# Overview:

* The visitor design pattern represents an operation (method) to be performed on the elements of an object structure (collection, list, etc.)
  + Let’s you define a new operation without changing the classes of the elements on which it operates.
* Helps us to add new functionalities to an existing object structure in such a way that the old structure remains unaffected by these changes:
  + We can follow the open/close principle here.
* Allows one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure.
* Used when we have to perform an operation on a group of similar kinds of objects
  + We can move the operational logic from the objects to another class
* Useful for adding new operations without affecting the existing structure.

# Examples:

* Consider a taxi booking scenario:
  + Taxi arrives at the defined location for the pickup.
  + Once we enter into it, the visiting taxi takes control of the transportation.
    - It can choose a different way toward our destination and we may or may not have any prior knowledge of that way.
* A shopping cart where we can add different type of items (Elements)
  + When we click on checkout button, It calculates the total amount to be paid
  + We can move out the calculation logic from an item class to another class using the visitor pattern (less coupling).
* This pattern is very useful when plugging into public APIs
  + Clients can perform operations on a class with a visiting class without modifying the source.

# When to use visitor pattern?

* When an object structure (collection or list) contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
  + Visitor lets you keep related operations together by defining them in one class
    - Encourages cleaner code
* When you want to decouple some logical code from the elements that you are using as input.
* When you to add capabilities to a composite of objects and encapsulation is not important.
* When he object structure is shared by many applications, use the visitor pattern to put operations in just those applications that need them.

# Implementation:

* **Visitor:**
  + Used to declare the visit operations for all the types of visitable classes.
  + The operation’s name and signature identifies the class that sends the Visit request to the visitor:
    - Let’s the Visitor determine the concrete class of the element being visited.
      * The visitor can then access the element directly through its particular interface.
* **Concrete Visitor:**
  + Implements each method declared by Visitor.
  + Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure.
  + Provides the context for the algorithm and stores its local state:
    - State often accumulates results during the traversal of the structure.
* **Element:**
  + Defines an Accept method that takes a visitor as an argument:
    - The entry point, which enables an object to be “visited” by the visitor object.
* **ConcreteElement:**
  + Implements an Accept method that takes a visitor as an argument.
    - The visitor object is passed to this object using the accept method.
* **ObjectStructure:**
  + Can enumerate its elements.
  + May provide a high-level interface to allow the visitor to visit its elements.
  + May be either a composite or a collection such as a list or a set.
* **Client:**
  + A consumer of the classes of the visitor design pattern.
  + Has access to the data structure objects and can instruct them to accept a Visitor to perform the appropriate processing.

# Summary:

* The **Visitor** pattern will create an external class that uses data in other classes.
* The core of this pattern is the Visitor interface:
  + Defines a visit operation for each type of ConcreteElement in the object structure (composite or collection).
* The **ConcreteVisitor** implements the operations defined in the Visitor interface:
  + Will store local state, typically, as it traverses the set of elements.
* The element interface simply defines an accept method to allow the visitor to run some action over that element:
  + The ConcreteElement will implement this accept method.
* Visitor operations are controlled in a unified manner.
* Allows you to add operations to a Composite structure (or collection) without changing the structure itself.
* Very easy to add operations that depend on the components of complex objects.
  + You can define a new operation over an object structure simply by adding a new visitor.
* A visitor gathers related operations and separates unrelated ones.
  + Localized in a visitor.
  + Unrelated sets of behavior are partitioned in their own visitor subclasses.
  + Simplifies both the classes defining the elements and the algorithms defined in the visitors.
* Class encapsulation may need to be compromised when visitors are used:
  + If the existing structure is complex, the traversal mechanism becomes complex.
* A drawback of visitor pattern is that we should know the return type of visit () methods at the time of designing.
  + Otherwise, we will have to change the interface and all of its implementations.
* Another drawback is that if there are too many implementations of the visitor interface, it makes it hard to extend.