# Overview:

* The Strategy design pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable
  + Let’s the algorithm vary independently from client to client
  + Conceptually, all of these algorithms do the same things:
    - Just have different implementations
* We can select the behavior of an algorithm dynamically at runtime
  + Can help us to avoid dealing with complex algorithm (specific data structures).
  + We let the client application pass the algorithm to be used as a parameter.
* Allows us to create objects that represent various strategies and a context object whose behavior varies as per its strategy object.
  + The strategy object changes the executing algorithm of the context object.

# Examples:

* In a soccer match, strategies will differ:
  + If Team A is leading Team B by a score of 1-0.
  + Instead of attacking, Team A becomes defensive.
  + Team B goes for an all-out attack to score.
* We can think two dedicated storage devices:
  + When one device becomes “full”, we start storing the data in the second available device.
  + A runtime check is necessary before the storing of data, and based on the situation, we will proceed.
* The Collections.sort() method from the Java API uses the strategy pattern
  + Takes a Comparator parameter
  + Based on the different implementations of Comparator interfaces
    - The Objects are getting sorted in different ways.
* Suppose we have different algorithms for breaking a stream of text into lines.
* Hard wiring all suck algorithms into the classes that require them is not desirable for several reasons.
  + Clients that need line breaking get more complex if they include the line-breaking code.
    - Makes clients bigger and harder to maintain.
  + Different algorithms will be appropriate at different times
    - Do not want to support multiple line breaking algorithms if we do not use them all
  + Difficult to add new algorithms and vary existing ones when line breaking is an integral part of a client.
* We can avoid the above problems by defining classes that encapsulate different line-breaking algorithms (Strategy pattern).

# Principles of the Strategy Pattern:

* Objects have responsibilities.
* Different, specific implementations of these responsibilities are manifested through the use of polymorphism.
* There is a need to manage several different implementations of what is, conceptually, the same algorithm.
* It is a good design practice to separate behaviors that occur in the problem domain from each other (decoupling)
  + Allows me to change the class responsible for one behavior without adversely affecting another.

# Strategy versus State Pattern:

* The strategy and state pattern are very similar.
* Think of the Strategy Pattern as subclasses decide how to implement steps in an algorithm.
* Think of the State Pattern as an alternative to putting lots of conditionals in your context:
  + Encapsulate interchangeable behaviors and use delegation to decide which behavior to use.
* One difference between the two is that with the state pattern the Context contains state as an instance variable:
  + There can be multiple tasks whose implementation can be dependent on the state.
  + In the strategy pattern, strategy is passed as argument to the method and the context object does not have any variable to store it.

# Advantages:

* By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
* Applications can switch strategies at run-time (polymorphism).
* Enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.
* Simplifies unit testing because each algorithm is in its own class and can be tested through its interface alone.
  + The developer does not need to worry about interactions caused by coupling.
  + Developer is able to test each algorithm independently and not worry about all the combinations possible.