Visitor Design Pattern

===

[Understanding the Visitor Design Pattern](https://www.youtube.com/watch?vf =TeZqKnC2gvA) [9:49]

What is it?

Allows for the separation of new operations to existing object structures without modifying the original structure. No need to modify an individual part of a class for each class in the main interface to add a new operator to an overall main interface.

In the original data structure, you must add a new method that accepts the visitor method:

```
    /*
Interface Element{
    *   main code
    */

   // accept visitor
   accept(visitor);
}

ConcreteElementA(){
/*
    *   main code
    */

   // accept visitor
   accept(visitor);
}

ConcreteElementB(){
    /**
    *   main code
    */

   // accept visitor
   accept(visitor);
}
```

You can then create a new interface for the visitors and create the visitor for each method you would like to add:

```
Interface Visitor(){
    /**
    *   main code
    */

   // visit method
   visit(ConcreteElementA);
   visit(ConcreteElementB);
}

ConcreteVisitor1(){
    /**
    *   main code
    */

   // visit method
   visit(ConcreteElementA);
   visit(ConcreteElementB);
}

ConcreteVisitor2(){
    /**
     *  main code
    */

   // visit method
   visit(ConcreteElementA);
   visit(ConcreteElementB);
}
```

Each concrete visitor is a method that would be added to the original data structure. The main visitor interface will take in each visitor method and combine it as one method, which is called in the original elements of the main data structure, where the method in each element is overloaded with the `accept(visitor)` method.

Single vs. Double Dispatch (Polymorphism)

```
// Example from Video
/**
 * Differentiating single and double dispatch
 */

public class Runner {
    public static void main(String[] args){
        // old code
        Animal dog = new Dog();
        Animal cat = new Cat();

        // new code
        List<Animal> myAnimals = new ArrayList<Animal>();
        myAnimals.add(dog);
        myAnimals.add(cat);

        dog.makeSound();
```

```java
        cat.makeSound();
        // this is possible because each element(animal(dog/cat)) is part
of the main interface or superclass(Animal)
    }
}

public interface Animal {
    void makeSound();
}

public class Cat implements Animal {
    public void makeSound(){
        System.out.println("Meow");
    }
}

public class Dog implements Animal {
    public void makeSound(){
        System.out.println("Bark");
    }
}
```

Real World Example:

A bank has 3 credit cards:

- Bronze: $0 -> Cash Back: Gas 1%, Food 2%, Hotel 1%, Other 1%
- Silver: $100 -> Cash Back: Gas 2%, Food 3%, Hotel 2%, Other 5%
- Gold: $500 -> Cash Back: Gas 5%, Food 10%, Hotel 3%, Other 20%

What happens when one object interacts with another? Card with cash back for each category:

- Item 1: Credit Card
- Item 2: Offer

Create a single generic implementation for each card type to compute offer instead of creating a new method for each offer type and each card.

Runner

Credit Card Package

Offers Package