Design Patterns in Java

A design patterns are **well-proved solution** for solving the specific problem/task.

Now, a question will be arising in your mind what kind of specific problem? Let me explain by taking an example.

**ProblemGiven:**  
Suppose you want to create a class for which only a single instance (or object) should be created and that single object can be used by all other classes.

**Solution:**  
**Singleton design pattern** is the best solution of above specific problem. So, every design pattern has **some specification or set of rules** for solving the problems

But remember one-thing, design patterns are programming language independent strategies for solving the common object-oriented design problems. That means, a design pattern represents an idea, not a particular implementation.

By using the design patterns you can make your code more flexible, reusable and maintainable. It is the most important part because java internally follows design patterns.

To become a professional software developer, you must know at least some popular solutions (i.e. design patterns) to the coding problems.

Advantage of design pattern:

1. They are reusable in multiple projects.
2. They provide the solutions that help to define the system architecture.
3. They capture the software engineering experiences.
4. They provide transparency to the design of an application.
5. They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
6. Design patterns don?t guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

When should we use the design patterns?

We must use the design patterns **during the analysis and requirement phase of SDLC**(Software Development Life Cycle).

Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

Categorization of design patterns:

Basically, design patterns are categorized into two parts:

1. Core Java (or JSE) Design Patterns.
2. JEE Design Patterns.

Core Java Design Patterns

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

1.Creational Design Pattern

1. Factory Pattern
2. Abstract Factory Pattern
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern.

2. Structural Design Pattern

1. Adapter Pattern
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

3. Behavioral Design Pattern

1. Chain Of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern

Singleton design pattern

Singleton Pattern says that just**"define a class that has only one instance and provides a global point of access to it".**

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes

To create the singleton class, we need to have static member of class, private constructor and static factory method.

* **Static member:** It gets memory only once because of static, itcontains the instance of the Singleton class.
* **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
* **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

Understanding early Instantiation of Singleton Pattern

In such case, we create the instance of the class at the time of declaring the static data member, so instance of the class is created at the time of classloading.

Let's see the example of singleton design pattern using early instantiation.

*File: A.java*

**class** A{

**private** **static** A obj=**new** A();//Early, instance will be created at load time

**private** A(){}

**public** **static** A getA(){

**return** obj;

 }

**public** **void** doSomething(){

 //write your code

 }

}

# Factory Method Pattern

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as **Virtual Constructor.**

#### **Advantage of Factory Design Pattern**

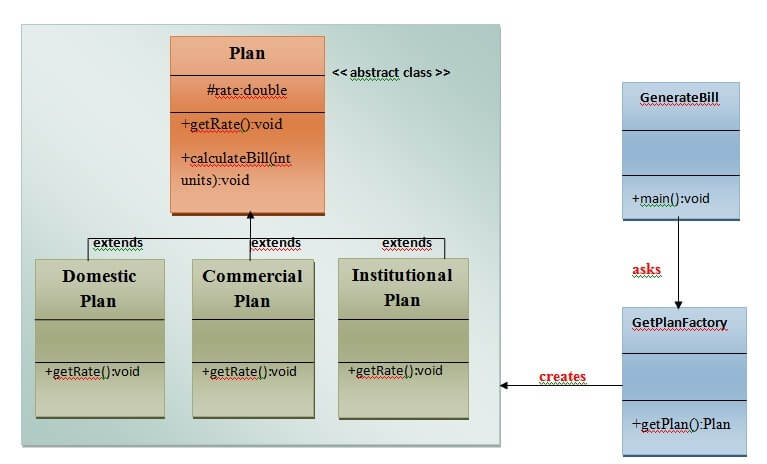
* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

#### **Usage of Factory Design Pattern**

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.

#### **UML for Factory Method Pattern**

* We are going to create a Plan abstract class and concrete classes that extends the Plan abstract class. A factory class GetPlanFactory is defined as a next step.
* GenerateBill class will use GetPlanFactory to get a Plan object. It will pass information (DOMESTICPLAN / COMMERCIALPLAN / INSTITUTIONALPLAN) to GetPalnFactory to get the type of object it needs.



### Calculate Electricity Bill : A Real World Example of Factory Method

**Step 1:**Create a Plan abstract class.

**import** java.io.\*;

**abstract** **class** Plan{

**protected** **double** rate;

**abstract** **void** getRate();

**public** **void** calculateBill(**int** units){

              System.out.println(units\*rate);

          }

}//end of Plan class.

**Step 2:**Create the concrete classes that extends Plan abstract class.

**class**  DomesticPlan **extends** Plan{

        //@override

**public** **void** getRate(){

             rate=3.50;

        }

   }//end of DomesticPlan class.

**class**  CommercialPlan **extends** Plan{

   //@override

**public** **void** getRate(){

        rate=7.50;

   }

//end of CommercialPlan **class**.

**class**  InstitutionalPlan **extends** Plan{

   //@override

**public** **void** getRate(){

        rate=5.50;

   }

//end of InstitutionalPlan **class**.

**Step 3:**Create a GetPlanFactory to generate object of concrete classes based on given information..

**class** GetPlanFactory{

   //use getPlan method to get object of type Plan

**public** Plan getPlan(String planType){

**if**(planType == **null**){

**return** **null**;

            }

**if**(planType.equalsIgnoreCase("DOMESTICPLAN")) {

**return** **new** DomesticPlan();

               }

**else** **if**(planType.equalsIgnoreCase("COMMERCIALPLAN")){

**return** **new** CommercialPlan();

            }

**else** **if**(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {

**return** **new** InstitutionalPlan();

          }

**return** **null**;

   }

}//end of GetPlanFactory class.

**Step 4:**Generate Bill by using the GetPlanFactory to get the object of concrete classes by passing an information such as type of plan DOMESTICPLAN or COMMERCIALPLAN or INSTITUTIONALPLAN.

**import** java.io.\*;

**class** GenerateBill{

**public** **static** **void** main(String args[])**throws** IOException{

      GetPlanFactory planFactory = **new** GetPlanFactory();

      System.out.print("Enter the name of plan for which the bill will be generated: ");

      BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));

      String planName=br.readLine();

      System.out.print("Enter the number of units for bill will be calculated: ");

**int** units=Integer.parseInt(br.readLine());

      Plan p = planFactory.getPlan(planName);

      //call getRate() method and calculateBill()method of DomesticPaln.

       System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");

           p.getRate();

           p.calculateBill(units);

            }

    }//end of GenerateBill class.

Abstract Factory Pattern

Abstract Factory Pattern says that just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.**That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

An Abstract Factory Pattern is also known as **Kit.**

Advantage of Abstract Factory Pattern

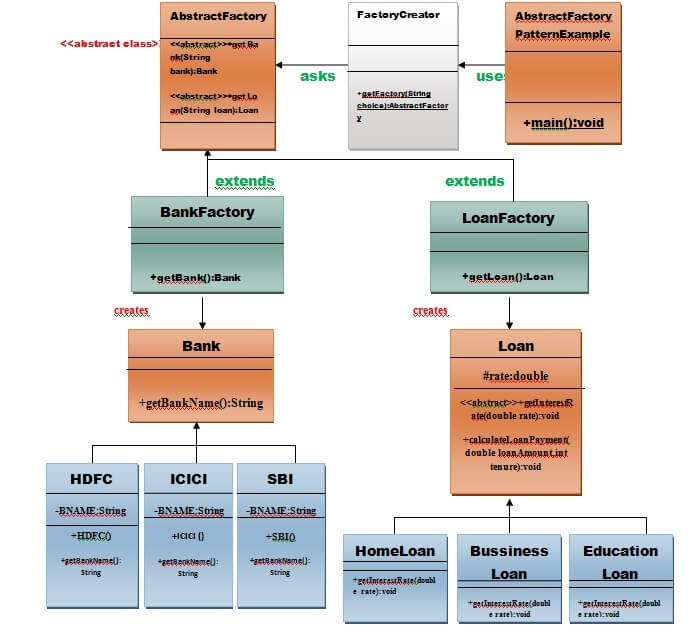
* Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
* It eases the exchanging of object families.
* It promotes consistency among objects.

Usage of Abstract Factory Pattern

* When the system needs to be independent of how its object are created, composed, and represented.
* When the family of related objects has to be used together, then this constraint needs to be enforced.
* When you want to provide a library of objects that does not show implementations and only reveals interfaces.
* When the system needs to be configured with one of a multiple family of objects.

UML for Abstract Factory Pattern

* We are going to create a **Bank interface** and a **Loan abstract class** as well as their sub-classes.
* Then we will create **AbstractFactory** class as next step.
* Then after we will create concrete classes, **BankFactory,** and **LoanFactory** that will extends **AbstractFactory class**
* After that, **AbstractFactoryPatternExample** class uses the **FactoryCreator** to get an object of **AbstractFactory** class.
* See the diagram carefully which is given below:



Example of Abstract Factory Pattern

Here, we are calculating the loan payment for different banks like HDFC, ICICI, SBI etc.

**Step 1:** Create a Bank interface

**import** java.io.\*;

**interface** Bank{

        String getBankName();

}

**Step 2:** Create concrete classes that implement the Bank interface.

**class** HDFC **implements** Bank{

**private** **final** String BNAME;

**public** HDFC(){

                BNAME="HDFC BANK";

        }

**public** String getBankName() {

**return** BNAME;

        }

}

**class** ICICI **implements** Bank{

**private** **final** String BNAME;

       ICICI(){

                BNAME="ICICI BANK";

        }

**public** String getBankName() {

**return** BNAME;

       }

}

**class** SBI **implements** Bank{

**private** **final** String BNAME;

**public** SBI(){

                BNAME="SBI BANK";

        }

**public** String getBankName(){

**return** BNAME;

       }

}

**Step 3:** Create the Loan abstract class.

**abstract** **class** Loan{

**protected** **double** rate;

**abstract** **void** getInterestRate(**double** rate);

**public** **void** calculateLoanPayment(**double** loanamount, **int** years)

   {

        /\*

              to calculate the monthly loan payment i.e. EMI

              rate=annual interest rate/12\*100;

              n=number of monthly installments;

              1year=12 months.

              so, n=years\*12;

            \*/

**double** EMI;

**int** n;

         n=years\*12;

         rate=rate/1200;

         EMI=((rate\*Math.pow((1+rate),n))/((Math.pow((1+rate),n))-1))\*loanamount;

System.out.println("your monthly EMI is "+ EMI +" for the amount"+loanamount+" you have borrowed");

 }

}// end of the Loan abstract class.

**Step 4:** Create concrete classes that extend the Loan abstract class..

**class** HomeLoan **extends** Loan{

**public** **void** getInterestRate(**double** r){

         rate=r;

    }

}//End of the HomeLoan class.

**class** BussinessLoan **extends** Loan{

**public** **void** getInterestRate(**double** r){

          rate=r;

     }

}//End of the BusssinessLoan class.

**class** EducationLoan **extends** Loan{

**public** **void** getInterestRate(**double** r){

       rate=r;

 }

}//End of the EducationLoan class.

**Step 5:** Create an abstract class (i.e AbstractFactory) to get the factories for Bank and Loan Objects.

**abstract** **class** AbstractFactory{

**public** **abstract** Bank getBank(String bank);

**public** **abstract** Loan getLoan(String loan);

}

**Step 6:** Create the factory classes that inherit AbstractFactory class to generate the object of concrete class based on given information.

**class** BankFactory **extends** AbstractFactory{

**public** Bank getBank(String bank){

**if**(bank == **null**){

**return** **null**;

      }

**if**(bank.equalsIgnoreCase("HDFC")){

**return** **new** HDFC();

      } **else** **if**(bank.equalsIgnoreCase("ICICI")){

**return** **new** ICICI();

      } **else** **if**(bank.equalsIgnoreCase("SBI")){

**return** **new** SBI();

      }

**return** **null**;

   }

**public** Loan getLoan(String loan) {

**return** **null**;

   }

}//End of the BankFactory class.

**class** LoanFactory **extends** AbstractFactory{

**public** Bank getBank(String bank){

**return** **null**;

          }

**public** Loan getLoan(String loan){

**if**(loan == **null**){

**return** **null**;

      }

**if**(loan.equalsIgnoreCase("Home")){

**return** **new** HomeLoan();

      } **else** **if**(loan.equalsIgnoreCase("Business")){

**return** **new** BussinessLoan();

      } **else** **if**(loan.equalsIgnoreCase("Education")){

**return** **new** EducationLoan();

      }

**return** **null**;

   }

}

**Step 7:** Create a FactoryCreator class to get the factories by passing an information such as Bank or Loan.

**class** FactoryCreator {

**public** **static** AbstractFactory getFactory(String choice){

**if**(choice.equalsIgnoreCase("Bank")){

**return** **new** BankFactory();

      } **else** **if**(choice.equalsIgnoreCase("Loan")){

**return** **new** LoanFactory();

      }

**return** **null**;

   }

}//End of the FactoryCreator.

**Step 8:** Use the FactoryCreator to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

**import** java.io.\*;

**class** AbstractFactoryPatternExample {

**public** **static** **void** main(String args[])**throws** IOException {

      BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.in));

      System.out.print("Enter the name of Bank from where you want to take loan amount: ");

      String bankName=br.readLine();

System.out.print("\n");

System.out.print("Enter the type of loan e.g. home loan or business loan or education loan : ");

String loanName=br.readLine();

AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");

Bank b=bankFactory.getBank(bankName);

System.out.print("\n");

System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");

**double** rate=Double.parseDouble(br.readLine());

System.out.print("\n");

System.out.print("Enter the loan amount you want to take: ");

**double** loanAmount=Double.parseDouble(br.readLine());

System.out.print("\n");

System.out.print("Enter the number of years to pay your entire loan amount: ");

**int** years=Integer.parseInt(br.readLine());

System.out.print("\n");

System.out.println("you are taking the loan from "+ b.getBankName());

AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");

           Loan l=loanFactory.getLoan(loanName);

           l.getInterestRate(rate);

           l.calculateLoanPayment(loanAmount,years);

  }

}//End of the  AbstractFactoryPatternExample

**Output**

Enter the name of Bank from where you want to take loan amount: hdfc

Enter the type of loan e.g. home loan or business loan or education loan : business

Enter the interest rate for HDFC BANK: 12.95

Enter the loan amount you want to take: 5000

Enter the number of years to pay your entire loan amount: 10

you are taking the loan from HDFC BANK

your monthly EMI is 74.50798631159589 for the amount5000.0 you have borrowed

Builder Design Pattern

Builder Pattern says that **"construct a complex object from simple objects using step-by-step approach"**

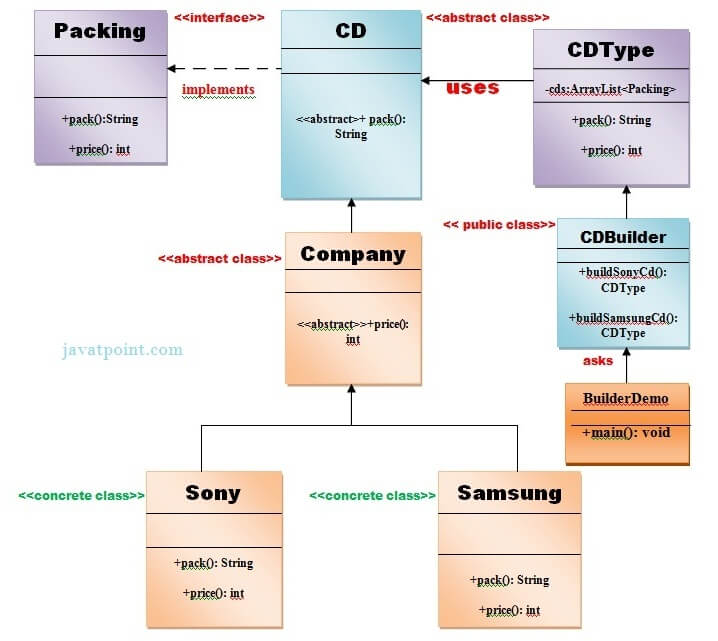
It is mostly used when object can't be created in single step like in the de-serialization of a complex object.

Advantage of Builder Design Pattern

The main advantages of Builder Pattern are as follows:

* It provides clear separation between the construction and representation of an object.
* It provides better control over construction process.
* It supports to change the internal representation of objects.

UML for Builder Pattern Example



Example of Builder Design Pattern

To create simple example of builder design pattern, you need to follow 6 following steps. Create Packing interface

1. Create 2 abstract classes CD and Company
2. Create 2 implementation classes of Company: Sony and Samsung
3. Create the CDType class
4. Create the CDBuilder class
5. Create the BuilderDemo class

1) Create Packing interface

*File: Packing.java*

**public** **interface** Packing {

**public** String pack();

**public** **int** price();

}

2) Create 2 abstract classes CD and Company

Create an abstract class CD which will implement Packing interface.

*File: CD.java*

**public** **abstract** **class** CD **implements** Packing{

**public** **abstract** String pack();

}

*File: Company.java*

**public** **abstract** **class** Company **extends** CD{

**public** **abstract** **int** price();

}

3) Create 2 implementation classes of Company: Sony and Samsung

*File: Sony.java*

**public** **class** Sony **extends** Company{

    @Override

**public** **int** price(){

**return** 20;

      }

    @Override

**public** String pack(){

**return** "Sony CD";

        }

}//End of the Sony class.

*File: Samsung.java*

**public** **class** Samsung **extends** Company {

    @Override

**public** **int** price(){

**return** 15;

    }

    @Override

**public** String pack(){

**return** "Samsung CD";

        }

}//End of the Samsung class.

4) Create the CDType class

*File: CDType.java*

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** CDType {

**private** List<Packing> items=**new** ArrayList<Packing>();

**public** **void** addItem(Packing packs) {

                    items.add(packs);

             }

**public** **void** getCost(){

**for** (Packing packs : items) {

                        packs.price();

              }

             }

**public** **void** showItems(){

**for** (Packing packing : items){

             System.out.print("CD name : "+packing.pack());

             System.out.println(", Price : "+packing.price());

          }

            }

}//End of the CDType class.

5) Create the CDBuilder class

*File: CDBuilder.java*

**public** **class** CDBuilder {

**public** CDType buildSonyCD(){

                     CDType cds=**new** CDType();

                     cds.addItem(**new** Sony());

**return** cds;

              }

**public** CDType buildSamsungCD(){

             CDType cds=**new** CDType();

             cds.addItem(**new** Samsung());

**return** cds;

              }

}// End of the CDBuilder class.

6) Create the BuilderDemo class

*File: BuilderDemo.java*

**public** **class** BuilderDemo{

**public** **static** **void** main(String args[]){

   CDBuilder cdBuilder=**new** CDBuilder();

   CDType cdType1=cdBuilder.buildSonyCD();

   cdType1.showItems();

   CDType cdType2=cdBuilder.buildSamsungCD();

   cdType2.showItems();

 }

}

Output

CD name : Sony CD, Price : 20

CD name : Samsung CD, Price : 15

# Template Pattern

A Template Pattern says that "just define the skeleton of a function in an operation, deferring some steps to its subclasses".

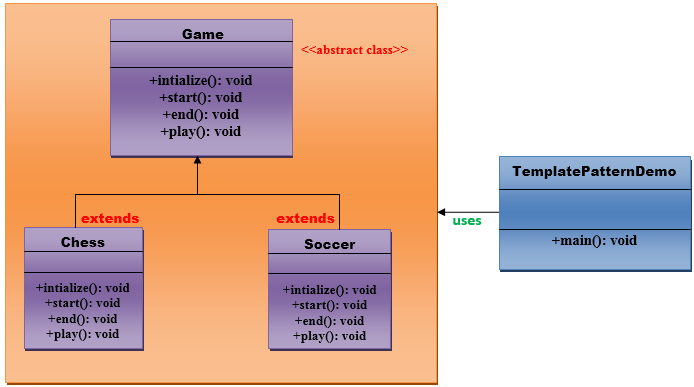
## Benefits:

* It is very common technique for reusing the code. This is only the main benefit of it.

## Usage:

* It is used when the common behavior among sub-classes should be moved to a single common class by avoiding the duplication.

## UML for Template Pattern:



## Implementation of Template Pattern:

**Step 1:**

Create a Game *abstract* class.

//This is an abstract class.

**public** **abstract** **class** Game {

**abstract** **void** initialize();

**abstract** **void** start();

**abstract** **void** end();

**public** **final** **void** play(){

          //initialize the game

          initialize();

          //start game

          start();

          //end game

          end();

       }

}// End of the Game abstract class.

**Step 2:**

Create a *Chess* class that will extend Game abstract class for giving the definition to its method.

//This is a class.

**public** **class** Chess **extends** Game {

     @Override

**void** initialize() {

          System.out.println("Chess Game Initialized! Start playing.");

       }

     @Override

**void** start() {

          System.out.println("Game Started. Welcome to in the chess game!");

       }

    @Override

**void** end() {

          System.out.println("Game Finished!");

       }

}// End of the Chess class.

**Step 3:**

Create a *Soccer* class that will extend Game abstract class for giving the definition to its method.

//This is a class.

**public** **class** Soccer **extends** Game {

    @Override

**void** initialize() {

          System.out.println("Soccer Game Initialized! Start playing.");

       }

    @Override

**void** start() {

          System.out.println("Game Started. Welcome to in the Soccer game!");

       }

    @Override

**void** end() {

          System.out.println("Game Finished!");

       }

}// End of the Soccer class.

**Step 4:**

Create a *TemplatePatternDemo* class.

//This is a class.

**public** **class** TemplatePatternDemo {

**public** **static** **void** main(String[] args) **throws** InstantiationException, IllegalAccessException, ClassNotFoundException {

         Class c=Class.forName(args[0]);

         Game game=(Game) c.newInstance();

         game.play();

       }

}// End of the Soccer class.

**Output:**

Bridge Pattern

A Bridge Pattern says that just **"decouple the functional abstraction from the implementation so that the two can vary independently".**

The Bridge Pattern is also known as **Handle or Body.**

Advantage of Bridge Pattern

* It enables the separation of implementation from the interface.
* It improves the extensibility.
* It allows the hiding of implementation details from the client.

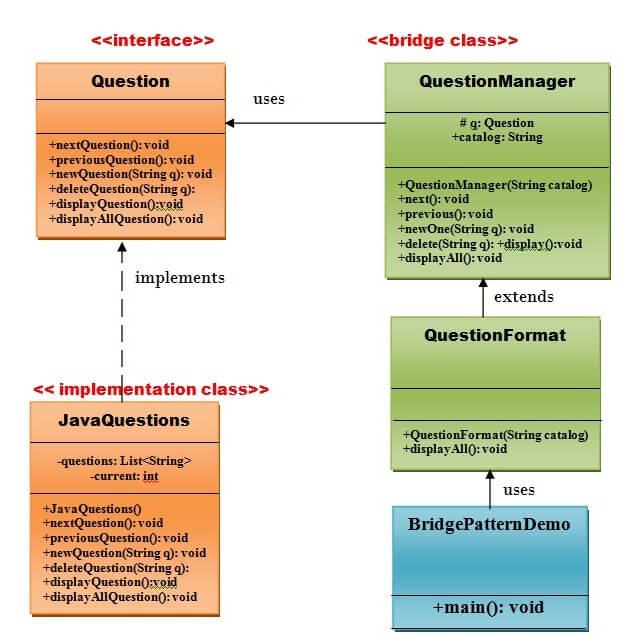
Usage of Bridge Pattern

* When you don't want a permanent binding between the functional abstraction and its implementation.
* When both the functional abstraction and its implementation need to extended using sub-classes.
* It is mostly used in those places where changes are made in the implementation does not affect the clients.

Example of Bridge Pattern

The UML given below describes the example of bridge pattern.

UML for Bridge Pattern:



Implementation of above UML:

Step 1

Create a **Question** interface that provides the navigation from one question to another or vice-versa.

// this is the Question interface.

**public** **interface** Question {

**public** **void** nextQuestion();

**public** **void** previousQuestion();

**public** **void** newQuestion(String q);

**public** **void** deleteQuestion(String q);

**public** **void** displayQuestion();

**public** **void** displayAllQuestions();

}

// End of the Question interface.

Step 2

Create a **JavaQuestions** implementation class that will implement **Question** interface.

// this is the JavaQuestions class.

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** JavaQuestions **implements** Question {

**private** List <String> questions = **new** ArrayList<String>();

**private** **int** current = 0;

**public** JavaQuestions(){

    questions.add("What is class? ");

    questions.add("What is interface? ");

    questions.add("What is abstraction? ");

    questions.add("How multiple polymorphism is achieved in java? ");

    questions.add("How many types of exception  handling are there in java? ");

    questions.add("Define the keyword final for  variable, method, and class in java? ");

    questions.add("What is abstract class? ");

    questions.add("What is multi-threading? ");

 }

**public** **void** nextQuestion() {

**if**( current <= questions.size()-1 )

    current++;

    System.out.print(current);

 }

**public** **void** previousQuestion() {

**if**( current > 0 )

    current--;

 }

**public** **void** newQuestion(String quest) {

    questions.add(quest);

 }

**public** **void** deleteQuestion(String quest) {

    questions.remove(quest);

 }

**public** **void** displayQuestion() {

    System.out.println( questions.get(current) );

 }

**public** **void** displayAllQuestions() {

**for** (String quest : questions) {

    System.out.println(quest);

 }

}

}// End of the JavaQuestions class.

Step 3

Create a **QuestionManager** class that will use **Question** interface which will act as a bridge

// this is the QuestionManager class.

**public** **class** QuestionManager  {

**protected** Question q;

**public** String catalog;

**public** QuestionManager(String catalog) {

**this**.catalog=catalog;

    }

**public** **void** next() {

    q.nextQuestion();

    }

**public** **void** previous() {

    q.previousQuestion();

    }

**public** **void** newOne(String quest) {

    q.newQuestion(quest);

    }

**public** **void** delete(String quest) {

    q.deleteQuestion(quest);

    }

**public** **void** display() {

    q.displayQuestion();

    }

**public** **void** displayAll() {

        System.out.println("Question Paper: " + catalog);

    q.displayAllQuestions();

    }

}// End of the QuestionManager class.

Step 4

Create a **QuestionFormat** class that will extend the **QuestionManager** class

// this is the QuestionFormat class.

**public** **class** QuestionFormat **extends** QuestionManager {

**public** QuestionFormat(String catalog){

**super**(catalog);

    }

**public** **void** displayAll() {

        System.out.println("\n---------------------------------------------------------");

**super**.displayAll();

        System.out.println("-----------------------------------------------------------");

    }

}// End of the QuestionFormat class.

Step 5

Create a **BridgePatternDemo** class.

// this is the BridgePatternDemo class.

**public** **class** BridgePatternDemo {

**public** **static** **void** main(String[] args) {

    QuestionFormat questions = **new** QuestionFormat("Java Programming Language");

    questions.q = **new** JavaQuestions();

        questions.delete("what is class?");

        questions.display();

    questions.newOne("What is inheritance? ");

    questions.newOne("How many types of inheritance are there in java?");

    questions.displayAll();

    }

}// End of the BridgePatternDemo class.

Output

What is **interface**?

--------------------------------------------------------------------

Question Paper: Java Programming Language

What is **class**?

What is **interface**?

What is abstraction?

How multiple polymorphism is achieved in java?

How many types of exception  handling are there in java?

Define the keyword **final** **for**  variable, method, and **class** in java?

What is **abstract** **class**?

What is multi-threading?

What is inheritance?

How many types of inheritance are there in java?

-----------------------------------------------------------------------

Proxy Pattern

Simply, proxy means an object representing another object.

According to GoF, a Proxy Pattern **"provides the control for accessing the original object".**

So, we can perform many operations like hiding the information of original object, on demand loading etc.

Proxy pattern is also known as **Surrogate or Placeholder.**

RMI API uses proxy design pattern. Stub and Skeleton are two proxy objects used in RMI.

Advantage of Proxy Pattern

* It provides the protection to the original object from the outside world.

Usage of Proxy Pattern:

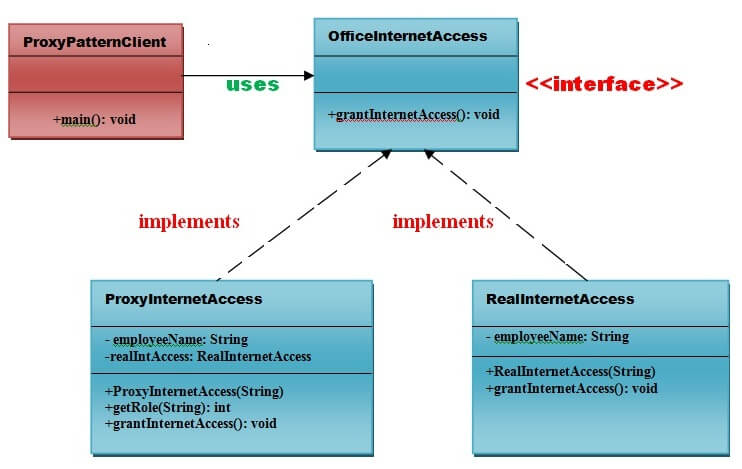
It is used:

* It can be used in **Virtual Proxy** scenario---Consider a situation where there is multiple database call to extract huge size image. Since this is an expensive operation so here we can use the proxy pattern which would create multiple proxies and point to the huge size memory consuming object for further processing. The real object gets created only when a client first requests/accesses the object and after that we can just refer to the proxy to reuse the object. This avoids duplication of the object and hence saving memory.
* It can be used in **Protective Proxy** scenario---It acts as an authorization layer to verify that whether the actual user has access the appropriate content or not. For example, a proxy server which provides restriction on internet access in office. Only the websites and contents which are valid will be allowed and the remaining ones will be blocked.
* It can be used in **Remote Proxy** scenario---A remote proxy can be thought about the stub in the RPC call. The remote proxy provides a local representation of the object which is present in the different address location. Another example can be providing interface for remote resources such as web service or REST resources.
* It can be used in **Smart Proxy** scenario---A smart proxy provides additional layer of security by interposing specific actions when the object is accessed. For example, to check whether the real object is locked or not before accessing it so that no other objects can change it.

Example of Proxy Pattern

Let's understand the example of proxy design pattern by the above UML diagram.

UML for Proxy Pattern:



Implementation of above UML:

Step 1

Create an **OfficeInternetAccess** interface.

**public** **interface** OfficeInternetAccess {

**public** **void** grantInternetAccess();

}

Step 2

Create a **RealInternetAccess** class that will implement **OfficeInternetAccess** interface for granting the permission to the specific employee.

*File: RealInternetAccess.java*

**public** **class** RealInternetAccess **implements** OfficeInternetAccess {

**private** String employeeName;

**public** RealInternetAccess(String empName) {

**this**.employeeName = empName;

    }

    @Override

**public** **void** grantInternetAccess() {

        System.out.println("Internet Access granted for employee: "+ employeeName);

    }

}

Step 3

Create a **ProxyInternetAccess** class that will implement **OfficeInternetAccess** interface for providing the object of **RealInternetAccess** class.

*File: ProxyInternetAccess.java*

**public** **class** ProxyInternetAccess **implements** OfficeInternetAccess {

**private** String employeeName;

**private** RealInternetAccess  realaccess;

**public** ProxyInternetAccess(String employeeName) {

**this**.employeeName = employeeName;

        }

        @Override

**public** **void** grantInternetAccess()

        {

**if** (getRole(employeeName) > 4)

            {

                realaccess = **new** RealInternetAccess(employeeName);

                realaccess.grantInternetAccess();

            }

**else**

            {

                System.out.println("No Internet access granted. Your job level is below 5");

            }

        }

**public** **int** getRole(String emplName) {

            // Check role from the database based on Name and designation

            // return job level or job designation.

**return** 9;

        }

}

Step 4

Now, Create a **ProxyPatternClient** class that can access the internet actually.

*File: ProxyPatternClient.java*

**public** **class** ProxyPatternClient {

**public** **static** **void** main(String[] args)

    {

        OfficeInternetAccess access = **new** ProxyInternetAccess("Ashwani Rajput");

        access.grantInternetAccess();

    }

}

Output

No Internet access granted. Your job level is below 5

# How to create Immutable class?

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable. We can also create immutable class by creating final class that have final data members as the example given below:

### Example to create Immutable class

In this example, we have created a final class named Employee. It have one final datamember, a parameterized constructor and getter method.

**ImmutableDemo.java**

**public** **final** **class** Employee

{

**Private final** String pancardNumber;

**public** Employee(String pancardNumber)

{

**this**.pancardNumber=pancardNumber;

}

**public** String getPancardNumber(){

**return** pancardNumber;

}

}

**public** **class** ImmutableDemo

{

**public** **static** **void** main(String ar[])

{

Employee e = **new** Employee("ABC123");

String s1 = e.getPancardNumber();

System.out.println("Pancard Number: " + s1);

}

}

**Output:**

Pancard Number: ABC123

The above class is immutable because:

* The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
* The class is final so we cannot create the subclass.
* There is no setter methods i.e. we have no option to change the value of the instance variable.

These points makes this class as immutable.