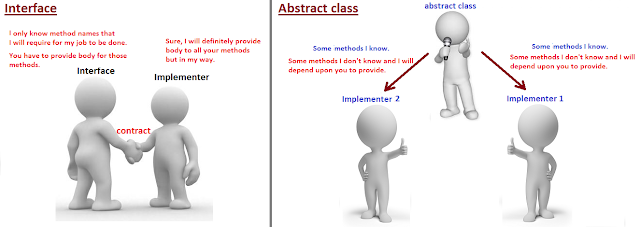
**Real time example of abstract class and interface in Java project.?**

Real time example of abstract class and interface in java projects. abstract class real world example. real time example of Interface in java.



**interface Vs abstract class in Java.**

**Interface:**

*Interface is used when you want to define a contract and you don't know anything about implementation. (here it is total abstraction as you don't know anything.)*

**Abstract class:**

*Abstract class is used when you know something and rely on others for what you don't know.(here it is partial abstraction as some of the things you know and some you don't know.)*

**Now, Let's understand above difference between Interface and Abstract class with real world project example.**

**When to use Interface**

**Scenario,**  
Consider we want to start a service like "[makemytrip.com](http://makemytrip.com/)" or "[expedia.com](http://expedia.com/)",  where we are responsible for displaying the flights from various flight service company and place an order from customer.   
Lets keep our service as simple as,

1. Displaying flights available from vendors like "airasia", "british airways" and "emirates".
2. Place and order for seat to respective vendor.

***How should we design our application considering interfaces and abstract class? In this scenario, interface is useful or abstract class?***  
  
Remember, In this application, we don't own any flight. we are just a middle man/aggregator and our task is to first enquire "airasia", then enquire "british airways" and at last enquire "emirates" about the list of flights available and later if customer opts for booking then inform the respective flight vendor to do booking.  
  
  
For this, first we need to tell "airasia", "british airways" and "emirates" to give us list of flights, internally how they are giving the list that we don't care.

1. **This means I only care for method "getAllAvailableFlights()"**  
     
   "getAllAvailableFlights()" from "airasia" may have used SOAP service to return list of flights.  
   "getAllAvailableFlights()" from "british airways" may have used REST service to return list of flights.  
   "getAllAvailableFlights()" from "emirates" may have used CORBA service to return list of flights.  
     
   but we don't care how it is internally implemented and what we care is the contract method "**getAllAvailableFlights**" that all the flight vendor should provide and return list of flights.
2. Similarly, for booking I only care for method "**booking()**" that all vendors should have, internally how this vendors are doing booking that I don't care.

**To conclude: We know contract.**   
So we can say that we know the contract that irrespective of who the Flight vendor is, we need "**getAllAvailableFlights()**" and "**booking()**" method from them to run our aggregator service.

*In this situation, Interface is useful because we are not aware of the implementation of all the 2 methods required, and what we know is the contract methods that vendor(implementer) should provide. so due to this total abstraction and for defining the contract, interface is useful in this place.*

**Technically, we need to design our interface somewhat like below,**  
  
FlightOpeartions.java(Contract)

[?](http://javabypatel.blogspot.com/2017/07/real-time-example-of-abstract-class-and-interface-in-java.html)

|  |  |
| --- | --- |
| 1  2  3  4 | interface FlightOpeartions{   void getAllAvailableFlights();   void booking(BookingObject bookingObj);  } |

BookingObject.java

[?](http://javabypatel.blogspot.com/2017/07/real-time-example-of-abstract-class-and-interface-in-java.html)

|  |  |
| --- | --- |
| 1 | class BookingObject{} |

BritishAirways.java (Vendor 1)

[?](http://javabypatel.blogspot.com/2017/07/real-time-example-of-abstract-class-and-interface-in-java.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class BritishAirways implements FlightOpeartions{     public void getAllAvailableFlights(){             //get british airways flights in the way             //they told us to fetch flight details.   }     public void booking(BookingObject flightDetails){            //place booking order in a way British airways            //told us to place order for seat.   }    } |

Emirates.java (Vendor 2)

[?](http://javabypatel.blogspot.com/2017/07/real-time-example-of-abstract-class-and-interface-in-java.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Emirates implements FlightOpeartions{     public void getAllAvailableFlights(){           //get Emirates flights in the way           //they told us to fetch flight details.   }     public void booking(BookingObject flightDetails){           //place booking order in a way Emirates airways           //told us to place order for seat.   }  } |

**When to use Abstract class**

**Scenario,**  
Consider we want to start a service like Bulk SMS sender, where we take orders from various telecom vendors like Airtel, France Telecom, Vodafone etc.  
  
For this, we don't have to setup our own infrastructure for sending SMS like Mobile towers but we need to take care of government rules like after 9PM, we should not send promotional SMS, we should also not send SMS to users registered under Do Not Disturb(DND) service etc. Remember, we need to take care of government rules for all the countries where we are sending SMS.  
  
***Note:****for infrastructure like towers, we will be relying on vendor who is going to give us order.*  
*Example, In case of,*  
*Vodafone request us for bulk messaging, in that case we will use Vodafine towers to send SMS.*  
*Airtel request us for bulk messaging, in that case we will use Airtel towers to send SMS.*  
*What our job is to manage Telecom Regulations for different countries where we are sending SMS.*  
    
So what all methods we require would be somewhat like below,

[?](http://javabypatel.blogspot.com/2017/07/real-time-example-of-abstract-class-and-interface-in-java.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public void eastablishConnectionWithYourTower(){     //connect using vendor way.     //we don't know how, candidate for abstract method  }    public void sendSMS(){     eastablishConnectionWithYourTower();     checkForDND();     checkForTelecomRules();     //sending SMS to numbers...numbers.     destroyConnectionWithYourTower()  }    public void destroyConnectionWithYourTower(){     //disconnect using vendor way.     //we don't know how, candidate for abstract method  }    public void checkForDND(){     //check for number present in DND.  }    public void checkForTelecomRules(){     //Check for telecom rules.  } |

Out of above 5 methods,

1. Methods we know is "sendSMS()", "checkForDND()", "checkForTelecomRules()".
2. Methods we don't know is "eastablishConnectionWithYourTower()", "destroyConnectionWithYourTower()".

we know how to check government rules for sending SMS as that is what our job is but  
we don't how to eastablish connection with tower and how to destroy connection with tower because this is purely customer specific, airtel has its own way, vodafone has its own way etc.   
  
*So in the given scenario, we know some methods but there also exist some methods which are unknown and depends on customers.*  
  
In this case, what will be helpful, abstarct class or interface?

***In this case, Abstract class will be helpful, because you know partial things like "checkForDND()", "checkForTelecomRules()" for sending sms to users but we don't know how to eastablishConnectionWithTower() and destroyConnectionWithTower() and need to depend on vendor specific way to connect and destroy connection from their towers.***

Let's see how our class will look like,

[?](http://javabypatel.blogspot.com/2017/07/real-time-example-of-abstract-class-and-interface-in-java.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | abstract class SMSSender{     abstract public void eastablishConnectionWithYourTower();     public void sendSMS(){    /\*eastablishConnectionWithYourTower();    checkForDND();    checkForTelecomRules();      sending SMS to numbers...numbers.\*/   }     abstract public void destroyConnectionWithYourTower();     public void checkForDND(){    //check for number present in DND.   }   public void checkForTelecomRules(){    //Check for telecom rules   }  }      class Vodafone extends SMSSender{     @Override   public void eastablishConnectionWithYourTower() {    //connecting using Vodafone way   }     @Override   public void destroyConnectionWithYourTower() {    //destroying connection using Vodafone way   }    }    class Airtel extends SMSSender{     @Override   public void eastablishConnectionWithYourTower() {    //connecting using Airtel way   }     @Override   public void destroyConnectionWithYourTower() {    //destroying connection using Airtel way   }    } |

**So to summarize,**

**For Interface:**

*Interface is used when you don't know anything about implementation but know the contract that implementer should have to accomplish the task.*

**For Abstract class:**

*Abstract class is used when you know partial implementation, where say out of 5 methods, you know implementation of 3 methods and don't know implemenatation of 2 methods in that case 2 methods will be abstract and you need to rely on implementer as a contract to must provide body of abstract methods to accomplish the task.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

## **Abstract Class**

*An abstract class is a class that is declared****abstract****—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed. An abstract class may have static fields and static methods. When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.*

*An abstract method is a method that is declared without an implementation (without braces and followed by a semicolon), like this:*

abstract void sum(int a, int b);

*Consider using abstract classes if any of these statements apply to your situation:*

*You want to share code among several closely related classes.*

*You expect that classes that extend your abstract class have many common methods or fields or require access modifiers other than public (such as protected and private).*

*You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.*

## **Interface**

*An interface is just the declaration of methods of an Object, it’s not the implementation. In an interface, we define what kind of operation an object can perform. These operations are defined by the classes that implement the interface. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler.*

interface Vehical {

// declaration

void changeGear(int newValue);

void speedUp(int increment);

void applyBrakes(int decrement);

}

class Car implements Vehical {

int speed = 0;

int gear = 1;

// implementation

void changeGear(int newValue) {

gear = newValue;

}

void speedUp(int increment) {

speed = speed + increment;

}

void applyBrakes(int decrement) {

speed = speed - decrement;

}

void printStates() {

System.out.println(" speed:" + speed + " gear:" + gear);

}

}

*Consider using interfaces if any of these statements apply to your situation:*

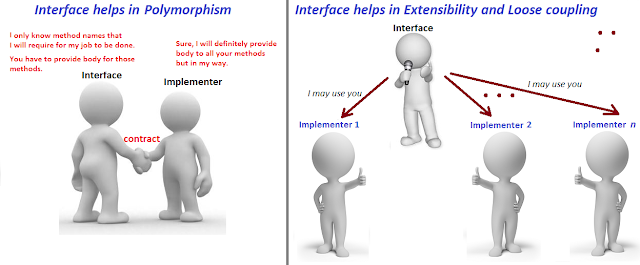
1. *You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.*
2. *You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.*
3. *You want to take advantage of multiple inheritances.*

### When to use Interface in java with example?

Interface is used in a situation,

1. When you know the contract methods but don't know anything about the implementation.
2. Your contract implementation can change in future.
3. You want to achieve dynamic polymorphishm and loose coupling that is by just changing one line of code, you should be able to switch between contract implementer.

**Lets understand above points in more details:**



**Interface in Java.**

**When to use Interface and Abstract class with Real example:**  
<http://javabypatel.blogspot.in/2017/07/when-to-use-abstract-class-and-interface-difference-between-them.html>

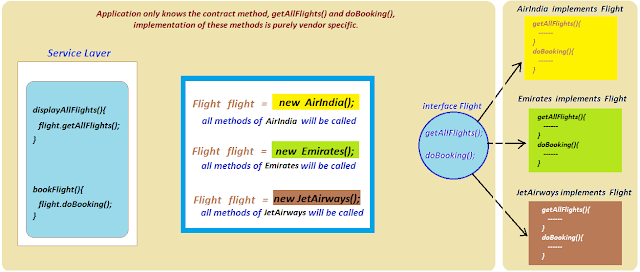
***When you know the contract methods but don't know anything about the implementation***

Lets say we want to start a service like "[makemytrip.com](http://makemytrip.com/)" or "[expedia.com](http://expedia.com/)",  where we are responsible for displaying the flights from various flight service company and place an order from customer.   
Lets keep our service as simple as, 

1. Displaying flights available from vendors like "AirIndia", "Emirates" and "JetAirways".
2. Place and order for seat to respective vendor.

Remember, In this application, we don't own any flight. we are just a middle man/aggregator and our task is to first enquire "AirIndia", then enquire "Emirates" and at last enquire "JetAirways" about the list of flights available and later if customer opts for booking then inform the respective flight vendor to do booking.  
  
  
For this, first we need to tell "AirIndia", "Emirates" and "JetAirways" to give us list of flights, internally how they are giving the list that we don't care.

1. **This means I only care for method name "getAllFlights()"**  
     
   "getAllFlights()" from "AirIndia" may have used SOAP service to return list of flights.  
   "getAllFlights()" from "Emirates" may have used REST service to return list of flights.  
   "getAllFlights()" from "JetAirways" may have used CORBA service to return list of flights.  
     
   but we don't care how it is internally implemented and what we care is the contract method "**getAllFlights**" that all the flight vendor should provide and return list of flights.
2. Similarly, for booking I only care for method name "**doBooking()**" that all vendors should have, internally how this vendors are doing booking that I don't care.

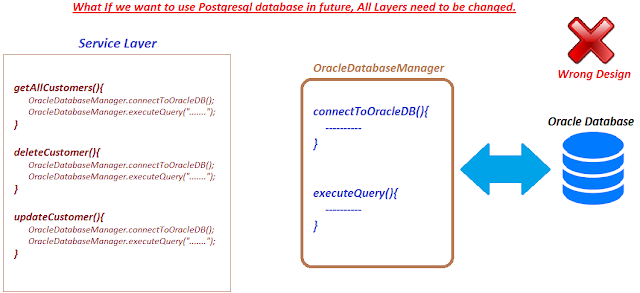


**To conclude: We know contract.** 

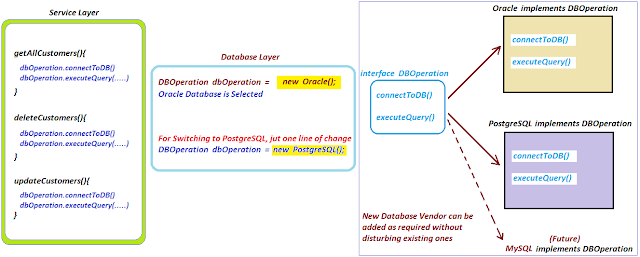
*So we can say that we know the contract that irrespective of who the Flight vendor is, you need "****getAllFlights()****" and "****doBooking()****" method from them to run our aggregator service.*

***Your contract implementation can change in future.***

Say we want to develop some business application which require usage of Database and we choose to use Oracle. So we will design something like below,



We are using Oracle database and all our layers are binded to use OracleDatabaseManager to fetch data using Oracle database.  
  
Now, say tomorrow we plan to change our database to Postgresql, We will end up changing all our layers, which is wrong. If we need to change all our layers, imagine testing and development efforts required.    
  
So how our design should be which can sustain future changes ? Interface will come to rescue.  
Instead of using calling concrete implementation menthods in service layer, we should call contract methods of interface which doesn't know anything about implementation.



In this appraoch, Service layer doesn't care as which implementation we are using.  
So in future if we switch from Oracle to PostgreSQL or to any vendor, Service layer will have no impact and it will remain as it is because it calls interface contract methods which doesn't care about implementation.  
  
Only change required is in Database layer which is also a minimal change.  
Also, this design helps us adding more vendor as an when required without disturbing the existing vendor implementation.  
  
  
***You want to achieve dynamic polymorphishm and loose coupling that is by just changing one line of code, you should be able to switch between contract implementer.***  
  
From above point, you can see, by only changing the middle layer, we are able to switch database implementation to different vendor without disturbing service layer.  
  
So we can say that our application or our service layer is not tightly coupled with database layer instead both layer works on contract which make them loosely coupled and changes on one layer doesn't affect another layer as contract still remains same.

### So to summarize,

### For Interface:

### *Interface is used when you don't know anything about implementation but know the contract that implementer should have to accomplish the task.*

### For Abstract class:

### *Abstract class is used when you know partial implementation, where say out of 5 methods, you know implementation of 3 methods and don't know implemenatation of 2 methods in that case 2 methods will be abstract and you need to rely on implementer as a contract to must provide body of abstract methods to accomplish the task.*

***Other differences between interface and abstract class in Java.***

| **No** | **abstract class** | **interface** |
| --- | --- | --- |
| **1** | Abstract class can have both abstract methods (incomplele. methods without body) and non-abstract methods(complete. methods with body). | Interface can only have abstract methods till Java 7.  In Java 8, Interface can have non-abstract default and static methods. |
| **2** | Abstract class can extends only one class and can implements multiple interfaces. | Interface can only extends other interfaces. |

## Difference between Abstract Class and Interface

1. abstract keyword is used to create an abstract class and it can be used with methods also whereas interface keyword is used to create interface and it can’t be used with methods.
2. Subclasses use extends keyword to extend an abstract class and they need to provide implementation of all the declared methods in the abstract class unless the subclass is also an abstract class whereas subclasses use implements keyword to implement interfaces and should provide implementation for all the methods declared in the interface.
3. Abstract classes can have methods with implementation whereas interface provides absolute abstraction and can’t have any method implementations.
4. Abstract classes can have constructors but interfaces can’t have constructors.
5. Abstract class have all the features of a normal java class except that we can’t instantiate it. We can use abstract keyword to make a class abstract but interfaces are a completely different type and can have only public static final constants and method declarations.
6. Abstract classes methods can have access modifiers as public, private, protected, static but interface methods are implicitly public and abstract, we can’t use any other access modifiers with interface methods.
7. A subclass can extend only one abstract class but it can implement multiple interfaces.
8. Abstract classes can extend other class and implement interfaces but interface can only extend other interfaces.
9. We can run an abstract class if it has main() method but we can’t run an interface because they can’t have main method implementation.
10. Interfaces are used to define contract for the subclasses whereas abstract class also define contract but it can provide other methods implementations for subclasses to use.

Thats all for the ***difference between interface and abstract classes***, now we can move on to know when should we use Interface over Abstract class and vice versa.

### Interface or Abstract Class

Whether to chose between Interface or abstract class for providing contract for subclasses is a design decision and depends on many factors, lets see when Interfaces are best choice and when can we use abstract classes.

1. Java doesn’t support multiple class level inheritance, so every class can extend only one superclass. But a class can implement multiple interfaces. So most of the times Interfaces are a good choice for providing base for class hierarchy and contract. Also coding in terms of interfaces is one of the best practices for coding in java.
2. If there are a lot of methods in the contract, then abstract class is more useful because we can provide default implementation for some of the methods that are common for all the subclasses. Also if subclasses don’t need to implement particular method, they can avoid providing the implementation but in case of interface, the subclass will have to provide implementation for all the methods even though it’s of no use and implementation is just empty block.
3. If our base contract keeps on changing then interfaces can cause issues because we can’t declare additional methods to the interface without changing all the implementation classes, with abstract class we can provide the default implementation and only change the implementation classes that are actually going to use the new methods.

### Use Abstract classes and Interface both

Actually most of the times, using Interfaces and abstract classes together is the best approach for designing a system, for example in JDK java.util.List is an interface that contains a lot of methods, so there is an abstract class java.util.AbstractList that provides skeletal implementation for all the methods of List interface so that any subclass can extend this class and implement only required methods.

We should always start with an interface as base and define methods that every subclasses should implement and then if there are some methods that only certain subclass should implement, we can extend the base interface and create a new interface with those methods. The subclasses will have option to chose between the base interface or the child interface to implement according to its requirements. If the number of methods grows a lot, its not a bad idea to provide a skeletal abstract class implementing the child interface and providing flexibility to the subclasses to chose between interface and abstract class.

### Java 8 interface changes

From Java 8 onwards, we can have method implementations in the interfaces. We can create default as well as static methods in the interfaces and provide implementation for them. This has bridge the gap between abstract classes and interfaces and now interfaces are the way to go because we can extend it further by providing default implementations for new methods. For more details, check out Java 8 interface default static methods.