# Strategy Pattern

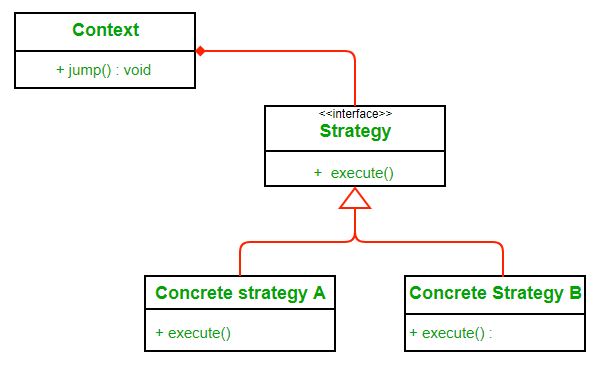
**Definition:**

“In computer programming, the strategy pattern (also known as the policy pattern) is a software design pattern that enables an algorithm’s behavior to be selected at runtime. The strategy pattern

* defines a family of algorithms,
* encapsulates each algorithm, and
* makes the algorithms interchangeable within that family.”

In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern. In Strategy pattern, we create classes which represent various strategies and a context class whose behavior varies as per its strategy class. The strategy class changes the executing algorithm of the context class.

***Class* Diagram:**

****

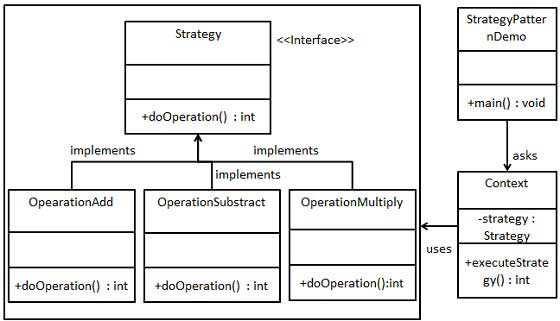
Here we rely on composition instead of inheritance for reuse. **Context**is composed of a **Strategy**. Instead of implementing a behavior the **Context** delegates it to **Strategy**. The context would be the class that would require changing behaviors. We can change behavior dynamically. **Strategy** is implemented as interface so that we can change behavior without affecting our context.

We will have a clearer understanding of strategy pattern when we will use it to solve our problem.

**Implementation**

We are going to create a *Strategy* interface defining an action and concrete strategy classes implementing the *Strategy* interface. *Context* is a class which uses a Strategy.

*StrategyPatternDemo*, our demo class, will use *Context* and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses.



**Advantages:**

1. A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.
2. By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
3. The application can switch strategies at run-time.
4. Strategy enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.
5. Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.

**Disadvantages:**

1. The application must be aware of all the strategies to select the right one for the right situation.
2. Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.
3. In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.

Step 1

Create an interface.

*Strategy.java*

public interface Strategy {

public int doOperation(int num1, int num2);

}

Step 2

Create concrete classes implementing the same interface.

*OperationAdd.java*

public class OperationAdd implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 + num2;

}

}

*OperationSubstract.java*

public class OperationSubstract implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 - num2;

}

}

*OperationMultiply.java*

public class OperationMultiply implements Strategy{

@Override

public int doOperation(int num1, int num2) {

return num1 \* num2;

}

}

Step 3

Create *Context* Class.

*Context.java*

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);

}

}

Step 4

Use the *Context* to see change in behaviour when it changes its *Strategy*.

*StrategyPatternDemo.java*

public class StrategyPatternDemo {

public static void main(String[] args) {

Context context = new Context(new OperationAdd());

System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationSubstract());

System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationMultiply());

System.out.println("10 \* 5 = " + context.executeStrategy(10, 5));

}

}

Step 5

Verify the output.

10 + 5 = 15

10 - 5 = 5

10 \* 5 = 50