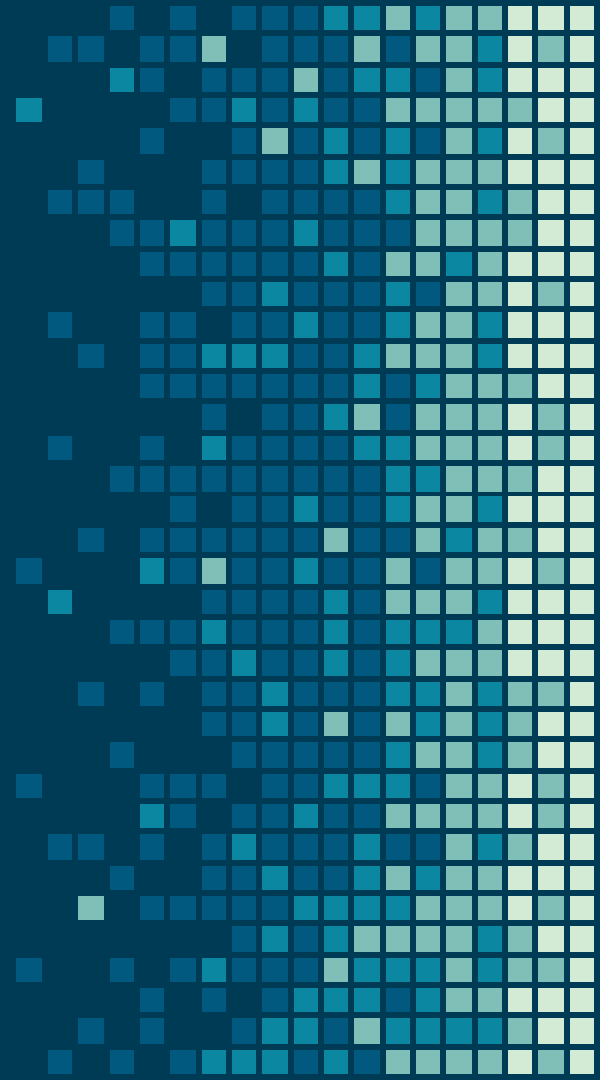


# Advanced Programming

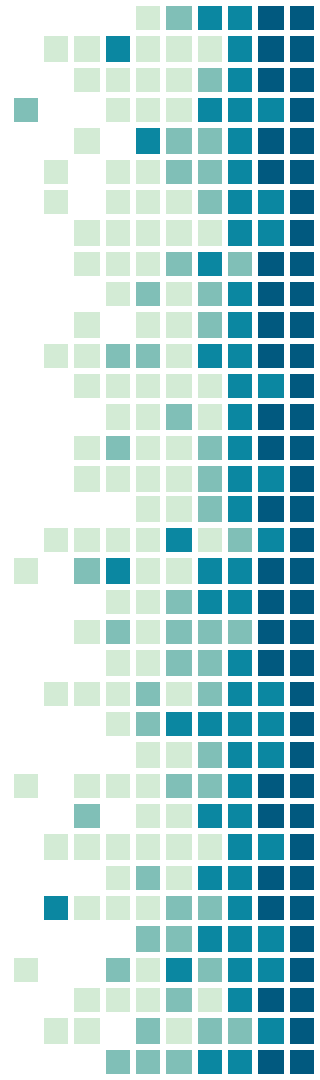
→ Behavioral Design Pattern

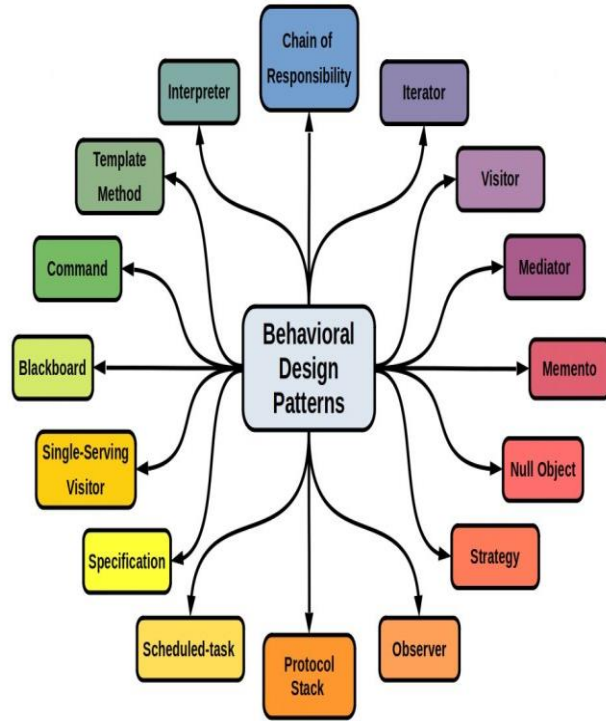




# HELLO!

**I AM HAMZEH ASEFAN**





# Behavioral Design Patterns

- Identify common communication patterns between objects and realize these patterns.
- Increase flexibility in carrying out this communication.

1.

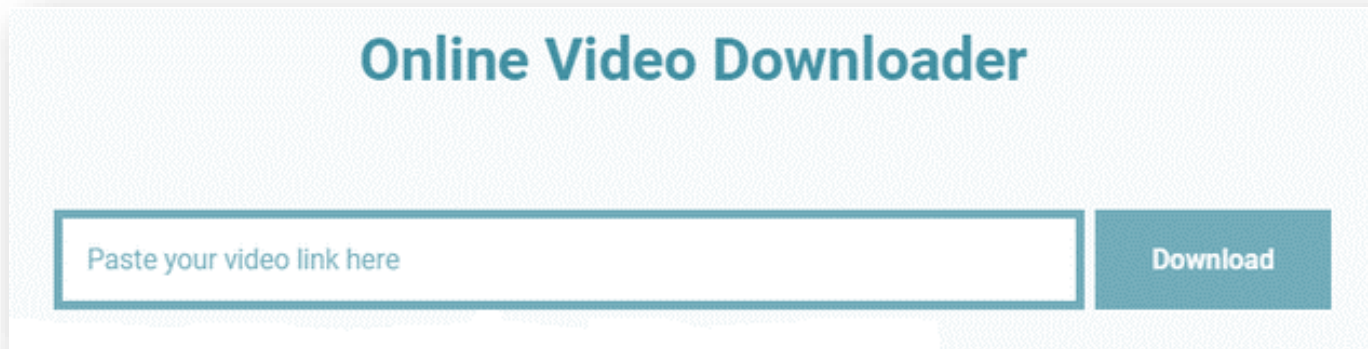
# Chain of Responsibility

Allows passing request along the chain of potential handlers until one of them handles request.



“ A request from the client is passed to a chain of objects to process them. Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

Example:

A mockup of an online video downloader interface. It features a light blue header with the title "Online Video Downloader" in a bold, dark blue font. Below the header is a white input field with a thin blue border, containing the placeholder text "Paste your video link here". To the right of the input field is a solid blue button with the word "Download" in white text. The entire interface is set against a light blue background with a subtle grid pattern. On the right side of the slide, there is a decorative vertical bar composed of a grid of squares in various shades of blue and green, with some squares missing, creating a pixelated effect.

## Online Video Downloader

Download

# Example:

```
public class Video {  
  
    private String link;  
  
    public Video(String link) {  
        this.link = link;  
    }  
  
    public String getLink() {  
        return link;  
    }  
  
    public void setLink(String link) {  
        this.link = link;  
    }  
  
}
```

```
public interface Handler {  
  
    void setNextHandler(Handler handler);  
    void handleLink(Video video);  
  
}
```

# Example:

```
public class YoutubeHandler implements Handler {  
  
    private Handler handler;  
  
    @Override  
    public void setNextHandler(Handler handler) {  
        this.handler = handler;  
    }  
  
    @Override  
    public void handleLink(Video video) {  
        if (video.getLink().toLowerCase().contains("youtube")) {  
            System.out.println("YoutubeHandler --> I can handle it");  
        } else {  
            System.out.println("YoutubeHandler --> I can't handle it");  
            handler.handleLink(video);  
        }  
    }  
}
```



# Example:

```
public class FacebookHandler implements Handler {  
  
    private Handler handler;  
  
    @Override  
    public void setNextHandler(Handler handler) {  
        this.handler = handler;  
    }  
  
    @Override  
    public void handleLink(Video video) {  
        if (video.getLink().toLowerCase().contains("facebook")) {  
            System.out.println("FacebookHandler --> I can handle it");  
        } else {  
            System.out.println("FacebookHandler --> I can't handle it");  
            handler.handleLink(video);  
        }  
    }  
}
```

# Example:

```
public class TwitterHandler implements Handler {  
  
    private Handler handler;  
  
    @Override  
    public void setNextHandler(Handler handler) {  
        this.handler = handler;  
    }  
  
    @Override  
    public void handleLink(Video video) {  
        if (video.getLink().toLowerCase().contains("twitter")) {  
            System.out.println("TwitterHandler --> I can handle it");  
        } else {  
            System.out.println("TwitterHandler --> I can't handle it");  
            handler.handleLink(video);  
        }  
    }  
}
```

# Example:

```
public class Client {  
  
    public static void main(String[] args) {  
  
        YoutubeHandler handler1 = new YoutubeHandler();  
        FacebookHandler handler2 = new FacebookHandler();  
        TwitterHandler handler3 = new TwitterHandler();  
  
        handler1.setNextHandler(handler2);  
        handler2.setNextHandler(handler3);  
  
        handler1.handleLink(new Video("https://twitter.com/nhamzahn"));  
  
    }  
}
```

# 2.

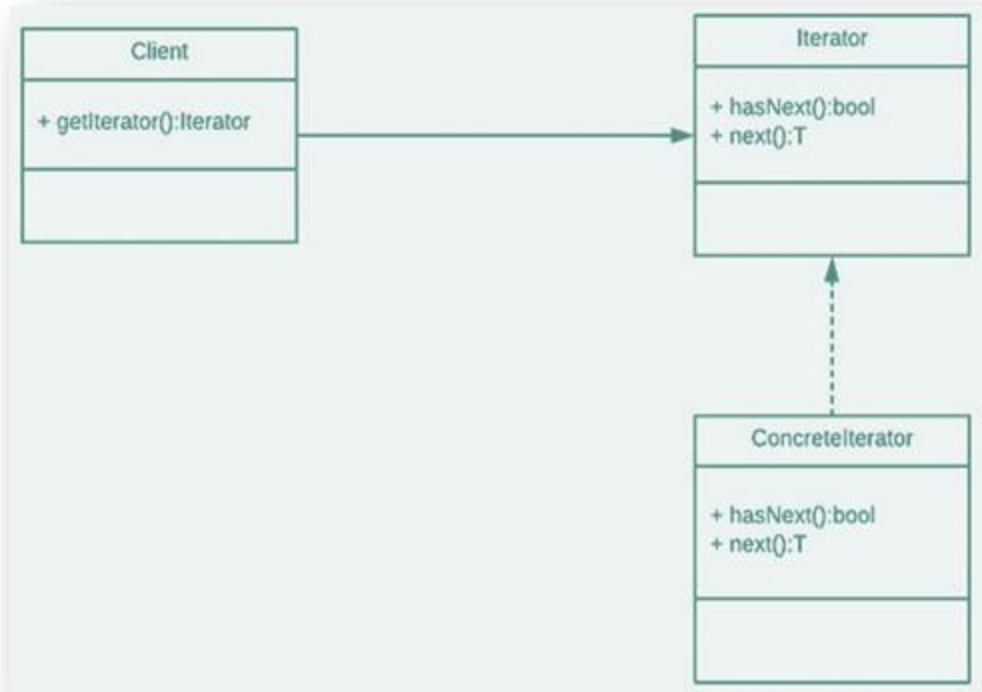
## Iterator

Used to get a way to access the elements of a collection object in sequential manner



“ The idea of the iterator pattern is to take the responsibility of accessing and passing through the objects of the collection and put it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.

# Iterator UML Class Diagram



# Java Enum

- An enum is a special "class" that represents a group of constants → Unchangeable variables.
- To create an enum, use the enum keyword (instead of class or interface).
- Separate the constants with a comma.
- Enum constants are public, static and final.
- An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).
- Use enums when you have values that you know aren't going to change, like days, colors, etc.

# Java Enum Example:

```
public enum Season {  
    WINTER, SPRING, SUMMER, FALL  
}
```

```
public class TestEnum {  
  
    public static void main(String[] args) {  
  
        for (Season s : Season.values())  
            System.out.println(s);  
  
        for (Season s : Season.values())  
            System.out.println(s + " at index " + s.ordinal());  
  
    }  
}
```

```
WINTER  
SPRING  
SUMMER  
FALL  
WINTER at index 0  
SPRING at index 1  
SUMMER at index 2  
FALL at index 3
```



# Iterator Example:

```
import java.util.ArrayList;
import java.util.Iterator;

public class TestIterator {

    public static void main(String[] args) {

        ArrayList<String> languages = new ArrayList<String>();
        languages.add("Python");
        languages.add("Java");
        languages.add("JavaScript");
        languages.add("C++");
        languages.add("PHP");

        Iterator<String> iterate = languages.iterator();
        while (iterate.hasNext())
            System.out.println(iterate.next());
    }
}
```

Python  
Java  
JavaScript  
C++  
PHP

# Inner Class in Java

- In Java, inner class refers to the class that is declared inside class:

```
class OuterClass {  
    ...  
    class InnerClass {  
        ...  
    }  
}
```

- Inner class can be private or protected.
- Inner class can also be static → you can access it without creating an object of the outer class.

# Iterator Example:

```
public interface Iterator<T> {  
  
    boolean hasNext();  
  
    T next();  
}
```

```
public enum ThemeColor {  
  
    RED,  
    GREEN,  
    BLUE,  
    BLACK,  
    WHITE;  
  
    public static Iterator<ThemeColor> getIterator() {  
        return new ThemeColorIterator();  
    }  
  
    private static class ThemeColorIterator implements Iterator<ThemeColor> {  
  
        private int position;  
  
        @Override  
        public boolean hasNext() {  
            return position < ThemeColor.values().length;  
        }  
  
        @Override  
        public ThemeColor next() {  
            return ThemeColor.values()[position++];  
        }  
    }  
}
```

# Iterator Example:

```
public class Client {  
  
    public static void main(String[] args) {  
        Iterator<ThemeColor> iter = ThemeColor.getIterator();  
        while (iter.hasNext())  
            System.out.println(iter.next());  
    }  
}
```

# Iterator VS For:

- For loop isn't always possible because it uses indices. Ex, Lists have indices, but Sets don't, because they're unordered collections. Iterator works with every kind of Iterable collection or array.
- You can remove elements while you're iterating, foreach loop can't.
- Lists also offer iterators that can iterate in both directions. A foreach loop only iterates from the beginning to an end.
- Iterator is less readable than foreach.

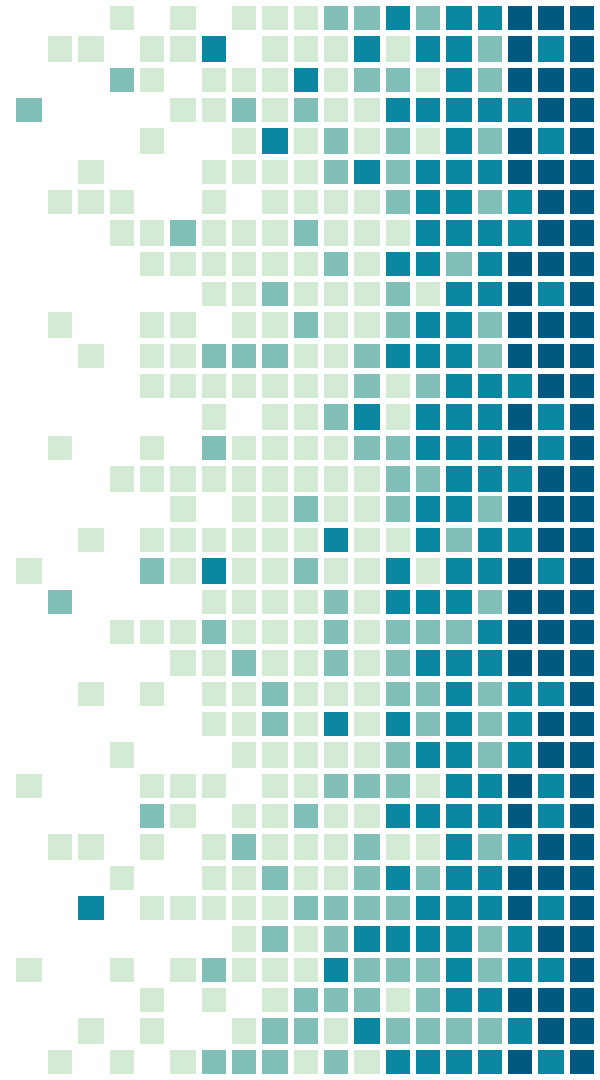
# Iterator VS For:

- Using indices to access elements is slightly more efficient with collections backed by an array. But if you use a LinkedList instead of an ArrayList, the performance will be awful.
  - ➔ Each time you access `list.get(i)`, the linked list will have to loop through all its elements until the *i*th one.
  - ➔ Iterator always uses the best possible way to iterate through elements of the given collection, because the collection itself has its own Iterator implementation.



# 3. Template

We can define an algorithm in a method as a series of steps.

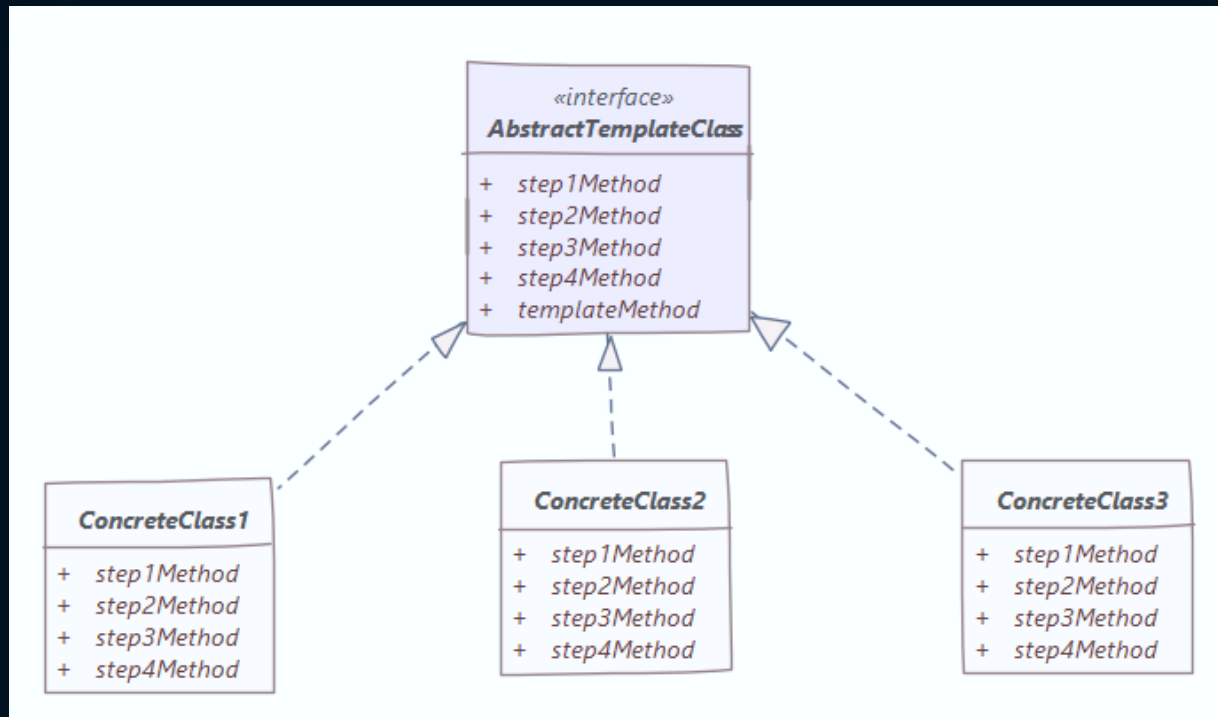




# Template

- Templates allows you to defines a skeleton of an algorithm in a base class and let subclasses override the steps without changing the overall algorithm's structure.
- It is important that subclasses do not override the template method itself → Final.

# Template Class Diagram



# Example:

```
public class Order {  
  
    private String id;  
    private LocalDate date;  
    private Map<String, Double> items = new HashMap<>();  
  
    public Order(String id) {  
        this.id = id;  
        date = LocalDate.now();  
    }  
  
    public String getId() {  
        return id;  
    }  
  
    public LocalDate getDate() {  
        return date;  
    }  
  
    public Map<String, Double> getItems() {  
        return items;  
    }  
  
    public void addItem(String name, double price) {  
        items.put(name, price);  
    }  
}
```

# Example:

```
public abstract class OrderPrinter {  
  
    public final void printOrder(Order order, String filename) throws IOException {  
        try(PrintWriter writer = new PrintWriter(filename)){  
            writer.println(start());  
            writer.println(formatOrderNumber(order));  
            writer.println(formatItems(order));  
            writer.println(end());  
        }  
    }  
  
    protected abstract String start();  
  
    protected abstract String formatOrderNumber(Order order);  
  
    protected abstract String formatItems(Order order);  
  
    protected abstract String end();  
}
```

# Example:

```
public class TextPrinter extends OrderPrinter {

    @Override
    protected String start() {
        return "Order Details";
    }

    @Override
    protected String formatOrderNumber(Order order) {
        return "Order #" + order.getId();
    }

    @Override
    protected String formatItems(Order order) {
        StringBuilder builder = new StringBuilder("Items\n-----\n");

        for (Map.Entry<String, Double> entry : order.getItems().entrySet()) {
            builder.append(entry.getKey() + " $" + entry.getValue() + "\n");
        }
        builder.append("-----");
        return builder.toString();
    }

    @Override
    protected String end() {
        return "";
    }
}
```

# Example:

```
public class HtmlPrinter extends OrderPrinter {

    @Override
    protected String start() {
        return "<html><head><title>Order Details</title></head><body>";
    }

    @Override
    protected String formatOrderNumber(Order order) {
        return "<h1>Order #" + order.getId() + "</h1>";
    }

    @Override
    protected String formatItems(Order order) {
        StringBuilder builder = new StringBuilder("<p><ul>");
        for (Map.Entry<String, Double> e : order.getItems().entrySet()) {
            builder.append("<li>" + e.getKey() + " $" + e.getValue() + "</li>");
        }
        builder.append("</ul></p>");
        return builder.toString();
    }

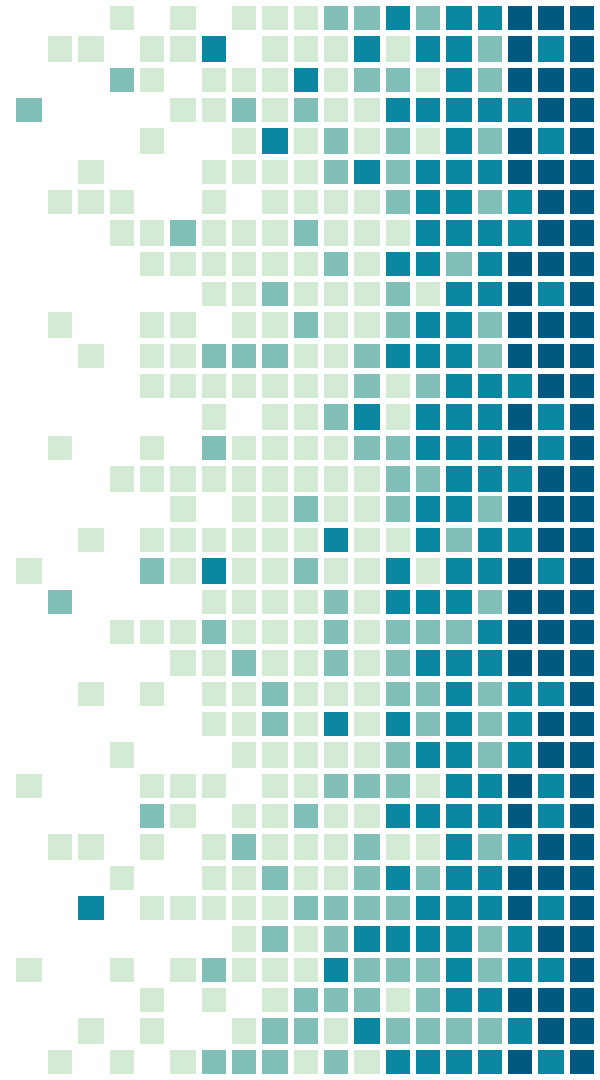
    @Override
    protected String end() {
        return "</body></html>";
    }
}
```

# Example:

```
public class Client {  
  
    public static void main(String[] args) throws IOException {  
        Order order = new Order("1001");  
  
        order.addItem("Sony Xperia 5 III", 900);  
        order.addItem("Iphone 13", 1000);  
        order.addItem("S22 Ultra", 1200);  
  
        OrderPrinter printer = new HtmlPrinter(); //new TextPrinter();  
        printer.printOrder(order, "d:\\a.html"); //d:\\a.txt  
    }  
}
```

# 4. Strategy

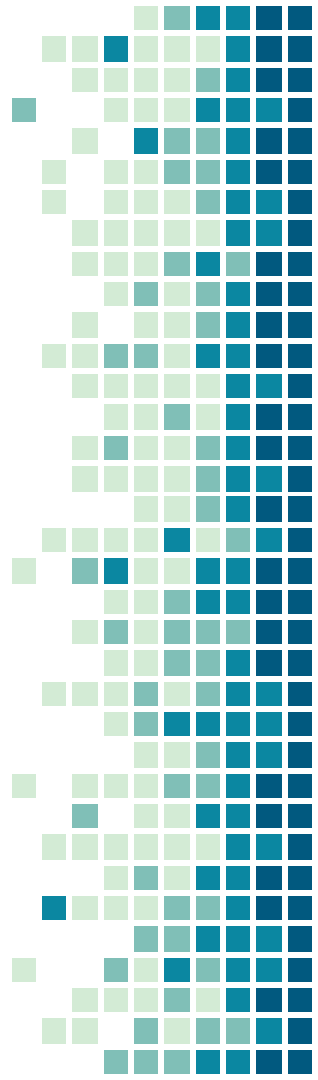
The strategy pattern allows us to change the behavior of an algorithm at runtime.





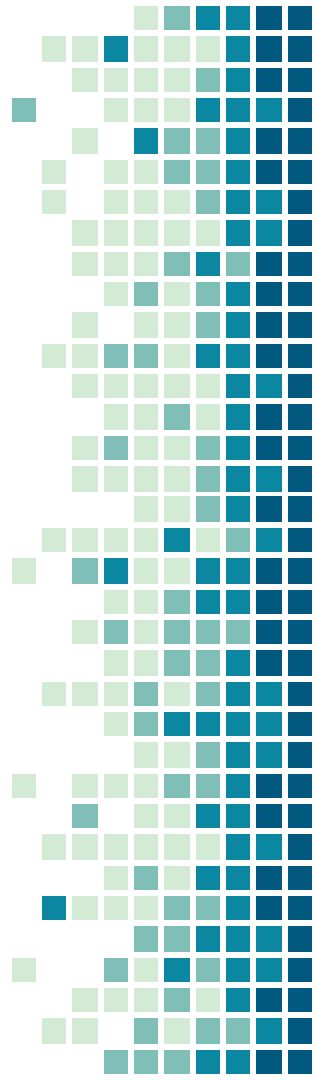
# Strategy

- Define a family of algorithms, encapsulate each one and make them interchangeable.
- The strategy pattern allows us to change the behavior of an algorithm at runtime.
- The client works with all strategies through the context. As a result, the client can work in same way with all algorithms.

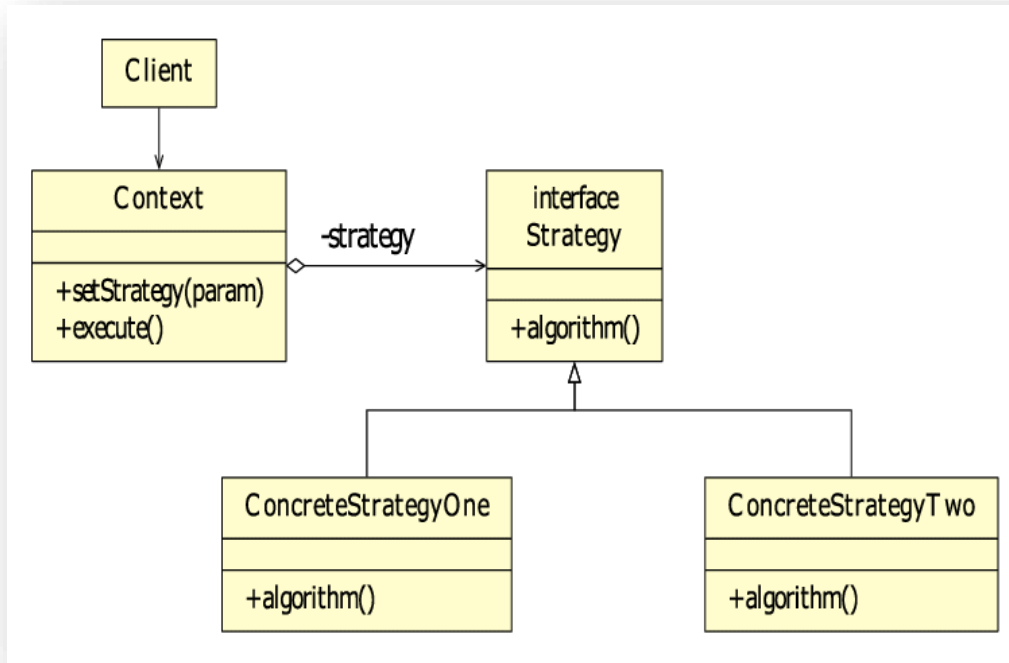


# Strategy

- Create interface which is used to apply an algorithm.
- Implement it multiple times for each possible algorithm.
- Create context class.
- Use the Context to see change in behavior when it changes its Strategy.



# Strategy Class Diagram



# Example 1:

```
public interface Strategy {  
    int doOperation(int num1, int num2);  
}
```

```
public class OperationAdd implements Strategy {  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 + num2;  
    }  
}
```

# Example 1:

```
public class OperationSubstract implements Strategy {  
  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 - num2;  
    }  
  
}
```

```
public class OperationMultiply implements Strategy {  
  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 * num2;  
    }  
  
}
```

# Example 1:

```
public class Context {  
  
    private Strategy strategy;  
  
    public Context(Strategy strategy) {  
        this.strategy = strategy;  
    }  
  
    public int executeStrategy(int num1, int num2) {  
        return strategy.doOperation(num1, num2);  
    }  
  
}
```

# Example 1:

```
public class Client {  
  
    public static void main(String[] args) {  
  
        int num1 = 7;  
        int num2 = 3;  
  
        Context context = new Context(new OperationAdd());  
        System.out.println(num1 + " + " + num2 + " = " + context.executeStrategy(num1, num2));  
  
        context = new Context(new OperationSubtract());  
        System.out.println(num1 + " - " + num2 + " = " + context.executeStrategy(num1, num2));  
  
        context = new Context(new OperationMultiply());  
        System.out.println(num1 + " * " + num2 + " = " + context.executeStrategy(num1, num2));  
  
    }  
}
```

7 + 3 = 10
7 - 3 = 4
7 * 3 = 21

## Example 2:

```
public class Order {  
  
    private String id;  
    private LocalDate date;  
    private Map<String, Double> items = new HashMap<>();  
  
    public Order(String id) {  
        this.id = id;  
        date = LocalDate.now();  
    }  
  
    public void addItem(String name, double price) {  
        items.put(name, price);  
    }  
  
    public String getId() {  
        return id;  
    }  
  
    public LocalDate getDate() {  
        return date;  
    }  
  
    public Map<String, Double> getItems() {  
        return items;  
    }  
}
```

```
//Strategy  
public interface OrderPrinter {  
  
    void print(Collection<Order> orders);  
}
```



## Example 2:

```
//Concrete strategy
public class SummaryPrinter implements OrderPrinter {

    @Override
    public void print(Collection<Order> orders) {
        System.out.println("***** Summary Report *****");
        Iterator<Order> iterator = orders.iterator();

        while (iterator.hasNext()) {
            Order order = iterator.next();
            System.out.println("\t" + order.getId() + ". \t" + order.getDate() + "\t" + order.getItems().size());
        }
        System.out.println("*****");
    }
}
```

## Example 2:

```
public class DetailPrinter implements OrderPrinter {

    @Override
    public void print(Collection<Order> orders) {
        System.out.println("***** Detail Report *****");
        System.out.println("-----");

        Iterator<Order> iter = orders.iterator();

        while (iter.hasNext()) {
            Order order = iter.next();
            System.out.println(order.getId() + ". " + order.getDate() + ": ");
            for (Map.Entry<String, Double> entry : order.getItems().entrySet()) {
                System.out.println("\t" + entry.getKey() + " --> " + entry.getValue() + "$");
            }

            System.out.println("-----");
        }
        System.out.println("*****");
    }
}
```

## Example 2:

```
//Context
public class PrintService {

    private OrderPrinter printer;

    public PrintService(OrderPrinter printer) {
        this.printer = printer;
    }

    public void printOrders(LinkedList<Order> orders) {
        printer.print(orders);
    }
}
```

## Example 2:

```
public class Client {  
  
    private static LinkedList<Order> orders = new LinkedList<>();  
    public static void main(String[] args) {  
        createOrders();  
        PrintService service = new PrintService(new DetailPrinter());  
        //PrintService service = new PrintService(new SummaryPrinter());  
        service.printOrders(orders);  
    }  
  
    private static void createOrders() {  
        Order order = null;  
  
        order = new Order("1");  
        order.addItem("S21", 700);  
        order.addItem("S21 Ultra", 800);  
        orders.add(order);  
  
        order = new Order("2");  
        order.addItem("Iphone 13", 1000);  
        order.addItem("Iphone 13 Pro Max", 1400);  
        orders.add(order);  
  
        order = new Order("3");  
        order.addItem("Sony Xperia 1 III", 1000);  
        order.addItem("Sony Xperia 5 III", 900);  
        orders.add(order);  
    }  
}
```

# THANKS!

Any questions?