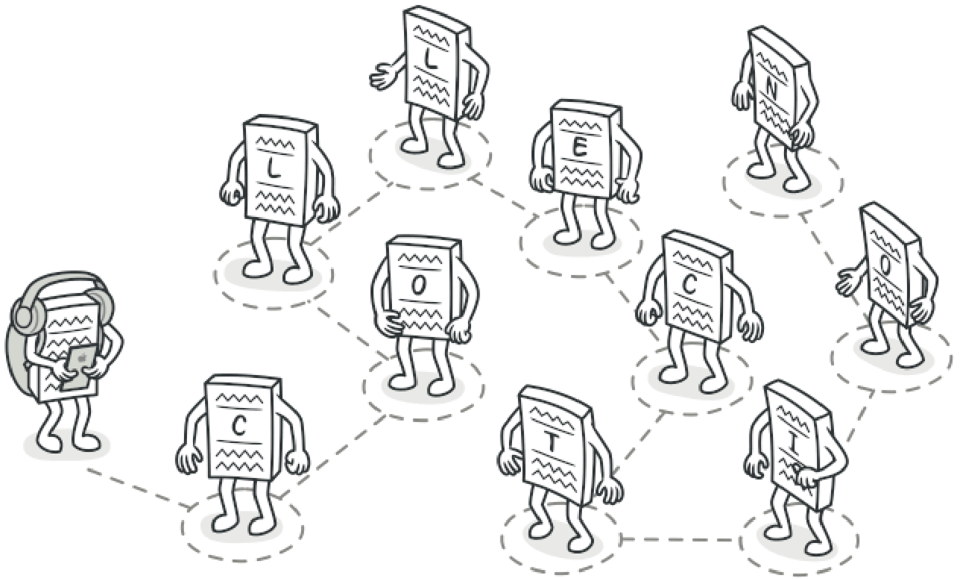
Design Patterns Behavioural

Behavioural Pattern – Iterator

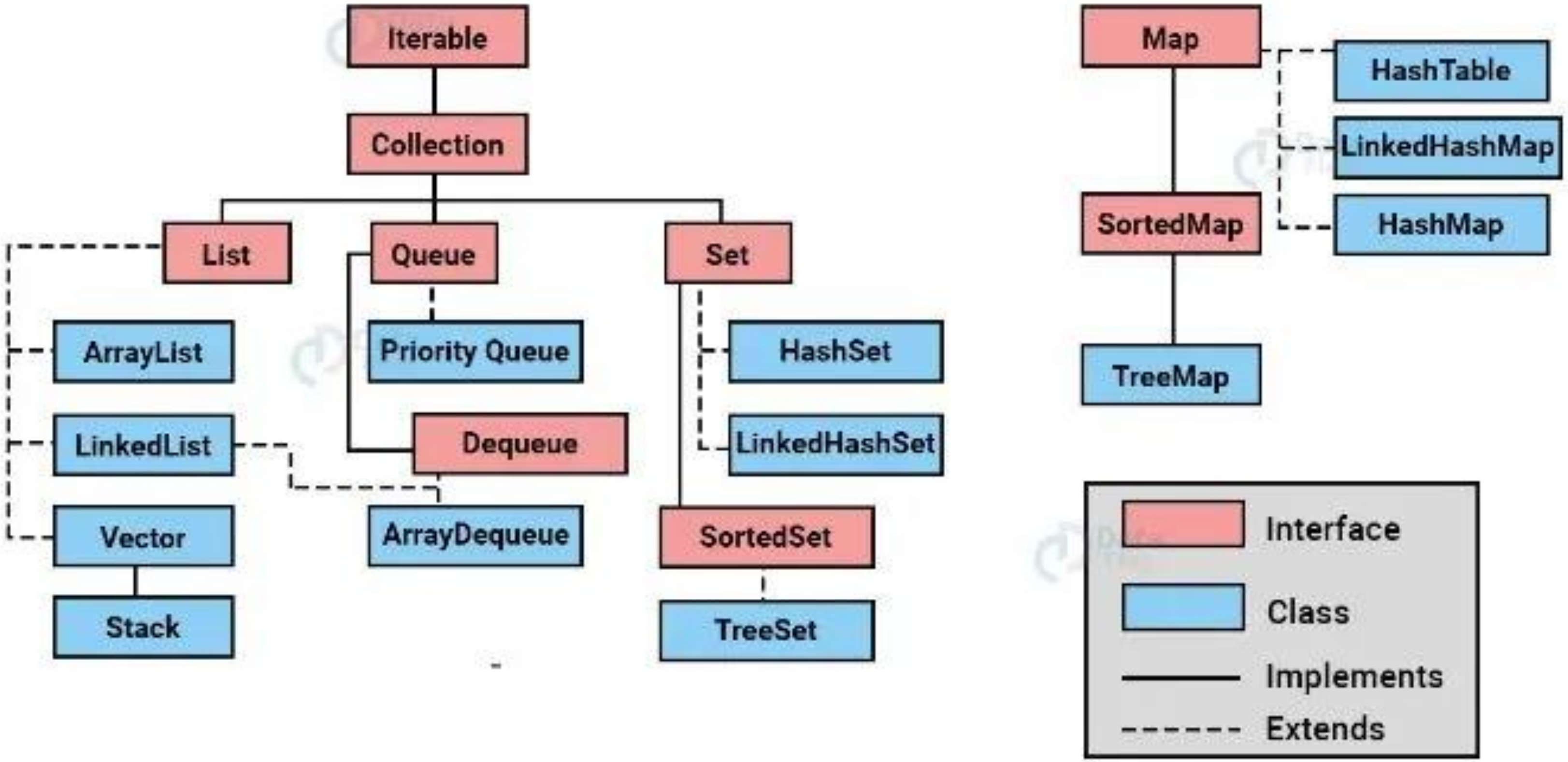
Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



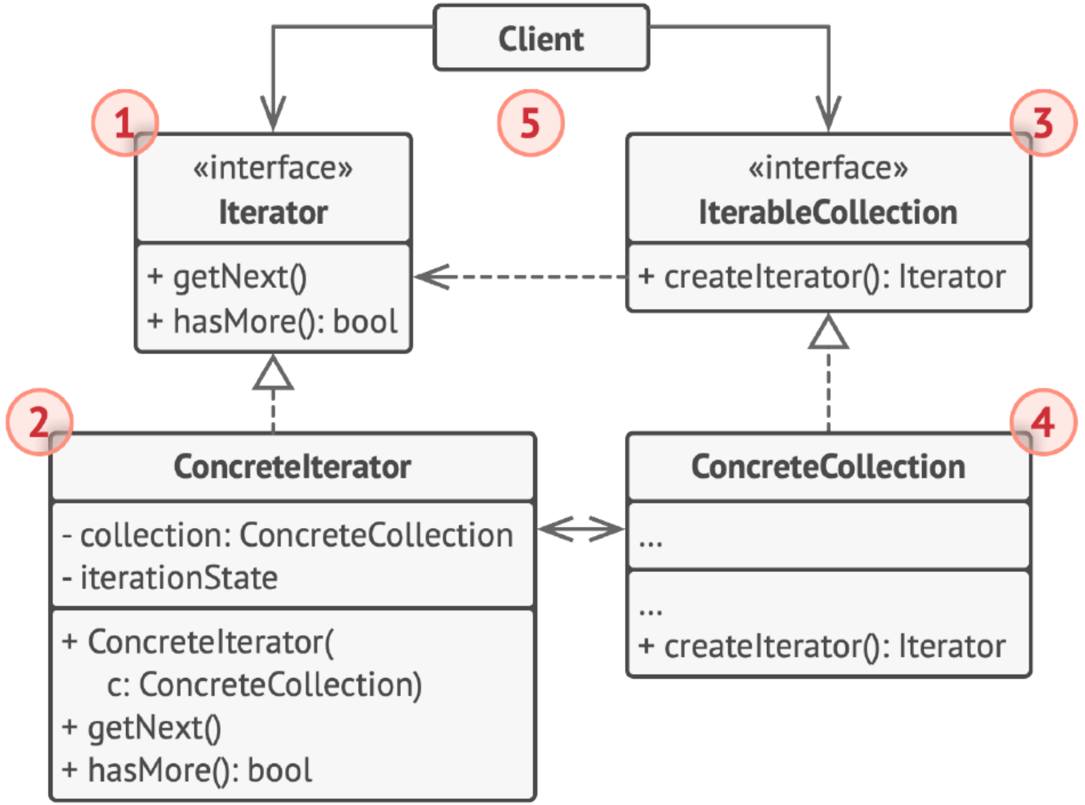
Iterator– When to use

- Use the Iterator pattern when your collection has a **complex data structure under the hood**, but you want to hide its complexity from clients (either for convenience or security reasons).
- Use the Iterator when you want your code to be able to traverse **different data structures or when types of these structures are unknown beforehand**.
- Use the Iterator when you need to support **different ways of traversing** a collection (e.g., depth-first, breadth-first, skipping elements), you can create different iterators without changing the collection class.

Hierarchy of Collection Framework in Java



Iterator – UML

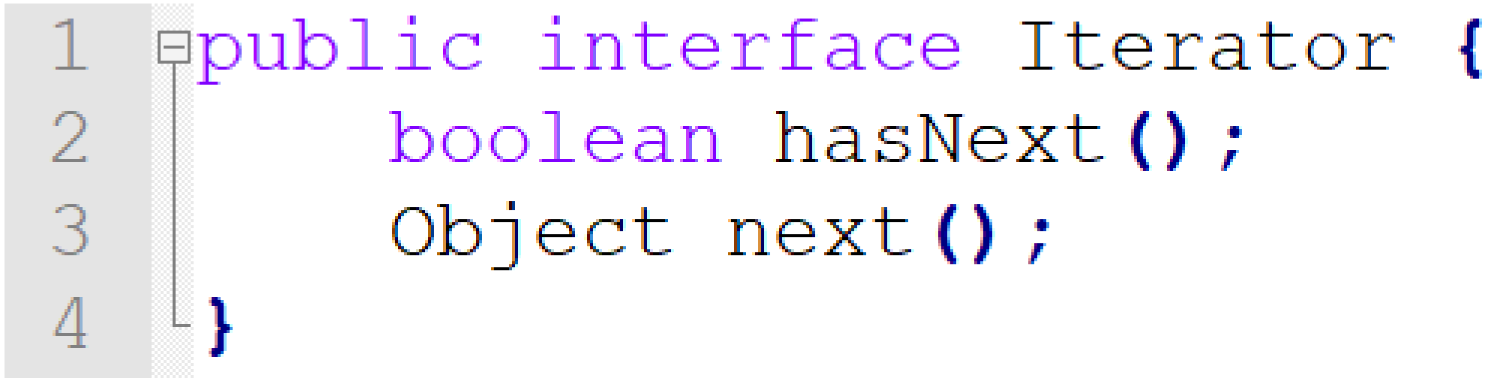


Iterator – Participants

- **Iterator**: This is an interface or abstract class that defines the methods necessary for accessing and traversing elements in the collection. Common methods in an iterator include:
  - **next()**: Returns the next element in the collection.
  - **hasNext()**: Returns a boolean indicating whether there are more elements to be iterated over.
  - **remove()**: Removes the last element returned by the iterator (optional operation).
- **Concrete Iterator**: This is a concrete class that implements the Iterator interface. It keeps track of the current position in the traversal of the aggregate object.
- **IterableCollection**: This is an interface or abstract class that represents the collection of objects. It typically defines a method to create an iterator object. The Concrete Aggregate is the concrete class that implements this interface and contains the collection of objects.
- **Concrete Collection**: This is a concrete class that implements the **IterableCollection** interface. It contains the collection of objects and implements the method to create and return an instance of the Concrete Iterator.

Iterator – Implementation step 1

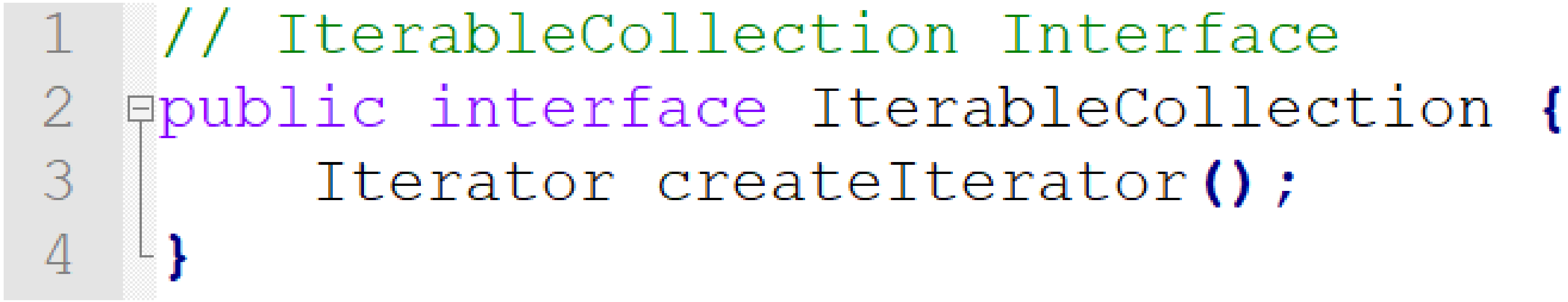
Declare the Iterator Interface



```
public interface Iterator {
      boolean hasNext();
      Object next();
}
```

Adapter – Implementation step 2

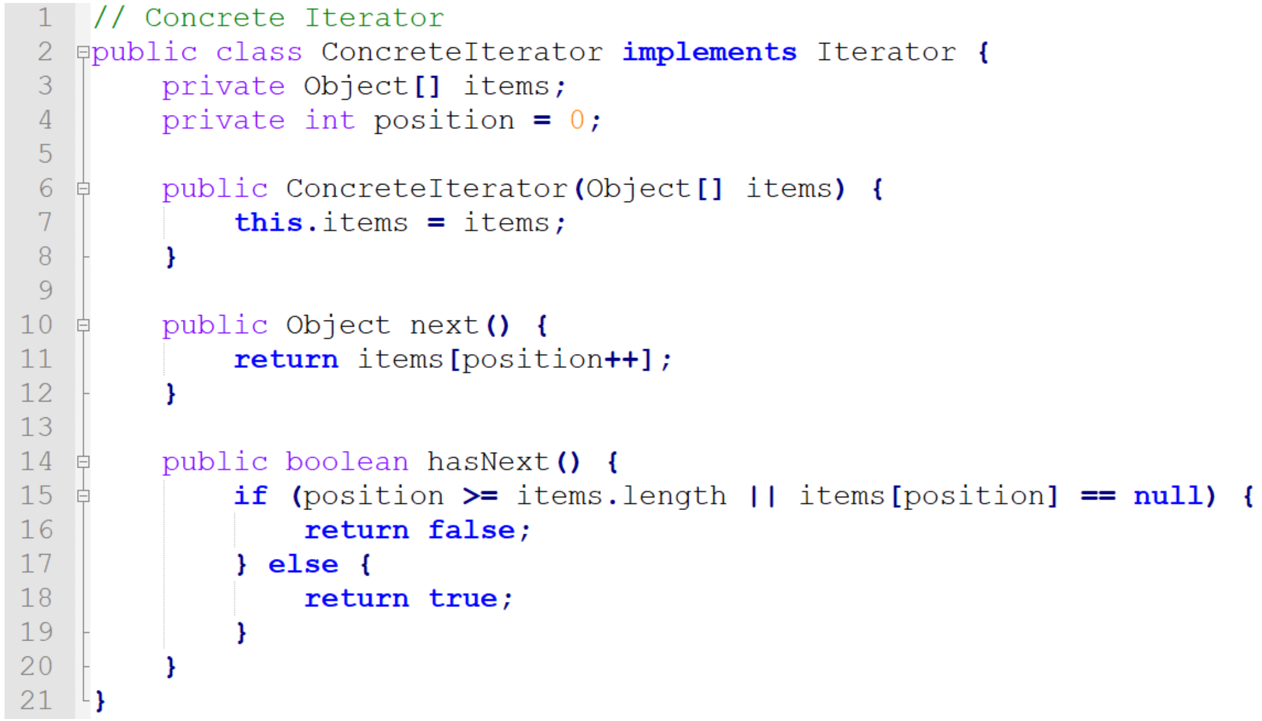
Declare the Collection Interface



```java
// IterableCollection Interface
public interface IterableCollection {
    Iterator createIterator();
}
```

Iterator – Implementation step 3

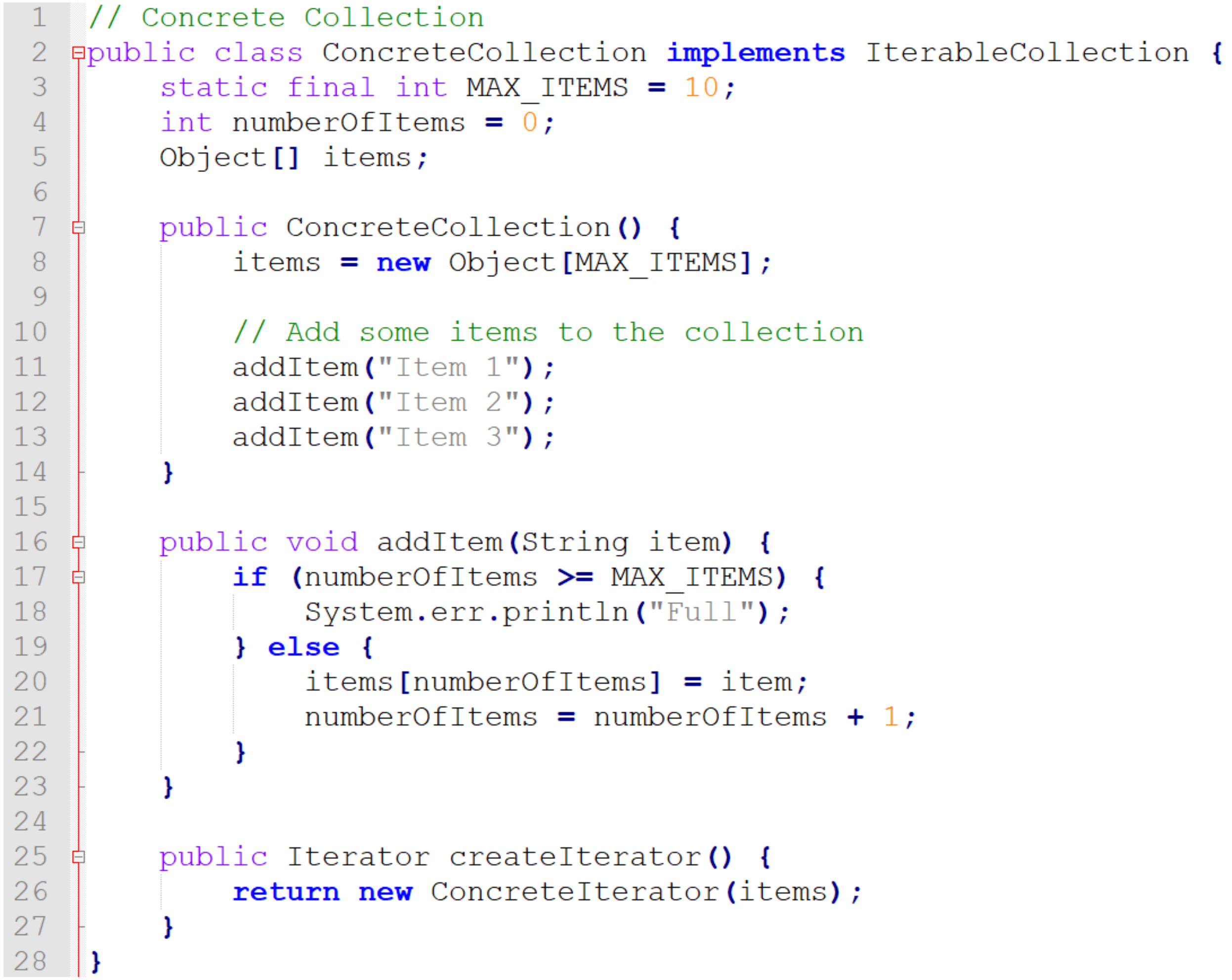
Implement Concrete Iterator



```java
// Concrete Iterator
public class ConcreteIterator implements Iterator {
    private Object[] items;
    private int position = 0;

    public ConcreteIterator(Object[] items) {
        this.items = items;
    }

    public Object next() {
        return items[position++];
    }

    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }
}
```

Iterator–Implementation step 4

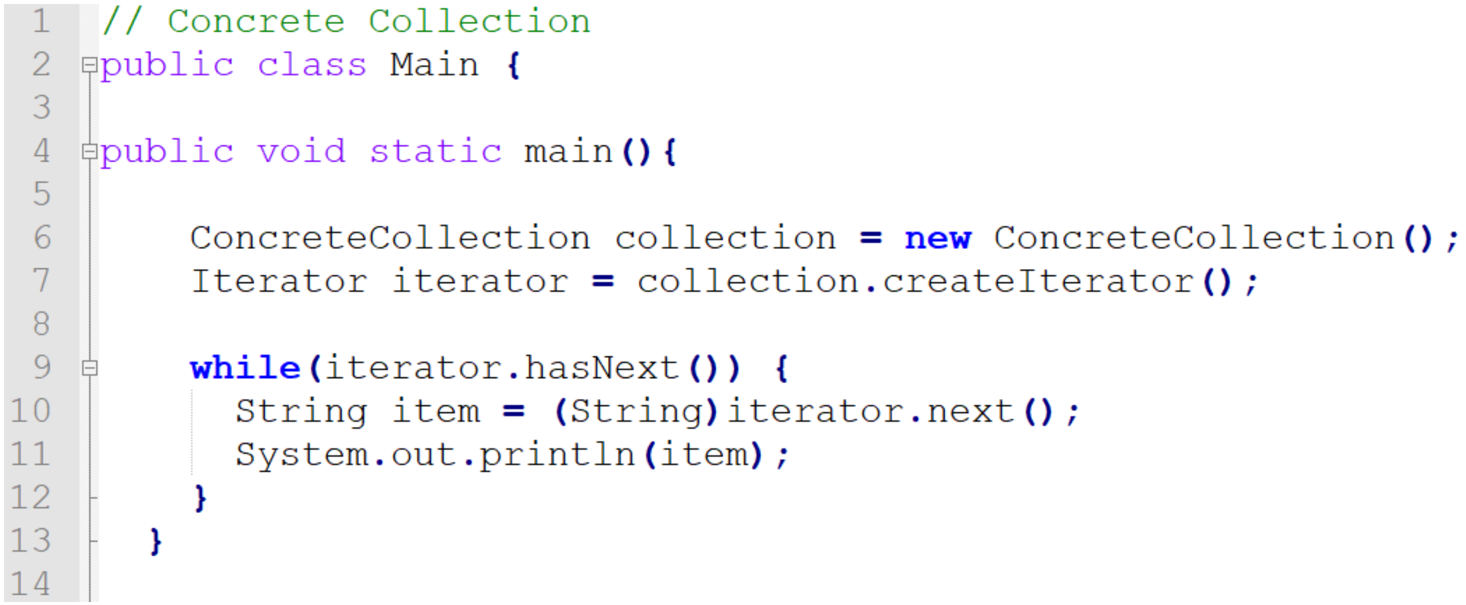
Implement the Collection Interface



```java
// Concrete Collection
public class ConcreteCollection implements IterableCollection {
    static final int MAX_ITEMS = 10;
    int numberOfItems = 0;
    Object[] items;

    public ConcreteCollection() {
        items = new Object[MAX_ITEMS];

        // Add some items to the collection
        addItem("Item 1");
        addItem("Item 2");
        addItem("Item 3");
    }

    public void addItem(String item) {
        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Full");
        } else {
            items[numberOfItems] = item;
            numberOfItems = numberOfItems + 1;
        }
    }

    public Iterator createIterator() {
        return new ConcreteIterator(items);
    }
}
```

Iterator – Implementation step 5

Use Iterator in Client Code



```
// Concrete Collection
public class Main {

public void static main(){

        ConcreteCollection collection = new ConcreteCollection();
        Iterator iterator = collection.createIterator();

        while(iterator.hasNext()) {
          String item = (String)iterator.next();
          System.out.println(item);
        }
    }

```

Iterator – Pros and Cons

- *Single Responsibility Principle*. You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes.
- *Open/Closed Principle*. You can implement new types of collections and iterators and pass them to existing code without breaking anything.
- You can iterate over the same collection in parallel because each iterator object contains its own iteration state.
- For the same reason, you can delay an iteration and continue it when needed.

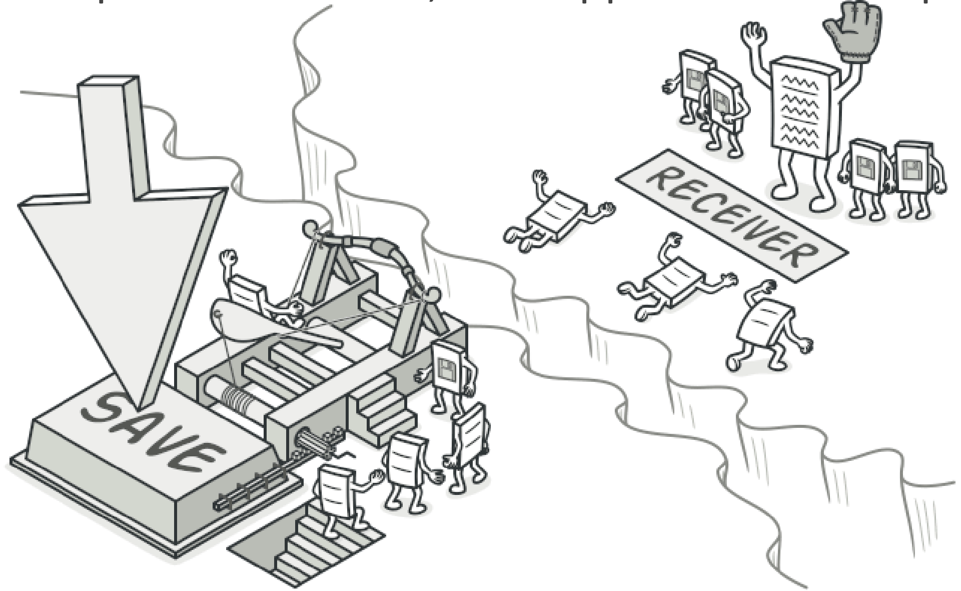
- Applying the pattern can be an overkill if your app only works with simple collections.
- Using an iterator may be less efficient than going through elements of some specialized collections directly.

Iterator – Use case in Java

- All implementations of **java.util.Iterator** (also **java.util.Scanner**).
- All implementations of **java.util.Enumeration**.

Behavioural Pattern – Command

Command pattern turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request’s execution, and support undoable operations.



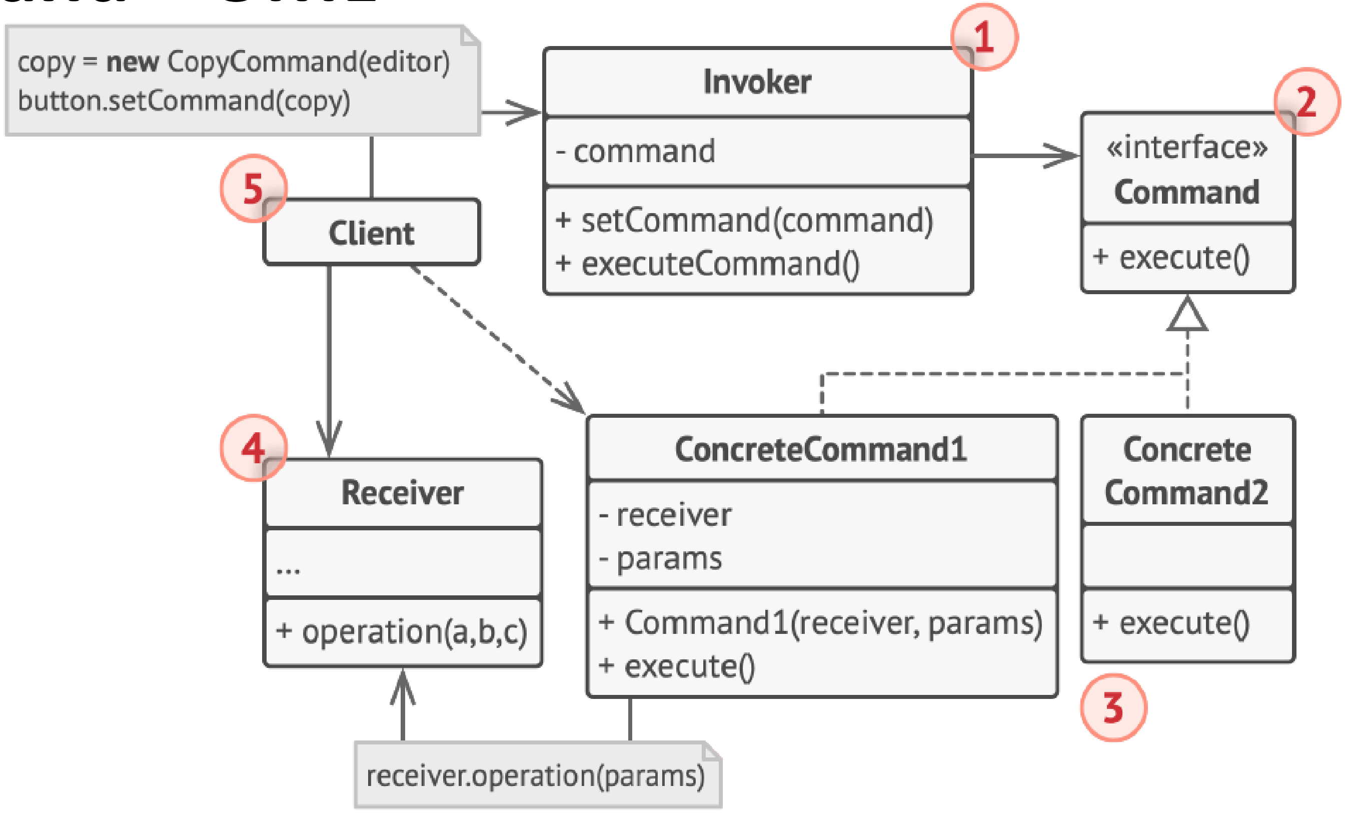
Command – When to use

- Use the Command pattern when you want to parametrize objects with operations.
- Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely.
- Use the Command pattern when you want to implement reversible operations.

Command – Participants

- **Command**: This is an interface that declares a method for executing an operation.
- **ConcreteCommand**: Classes that implement the Command interface. They define a binding between a Receiver object and an action. The ConcreteCommand's **execute()** method invokes the corresponding operation on the Receiver to perform the action.
- **Client**: This component creates ConcreteCommand objects and associates them with the appropriate Receiver objects.
- **Invoker**: This component asks the command to execute the request. It holds a command object and triggers its **execute()** method.
- **Receiver**: This component knows how to perform the operations associated with the request. The ConcreteCommand passes the request to the appropriate method of the Receiver.

Command – UML

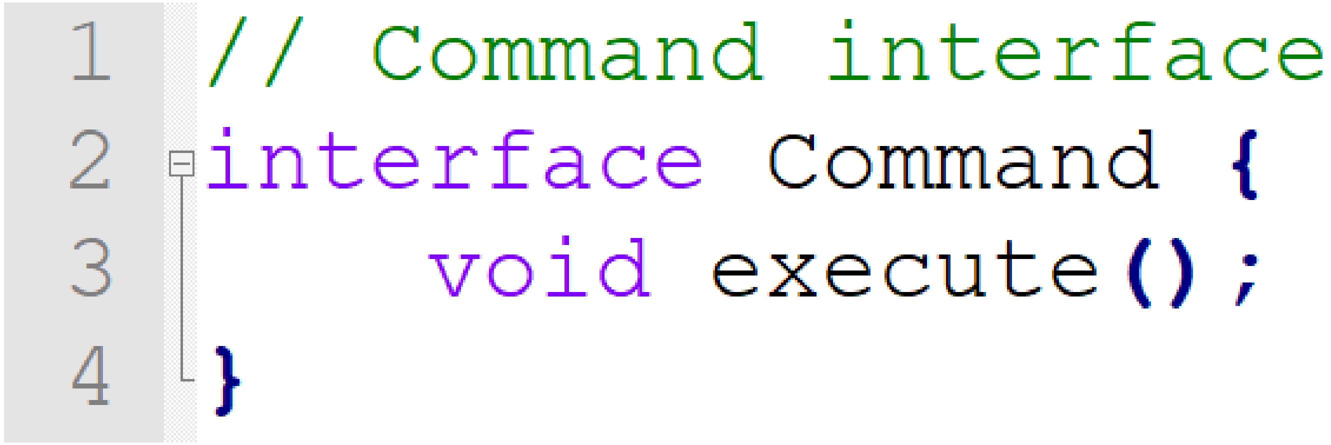

copy = **new** CopyCommand(editor)
button.setCommand(copy)
1
Invoker
- command
+ setCommand(command)
+ executeCommand()
2
«interface»
Command
+ execute()
5
Client
4
Receiver
...
+ operation(a,b,c)
ConcreteCommand1
- receiver
- params
+ Command1(receiver, params)
+ execute()
Concrete Command2
+ execute()
3
receiver.operation(params)

Implementation Description in Brief

- Create Command Interface that has execute and unexecute methods.
- Create Specific Concrete Commands to Implement Command Interface
- Create Receiver. Receivers are actually the entities that have the actual business logic
- Set those Receivers into Concrete Command from Its constructor.
- Create Invoker that has a list of Commands. So that Invoker can perform some operations through Command execute method

Command – Implementation step 1

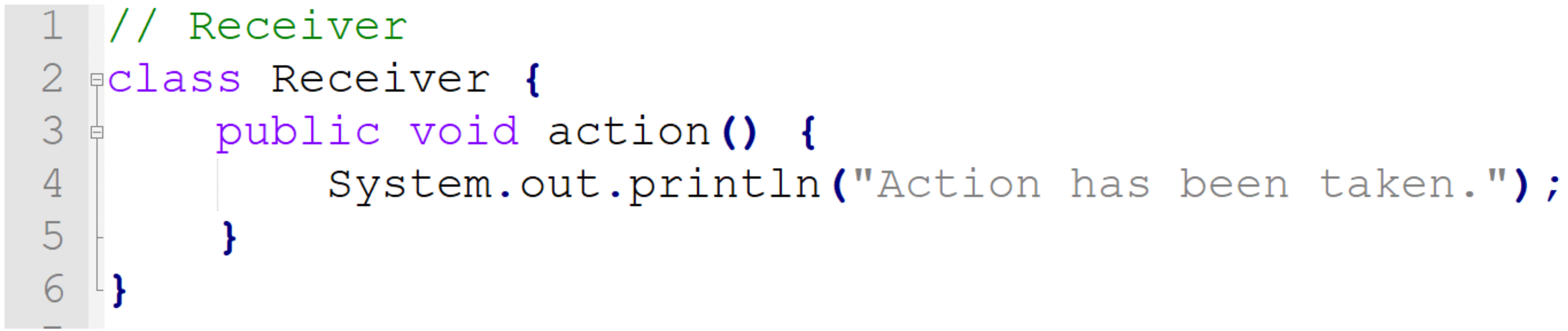
Define the Command Interface



```
// Command interface
interface Command {
    void execute();
}
```

Command– Implementation step 2

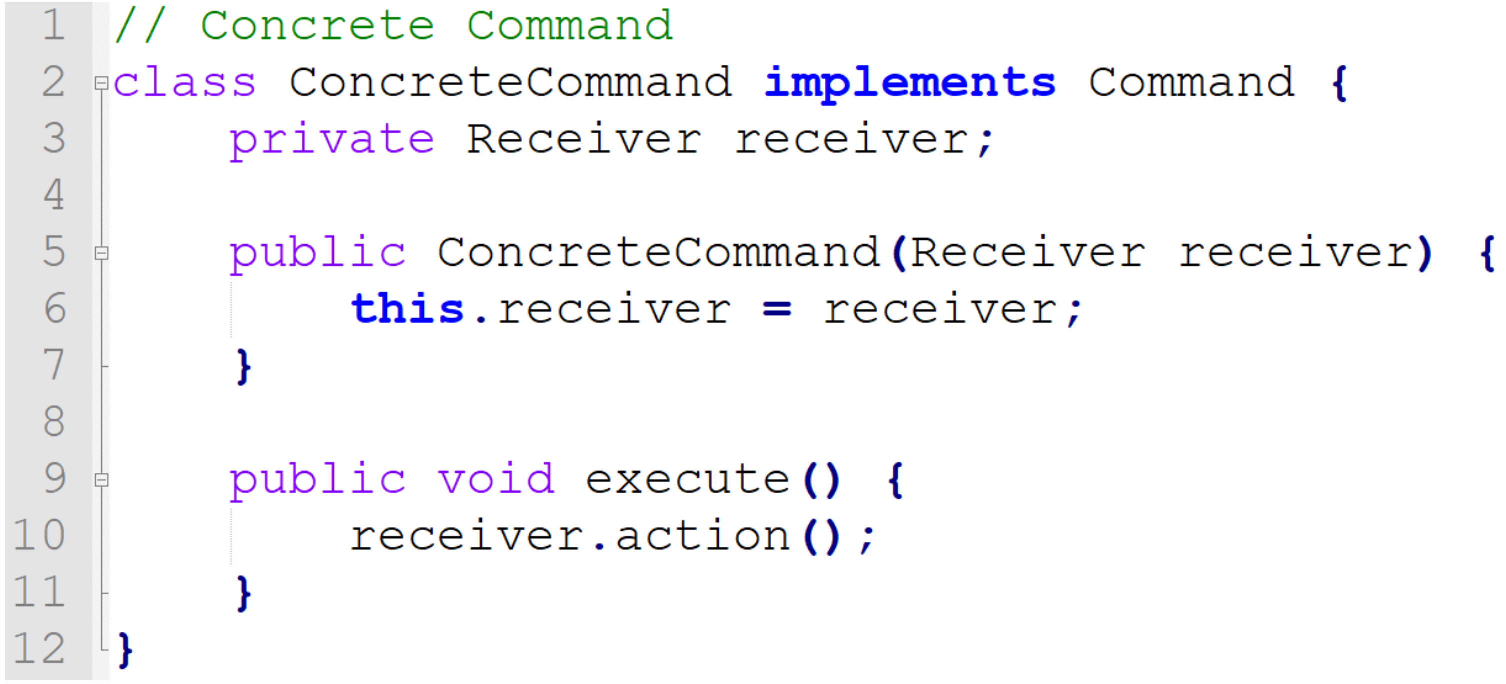
Define the Receiver



```java
// Receiver
class Receiver {
    public void action() {
        System.out.println("Action has been taken.");
    }
}
```

Command– Implementation step 3

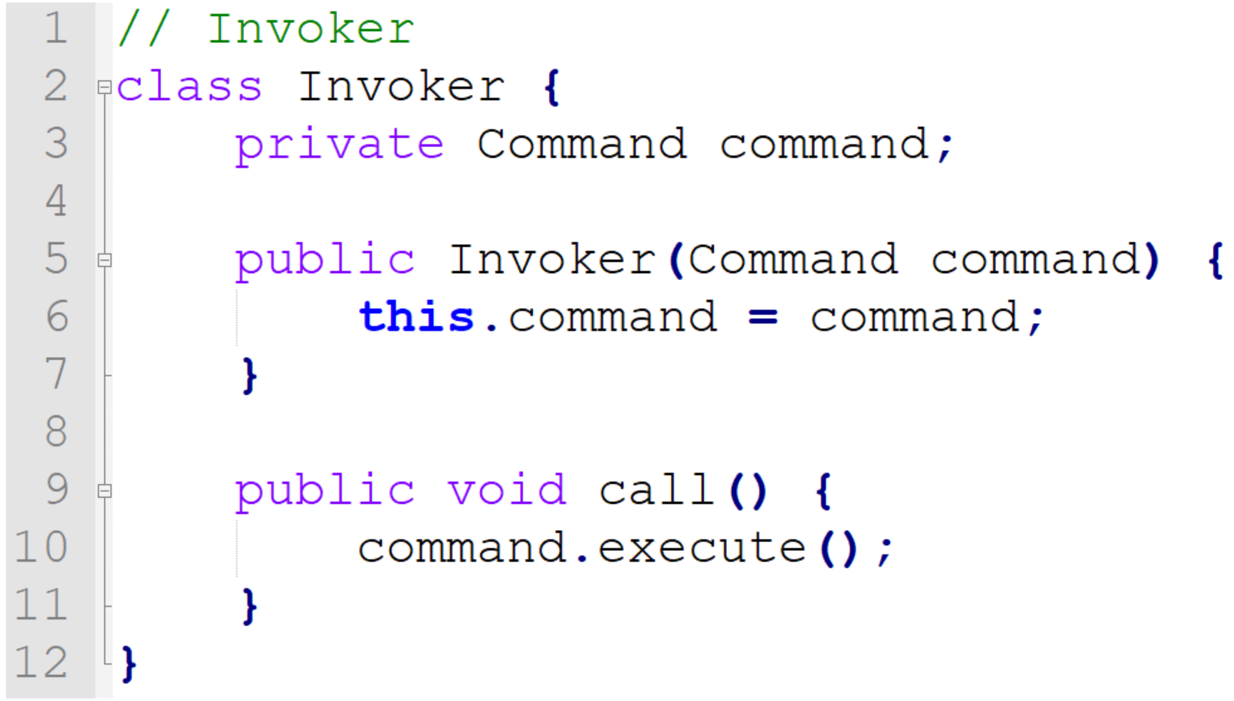
Implement Concrete Commands:



```
// Concrete Command
class ConcreteCommand implements Command {
    private Receiver receiver;

    public ConcreteCommand(Receiver receiver) {
        this.receiver = receiver;
    }

    public void execute() {
        receiver.action();
    }
}
```

Command – Implementation step 4

Define the Invoker



```java
// Invoker
class Invoker {
    private Command command;

    public Invoker(Command command) {
        this.command = command;
    }

    public void call() {
        command.execute();
    }
}
```

Command – Implementation step 5

Use the Command Pattern in the Client Code

```java
// Invoker
// Client
public class Client {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command command = new ConcreteCommand(receiver);
        Invoker invoker = new Invoker(command);
        invoker.call();
    }
}
```

Command – Pros and Cons

- *Single Responsibility Principle*. You can decouple classes that invoke operations from classes that perform these operations.
- *Open/Closed Principle*. You can introduce new commands into the app without breaking existing client code.
- You can implement undo/redo.
- You can implement deferred execution of operations.
- You can assemble a set of simple commands into a complex one.

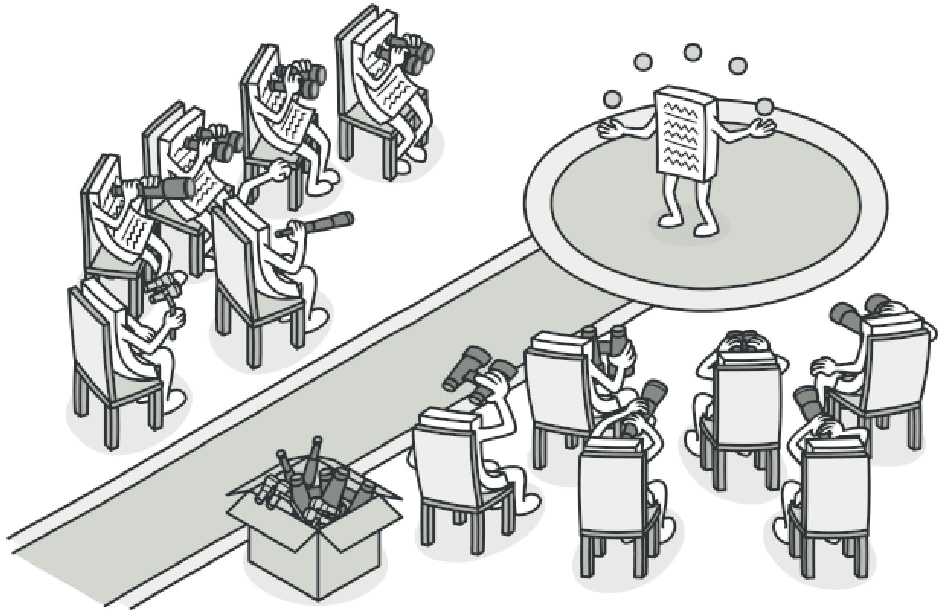
- The code may become more complicated since you’re introducing a whole new layer between senders and receivers.

Command – Use case in Java

- All implementations of java.lang.Runnable
- All implementations of javax.swing.Action

Behavioural Pattern – Observer

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

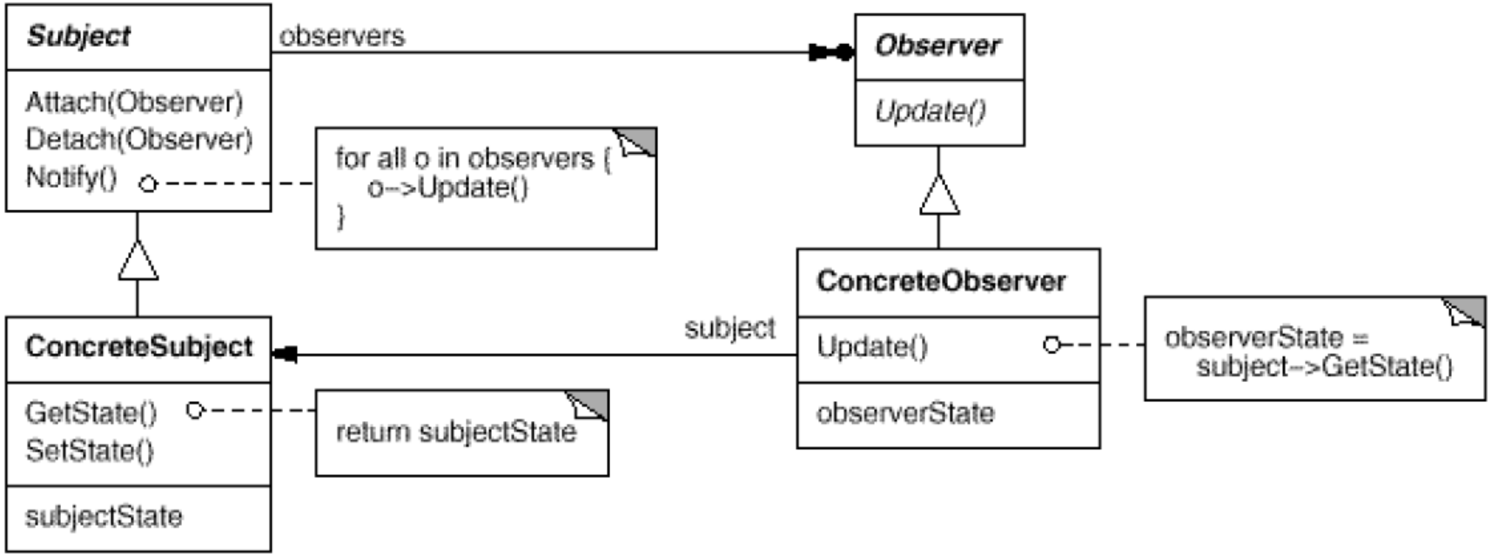


Observer – When to use

- Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.

- Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases.

Observer – UML

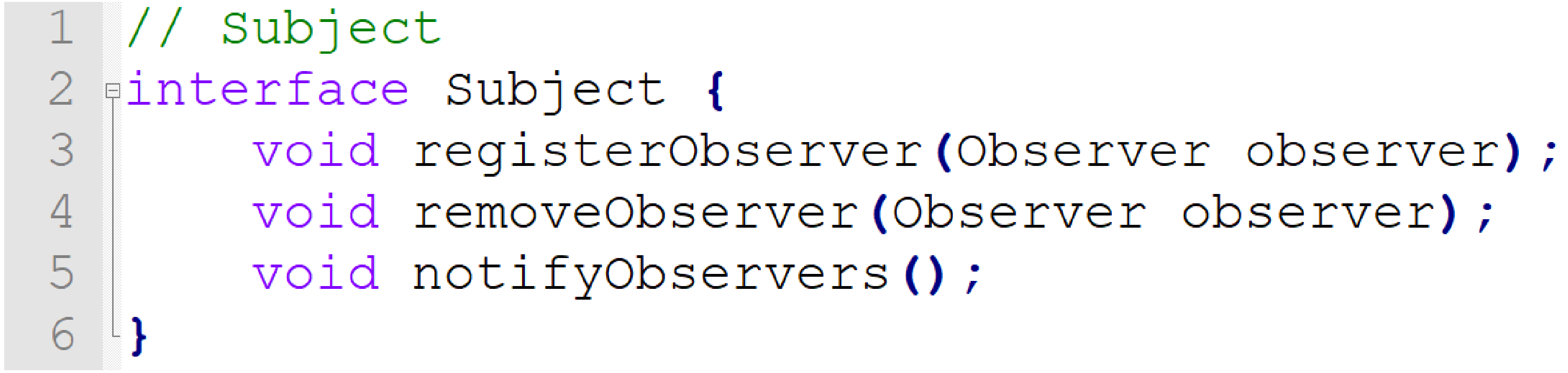


Observer – Participants

- **Subject (or Observable)**: The Subject is an object that holds the state and notifies observers about any state changes. It maintains a list of observers and provides methods to add and remove observers from this list. It also contains a method to notify all the observers when a change occurs.
- **Observer**: The Observer is an interface that defines the **update** method, which observers must implement. This method is called by the Subject when there is a change in its state. Observers use this method to update themselves based on the change.
- **Concrete Observer**: Concrete Observers are the actual objects that observe the Subject. They implement the Observer interface and define the **update** method. When this method is called, the Concrete Observer can execute code to reflect the changes in the Subject.
- **Concrete Subject (or Concrete Observable)**: The Concrete Subject is an object which extends or implements the Subject. It has the state that the Concrete Observers are interested in. When the state changes, the Concrete Subject notifies all registered observers by calling their **update** method with the updated state.

Observer – Implementation step 1

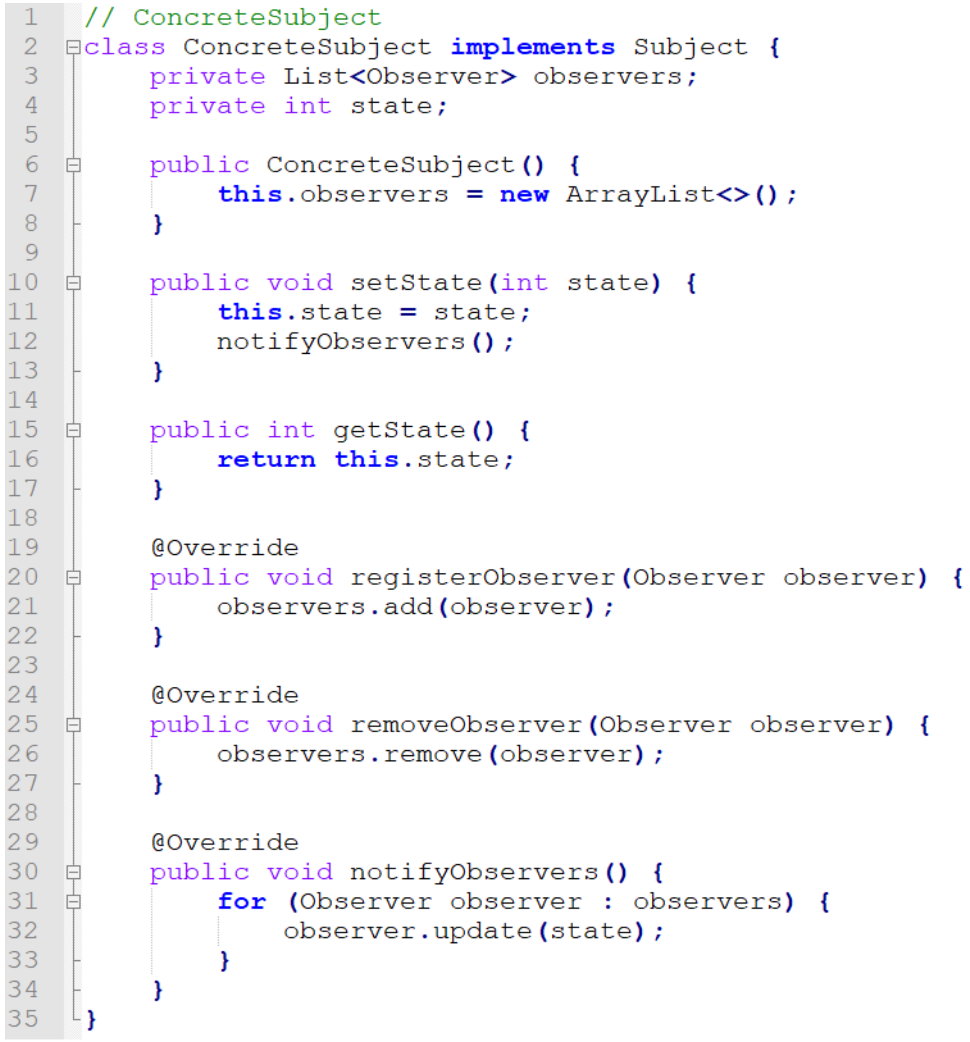
Define the Subject Interface



```
// Subject
interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
```

Observer – Implementation step 2

Concrete Subject

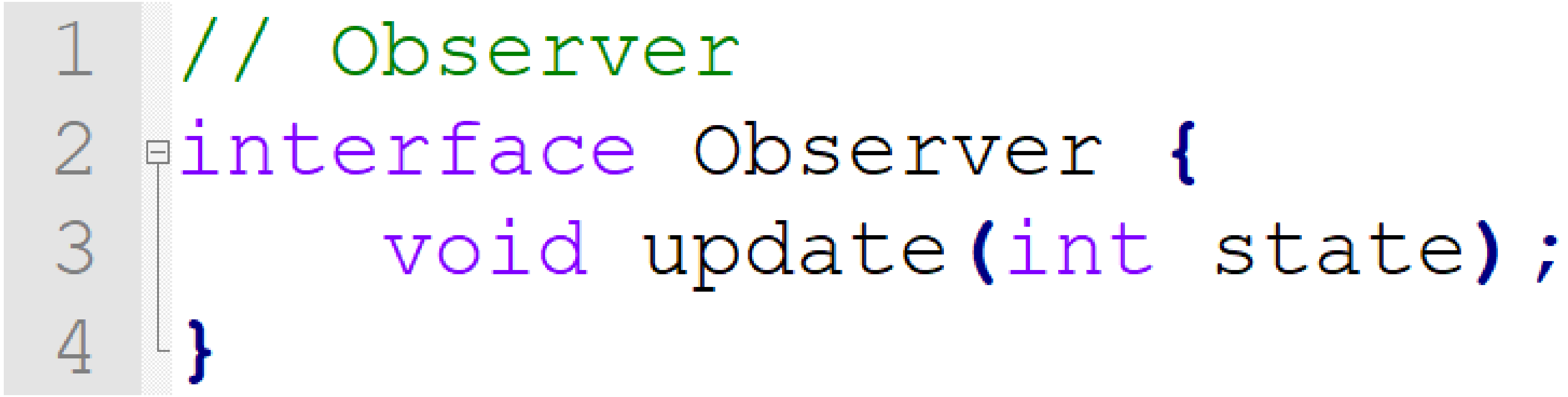


```java
// ConcreteSubject
class ConcreteSubject implements Subject {
    private List<Observer> observers;
    private int state;

    public ConcreteSubject() {
        this.observers = new ArrayList<>();
    }

    public void setState(int state) {
        this.state = state;
        notifyObservers();
    }

    public int getState() {
        return this.state;
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(state);
        }
    }
}
```

Observer – Implementation step 3

Define the Observer Interface

create an interface named **Subject** that declares methods for registering, unregistering, and notifying observers.



```
// Observer
interface Observer {
    void update(int state);
}
```

Observer – Implementation step 4

Implement the Concrete Observer

```java
// ConcreteObserver
class ConcreteObserver implements Observer {
    private int state;

    @Override
    public void update(int state) {
        this.state = state;
        System.out.println("State updated to: " + state);
    }
}
```

Observer – Implementation step 5

Demonstrate the Observer Pattern

```java
// Client
public class Client {
    public static void main(String[] args) {
        ConcreteSubject subject = new ConcreteSubject();
        ConcreteObserver observer1 = new ConcreteObserver();
        ConcreteObserver observer2 = new ConcreteObserver();

        subject.registerObserver(observer1);
        subject.registerObserver(observer2);

        subject.setState(1);  // All observers get updated
        subject.setState(2);  // All observers get updated
    }
}
```

Observer – Pros and Cons

- *Open/Closed Principle*. You can introduce new subscriber classes without having to change the publisher's code (and vice versa if there's a publisher interface).
- You can establish relations between objects at runtime.

- Subscribers are notified in random order.

Observer – Use case in Java

- All implementations of java.util.EventListener (practically all over Swing components)
- javax.servlet.http.HttpSessionBindingListener
- javax.servlet.http.HttpSessionAttributeListener

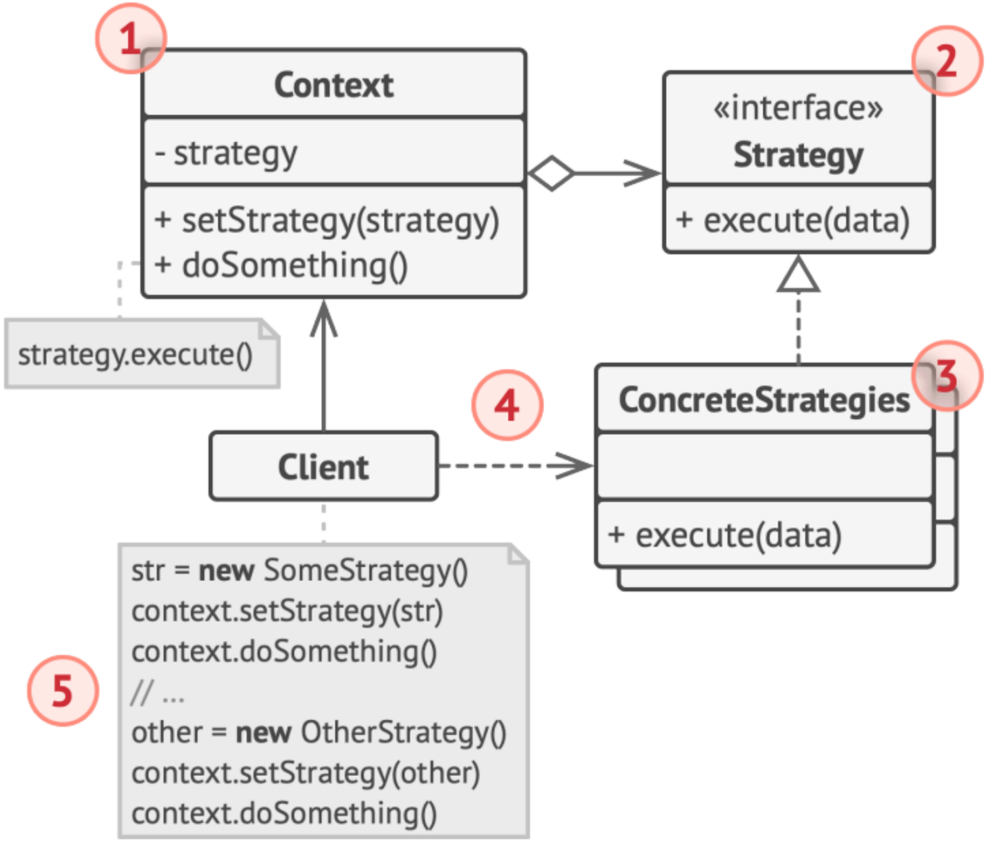
Behavioural Pattern – Strategy

Strategy design pattern lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Strategy – When to use

- when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
- when you have a lot of similar classes that only differ in the way they execute some behavior.
- when your class has a massive conditional statement that switches between different variants of the same algorithm.
- Use the pattern to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.

Strategy – UML



```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

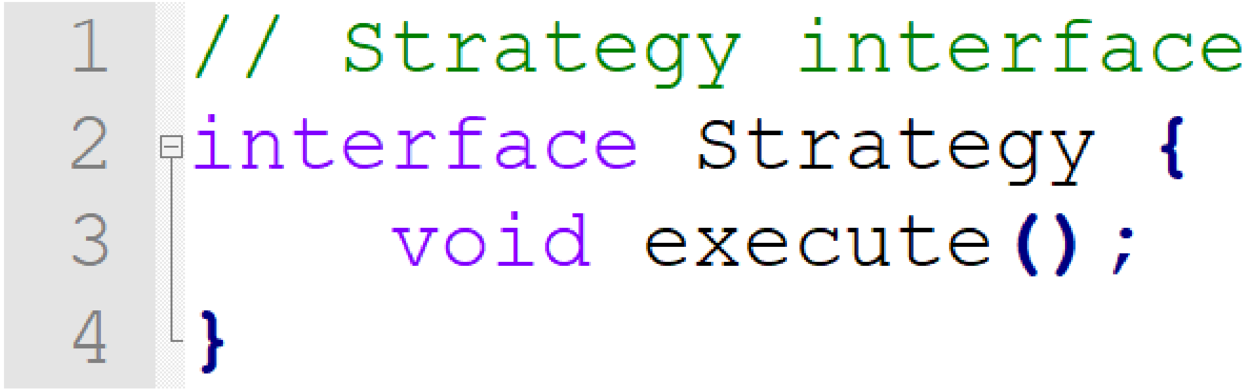
Strategy – Participants

- **Strategy Interface**: This is an interface that is common to all algorithms. It usually has a single method that the algorithms must implement.
- **Concrete Strategies**: These are the classes that implement the strategy interface. Each class represents a different algorithm.
- **Context**: This is the class that uses a strategy. It holds a reference to a strategy object and delegates it executing the algorithm. The context might accept strategy through the constructor and provide a setter to change it at runtime.

Strategy – Implementation step 1

Define the Strategy Interface

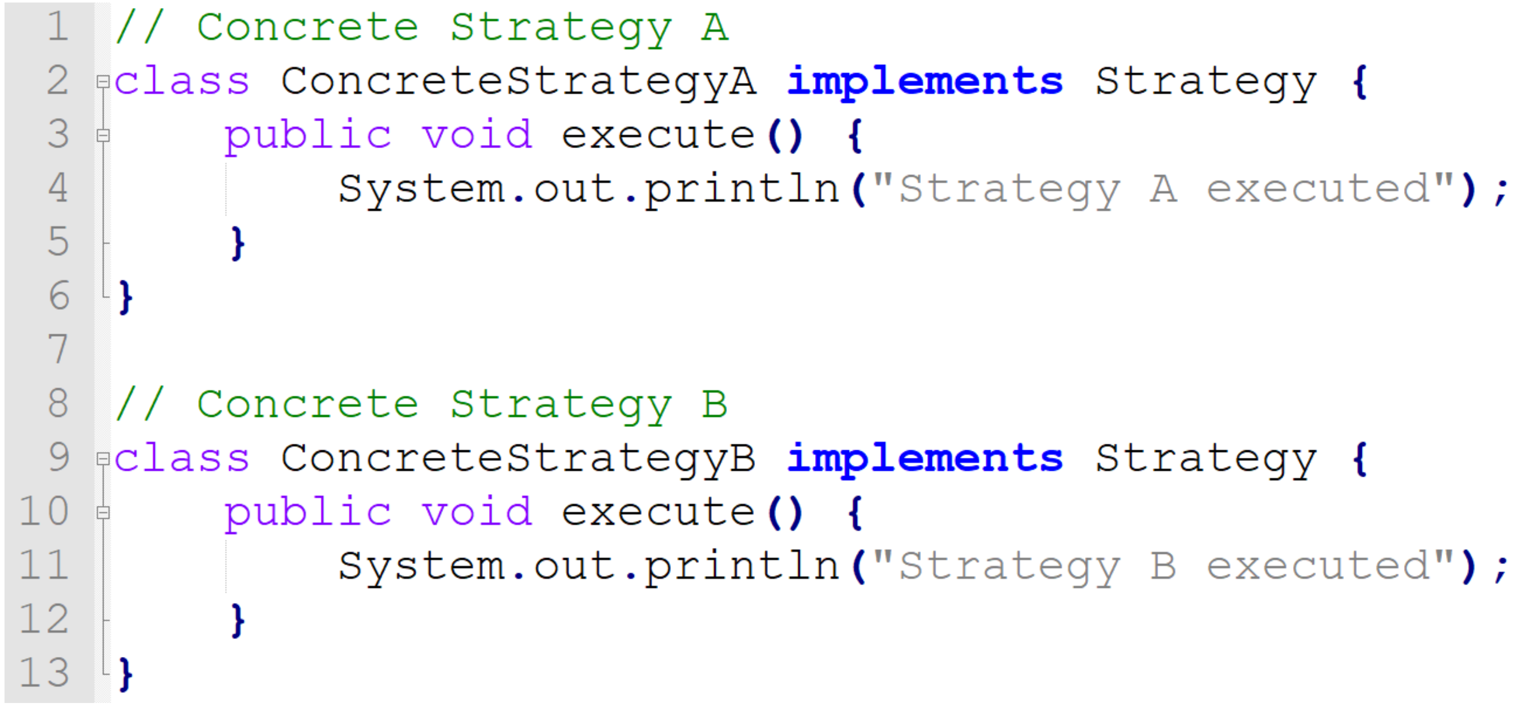
This is the interface that all of the algorithms will implement.



```
// Strategy interface
interface Strategy {
    void execute();
}
```

Strategy – Implementation step 2

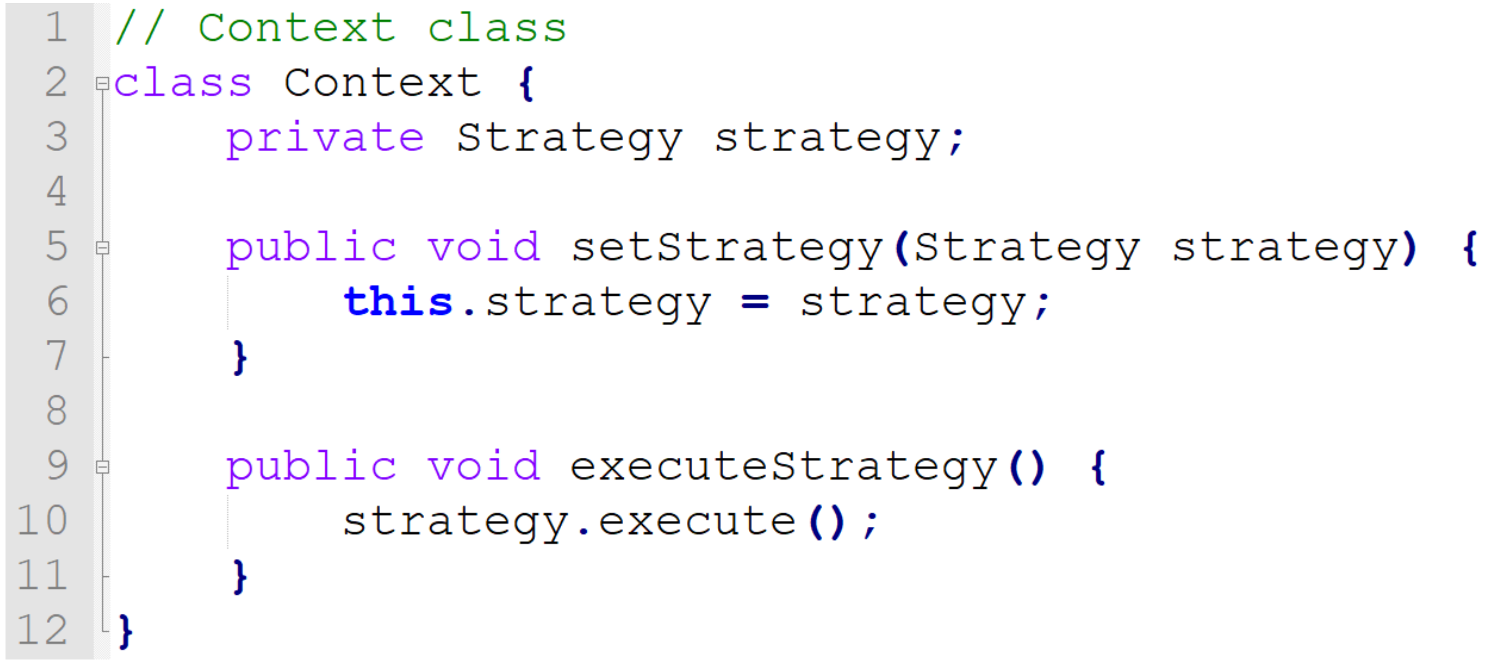
Implement Concrete Strategies



```java
// Concrete Strategy A
class ConcreteStrategyA implements Strategy {
    public void execute() {
        System.out.println("Strategy A executed");
    }
}

// Concrete Strategy B
class ConcreteStrategyB implements Strategy {
    public void execute() {
        System.out.println("Strategy B executed");
    }
}
```

Strategy – Implementation step 3

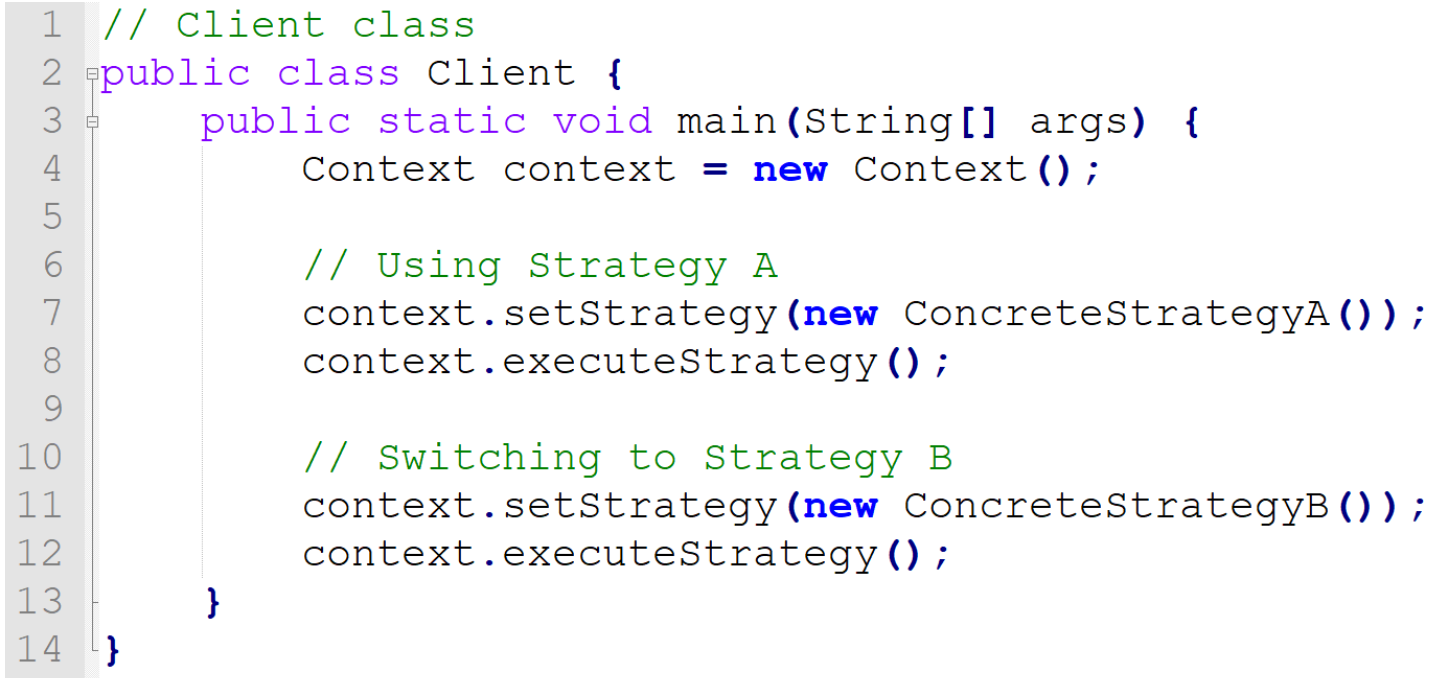
Create the Context Class



```
// Context class
class Context {
    private Strategy strategy;

    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    public void executeStrategy() {
        strategy.execute();
    }
}
```

Strategy – Implementation step 4

Use the Strategy Pattern in Client Code



```java
// Client class
public class Client {
    public static void main(String[] args) {
        Context context = new Context();

        // Using Strategy A
        context.setStrategy(new ConcreteStrategyA());
        context.executeStrategy();

        // Switching to Strategy B
        context.setStrategy(new ConcreteStrategyB());
        context.executeStrategy();
    }
}
```

Strategy – Pros and Cons

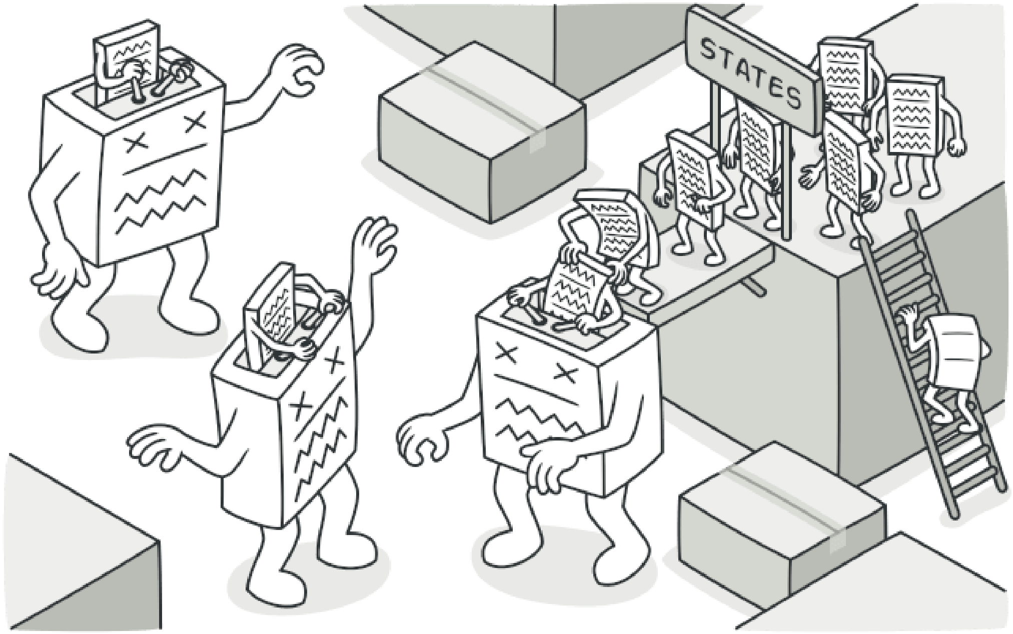
- You can swap algorithms used inside an object at runtime.
- You can isolate the implementation details of an algorithm from the code that uses it.
- You can replace inheritance with composition.
- *Open/Closed Principle*. You can introduce new strategies without having to change the context.
- If you only have a couple of algorithms and they rarely change, there's no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
- Clients must be aware of the differences between strategies to be able to select a proper one.
- A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you'd have used the strategy objects, but without bloating your code with extra classes and interfaces.

Strategy – Use case in Java

- java.util.Comparator#compare() called from Collections#sort().
- javax.servlet.http.HttpServlet: service() method, plus all of the doXXX() methods that accept HttpServletRequest and HttpServletResponse objects as arguments.
- javax.servlet.Filter#doFilter()

Behavioural Pattern – State

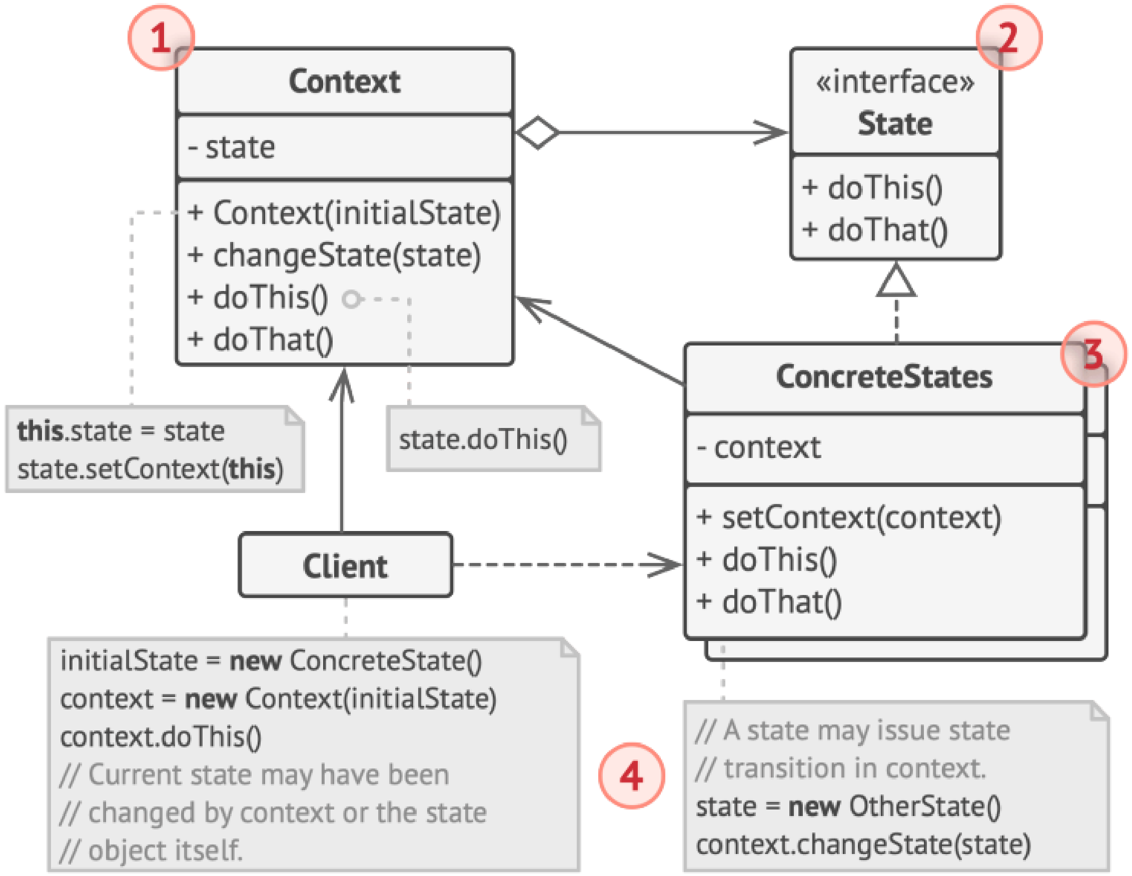
State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



State – When to use

- when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently.
- when you have a class polluted with massive conditionals that alter how the class behaves according to the current values of the class’s fields.
- when you have a lot of duplicate code across similar states and transitions of a condition-based state machine.

State – UML



```
this.state = state
state.setContext(this)
```

```
state.doThis()
```

```
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// Current state may have been
// changed by context or the state
// object itself.
```

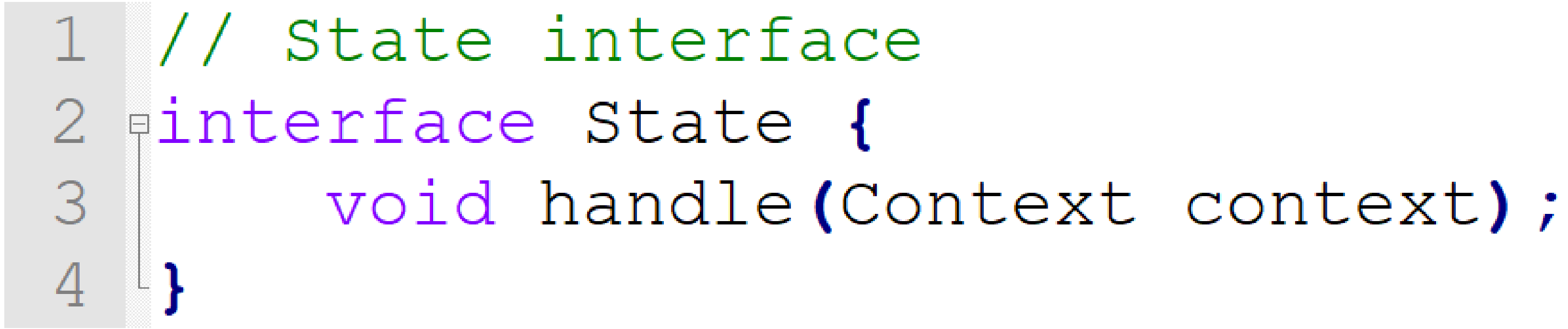
```
// A state may issue state
// transition in context.
state = new OtherState()
context.changeState(state)
```

State – Participants

- **Context**: This class holds an instance of a ConcreteState subclass that defines the current state.
- **State Interface**: This is an interface that is common to all concrete states. It usually has a method or methods that represent the behavior of the object.
- **Concrete States**: These are the classes that implement the State interface. Each class represents a different state of the Context.

State– Implementation step 1

Define the State Interface



```
// State interface
interface State {
    void handle(Context context);
}
```

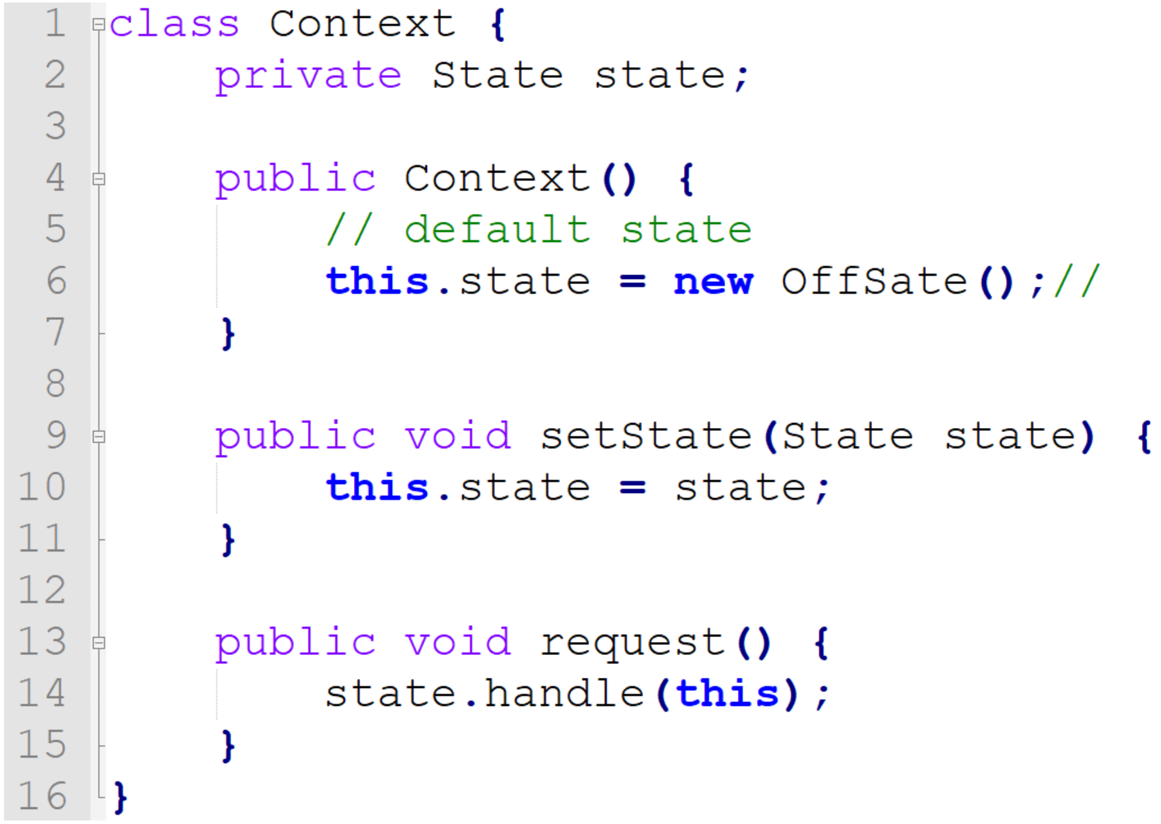
State – Implementation step 2

Implement Concrete States

```java
// Concrete States
class OnState implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Switching to Off State");
        context.setState(new OffState());
    }
}

class OffState implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Switching to On State");
        context.setState(new OnState());
    }
}
```

State – Implementation step 3

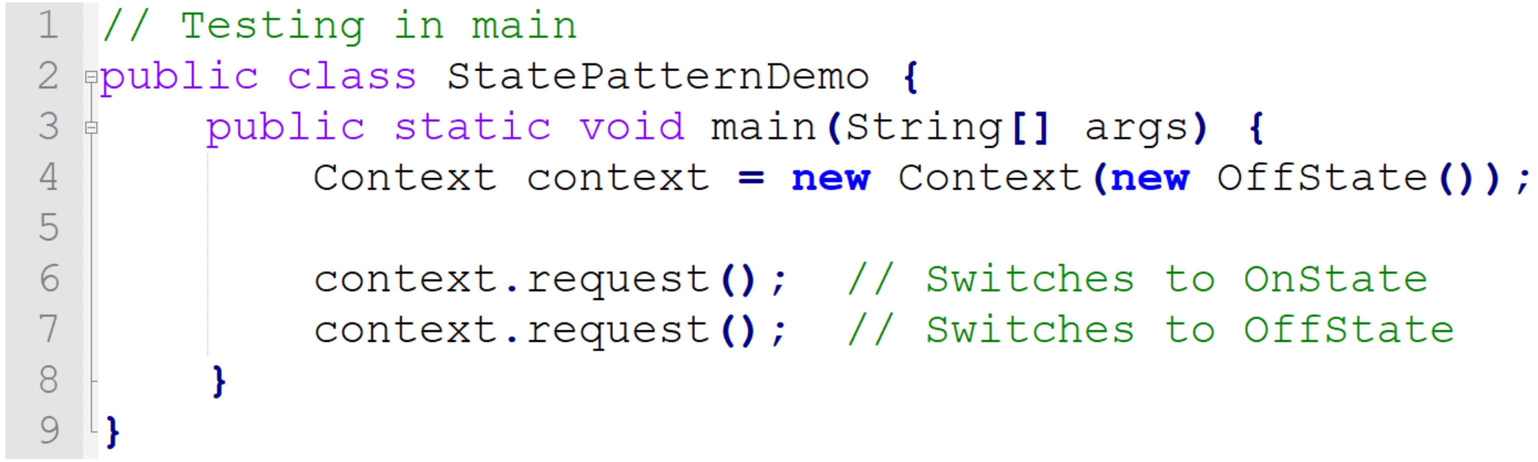
Create the Context Class



```java
class Context {
	private State state;

	public Context() {
		// default state
		this.state = new OffSate();//
	}

	public void setState(State state) {
		this.state = state;
	}

	public void request() {
		state.handle(this);
	}
}
```

State– Implementation step 4

Use the State Pattern in Client Code



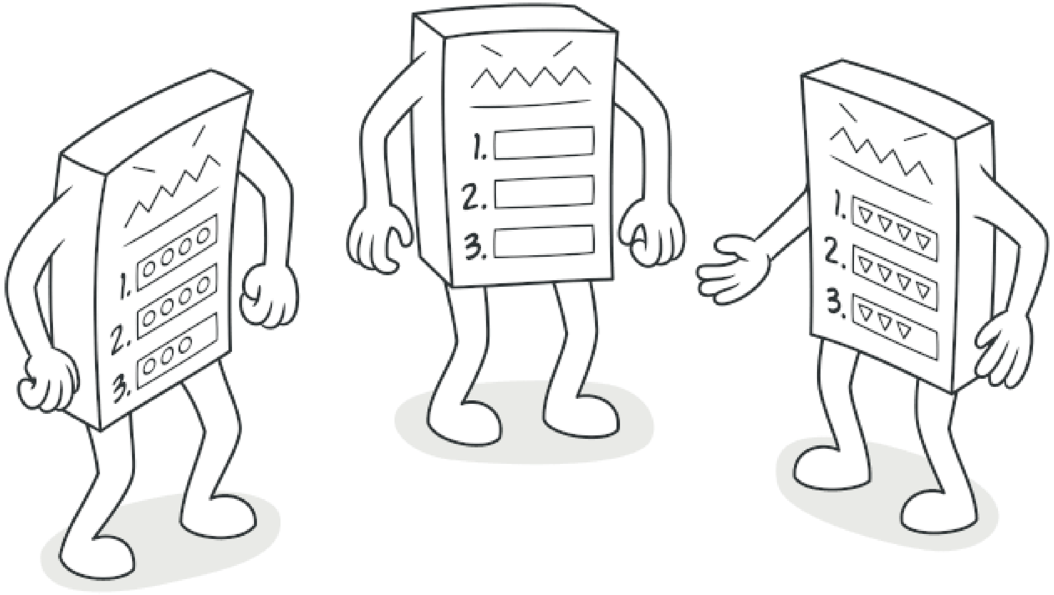
```java
// Testing in main
public class StatePatternDemo {
    public static void main(String[] args) {
        Context context = new Context(new OffState());

        context.request();  // Switches to OnState
        context.request();  // Switches to OffState
    }
}
```

State – Pros and Cons

- *Single Responsibility Principle*. Organize the code related to particular states into separate classes.
- *Open/Closed Principle*. Introduce new states without changing existing state classes or the context.
- Simplify the code of the context by eliminating bulky state machine conditionals.
- Applying the pattern can be overkill if a state machine has only a few states or rarely changes.

Behavioural Pattern – Template Method

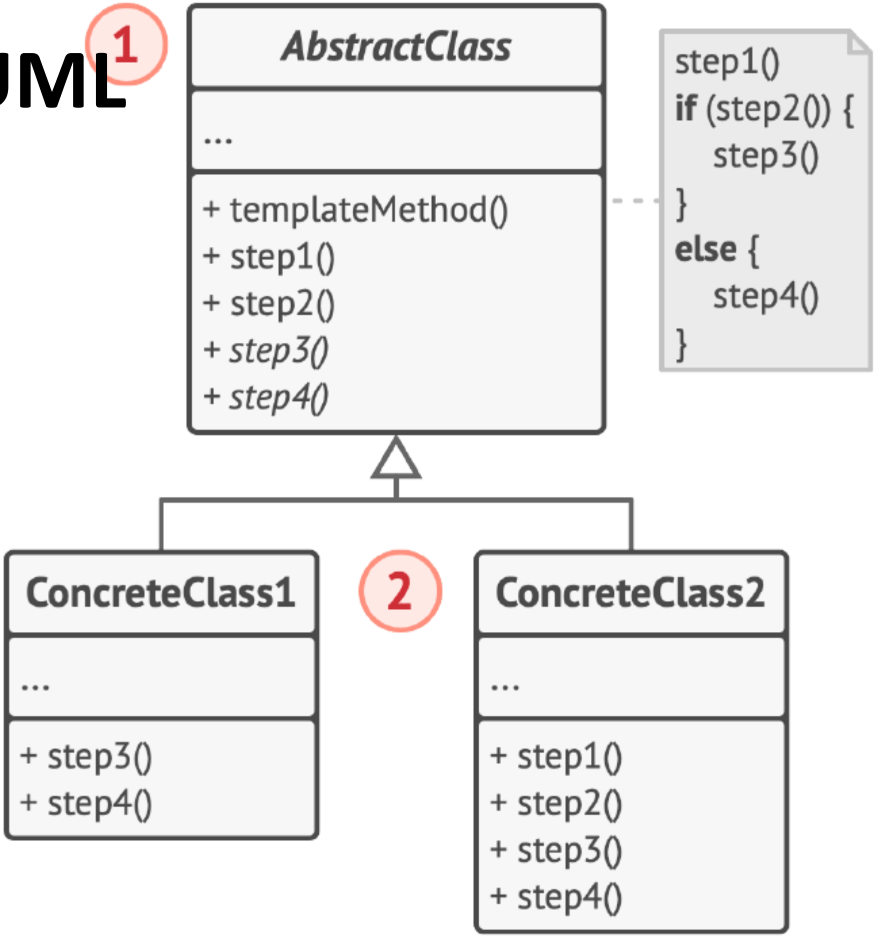
Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.



Template Method – When to use

- **When you want to provide a template for an algorithm**: The main feature of the Template Method pattern is that it provides a blueprint for an algorithm, typically in a base class, while allowing derived classes to provide specific implementations for some steps of the algorithm.
- **When there are similar classes with only a few differences in behavior**: If you have classes that perform very similar tasks but differ slightly, you can use the Template Method pattern. The common tasks can be implemented in the base class, and the varying parts can be implemented in the subclasses.
- **When you want to enforce a specific order of method execution**: The Template Method pattern allows you to enforce that methods are executed in a specific order. The abstract template method in the base class defines the order, and subclasses implement the methods in this predefined order.
- **When you want to provide hooks for subclasses**: The Template Method pattern allows you to provide default behavior in a method, which subclasses can override if necessary. This is typically done for optional parts of the algorithm.
- **When you want to prevent code duplication**: If the algorithm involves similar code in different classes, you can use the Template Method pattern to prevent this duplication. The common code can be factored out into a base class, leaving only the differences in the subclasses.

Template Method – UML

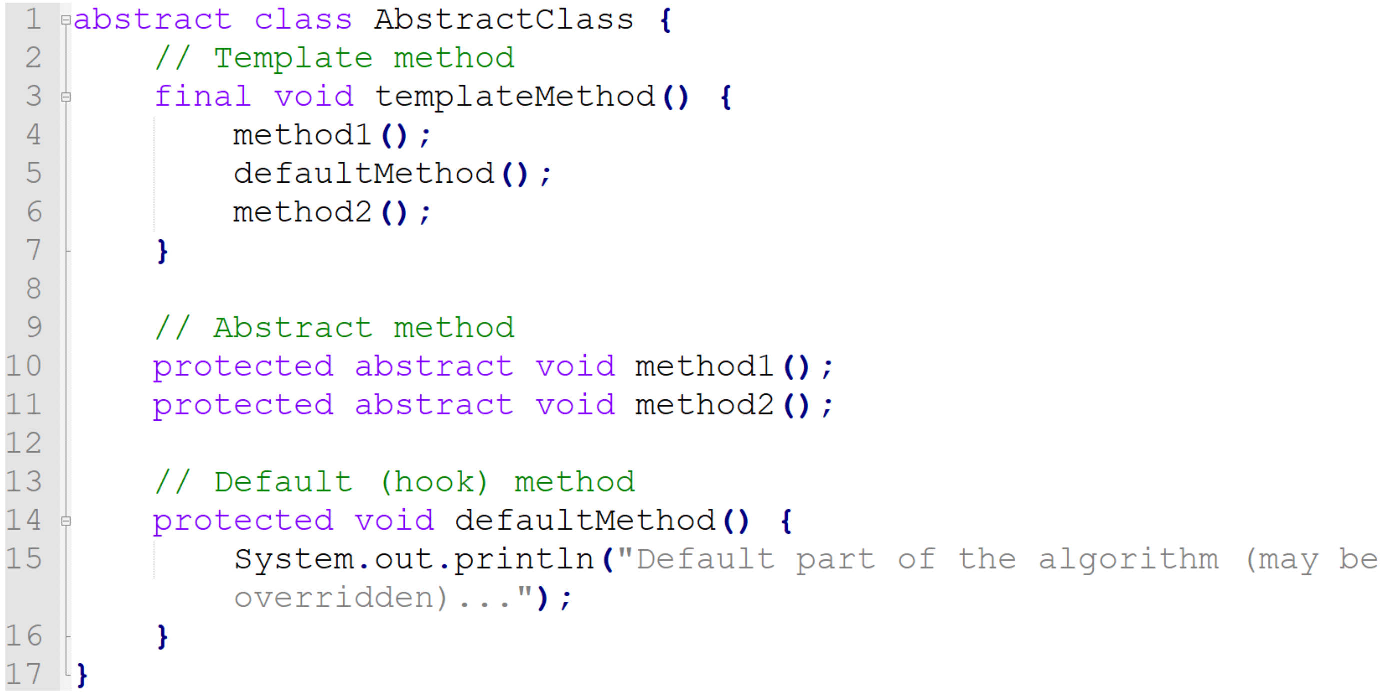


Template Method – Participants

- **Abstract Class**: Defines the structure of an algorithm, including some steps that are implemented directly in this class and others that it requires subclasses to implement.
- **Concrete Class**: Subclasses of the abstract class. They provide implementation for the steps required by the abstract class, without changing the overall structure of the algorithm.

Template Method – Implementation step 1

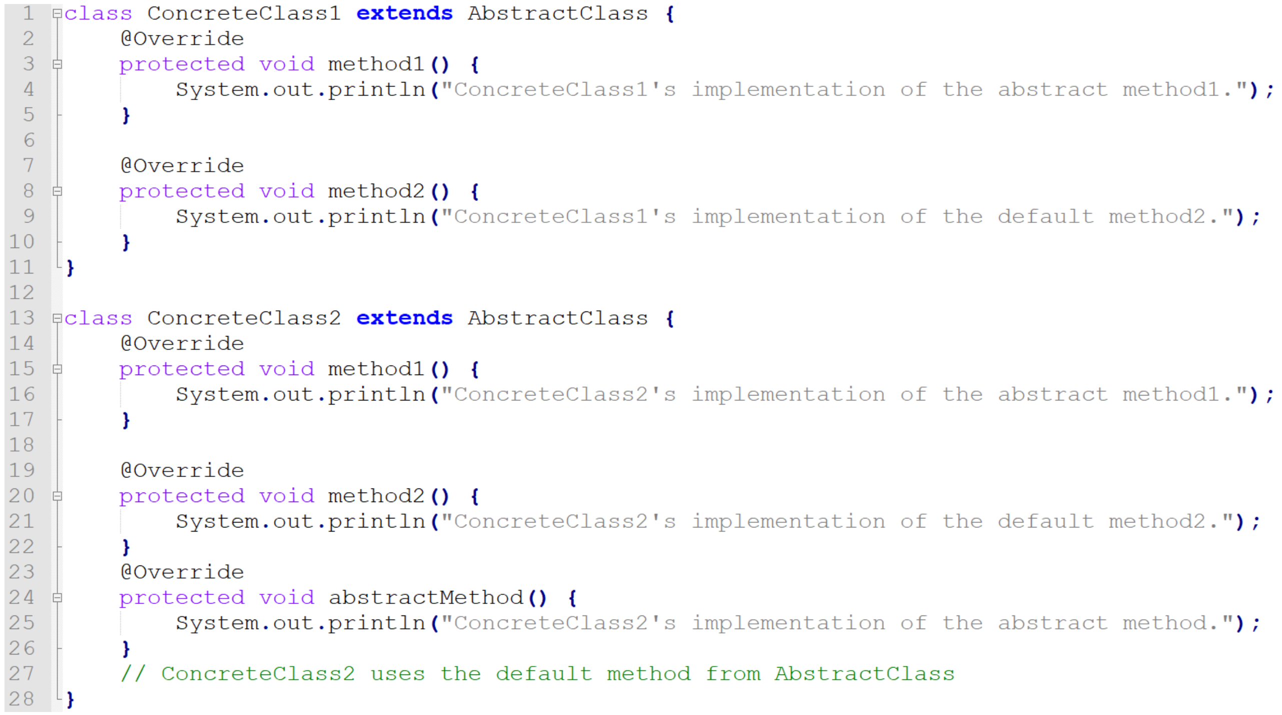
Define AbstractCLass



```java
abstract class AbstractClass {
    // Template method
    final void templateMethod() {
        method1();
        defaultMethod();
        method2();
    }

    // Abstract method
    protected abstract void method1();
    protected abstract void method2();

    // Default (hook) method
    protected void defaultMethod() {
        System.out.println("Default part of the algorithm (may be overridden)...");
    }
}
```

Template Method – Implementation step 2

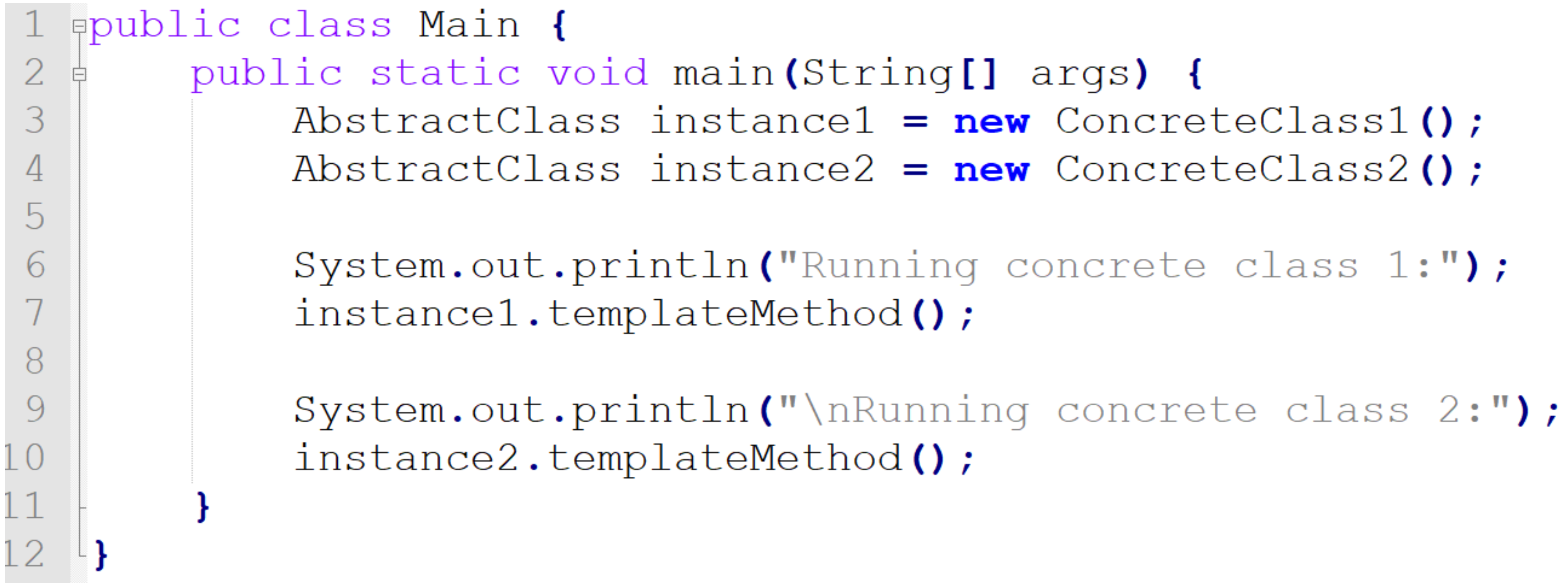
Define ConcreteCLass



```java
class ConcreteClass1 extends AbstractClass {
    @Override
    protected void method1() {
        System.out.println("ConcreteClass1's implementation of the abstract method1.");
    }

    @Override
    protected void method2() {
        System.out.println("ConcreteClass1's implementation of the default method2.");
    }
}

class ConcreteClass2 extends AbstractClass {
    @Override
    protected void method1() {
        System.out.println("ConcreteClass2's implementation of the abstract method1.");
    }

    @Override
    protected void method2() {
        System.out.println("ConcreteClass2's implementation of the default method2.");
    }
    @Override
    protected void abstractMethod() {
        System.out.println("ConcreteClass2's implementation of the abstract method.");
    }
    // ConcreteClass2 uses the default method from AbstractClass
}
```

Template Method – Implementation step 3

Use the pattern



```java
public class Main {
    public static void main(String[] args) {
        AbstractClass instance1 = new ConcreteClass1();
        AbstractClass instance2 = new ConcreteClass2();

        System.out.println("Running concrete class 1:");
        instance1.templateMethod();

        System.out.println("\nRunning concrete class 2:");
        instance2.templateMethod();
    }
}
```

Template – Pros and Cons

- You can let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm.
- You can pull the duplicate code into a superclass.
- Some clients may be limited by the provided skeleton of an algorithm.
- You might violate the *Liskov Substitution Principle* by suppressing a default step implementation via a subclass.
- Template methods tend to be harder to maintain the more steps they have.