* This pattern defines family of algorithms and allows then to be selected and used interchangeably at runtime

In the Strategy Pattern, the behavior of a class or its algorithm can be changed at runtime. This type of design pattern is a behavioral pattern.

In the strategy pattern, we create objects representing various strategies and a context object whose behavior changes as the strategy object changes. The strategy object changes the execution algorithm of the context object.

**introduce**

**Intent**: To define a series of algorithms, encapsulate them one by one, and make them interchangeable.

**The main solution**: the complexity and difficult maintenance of using if…else when there are multiple algorithms that are similar.

**When to use**: A system has many, many classes that are distinguished only by their direct behavior.

**Keycode**: Implement the same interface.

**Usage scenarios**: 1. If there are many classes in a system, and the difference between them is only their behavior, then using the strategy pattern can dynamically allow an object to choose one behavior among many behaviors. 2. A system needs to dynamically choose one of several algorithms. 3. If an object has a lot of behaviors, if the appropriate mode is not used, these behaviors have to be implemented using multiple conditional selection statements.

**Note**: If a system has more than four policies, you need to consider using a hybrid mode to solve the problem of policy class expansion.

**accomplish**

We’ll create a Strategy interface that defines the activity and an entity strategy class that implements the Strategy interface. Context is a class that uses a certain strategy.

public interface Strategy {  
 public int doOperation(int num1, int num2);  
}public class OperationSubstract implements Strategy{  
 [@Override](http://twitter.com/Override)  
 public int doOperation(int num1, int num2) {  
 return num1 - num2;  
 }  
}public class OperationMultiply implements Strategy{  
 [@Override](http://twitter.com/Override)  
 public int doOperation(int num1, int num2) {  
 return num1 \* num2;  
 }  
}

Step 3

|  |
| --- |
| 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 how its behavior changes when it changes the Strategy.

Step 5

Execute the program and output the result:

10 + 5 = 15  
10 - 5 = 5  
10 \* 5 = 50

Strategy — this design pattern allows us using different strategies by writing generic code. Examples are:

* [java.util.Comparator](https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Comparator.html)
* javax.servlet.Filter — doFilter()