# Strategy pattern

More on the strategy pattern

# Description

Encapsulating algorithms by creating a family of algorithms, which can be changed at runtime.

Where we have options of choosing one of several alternative algorithms for solving some problem, it could be wise to encapsulate the algorithm in an object.

This allows us to create new implementations of the algorithm without having to change client code.

We can even select a method at runtime (from user input or other reason).

# Encapsulate what varies

If we have a point in our application where an algorithm is applied (typically a method call, but *`algorithm`* is a word which makes us sound smarter).

If we discover (during the design) that exactly how this method will be implemented might vary from situation to situation, we can remove it from our client code and put it an object instead.

This allows us to follow the open-closed principle.

Our client code will stay the same (closed) but the implementation of the method is open for extension (modification and adding more versions).

# Some examples from the Java API

When comparing two objects, we have basically two options:

- Letting the class of the object implement Comparable
- Using an external object which can compare two objects
  - The external object will be of a class implementing Comparator

Of course, we can also combine the two.

# Who needs to compare objects?

Whenever we are ordering or ranking objects in some way, we need to be able to compare an object with an object of the same type.

The sorting and searching methods in java.util.Arrays and java.util.Collections need to be able to tell which of two objects should be considered “less than” the other.

The simplest versions of the sort method, simply take a list (or array) as an argument and - if the objects are of a class implementing Comparable - they can sort the list.

# If we want to sort a list of non-comparables?

Sometimes we need to sort a list of objects, whose class does not implement Comparable. Then we need to use an overloaded sort method, and provide an object which is capable of comparing two objects at the time and give the same answer as compareTo() would have done (if the objects had implemented the Comparable interface).

For this, there's another interface called java.util.Comparator

```
Arrays.sort(products, new ProductPriceComparator());
```

# The good news is...

Since the sort methods have outsourced the comparing of objects in the list to an object, it needs only to care about making sure that the comparator object is of the right type - that it implements Comparator.

Since Comparator<T> is a generic interface which declares a

```
compare(T o1, T o2)
```

method, the sorting methods can stay relaxed (and unchanged) and simply use it when it needs to compare the elements in the list.

This means that we can create as many comparators as we need, and they will all fit perfectly with the call to sort().

# Another example

In the Swing API, they have outsorced the handling of laying out the components within a container to a Layout object. This means that we can specify (even in runtime) what strategy for laying out (organizing) the components in a container, by using an object of some specific layout type.

Heck, we can even create our own layout class by implementing the interface LayoutManager.

https://docs.oracle.com/javase/tutorial/uiswing/layout/custom.html

# Filtering

Another typical use case for the strategy pattern is to use it for filtering.

A method which operates on a collection might need to be able to make informed choices about whether to operate on every object or not.

If we write the method in a way such that it asks a filter about each element before operating on it, we can encapsulate the actual decision (filtering) to an object.

We can create a filter interface and pass a reference to an object of its type as a second parameter to the method.

# Filtering example

```
public interface CarFilter{
  public boolean accept(Car c);
}
public class SelectForInspectionFilter implements CarFilter{
  public boolean accept(Car c){
    return c.year() < 2000;
  }
}
// Client code
for (Car c : carsToCheck){
  if(filter.accept(c)){
    scheduleForMaintenance(c);
  }
}
```

# Filtering example

```
public interface CarFilter{
  public boolean accept(Car c);
}
public class SelectForEngineCheck implements CarFilter{
  public boolean accept(Car c){
    return c.milesRun() > 7500; // new condition
  }
}
// Client code
for (Car c : carsToCheck){      // this code hasn't changed
  if(filter.accept(c)){
    scheduleForMaintenance(c);
  }
}
```