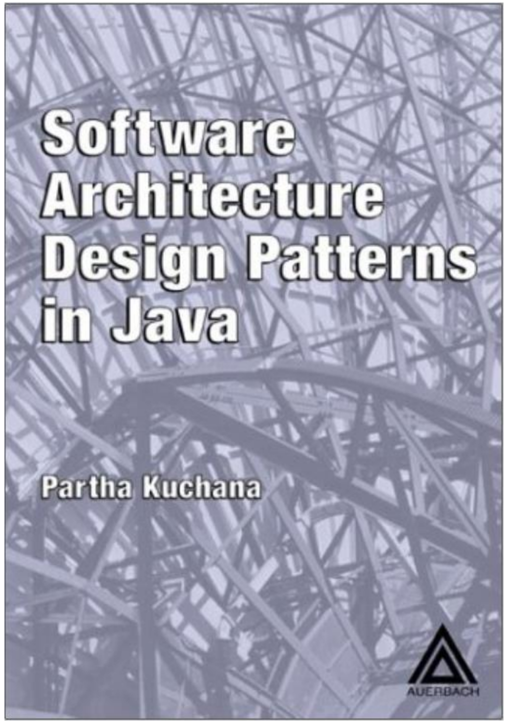
# Pola Desain Perangkat Lunak (PDPL)

[Minggu 2] – 24 Februari 2020
01_PDPL_BasicPattern

Prepared by Tifanny Nabarian (nabarian@gmail.com)

Tribute to Hilmi A.T. for magnificent previous lecture slides

## Reference



Kuchana, Partha. 2004. Software Architecture Design Patterns in Java. CRC Press.

# Topics

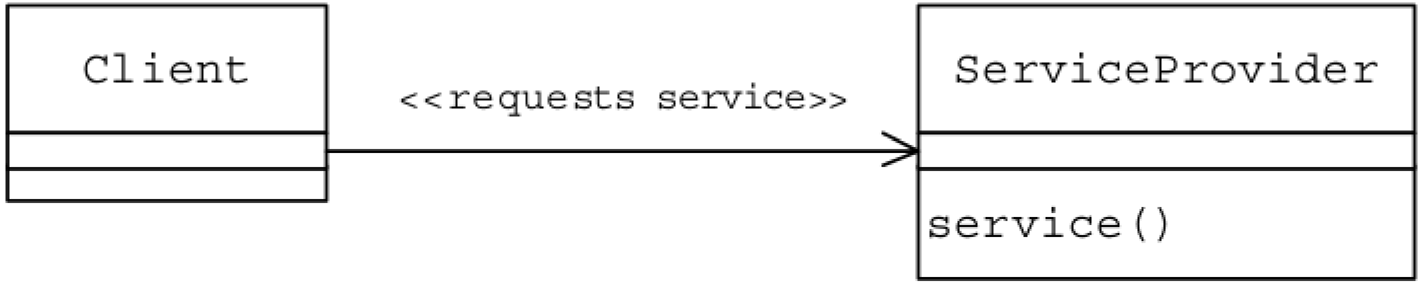
| | |
|---|---|
| **Interface** | Can be used to design a set of service provider classes that offer the same service so that a client object can use different classes of service provider objects in a seamless manner without having to alter the client implementation. |
| **Abstract Parent Class** | Useful for designing a framework for the consistent implementation of the functionality common to a set of related classes. |
| **Private Methods** | Provide a way of designing a class behavior so that external objects are not permitted to access the behavior that is meant only for the internal use. |
| **Accessor Methods** | Provide a way of accessing an object's state using specific methods. This approach discourages different client objects from directly accessing the attributes of an object, resulting in a more maintainable class structure. |
| **Constant Data Managet** | Useful for designing an easy to maintain, centralized repository for the constant data in an application. |
| **Immutable Object** | Used to ensure that the state of an object cannot be changed. May be used to ensure that the concurrent access to a data object by several client objects does not result in race conditions. |

## Interface

## Interface

- In general, the functionality of an object-oriented system is encapsulated in the form of a set of objects.
- These objects provide different services either on their own or by interacting with other objects.
- In other words, a given object may rely upon the services offered by a different object to provide the service it is designed for.
- An object that requests a service from another object is referred as a client object. Some other objects in the system may seek the services offered by the client object

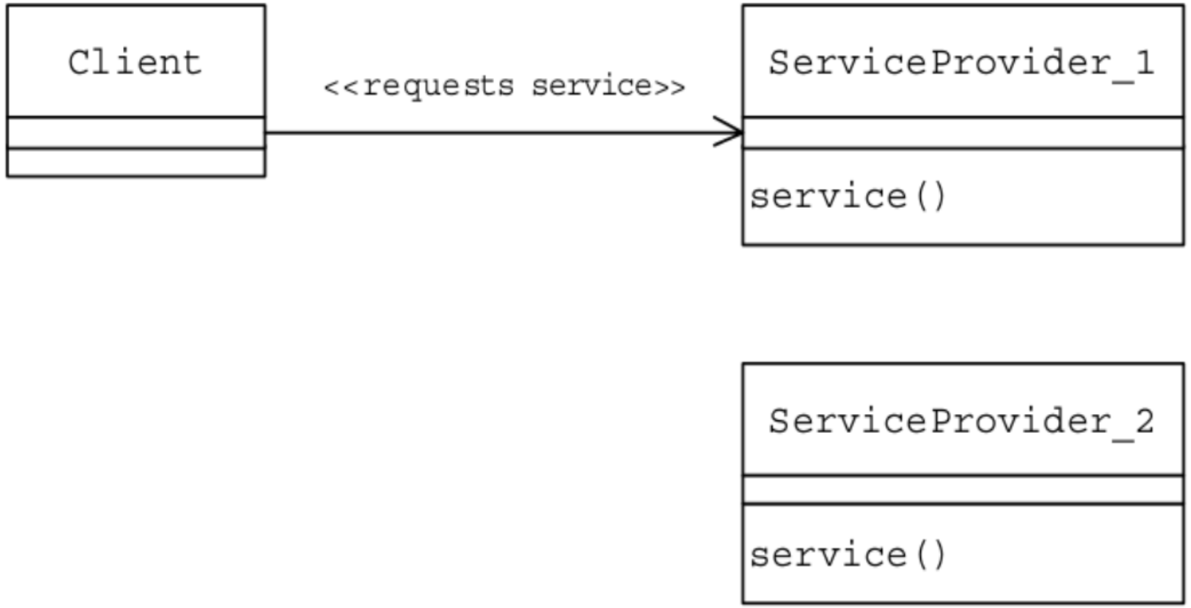
## Interface



- This type of direct interaction **ties the client with a specific class type** for a given service request.
- This approach works fine when there is only one class of objects offering a given service, but may not be adequate when **there is more than one class of objects that provide the same service** required by the client

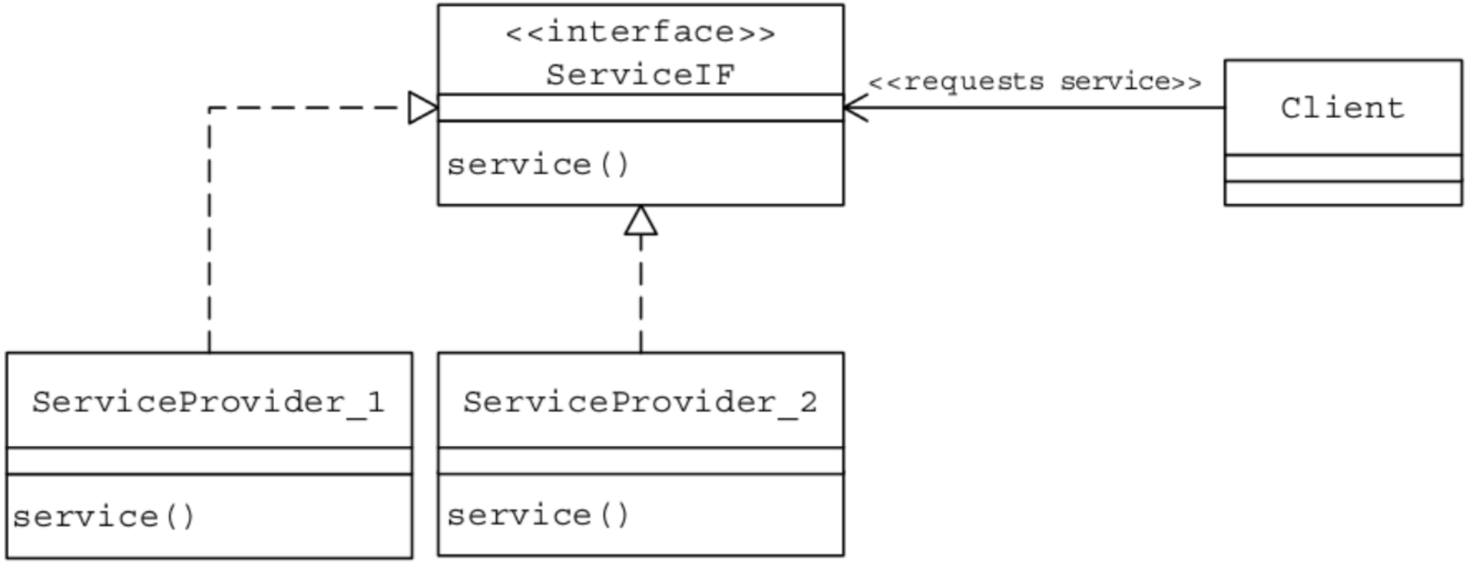
## Interface

Problem..



## Interface

Solution...



## Interface

In the Java programming language, an interface is a reference type, **similar to a class**, that can contain only **constants**, **method signatures**, default **methods**, **static methods**, and **nested type**

## Interface (example..)

Class CategoryA to calculate salary with Category A

```
public class CategoryA {
    double baseSalary;
    double OT;
    public CategoryA(double base, double overTime) {
      baseSalary = base;
      OT = overTime;
    }
    public double getSalary() {
      return (baseSalary + OT);
    }
}
```

## Interface (example..)

Employee class

```
public class Employee {
  CategoryA salaryCalculator;
  String name;
  public Employee(String s, CategoryA c) {
    name = s;
    salaryCalculator = c;
  }
  public void display() {
    System.out.println("Name=" + name);
    System.out.println("salary= " +
                           salaryCalculator.getSalary());
  }
}
```

## Interface (example..)

### Main application

```
public class MainApp {
  public static void main(String [] args) {
    CategoryA c = new CategoryA(10000, 200);
    Employee e = new Employee ("Jennifer,"c);
    e.display();
  }
}
```

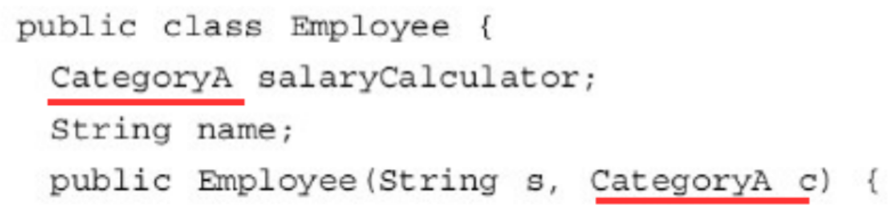
## Interface (example..)

What about salary with category B??

```
public class CategoryB {
  double salesAmt;
  double baseSalary;
  final static double commission = 0.02;
  public CategoryB(double sa, double base) {
    baseSalary = base;
     salesAmt = sa;
  }
  public double getSalary() {
    return (baseSalary + (commission * salesAmt));
  }
}
```

# Interface (example..)

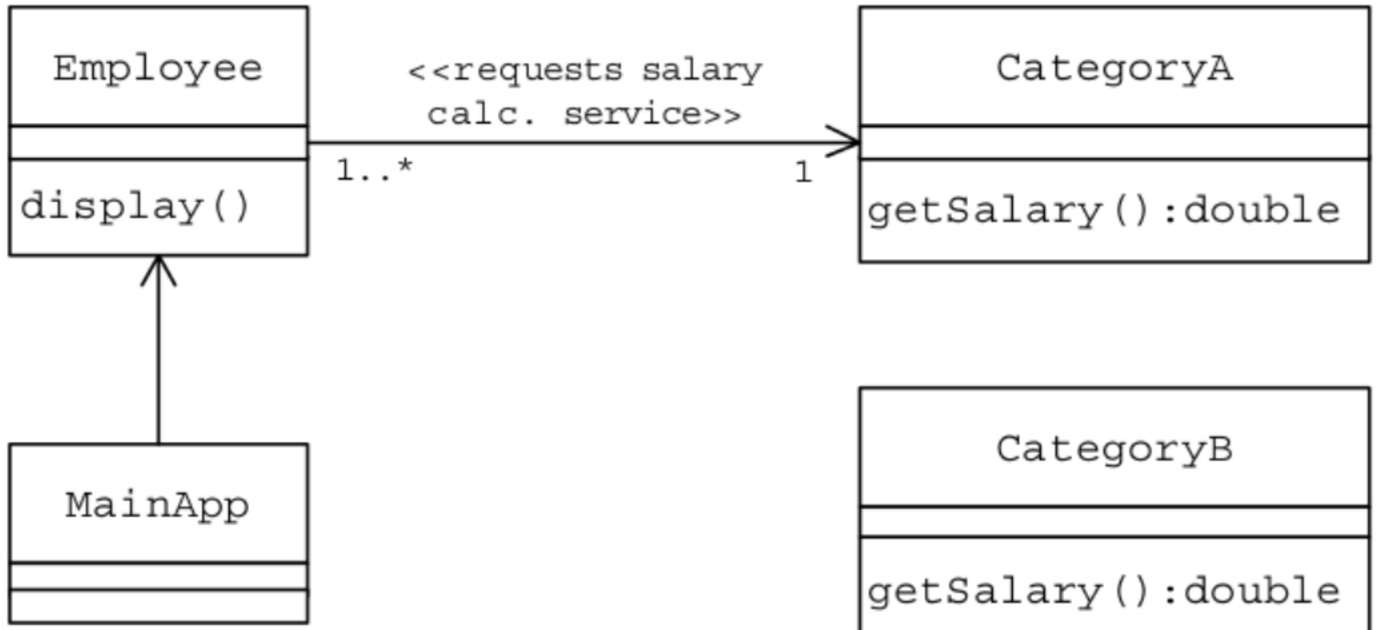
To accommodate category B, we have to modify Employee class. What about category C, D, …, n ??



```
public class Employee {
  CategoryA salaryCalculator;
  String name;
  public Employee(String s, CategoryA c) {
```

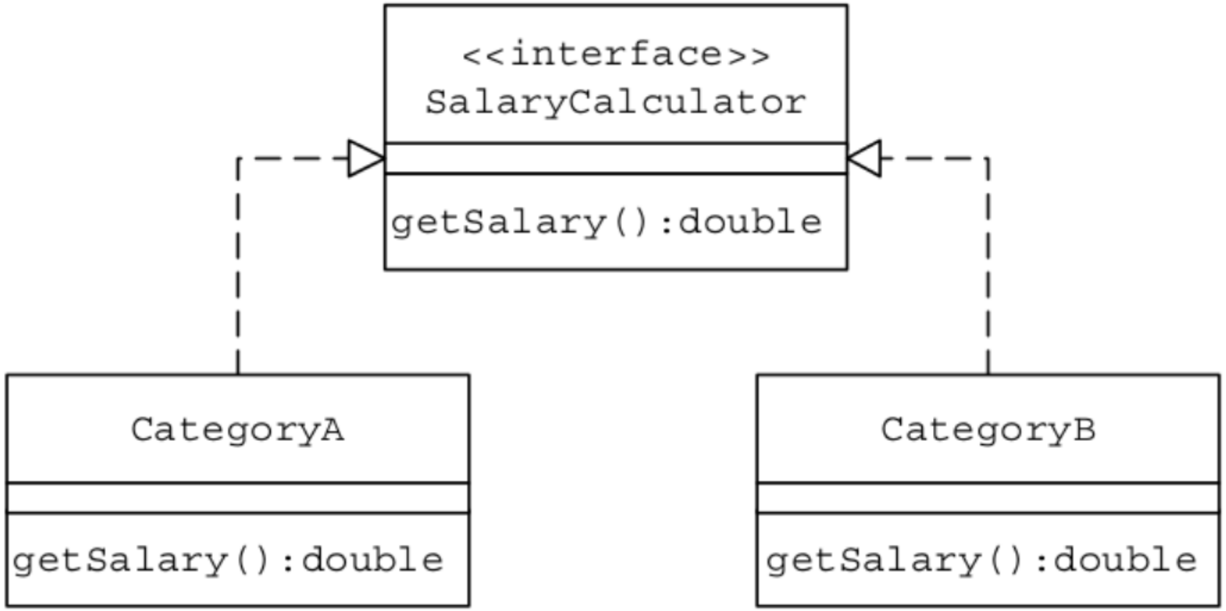
We have to create different objects and constructor for every category!!!

## Interface (example..)



# Interface (example..)

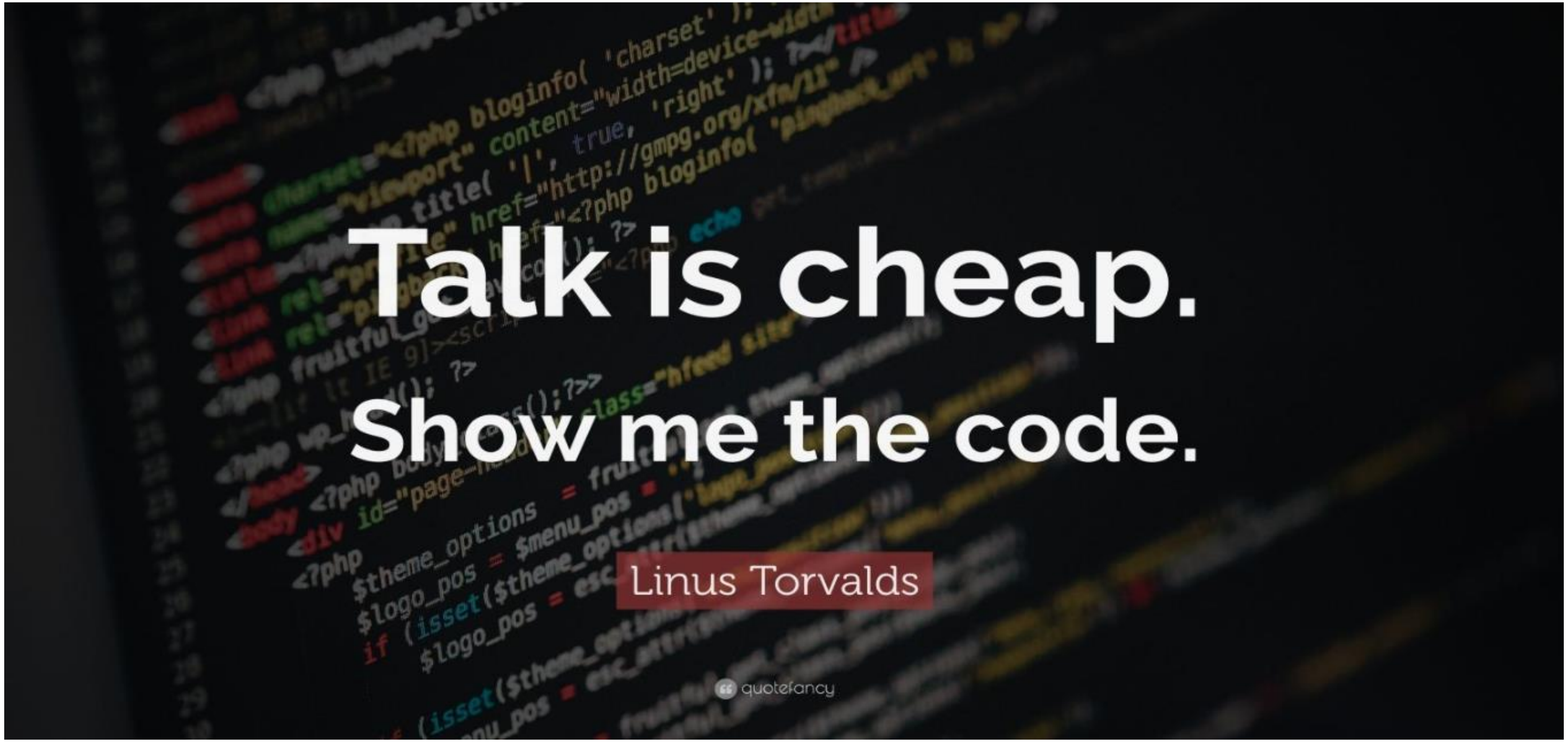
Solution:



## Interface (example..)

- Main application

```
public class MainApp {
   public static void main(String [] args) {
      SalaryCalculator c = new CategoryA(10000, 200);
      Employee e = new Employee ("Jennifer",c);
      e.display();
      c = new CategoryB(20000, 800);
      e = new Employee ("Shania",c);
      e.display();
   }
}
```


Talk is cheap.
Show me the code.
Linus Torvalds
quotefancy

## Abstract Parent Class

## Abstract Parent Class..

- The Abstract Parent Class pattern is useful for designing a framework for the consistent implementation of functionality common to a set of related classes.
- An abstract method is a method that is declared, but contains no implementation.
- An abstract class is a class with one or more abstract methods.
- Abstract methods, with more than one possible implementation, represent variable parts of the behavior of an abstract class.
- An abstract class may contain implementations for other methods, which represent the invariable parts of the class functionality.

# Abstract Parent Class..

- Different subclasses may be designed when the functionality outlined by abstract methods in an abstract class needs to be implemented differently.
- An abstract class, as is, may not be directly instantiated. When a class is designed as a subclass of an abstract class, it must implement all of the abstract methods declared in the parent abstract class. Otherwise the subclass itself becomes an abstract class.
- Only non abstract subclasses of an abstract class can be instantiated.
- The requirement that every concrete subclass of an abstract class must implement all of its abstract methods ensures that the variable part of the functionality will be implemented in a consistent manner in terms of the method signatures.
- The set of methods implemented by the abstract parent class is automatically inherited by all subclasses. This eliminates the need for redundant implementations of these methods by each subclass.

## Abstract Parent Class..

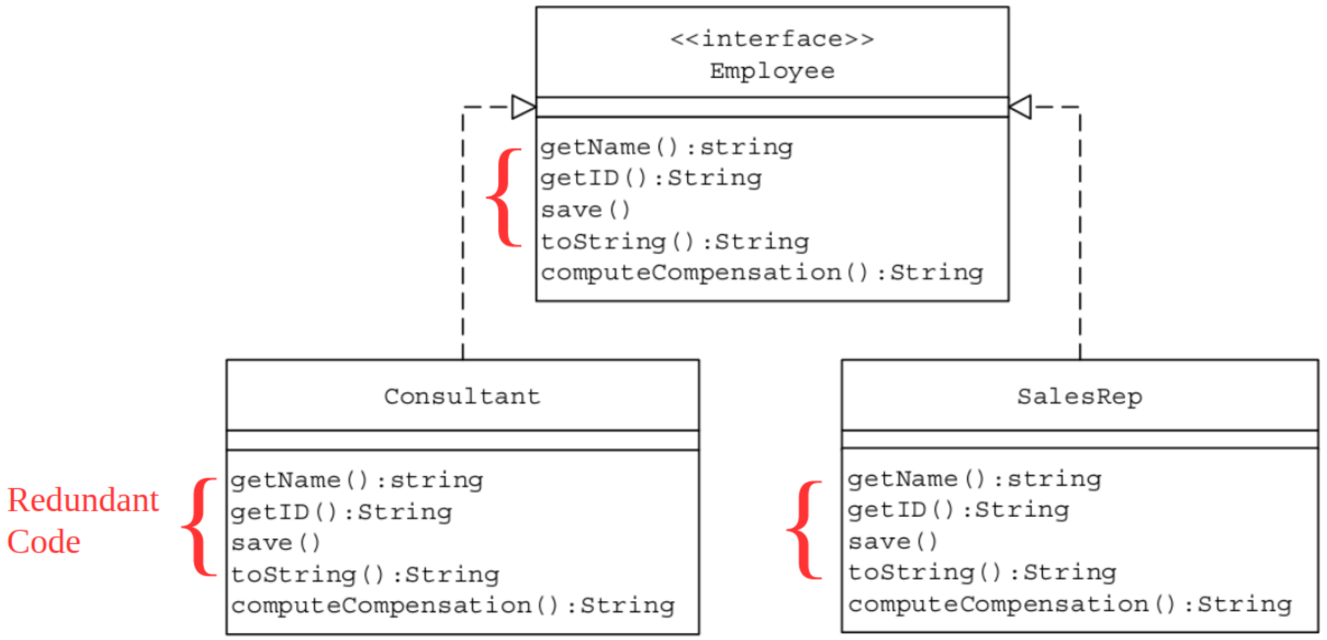
- In the Java programming language there is no support for multiple inheritance.
- That means a class can inherit only from one single class. Hence inheritance should be used only when it is absolutely necessary.
- Whenever possible, methods denoting the common behavior should be declared in the form of a Java interface to be implemented by different implementer classes. But interfaces suffer from the limitation that they cannot provide method implementations.
- This means that every implementer of an interface must explicitly implement all methods declared in an interface, even when some of these methods represent the invariable part of the functionality and have exactly the same implementation in all of the implementer classes. This leads to redundant code.

## Abstract Parent Class..

- Let us consider the following operations as part of designing the representation of an employee.
  1. Save employee data
  2. Display employee data
  3. Access employee attributes such as name and ID
  4. **Calculate compensation**
- Operation 1 through Operation 3 remain the same for all employees, the compensation calculation will be different for employees with different designations.

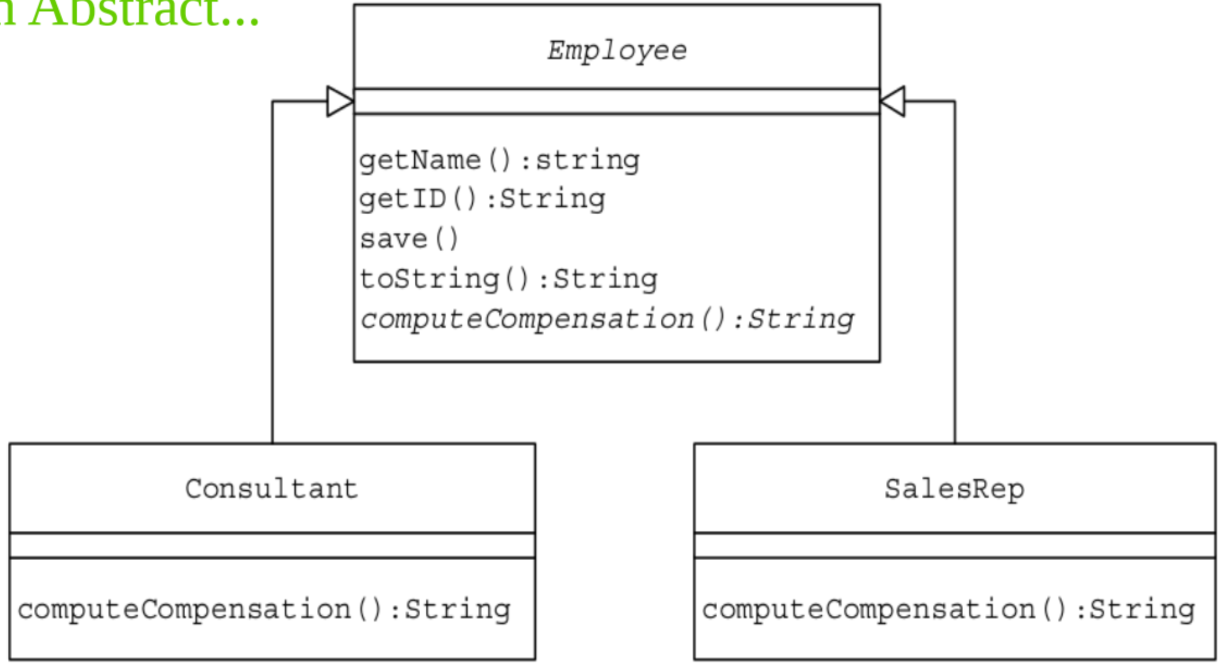
# Abstract Parent Class..

With interface...



## Abstract Parent Class..

With Abstract...


*Employee*
getName():string
getID():String
save()
toString():String
*computeCompensation():String*
Consultant
computeCompensation():String
SalesRep
computeCompensation():String

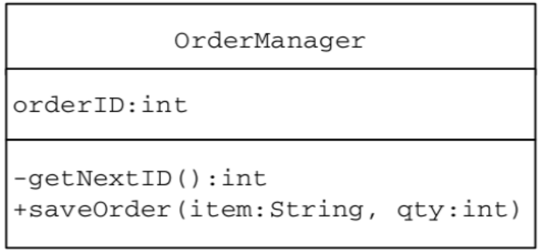
## Private Methods

## Private Methods..

- Typically a class is designed to offer a well-defined and related set of services to its clients. These services are offered in the form of its methods, which constitute the overall behavior of that object.
- In case of a well-designed class, each method is designed to perform a single, defined task. Some of these methods may use the functionality offered by other methods or even other objects to perform the task they are designed for.
- Not all methods of a class are always meant to be used by external client objects. Those methods that offer defined services to different client objects make up an object's public protocol and are to be declared as public methods.
- Some of the other methods may exist to be used internally by other methods or inner classes of the same object. The Private Methods pattern recommends designing such methods as private methods

# Private Methods (example)..

```
public class OrderManager {
  private int orderID = 0;
  //Meant to be used internally
  private int getNextID() {
    ++orderID;
    return orderID;
  }
  //public method to be used by client objects
  public void saveOrder(String item, int qty) {
    int ID = getNextID();
    System.out.println("Order ID=" + ID + "; Item=" + item +
                       "; Qty=" + qty + " is saved. ");
  }
}
```



## Accessor Methods

## Accessor Methods

- The Accessor Methods pattern is one of the most commonly used patterns in the area of object-oriented programming.
- In general, the values of different instance variables of an object, at a given point of time, constitute its state. The state of an object can be grouped into **two categories — public and private**.
- The public state of an object is available to different client objects to access, whereas the private state of an object is meant to be used internally by the object itself and not to be accessed by other objects.
- All instance variables being declared as private and provide public methods known as accessor methods to access the public state of an object.

## Accessor Methods

- This prevents external client objects from accessing object instance variables directly.
- In addition, accessor methods hide from the client whether a property is stored as a direct attribute or as a derived one.
- An object can access its private variables directly. But doing so could greatly affect the maintainability of an application, which the object is part of.
- When there is a change in the way a particular instance variable is to be defined, it requires changes to be made in every place of the application code where the instance variable is referenced directly.

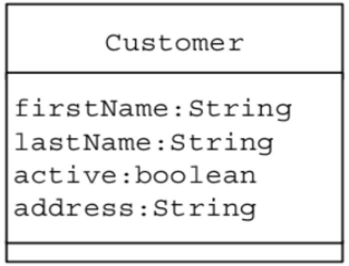
## Accessor Methods Nomenclature

- To access a **non-Boolean** instance variable:
  - Define a **getXXXX()** method to read the values of an instance variable XXXX
  - Define a **setXXXX(new value)** method to alter the value of an instance variable XXXX

- To access a **Boolean** instance variable
  – Define an **isXXXX()** method to check if the value of an instance variable XXXX is true or false.
  – Define a **setXXXX(new value)** method to alter the value of a Boolean instance variable XXXX

# Accessor Methods (example)

| *Variable* | *Method* | *Purpose* |
|---|---|---|
| `firstName` | `getFirstName` | To read the value of the `firstName` instance variable |
| | `setFirstName` | To alter the value of the `firstName` instance variable |
| `lastName` | `getLastName` | To read the value of the `lastName` instance variable |
| | `setLastName` | To alter the value of the `lastName` instance variable |
| `address` | `getAddress` | To read the value of the `address` instance variable |
| | `setAddress` | To alter the value of the `address` instance variable |
| `active` | `isActive` | To read the value of the `active Boolean` instance variable |
| | `setActive` | To alter the value of the `active Boolean` instance variable |



## Accessor Methods Example..

If we need to add the following two new methods to the Customer class.

1. **isValidCustomer** — To check if the customer data is valid.
2. **save** — To save the customer data to a data file.

We can use two different approach

- ◻ access variable directly
- ◻ access variable via accessor method

When there is a **change** in the definition of any of the instance variables, it requires a **change to the implementation of all the methods** that access these instance variables directly.

## Accessor Methods Example..

Approach 1 (access directly)

```
  public boolean isValidCustomer() {
    if ((firstName.length() > 0) && (lastName.length() > 0) &&
         (address.length() > 0))
      return true;
    return false;
  }
 public void save() {
    String data =
      firstName + ″," + lastName + ″," + address +
     ″," + active;
    FileUtil futil = new FileUtil();
    futil.writeToFile("customer.txt″,data, true, true);
  }
```

## Accessor Methods Example..

Approach 2 (access via accessor methods)

```
public boolean isValidCustomer() {
  if ((getFirstName().length() > 0) &&
      (getLastName().length() > 0) &&
      (getAddress().length() > 0))
    return true;
  return false;
}
public void save() {
  String data =
    getFirstName() + ″," + getLastName() + ″," +
    getAddress() + ″," + isActive();
  FileUtil futil = new FileUtil();
  futil.writeToFile("customer.txt″,data, true, true);
}
```

## Constant Data Manager

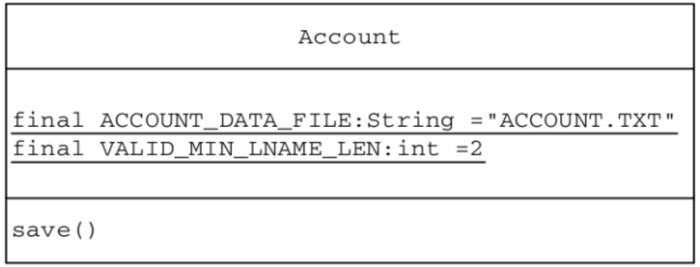
## Constant Data Manager

- Objects in an application usually make use of different types of data in offering the functionality they are designed for.
- Such data can either be variable data or constant data. The Constant Data Manager pattern is useful for designing an efficient storage mechanism for the constant data used by different objects in an application.
- In general, application objects access different types of constant data items such as data file names, button labels, maximum and minimum range values, error codes and error messages, etc.

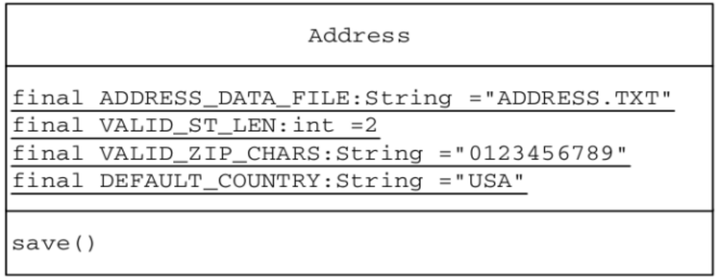
## Constant Data Manager

- Instead of allowing the constant data to be present in different objects, the Constant Data Manager pattern recommends all such data, which is considered as constant in an application, be kept in a separate object and accessed by other objects in the application.
- This type of separation provides an easy to maintain, centralized repository for the constant data in an application.

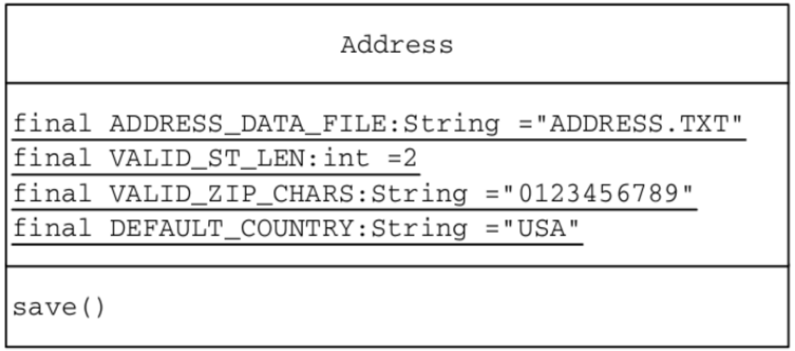
# Constant Data Manager (example)



```
Account
final ACCOUNT_DATA_FILE:String ="ACCOUNT.TXT"
final VALID_MIN_LNAME_LEN:int =2
save()
```

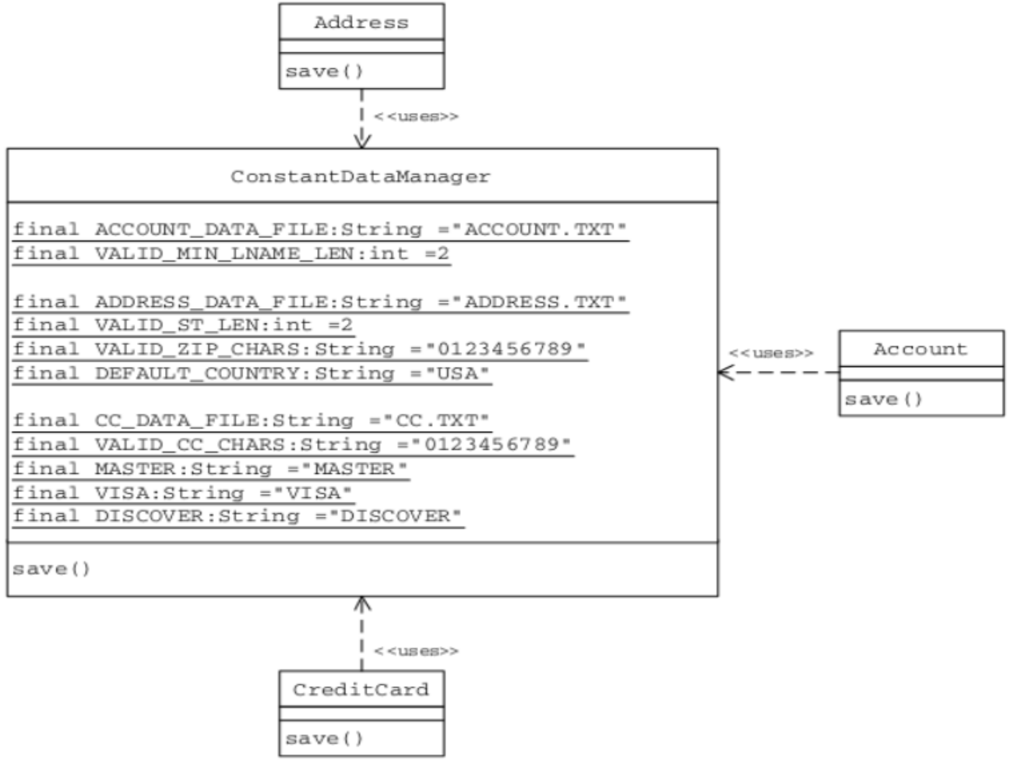


```
Address
final ADDRESS_DATA_FILE:String ="ADDRESS.TXT"
final VALID_ST_LEN:int =2
final VALID_ZIP_CHARS:String ="0123456789"
final DEFAULT_COUNTRY:String ="USA"
save()
```



```
Address
final ADDRESS_DATA_FILE:String ="ADDRESS.TXT"
final VALID_ST_LEN:int =2
final VALID_ZIP_CHARS:String ="0123456789"
final DEFAULT_COUNTRY:String ="USA"
save()
```

# Constant Data Manager (example)



Address
save()
<<uses>>
ConstantDataManager
final ACCOUNT_DATA_FILE:String ="ACCOUNT.TXT"
final VALID_MIN_LNAME_LEN:int =2
final ADDRESS_DATA_FILE:String ="ADDRESS.TXT"
final VALID_ST_LEN:int =2
final VALID_ZIP_CHARS:String ="0123456789"
final DEFAULT_COUNTRY:String ="USA"
final CC_DATA_FILE:String ="CC.TXT"
final VALID_CC_CHARS:String ="0123456789"
final MASTER:String ="MASTER"
final VISA:String ="VISA"
final DISCOVER:String ="DISCOVER"
save()
<<uses>>
Account
save()
<<uses>>
CreditCard
save()

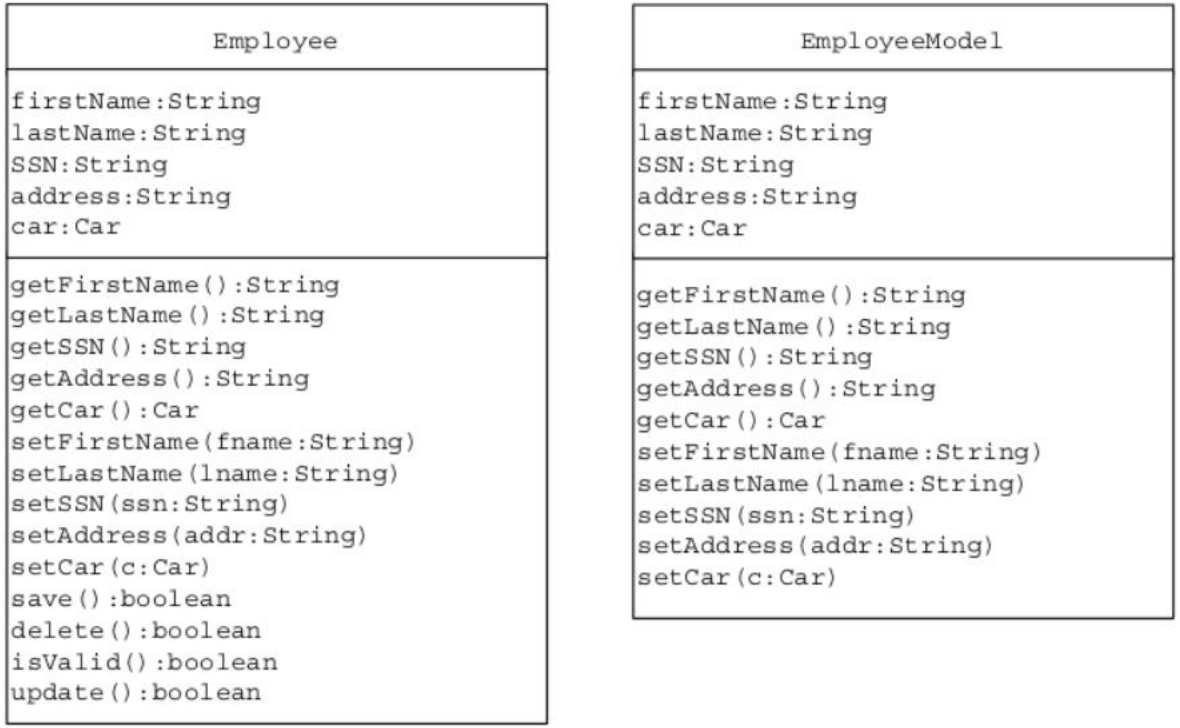
# Immutable Objects

- In general, classes in an application are designed to carry data and have behavior. Sometimes a class may be designed in such a way that its instances can be used just as carriers of related data without any specific behavior. Such classes can be called data model classes and instances of such classes are referred to as data objects.
- The Immutable Object pattern can be used to ensure that the concurrent access to a data object by several client objects does not result in any problem. The Immutable Object pattern accomplishes this without involving the overhead of synchronizing the methods to access the object data.

## Immutable Objects

- Applying the Immutable Object pattern, the data model class can be designed in such a way that the data carried by an instance of the data model class remains unchanged over its entire lifetime. That means the instances of the data model class become immutable.
- In general, concurrent access to an object creates problems when one thread can change data while a different thread is reading the same data. The fact that the data of an immutable object cannot be modified makes it automatically thread-safe and eliminates any concurrent access related problems.
- Though using the Immutable Object pattern opens up an application for all kinds of performance tuning tricks, it must be noted that designing an object as immutable is an important decision. Every now and then it turns out that objects that were once thought of as immutable are in fact mutable, which could result in difficult implementation changes.

# Immutable Objects.. (example)



# Thank You!

*Subhaanakallohumma wa bihamdika, asy-hadu alla ilaha illa anta, as-tagh-firuka wa atuubu ilaik*