it?

```
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");
    }
}
```

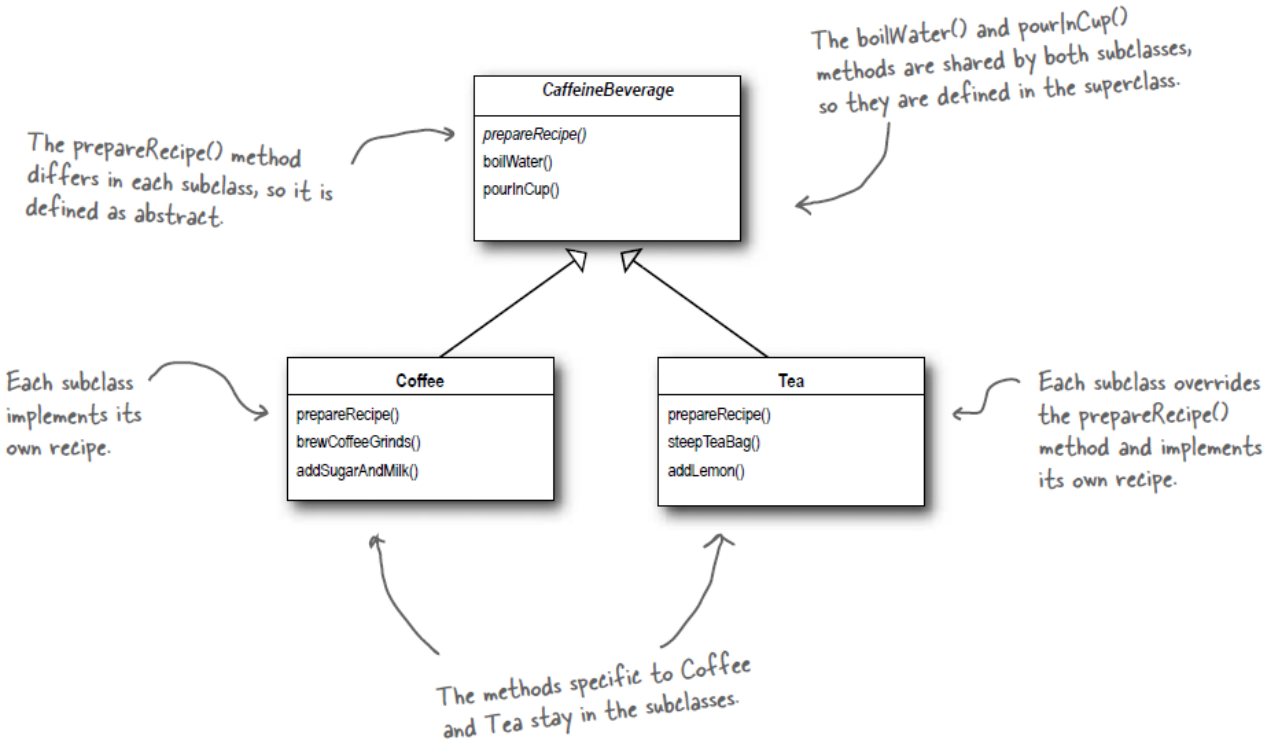
Recipe for coffee – each step is implemented as a separate method.

```
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 pourInCup() {
        System.out.println("Pouring into cup");
    }
    public void addLemon() {
        System.out.println("Adding Lemon");
    }
}
```

Very similar to coffee – boilWater and pourInCup are exactly the same.

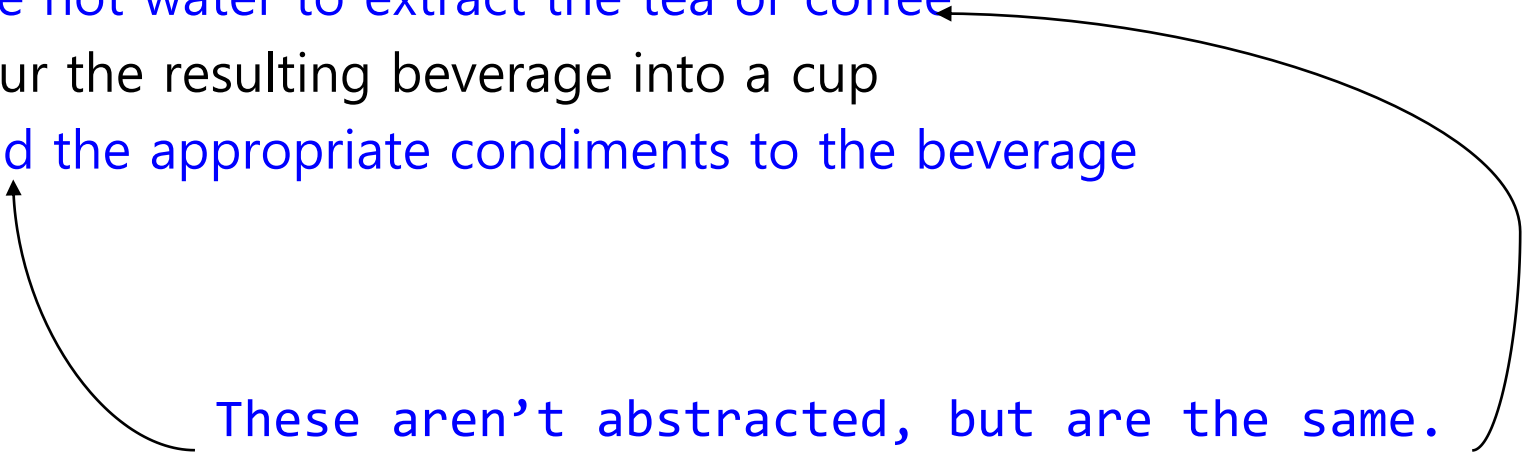
May I abstract your Coffee, Tea?

- Is this a good redesign? Are we overlooking some other commonality? What are other ways that Coffee and Tea are similar?



What else do they have in common?

- Both the recipes follow the same algorithm:
  1. Boil some water
  2. Use hot water to extract the tea or coffee
  3. Pour the resulting beverage into a cup
  4. Add the appropriate condiments to the beverage



Abstracting prepareRecipe()

- Provide a common interface for the different methods
  - Coffee uses brewCoffeeGrinds() and addSugarAndMilk() methods while Tea uses steepTeaBag() and addLemon() methods.
  - Steeping and brewing are pretty analogous. - so a common interface may be the ticket: brew() and addCondiments().

```
void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
}
```

Abstracting prepareRecipe()

```
public abstract class CaffeineBeverage {
    final void prepareRecipe() { // template method
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
    // methods that need to be supplied by the
    // subclass are declared abstract.
    abstract void brew();
    abstract void addCondiments();
    void boilWater() {
        System.out.println("Boiling water");
    }
    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

```
public class Tea extends CaffeineBeverage {
    @Override
    void brew() {
        System.out.println("Steeping the tea");
    }
    @Override
    void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
public class Coffee extends CaffeineBeverage {
    @Override
    void brew() {
        System.out.println("Dripping Coffee through
filter");
    }
    @Override
    void addCondiments() {
        System.out.println("Adding Sugar and
Milk");
    }
}
```

Let's make some tea..

- First, we need a Tea object.
- Then, we call the template method which follows the algorithm for making caffeine beverages.

```
Tea myTea = new Tea();
myTea.prepareRecipe(); // polymorphism ensures that
while the template controls everything, it still
calls the right methods.
```

Hooked on the Template Method

- A hook is a method that is declared in the abstract class, but only given an empty or default implementation.
  - Gives the subclasses the ability to "hook into" the algorithm at various points, if they wish; they can ignore the hook as well.

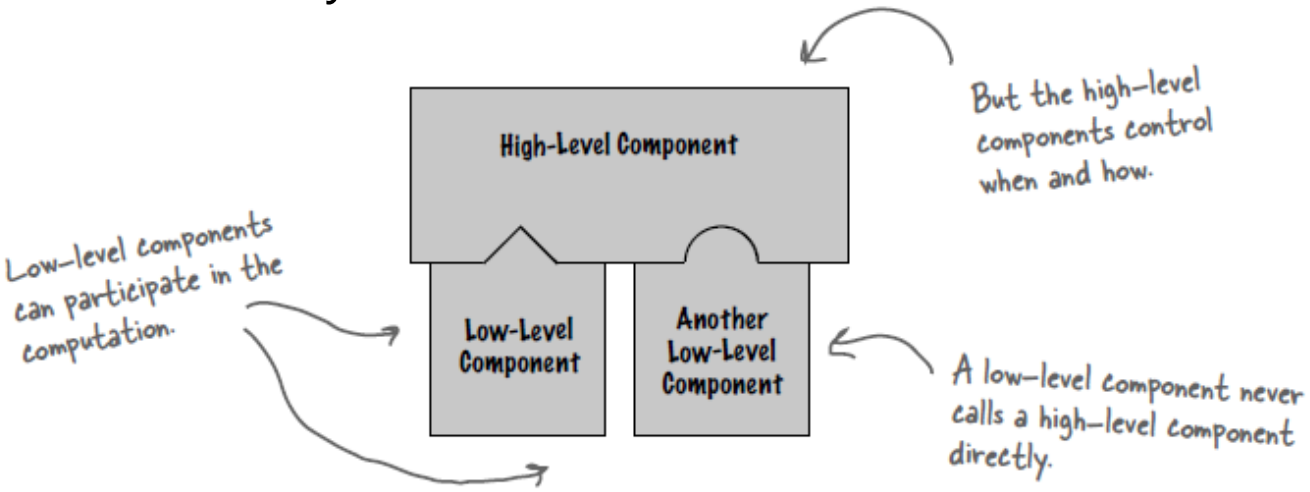
```
public abstract class CaffeineBeverageWithHook {
    final void prepareRecipe() { // template method
        boilWater();
        brew();
        pourInCup();
        if (wantCondiments()) addCondiments();
    }
    // This is a hook. If subclasses wan to use
    // the hook they simple override it!
    boolean wantCondiments() {
        return true;
    }
}
```

Using the Hook

```
public class CoffeeWithHook extends
CaffeineBeverageWithHook {
    // brew()
    // addCondiments()
    public boolean wantCondiments() {
        String answer = getUserInput();
        if (answer.toLowerCase().startWith("y"))
            return true;
        else
            return false;
    }
    public String getUserInput() {
        String answer = null;
        // get user input…
        return answer;
    }
}
```

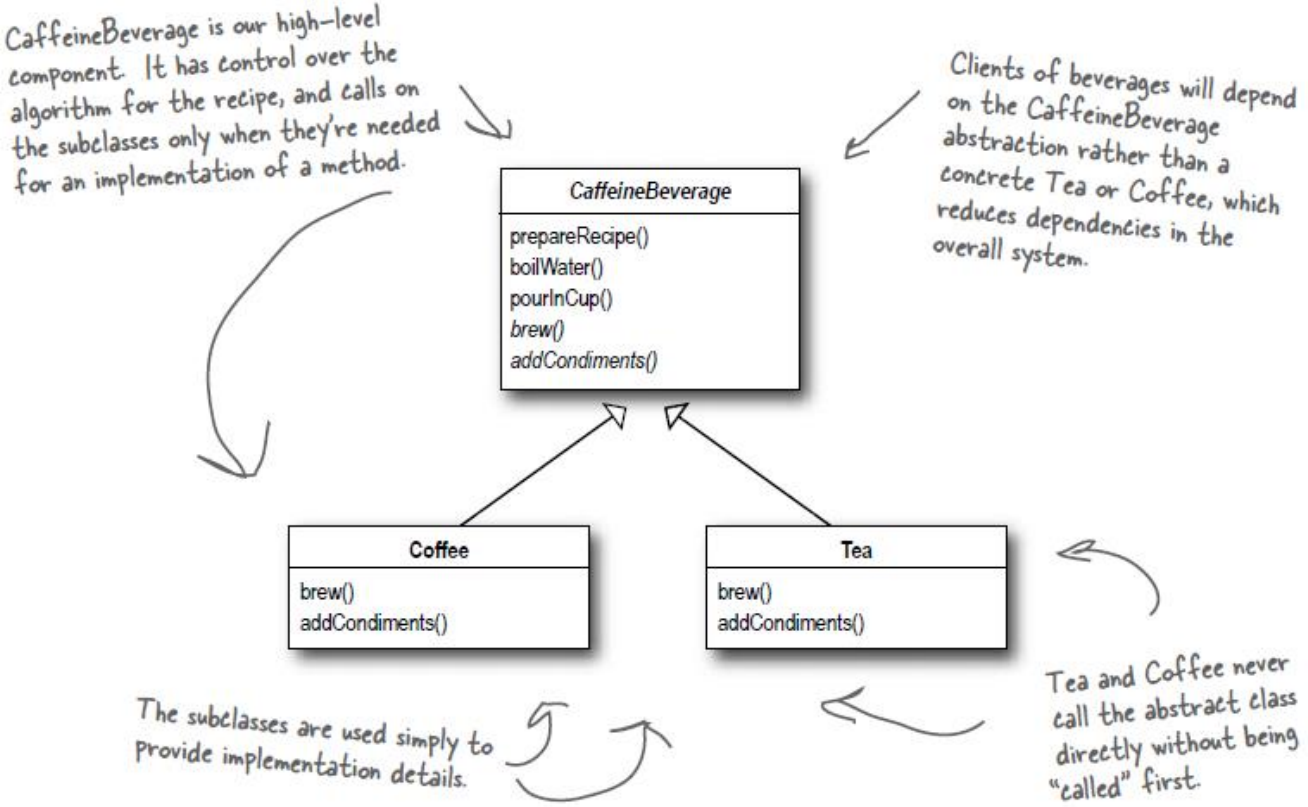
Hollywood Principle

- With the Hollywood principle, "Don't call us, we'll call you!",
  - We allow low level components to hook themselves into a system.
  - But high level components determine when they are needed and how.
  - High level components give the low-level components a "don't call us, we'll call you" treatment.

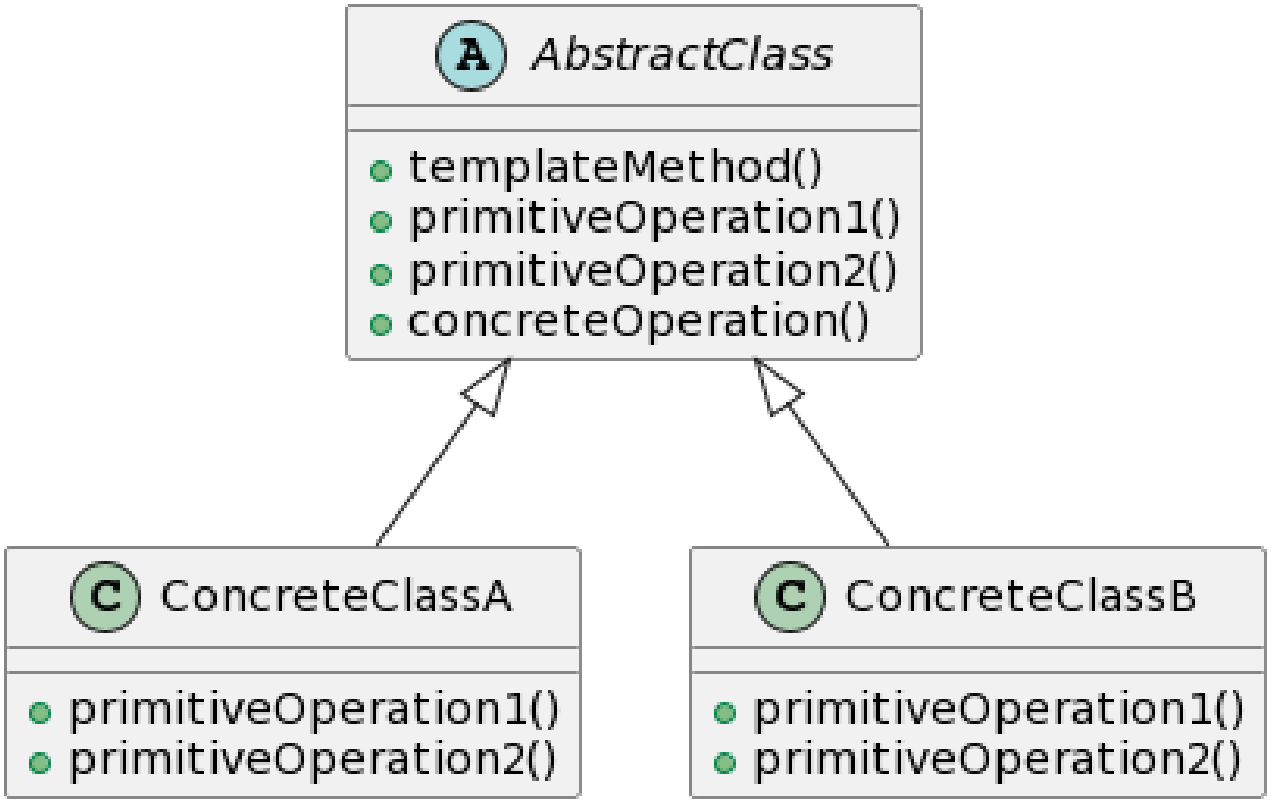


Hollywood Principle & Template Method

- When we design with the Template Method pattern, we're telling subclasses, "don't call us, we'll call you."



Template Method Pattern



Define Template Method Pattern

- AbstractClass
  - Defines an **algorithm that uses primitive operations** that are supplied by a subclass.
  - AbstractClass contains the **template method.** Template method can be recognized by behavioral methods that already have a "default" behavior defined by the base class.
- ConcreteClassA, ConcreteClassB
  - Implements the **primitive** operations to carry out subclass-specific steps of the algorithm.

Template Method Implementation

- Give primitive and hook methods protected access
  - These methods are intended to be called by a template method, and not directly by clients
- Declare primitive methods as abstract in the superclass.
  - Primitive methods must be implemented by subclasses.
- Declare hook methods as non-abstract
  - Hook methods may optionally be overridden by subclasses.
- Declare template methods as final
  - This prevents a subclass from overriding the method and interfering with it's algorithm structure.

Template Method Implementation

```
public abstract class AbstractTemplateWithHook {
    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }
    abstract void primitiveOperation1();
    abstract void primitiveOperation2();
    final void concreteOperation() {
        System.out.println("concrete operation");
    }
    // A subclass can override the hook if it wish
    boolean hook() {
        return true;
    }
}
```

```
public class ConcreteTemplateWithHook extends
AbstractTemplateWithHook {
    @Override
    void primitiveOperation1() {
        System.out.println("subclass operation1");
    }
    @Override
    void primitiveOperation2() {
        System.out.println("subclass operation2");
    }
    public boolean hook() {
        return false;
    }
}
public class TemplateMainTest {
    public static void main(String[] args) {
        AbstractTemplateWithHook temp
            = new ConcreteTemplateWithHook();
        temp.templateMethod();
    }
}
```

Sorting with Template Method

- Java Arrays class sort() method

```
// sort()
public static void sort(Object[] a) {
    Object aux[] = (Object[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
// mergeSort() contains the sort algorithm
private static void mergeSort(Object[] src,
Object[] dest, int low, int high, int off) {
    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, i, j-1); // concrete method
        }
    }
}
```

Sorting with Template Method

- Duck needs to implement the Comparable interface.

```
public class Duck implements Comparable {
    String name;
    double weight;
    public Duck(String name, double weight) {
        this.name = name;
        this.weight = weight;
    }
    public String toString() {
        return name + " " + weight;
    }
    public int compareTo(Object object) {
        Duck other = (Duck)object;
        return Double.compare(this.weight,
other.weight);
    }
}
```

Sorting with Template Method

```
public class DuckSortMainTest {
    public static void main(String[] args) {
        Duck[] ducks = { new Duck("Daffy", 8),
new Duck("Howard", 7), new Duck("Louie", 2),
new Duck("Donald", 10), new Duck("Huey", 5)};
        System.out.println("before sorting");
        display(ducks);
        Arrays.sort(ducks);
        System.out.println("after sorting");
        display(ducks);
    }
    public static void display(Duck[] ducks) {
        for (Duck duck : ducks)
            System.out.println(duck);
    }
}
```

Swingin' with Frame

- JFrame is the most basic Swing container and inherits a **paint()** method.
- Default behavior of paint() method - does nothing because it is a hook.
- By overriding the paint() method, you can insert yourself into JFrame's algorithm for displaying its area of screen and have your own graphic output incorporated into the JFrame.

Simple JFrame Example

```
public class MyFrame extends JFrame {
    public MyFrame(String title) {
        super(title);
        this.setSize(300,300);
        this.setVisible(true);
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    // JFrame update algorithm calls paint().
    // By default paint() does nothing. It's a hook.
    // We are overriding paint() to draw msg.
    public void paint(Graphics g) {
        super.paint(g);
        String msg = "I rule!!";
        g.drawstring(msg, 100, 100);
    }
    public static void main(String[] args) {
        new MyFrame("Simple Frame");
    }
}
```

Applets

- Applets provide numerous hooks. Concrete applets make extensive use of hooks to supply their own behaviors.

```
public class MyApplet extends Applet {
    String msg;
    public void init() { // init hook
        msg = "Hello, I'm alive!";
        repaint(); // concrete method in Applet
    }
    public void start() { // start hook
        msg = "Now, starting up"; repaint();
    }
    public void stop() { // stop hook
        msg = "Being stopped"; repaint();
    }
    public void destroy() { } // destroy hook
    public void paint(Graphics g) { // paint hook
        g.drawstring(msg, 5, 15);
    }
}
```

Template Method vs Strategy vs Factory

- 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.

Code Redundancy

- NaturalNumMaxFinder

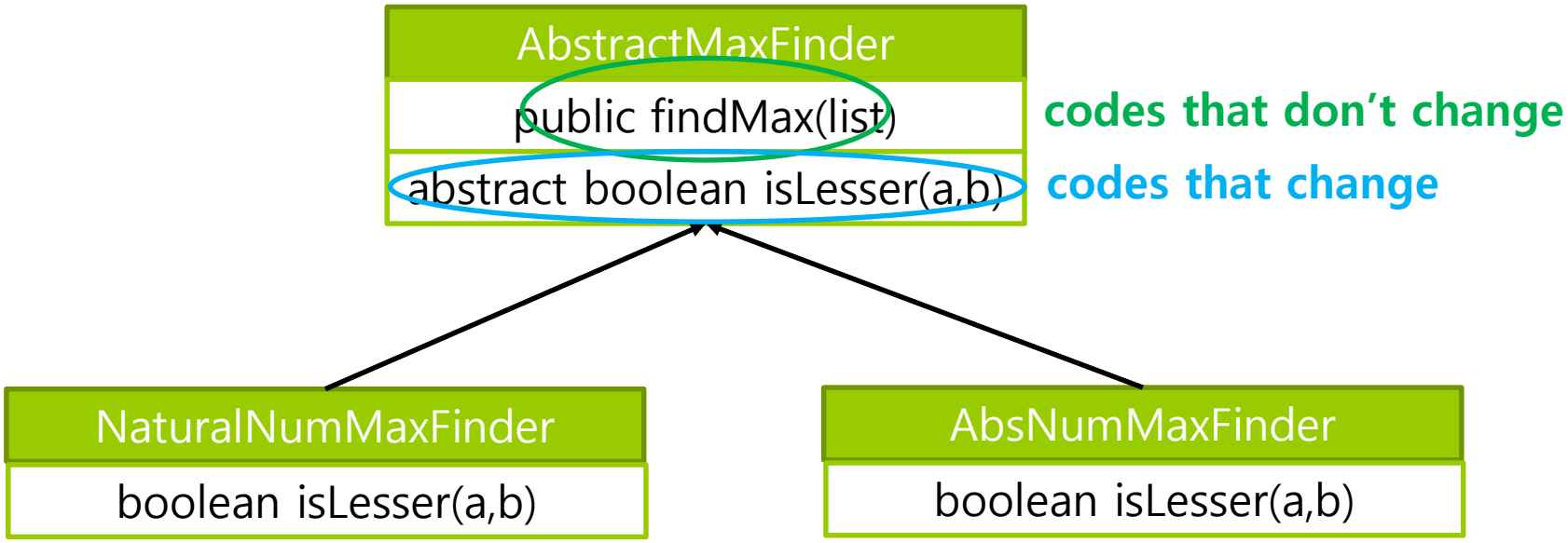
```
int max = numbers[0];
for (int i=1; i<numbers.length;
i++) {
    if (max < numbers[i])
        max = numbers[i];
}
return max;
```

- AbsNumMaxFinder

```
int max = numbers[0];
for (int i=1; i<numbers.length;
i++) {
    if (Math.abs(max) <
Math.abs(numbers[i]))
        max = numbers[i];
}
return max;
```

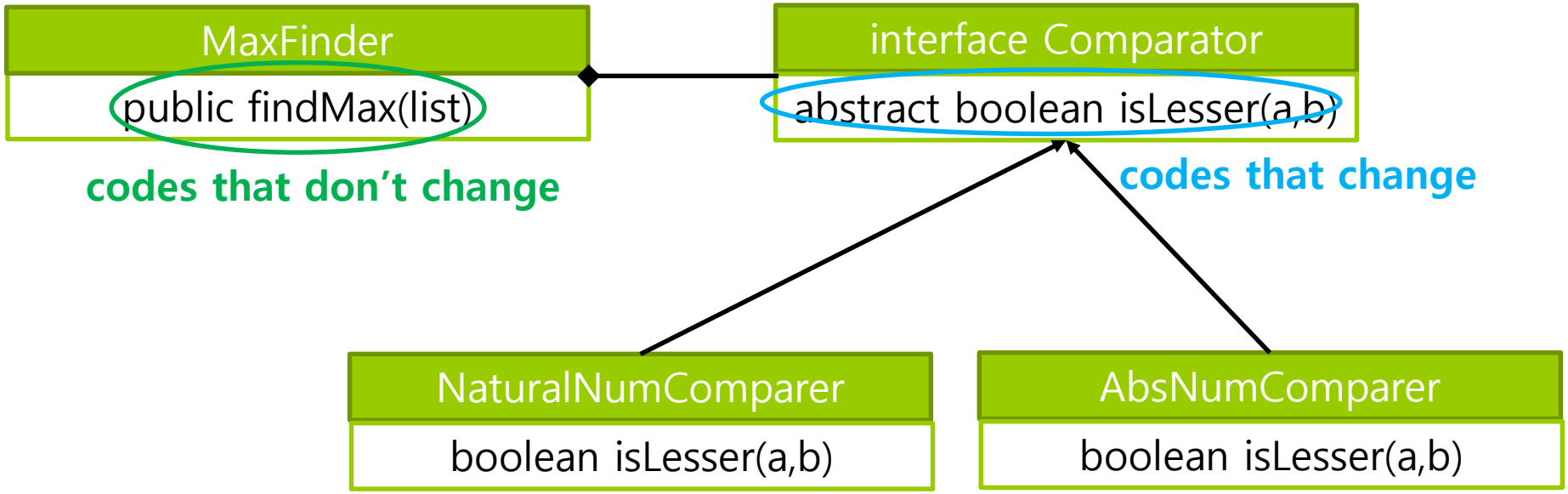
Template Method Pattern

- Template Method pattern **(abstraction through inheritance)**
  - Define the skeleton of the algorithm in the **template method**
  - The template method allows subclasses to override certain steps while keeping the structure of the algorithm intact.



Strategy Pattern

- Strategy Pattern **(abstraction through delegation)**
  - Define a group of algorithms and encapsulate each so that they can used interchangeably.
  - With strategy, you can change the algorithm independently of the client using the algorithm.



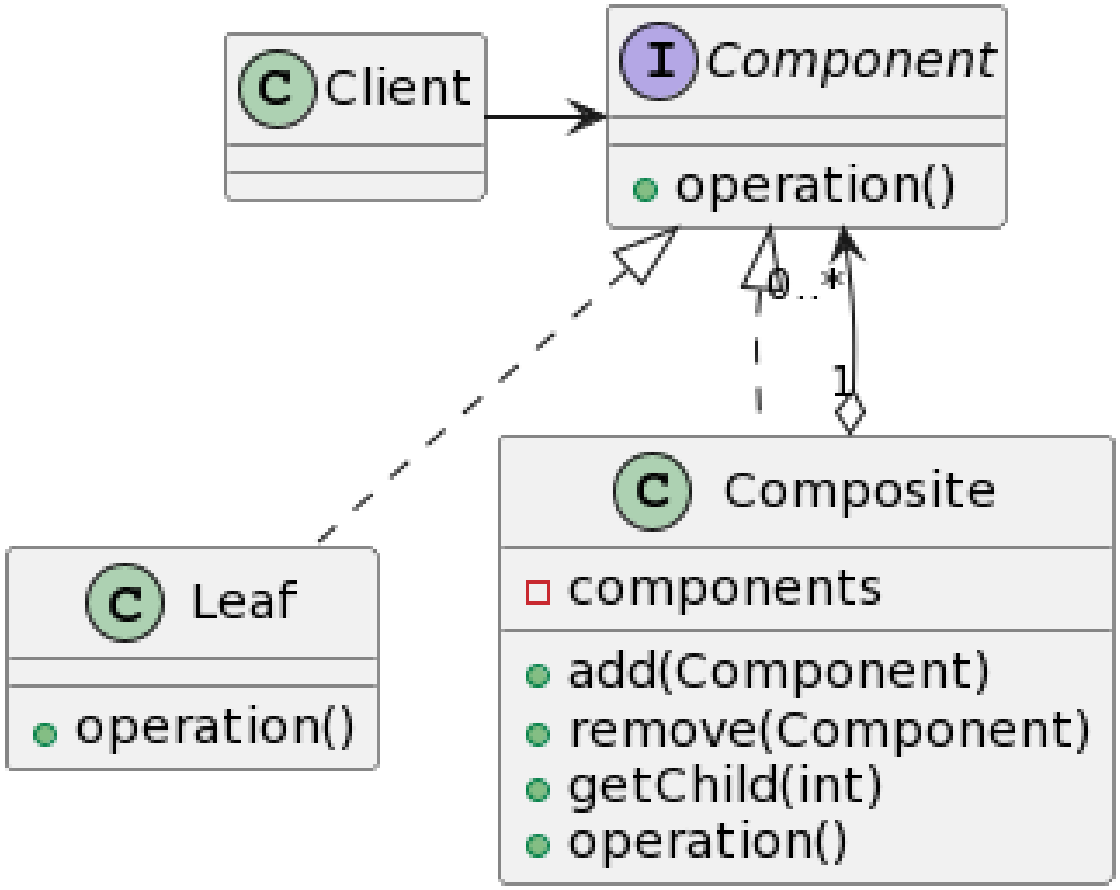
Composite Pattern

- "Compose objects into tree structure to represent **part-whole hierarchies**. Composite lets client treat individual objects and compositions of objects uniformly."
- Recursive composition
- The Composite pattern is used where we need to treat a group of objects in similar way as a single object.
- The Composite pattern composes objects in term of a **tree structure** to represent **part** as well as **whole** hierarchy.

Composite Pattern

	Description
Pattern	Composite
Problem	Application needs to manipulate a hierarchical collection of “primitive” and “composite” objects.
Solution	Using Composite pattern, we can **apply the same operations over both composites and individual objects**. In most cases we can **ignore** the differences between compositions of objects and individual objects.
Result	more maintainable code

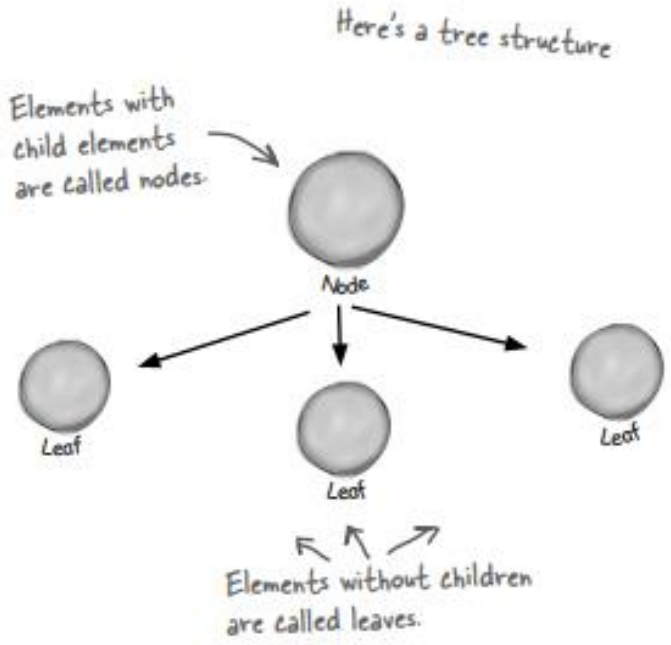
Composite Pattern

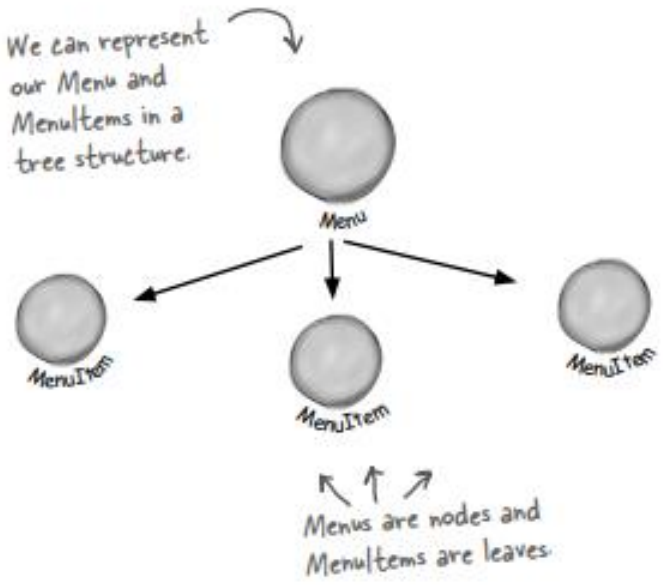


Define Composite Pattern

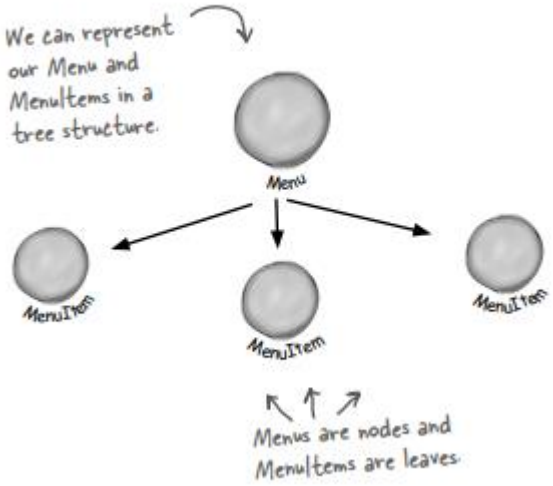
- Component
  - Component declares the interface for objects in the composition and for accessing and managing its child components. It also implements default behavior for the interface common to all classes as appropriate.
- Leaf
  - Leaf defines behavior for primitive objects in the composition. It represents leaf objects in the composition.
- Composite
  - Composite stores leaf components and implements leaf related operations in the component interface.

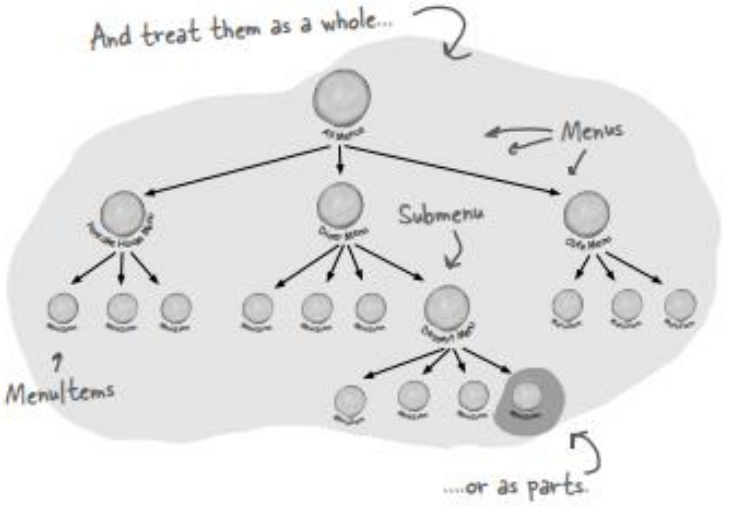
Define Composite Pattern

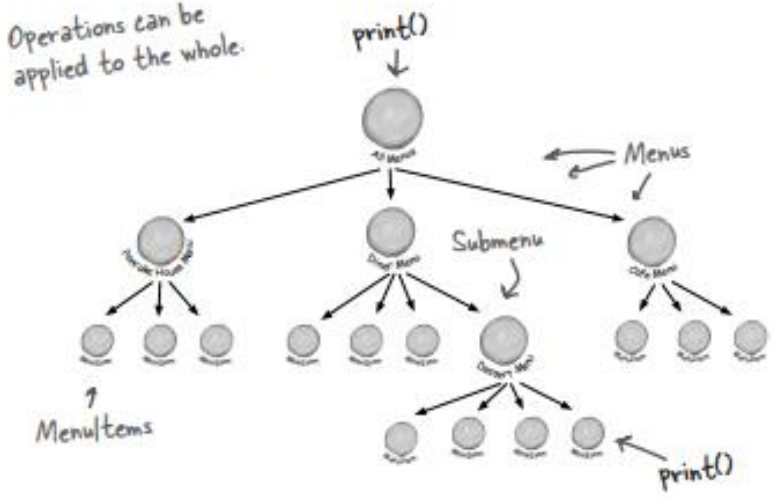




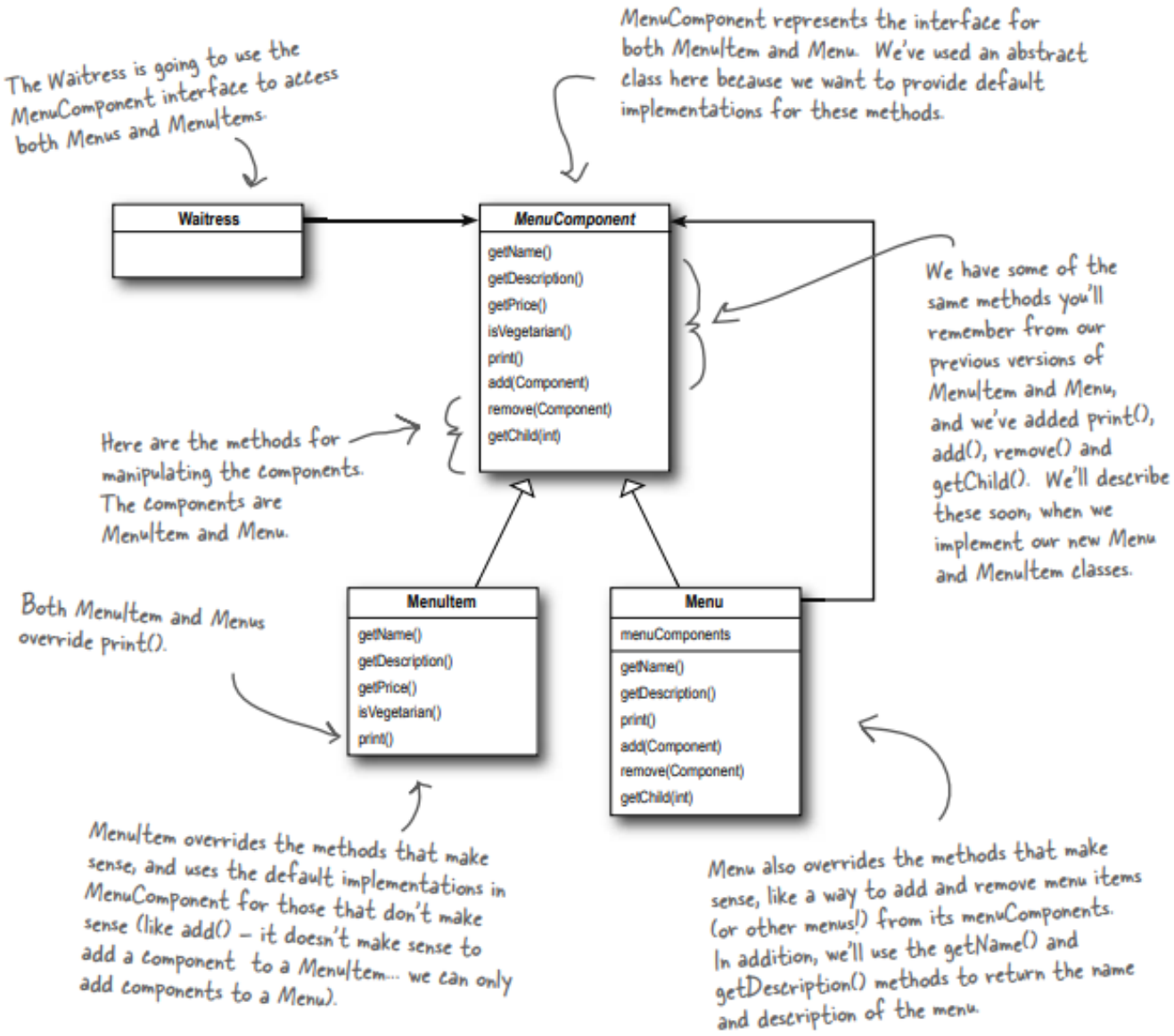
Define Composite Pattern





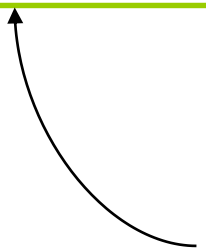


Menus (HFDP Ch. 9)



Menus (HFDP Ch. 9)

```
public abstract class MenuComponent {
  public void add(MenuComponent menuComponent) {
    throw new UnsupportedOperationException();
  }
  public void remove(MenuComponent menuComponent) {
    throw new UnsupportedOperationException();
  }
  public MenuComponent getChild(int i) {
    throw new UnsupportedOperationException();
  }
```



We grouped together the “composite” methods.

Menus (HFDP Ch. 9)

```
  public String getName() {
    throw new UnsupportedOperationException();
  }
  public String getDescription() {
    throw new UnsupportedOperationException();
  }
  public double getPrice() {
    throw new UnsupportedOperationException();
  }
  public boolean isVegetarian() {
    throw new UnsupportedOperationException();
  }
  public void print() {
    throw new UnsupportedOperationException();
  }
}
```

Here are “operation” methods.

Menus (HFDP Ch. 9)

```
public class MenuItem extends MenuComponent {
  String name;
  String description;
  boolean vegetarian;
  double price;

  public MenuItem(String name, String description,
                  Boolean vegetarian, double price) {
    this.name = name;
    this.description = description;
    this.vegetarian = vegetarian;
    this.price = price;
  }
  public String getName() {
    return name;
  }
```

Menus (HFDP Ch. 9)

```
  public String getDescription() {
    return description;
  }
  public double getPrice() {
    return price;
  }
  public boolean isVegetarian() {
    return vegetarian;
  }
  public void print() {
    System.out.println(" " + getName());
    if (isVegetarian()) System.out.println("(v)");
    System.out.println(" " + getPrice());
    System.out.println(" " + getDescription());
  }
}
```

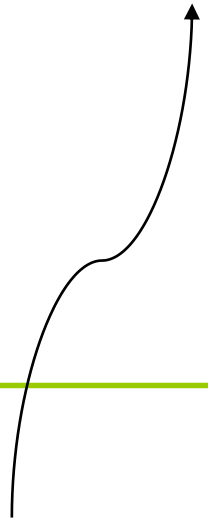
Menus (HFDP Ch. 9)

```
public class Menu extends MenuComponent {
  List<MenuComponent> items = new AraryList<>();
  String name;
  String description;
  // constructor
  // add, remove, get, getName, getDescription
  // print the menu & menu components
  public void print() {
    System.out.println(" " + getName());
    System.out.println(" " + getDescription());
    Iterator<MenuComponent> itr = items.iterator();
    while(itr.hasNext()) {
      MenuComponent item =
          (MenuComponent) itr.next();
      item.print();
    }
  }
}
```

Menus (HFDP Ch. 9)

```
public class Waitress {
  MenuComponent allMenus;

  public Waitress(MenuComponent allMenus) {
    this.allMenus = allMenus;
  }
  public void printMenu() {
    allMenus.print();
  }
}
```



Hands waitress the top level menu component.

Menus (HFDP Ch. 9)

```
public class MenuTestDrive {
  public static void main(String[] args) {
    MenuComponent pancakeHouseMenu =
      new Menu("PANCAKE HOUSE MENU", "Breakfast");
    MenuComponent dinerMenu =
      new Menu("DINER MENU", "Lunch");
    MenuComponent cafeMenu =
      new Menu("CAFE MENU", "Dinner");
    MenuComponent dessertMenu =
      new Menu("DESSERT MENU", "Dessert of
course");
    MenuComponent allMenus = new Menu("ALL MENUS",
"All menu combined"
    allMenus.add(pancakeHouseMenu);
    allMenus.add(dinerMenu);
    allMenus.add(cafeMenu);
```

Menus (HFDP Ch. 9)

```
    // add pancakeHouse menu items
    dinerMenu.add(new MenuItem("Pasta", "Spaghetti
with Marinara Sauce, and a slice of sourdough bread
", true, 3.89));
    // add more diner menu items
    dinerMenu.add(dessertMenu);
    dessertMenu.add(new MenuItem("Apple Pie",
"Apple pie with a flakey crust, topped with vanilla
icecream", true, 1.59));
    // add more dessert menu items
    // add café menu items
    Waitress waitress = new Waitress(allMenus);
    waitress.printMenu();
  }
}
```