Design Patterns

<http://java67.blogspot.sg/2012/09/top-10-java-design-pattern-interview-question-answer.html>

Top 18 Java Design Pattern Interview Questions Answers for Experienced

**Design pattern interview question in Java**

design pattern interview question are integral part of any good list of core Java interview questions. Java is a popular Object oriented programming language and have lots of design pattern and design principles, contributed by many developers and open source framework. As a Java programmer its expected from you to know OOPS concept like [Abstraction](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html), [Encapsulation](http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html) and [polymorphism](http://javarevisited.blogspot.sg/2011/08/what-is-polymorphism-in-java-example.html), What is design pattern in Java, Some popular Java design pattern and most importantly when to use those design pattern in Java application. purpose of asking design pattern interview question in Java is to check whether Java programmer are familiar to those essential design patterns or not. Design patterns in Java interviews are as important as [multi-threading](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html), [collection](http://javarevisited.blogspot.sg/2011/11/collection-interview-questions-answers.html) and [programming questions](http://javarevisited.blogspot.sg/2011/06/top-programming-interview-questions.html). If you are senior or experienced Java programmer than expect more complex and tough design pattern in Java interview e.g. Chain of responsibility design pattern and solving real time [software design questions](http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html).

### Top Java design pattern questions and answers

Here is my list of t*op 10 design pattern interview question in Java*. I have also provided answer of those Java design pattern question as link. no matter which level of Java interview are you going e.g. programmer, software engineer, senior software engineer in Java, you can expect few question from Java design pattern.

**1. When to use Strategy Design Pattern in Java?**

Strategy pattern in quite useful for implementing set of related algorithms e.g. compression algorithms, filtering strategies etc. Strategy design pattern allows you to create Context classes, which uses Strategy implementation classes for applying business rules. This pattern follow open closed design principle and quite useful in Java. One example of Strategy pattern from JDK itself is a Collections.sort() method and [Comparator interface](http://java67.blogspot.sg/2012/10/how-to-sort-object-in-java-comparator-comparable-example.html), which is a strategy interface and defines strategy for comparing objects. Because of this pattern, we don't need to modify sort() method (closed for modification) to compare any object, at same time we can implement Comparator interface to define new comparing strategy (open for extension).

**2. What is Observer design pattern in Java? When do you use Observer pattern in Java?**

This is one of the most common Java design pattern interview question. Observer pattern is based upon notification, there are two kinds of object Subject and Observer. Whenever there is change on subject's state observer will receive notification. See [What is Observer design pattern in Java with real life example](http://javarevisited.blogspot.sg/2011/12/observer-design-pattern-java-example.html) for more details.

**3. Difference between Strategy and State design Pattern in Java?**

This is an interesting Java design pattern interview questions as both Strategy and State pattern has same structure. If you look at UML class diagram for both pattern they look exactly same, but there intent is totally different. State design pattern is used to define and mange state of object, while Strategy pattern is used to define a set of interchangeable algorithm and let's client to choose one of them. So Strategy pattern is a client driven pattern while Object can manage there state itself.

**4. What is decorator pattern in Java? Can you give an example of Decorator pattern?**

Decorator pattern is another popular java design pattern question which is common because of its heavy usage in java.io package. BufferedReader and BufferedWriter are good example of decorator pattern in Java. See [How to use Decorator pattern in Java](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html) fore more details.

**5. When to use Composite design Pattern in Java? Have you used previously in your project?**

This design pattern question is asked on Java interview not just to check familiarity with Composite pattern but also, whether candidate has real life experience or not. *Composite pattern* is also a core Java design pattern, which allows you to treat both whole and part object to treat in similar way. Client code, which deals with Composite or individual object doesn't differentiate on them, it is possible because Composite class also implement same interface as there individual part. One of the good example of Composite pattern from JDK is JPanel class, which is both Component and Container.  When paint() method is called on JPanel, it internally called paint() method of individual components and let them draw themselves. On second part of this design pattern interview question, be truthful, if you have used then say yes, otherwise say that you are familiar with concept and used it by your own. By the way always remember, giving an example from your project creates better impression.

**6. What is Singleton pattern in Java?**

Singleton pattern in Java is a pattern which allows only one instance of Singleton class available in whole application. java.lang.Runtime is good example of Singleton pattern in Java. There are lot's of follow up questions on Singleton pattern see [10 Java singleton interview question answers](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) for those followups

**7. Can you write thread-safe Singleton in Java?**

There are multiple ways to write thread-safe singleton in Java e.g by writing singleton using double checked locking, by using static Singleton instance initialized during [class loading.](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html) By the way using Java enum to create thread-safe singleton is most simple way. See [Why Enum singleton is better in Java](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html) for more details.

**8. When to use Template method design Pattern in Java?**Template pattern is another popular core Java design pattern interview question. I have seen it appear many times in real life project itself. Template pattern outlines an algorithm in form of template method and let subclass implement individual steps. Key point to mention, while answering this question is that template method should be final, so that subclass can not override and change steps of algorithm, but same time individual step should be abstract, so that child classes can implement them.

**9. What is Factory pattern in Java? What is advantage of using static factory method to create object?**

Factory pattern in Java is a creation Java design pattern and favorite on many Java interviews.Factory pattern used to create object by providing static factory methods. There are many advantage of providing factory methods e.g. caching immutable objects, easy to introduce new objects etc. See [What is Factory pattern in Java and benefits](http://javarevisited.blogspot.sg/2011/12/factory-design-pattern-java-example.html) for more details.

**10. Difference between Decorator and Proxy pattern in Java?**Another tricky Java design pattern question and trick here is that both Decorator and Proxy implements interface of the object they decorate or encapsulate. As I said, many Java design pattern can have similar or exactly same structure but they differ in there intent. Decorator pattern is used to implement functionality on already created object, while Proxy pattern is used for controlling access to object. One more difference between Decorator and Proxy design pattern is that, Decorator doesn't create object, instead it get object in it's constructor, while Proxy actually creates objects.

**11. When to use Setter and Constructor Injection in Dependency Injection pattern?**

Use Setter injection to provide optional dependencies of an object, while use Constructor injection to provide mandatory dependency of an object, without which it can not work. This question is related to [Dependency Injection design pattern](http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html) and mostly asked in context of Spring framework, which is now become an standard for developing Java application. Since Spring provides IOC container, it also gives you way to specify dependencies either by using setter methods or constructors. You can also take a look my [previous post](http://javarevisited.blogspot.com/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html) on same topic.

**12. What is difference between Factory and Abstract factory in Java**

see  [here](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) to answer this Java design pattern interview question.

**13. When to use Adapter pattern in Java? Have you used it before in your project?**

Use Adapter pattern when you need to make two class work with incompatible interfaces. Adapter pattern can also be used to encapsulate third party code, so that your application only depends upon Adapter, which can adapt itself when third party code changes or you moved to a different third party library. By the way this Java design pattern question can also be asked by providing actual scenario.

**14. Can you write code to implement producer consumer design pattern in Java?**

Producer consumer design pattern is a concurrency design pattern in Java which can be implemented using multiple way. if you are working in Java 5 then its better to use Concurrency util to implement producer consumer pattern instead of plain old [wait and notify in Java](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html).  Here is a good example of implementing [producer consumer problem using BlockingQueue in Java](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html).

**15. What is Open closed design principle in Java?**

Open closed design principle is one of the SOLID principle defined by Robert C. Martin, popularly known as Uncle Bob. This principle advices that a code should be open for extension but close for modification. At first this may look conflicting but once you explore power of polymorphism, you will start finding patterns which can provide stability and flexibility of this principle. One of the key example of this is State and Strategy design pattern, where Context class is closed for modification and new functionality is provided by writing new code by implementing new state of strategy. See [this](http://javarevisited.blogspot.com/2011/11/great-example-of-open-closed-design.html) article to know more about Open closed principle.

**16. What is Builder design pattern in Java? When do you use Builder pattern ?**

Builder pattern in Java is another creational design pattern in Java and often asked in Java interviews because of its specific use when you need to build an object which requires multiple properties some optional and some mandatory. See [When to use Builder pattern in Java](http://javarevisited.blogspot.sg/2012/06/builder-design-pattern-in-java-example.html) for more details

**17. Can you give an example of  SOLID design principles in Java?**

There are lots of SOLID design pattern which forms acronym SOLID, read this [list of SOLID design principles for Java programmer](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html)  to answer this Java interview question.

**18. What is difference between Abstraction and Encapsulation in Java?**

I have already covered answer of this Java interview question in my previous post as [Difference between encapsulation and abstraction in Java](http://java67.blogspot.sg/2012/08/difference-between-abstraction-and-encapsulation-java-oops.html). See there to answer this question.

This was my list of **10 popular design pattern interview question in Jav**a. I have not included MVC (Model View Controller) design pattern because that is more specific to J2EE and [Servlet JSP interview](http://javarevisited.blogspot.sg/2011/09/servlet-interview-questions-answers.html), but if you are going for any Java interview which demands experience in J2EE than you must prepare MVC design pattern. That's all on Java design pattern interview question and answers. Please let us know if you have any other interesting question on Java design pattern.

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

<http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html>

20 Design pattern and Software design interview questions for Programmers

**Design patterns and software design questions** are essential part of any programming interview, no matter whether you are going for Java interview or C#  interview. In face programming and design skill complement each other quite well, people who are good programmer are often a good designer as well as they know how to break a problem in to piece of code or software design but these skill just doesn’t come. You need to keep designing, programming both small scale and large scale systems and keep learning from mistakes.Learning about [Object oriented design principles](http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html) is a good starting point. Anyway this article is about some design questions which has been repeatedly asked in various interviews. I have divided them on two category for beginners and intermediate for sake of clarity and difficulty level.

## Design pattern interview questions for Senior and experienced level

These are questions which not only relates to design patterns but also related to software design. These questions requires some amount of thinking and experience to answer. In most of the cases interviewer is not looking for absolute answers but looking for your approach, how do you think about a problem, do you able to think through, do you able to bring out things which are not told to you. This is where experience come in picture, What are things you consider while solving a problem etc. overall these design questions kicks off your thought process. Some time interviewer ask you to write code as well so be prepare for that. you can excel in these questions if you know the concept, example and application of your programming and design skill.

**1. Give an example where you prefer abstract class over interface ?**

This is common but yet *tricky design interview question*. both interface and abstract class follow "writing code for interface than implementation" design principle which adds flexibility in code, quite important to tackle with changing requirement. here are some pointers which help you to answer this question:

1. In Java you can only extend one class but implement multiple interface. So if you extend a class you lost your chance of extending another class.

2. Interface are used to represent adjective or behavior e.g. Runnable, Clonable, Serializable etc, so if you use an abstract class to represent behavior your class can not be Runnable and Clonable at same time because you can not extend two class in Java but if you use interface your class can have multiple behavior at same time.

3. On time critical application prefer abstract class is slightly faster than interface.

4. If there is a genuine common behavior across the inheritance hierarchy which can be coded better at one place than abstract class is preferred choice. Some time interface and abstract class can work together also where defining function in interface and default functionality on abstract class.

To learn more about interface in Java check my post [10 things to know about Java interfaces](http://javarevisited.blogspot.com/2012/04/10-points-on-interface-in-java-with.html)

**2. Design a Vending Machine which can accept different coins, deliver different products?**

This is an open design question which you can use as exercise, try producing **design document**, **code** and **Junit test** rather just solving the problem and check how much time it take you to come to solution and produce require artifacts, Ideally this question should be solve in 3 hours, at least a working version.

**3. You have a Smartphone class and will have derived classes like IPhone, AndroidPhone,WindowsMobilePhone**

**can be even phone names with brand, how would you design this system of Classes**.

This is another design pattern exercise where you need to apply your object oriented design skill to come with a design which is flexible enough to support future products and stable enough to support changes in existing model.

**4. When do you overload a method in Java and when do you override it ?**

Rather a simple question for experienced designer in Java. if you see different implementation of a class has different way of doing certain thing than overriding is the way to go while overloading is doing same thing but with different input. method signature varies in case of overloading but not in case of overriding in java.

**5. Design ATM Machine ?**

We all use ATM (Automated Teller Machine) , Just think how will you design an ATM ? for designing financial system one must requirement is that they should work as expected in all situation. so no matter whether its power outage ATM should maintain **correct state (transactions**), think about **locking**, **transaction**, **error condition**, **boundary condition** etc. even if you not able to come up exact design but if you be able to point out non functional requirement, raise some question , think about boundary condition will be good progress.

**6. You are writing classes to provide Market Data and you know that you can switch to different vendors overtime like Reuters, wombat and may be even to direct exchange feed , how do you design your Market Data system.**

This is very interesting design interview question and actually asked in one of big investment bank and rather common scenario if you have been writing code in Java. Key point is you will have a MarketData interface which will have methods required by client e.g. getBid(), getPrice(), getLevel() etc and MarketData should be composed with a MarketDataProvider by using **dependency injection**. So when you change your MarketData provider Client won't get affected because they access method form MarketData interface or class.

**7. Why is access to non-static variables not allowed from static methods in Java**

You can not access non-static data from static context in Java simply because non-static variables are associated with a particular instance of object while Static is not associated with any instance. You can also see my post [why non static variable are not accessible in static context](http://javarevisited.blogspot.com/2012/02/why-non-static-variable-cannot-be.html) for more detailed discussion.

**8. Design a Concurrent Rule pipeline in Java?**

Concurrent programming or concurrent design is very hot now days to leverage power of ever increasing cores in

advanced processor and Java being a multi-threaded language has benefit over others. Do design a concurrent system key point to note is [thread-safety](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html), immutability, local variables and avoid using static or [instance variables](http://javarevisited.blogspot.com/2012/02/difference-between-instance-class-and.html). you just to think that one class can be executed by multiple thread a same time, So best approach is that every thread work on its own data, doesn't interfere on other data and have minimal synchronization preferred at start of pipeline. This question can lead from initial discussion to full coding of classes and interface but if you remember key points and issues around concurrency e.g. [race condition](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html), [deadlock](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html), memory interference, atomicity, [ThreadLocal variables](http://javarevisited.blogspot.com/2012/05/how-to-use-threadlocal-in-java-benefits.html)  etc you can get around it.

## Design pattern interview questions for Beginners

These software design and design pattern questions are mostly asked at beginners level and just informative purpose that how much candidate is familiar with design patterns like does he know **what is a design pattern** or **what does a particular design pattern do** ? These questions can easily be answered by memorizing the concept but still has value in terms of information and knowledge.

**1. What is design patterns ? Have you used any design pattern in your code ?**

Design patterns are tried and tested way to solve particular design issues by various programmers in the world. Design patterns are extension of code reuse.

**2. Can you name few design patterns used in standard JDK library?**

[Decorator design pattern](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html) which is used in various Java IO classes, Singleton pattern which is used in Runtime , Calendar and various other classes, Factory pattern which is used along with various Immutable classes likes Boolean e.g. Boolean.valueOf and Observer pattern which is used in Swing and many event listener frameworks.

**3. What is Singleton design pattern in Java ? write code for thread-safe singleton in Java**

Singleton pattern focus on sharing of expensive object in whole system. Only one instance of a particular class is maintained in whole application which is shared by all modules. Java.lang.Runtime is a classical example of Singleton design pattern. You can also see my post [10 questions on Singleton pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) for more questions and discussion. From Java 5 onwards you can use [enum](http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html) to thread-safe singleton.

**4. What is main benefit of using factory pattern ? Where do you use it?**

Factory pattern’s main benefit is increased level of encapsulation while creating objects. If you use Factory to create object you can later replace original implementation of Products or classes with more advanced and high performance implementation without any change on client layer. See my post on [Factory pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) for more detailed explanation and benefits.

**5. What is observer design pattern in Java**

Observer design pattern is based on communicating changes in state of object to observers so that they can take there action. Simple example is a weather system where change in weather must be reflected in Views to show to public. Here weather object is Subject while different views are Observers. Look on [this](http://javarevisited.blogspot.sg/2011/12/observer-design-pattern-java-example.html) article for complete example of Observer pattern in Java.

6. Give example of decorator design pattern in Java ? Does it operate on object level or class level ?

Decorator pattern enhances capability of individual object. Java IO uses decorator pattern extensively and classical example is Buffered classes like BufferedReader and BufferedWriter which enhances Reader and Writer objects to perform Buffer level reading and writing for improved performance. Read more on [Decorator design pattern and Java](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html)

7. What is MVC design pattern ? Give one example of MVC design pattern ?

8. What is FrontController design pattern in Java ? Give an example of front controller pattern ?

9. What is Chain of Responsibility design pattern ?

10.What is Adapter design pattern ? Give examples of adapter design pattern in Java?

 These are left for your exercise, try finding out answers of these design pattern questions as part of your preparation.

These were some of the **design pattern questions** I have seen in most of interviews, there are many more specially in *software design* which is important in [google interviews](http://javarevisited.blogspot.com/2012/01/google-interview-questions-answers-top.html) and various other companies like Amazon, Microsoft etc. Please share if you have faced any interesting design questions which is worth sharing.

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

<http://javarevisited.blogspot.sg/2011/12/observer-design-pattern-java-example.html>

Observer design Pattern in Java with Real world code Example

**Observer design pattern** **in Java** is a fundamental core Java pattern where Observe watch for any change in state or property of Subject. For Example Company updates all its shareholders for any decision they make here Company is Subject and Shareholders are Observers, any change in policy of company and Company notifies all its Shareholders or Observer. This was simple real world explanation of Observer pattern. In this article we will in detail *what is Observer Design pattern*, what is *benefit of Observer design Pattern*, Example or Observer pattern in Java and few other points. Just like [Decorator design Pattern](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html) and [Factory Pattern in Java](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html), Observer pattern is also used in JDK.

## Observer design Pattern Java Code Example

### What is Observer design Pattern?

*Observer design pattern in Java* is very important pattern and as name suggest it’s used to observe things. Suppose you want to notify for change in a particular object than you observer that object and changes are notified to you. Object which is being observed is refereed as Subject and classes which observe subject are called Observer. This is beautiful pattern and used heavily along with Model View Controller Design pattern where change in model is propagated to view so that it can render it with modified information. Observer pattern is also a very [popular Java interview questions](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html) and mostly asked on Senior or mid senior level.

### Problem which is solved by Observer Pattern:

If we have requirement that if particular object change its state and on depending upon

This changes some or group of objects automatically change their state we need to implement observer pattern it will reduce coupling between objects.

In real world if try to find example see when we subscribe for New Phone connection whenever customer is registered with that company all other departments are notified accordingly and then depending upon the state the do their jobs like do the verification of their address then if customer state is verified then dispatch the welcome kit etc.

### How Observer Design Pattern is implemented in Java;

For implementation of this pattern java makes our task very easy, developer need not to do so much for implementing this pattern .In **java.util** package we can find interfaces ,classes and methods for implementing this pattern.

**Public Interface Observer:**

Any class who implements this interface must be notified when subject or observable object change its status.

Update (Observable Ob, Object arg): This method is called when subject is changed.

**Class Observable:**

It’s a subject to whom observer wants to observe.

**Some Important Method:**

addObserver(Observer o):add Observers in the set of observers for this subject or observalbel object.

deleteObserver(Observer o): delete Observers in the set of observers .

hasChanged():check if object has changed.

clearChanged():this method will indicate that subject has no changes or all the observers has been notified when changes is made.

notifyObservers(): notify all the observers if object has changed .

### Code Example of Observer design pattern in Java:

**Observer Design pattern** is generic than how it is implemented in Java. You are free to choose java.util.Observable or java.util.Observer or *writing your own Subject and Observer interface*. I prefer having my own Subject and Observer interface and this is how I am going to write my Code Example of Observer design Pattern in java. Mine Example is very Simple you have a Loan on which interest rate is subject to change and when it changes, Loan notifies to Newspaper or Internet media to display new loan interest rate. To implement this we have a **Subject** interface which contains methods for adding, removing and notifying Observers and an **Observer** interface which contains update(int interest) method which will be called by Subject implementation when interest rate changes.

**import** java.util.ArrayList;

**interface** Observer {

**public** **void** update(**float** interest);

}

**interface** Subject {

**public** **void** registerObserver(Observer observer);

**public** **void** removeObserver(Observer observer);

**public** **void** notifyObservers();

}

**class** Loan **implements** Subject {

**private** ArrayList<Observer> observers = **new** ArrayList<Observer>();

**private** String type;

**private** **float** interest;

**private** String bank;

**public** Loan(String type, **float** interest, String bank) {

**this**.type = type;

**this**.interest = interest;

**this**.bank = bank;

       }

**public** **float** getInterest() {

**return** interest;

       }

**public** **void** setInterest(**float** interest) {

**this**.interest = interest;

              notifyObservers();

       }

**public** String getBank() {

**return** **this**.bank;

       }

**public** String getType() {

**return** **this**.type;

       }

       @Override

**public** **void** registerObserver(Observer observer) {

              observers.add(observer);

       }

       @Override

**public** **void** removeObserver(Observer observer) {

              observers.remove(observer);

       }

       @Override

**public** **void** notifyObservers() {

**for** (Observer ob : observers) {

                     System.*out*

                                  .println("Notifying Observers on change in Loan interest rate");

                     ob.update(**this**.interest);

              }

       }

}

**class** Newspaper **implements** Observer {

       @Override

**public** **void** update(**float** interest) {

              System.*out*.println("Newspaper: Interest Rate updated, new Rate is: "

                           + interest);

       }

}

**class** Internet **implements** Observer {

       @Override

**public** **void** update(**float** interest) {

              System.*out*.println("Internet: Interest Rate updated, new Rate is: "

                           + interest);

       }

}

**public** **class** ObserverTest {

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

              // this will maintain all loans information

              Newspaper printMedia = **new** Newspaper();

              Internet onlineMedia = **new** Internet();

              Loan personalLoan = **new** Loan("Personal Loan", 12.5f,

                           "Standard Charterd");

              personalLoan.registerObserver(printMedia);

              personalLoan.registerObserver(onlineMedia);

              personalLoan.setInterest(3.5f);

       }

}

**Advantage of Observer Design Pattern in Java:**

Main advantage is **loose coupling** between objects called observer and observable. The subject only know the list of observers it don’t care about how they have their implementation.All the observers are notified by subject in a single event call as **Broadcast communication**

**Disadvantage of Observer Design Pattern in Java:**

·          The disadvantage is that the sometime if any problem comes, [debugging](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html) becomes very difficult because flow of control is implicitly between **observers** and **observable** we can predict that now observer is going to fire and if there is chain between observers then debugging become more complex.

·          Another issue is Memory management because subject will hold all the reference of all the observers if we not unregister the object it can create the memory issue.

That’s all on **Observer Pattern in Java**. Share your thought how and when you have used Observer Pattern in your Project and any issue you have faced?

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

<http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html>

Decorator design Pattern in Java with Example Java Tutorial

I was thinking to write on **decorator design pattern in Java** when I first wrote [10 interview questions on Singleton Pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html). Since design pattern is quite important while building software and it’s equally important on any Core Java Interview, It’s always good to have clear understanding of various design patterns in Java. In this article we will explore and learn **Decorator Design pattern in Java** which is a prominent core Java design pattern and you can see lot of its example in JDK itself. JDK use decorator pattern in IO package where it has decorated Reader and Writer Classes for various scenario, for example BufferedReader and BufferedWriter are example of decorator design pattern in Java. From design perspective its also good idea to learn how existing things work inside JDK itself for example [How HashMap works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html) or [How SubString method work in Java](http://javarevisited.blogspot.com/2011/10/how-substring-in-java-works.html), that will give you some idea of things you need to keep in mind while designing your Class or interface in Java. Now let’s Move on to **Decorator pattern in Java**.

## Java Decorator Design Pattern

In this Java tutorial we will see:

What is decorator pattern in Java?

When to use decorator pattern in Java?

How to use decorator pattern in Java?

Example of decorator design pattern

Advantage and Disadvantage of decorator pattern in Java

### What is decorator design pattern in Java?

·          Decorator design pattern is used to **enhance the functionality of a particular object at run-time** or dynamically.

·          At the same time **other instance of same class will not be affected by this** so individual object gets the new behavior.

·          Basically we wrap the original object through decorator object.

·          Decorator design pattern is based on abstract classes and we derive concrete implementation from that classes,

·          It’s a structural design pattern and most widely used.

I prefer to answer *What is decorator design pattern* in point format just to stress on important point like this pattern operator at individual object level. This question also asked in many [Core Java interviews in Investment banks](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html)

### Problem which is solved by Decorator Pattern:

Now the question is why this pattern has came into existence what is the problem with existing system, so the answer is if anyone wants to add some functionality to individual object or change the state of particular object at run time it is not possible what the possible is we can provide the specific behavior to all the object of that class at design time by the help of inheritance or using subclass, but **Decorator pattern** makes possible that we provide individual object of same class a specific behavior or state at run time. This doesn’t affect other object of same [Class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html).

**When to use Decorator pattern in Java**

·          When sub classing is become impractical and we need large number of different possibilities to make independent object or we can say we have number of combination for an object.

·          Secondly when we want to add functionality to individual object not to all object at run-time we use decorator design pattern.

**Code Example of decorator design pattern:**

To better understand concept of decorator design pattern let see a code example using Decorator Pattern in Java. You can also look inside JDK and find what are classes and packages which are using decorator pattern.

// Component on Decorator design pattern

public abstract class Currency {

String description = "Unknown currency";

public String getCurrencyDescription() {

return description;

}

public abstract double cost(double value);

}

// Concrete Component

public class Rupee extends Currency {

double value;

public Rupee() {

description = "indian rupees";

}

public double cost(double v){

value=v;

return value;

}

}

//Another Concrete Component

public class Dollar extends Currency{

double value;

public Dollar () {

description = "Dollar”;

}

public double cost(double v){

value=v;

return value;

}

}

// Decorator

public abstract class Decorator extends Currency{

public abstract String getDescription();

}

// Concrete Decorator

public class USDDecorator extends Decorator{

Currency currency;

public USDDecorator(Currency currency){

this.currency = currency;

}

public String getDescription(){

return currency.getDescription()+" ,its US Dollar";

}

}

//Another Concrete Decorator

public class SGDDecorator extends Decorator{

Currency currency;

public SGDDecorator(Currency currency){

this.currency = currency;

}

public String getDescription(){

return currency.getDescription()+" ,its singapore Dollar";

}

}

Now its time to check currency.

public class CurrencyCheck {

public static void main(String[] args) {

// without adding decorators

Currency curr = new Dollar();

System.out.println(curr.getDescription() +" dollar. "+curr.cost(2.0));

//adding decorators

Currency curr2 = new USDDecorator(new Dollar());

System.out.println(curr2.getDescription() +" dollar. "+curr2.cost(4.0));

Currency curr3 = new SGDDecorator(new Dollar());

System.out.println(curr3.getDescription() +" dollar. "+curr3.cost(4.0));

}

**Explanation of the code**:

We can understand this in following term;

1.      **Component Interface**: In our example Currency interface is component which used on its own or we need decorator for that.

2.      **Concrete Component: it** implements Component and we add new behavior to this object at dynamically. Dollar and Rupee are the concrete implementation of currency.

3.      **Decorator: Decorator** contains a HAS a Relationship in simple word we can say it has a instance variable that holds reference for component they implement same component which they are going to decorate. Here a Decorator is an abstract class which extends the currency.

4.      **Concrete Decorator:** it’s an implementation of Decorator So USD Dollar and SGD Dollar are the implementation of Decorator contains instance variable for component interface or the thing which they are going to decorate.

**Advantage of Decorator design Pattern in Java**

In brief we see what the main advantages of using decorator design patterns are.

1.      Decorator Pattern is flexible than inheritance because inheritance add responsibilities at compile time and it will add at run-time.

2.      Decorator pattern enhance or modify the object functionality

**Disadvantage**

Main disadvantage of using Decorator Pattern in Java is that the code maintenance can be a problem as it provides a lot of similar kind of small objects (each decorator).

That’s all on **decorator design pattern in Java**. To get mastery on decorator pattern I suggest looking inside JDK library itself and finding what classes are decorated, why they are decorated. Also think of scenario where inheritance is impractical and you look more flexibility and try to **use decorator pattern in Java** there.

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

<http://javarevisited.blogspot.sg/2011/12/factory-design-pattern-java-example.html>

What is Factory method Design Pattern in Java with Example – Tutorial

**Factory design pattern in Java** one of the core design pattern which is used heavily not only in JDK but also in various Open Source framework such as Spring, Struts and Apache along with decorator design pattern in Java. Factory Design pattern is based on [**Encapsulation**](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html)object oriented concept. Factory method is used to create different object from factory often refereed as Item and it encapsulate the creation code. So instead of having object creation code on client side we encapsulate inside **Factory method in Java**. One of the best examples of factory pattern in Java is BorderFactory Class of Swing API. In this Design pattern tutorial we will see **What is Factory method design pattern in Java**, What are main *advantages of factory pattern in Java* , Code example of Factory design pattern and What problem **Factory pattern** solves in Java or when to use Factory design pattern.  This article is in continuation of my design pattern article as [10 OOPS and SOLID design principles java programmer should know](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html) and How to use Observer pattern in Java

## 

## What is static factory method or factory design pattern

Factory design pattern is used to create objects or [Class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html) and it provides loose coupling and high cohesion. Factory pattern encapsulate object creation logic which makes it easy to change it later when you change how object gets created or you can even introduce new object with just change in one class. In GOF pattern list Factory pattern is listed as Creation design pattern. Factory should be an interface and clients first either creates factory or get factory which later used to create objects.

## Example of static factory method in JDK

 Best Example of Factory method design pattern is valueOf() method which is there in String and wrapper classes like Integer and Boolean and used for type conversion i.e. from converting String to Integer or String to double in java..

Some more examples of factory method design pattern from JDK is :

valueOf() method which returns object created by factory equivalent to value of parameter passed.

getInstance() method which creates instance of Singleton class.

newInstance() method which is used to create and return new instance from factory method every time called.

getType() and newType() equivalent of getInstance() and newInstance() factory method but used when factory method resides in separate class.

### Problem which is solved by Factory method Pattern in Java

Whenever we talk about **object oriented language** it will based upon some concept like [abstraction](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html), [polymorphism](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html) etc and on that [encapsulation](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html) and delegation are important concept any design will be called good if task are delegated to different object and some kind of encapsulation is there.

Some time our application or framework will not know that what kind of object it has to create at run-time it knows only the interface or abstract class and as we know we can not create object of interface or abstract class so main problem is frame work knows **when** it has to create but don’t know **what kind** of object.

Whenever we create object using new() we violate **principle of programming for interface rather than implementation** which eventually result in inflexible code and difficult to change in maintenance. By using Factory design pattern in Java we get rid of this problem.

Another problem we can face is class needs to contain objects of other classes or class hierarchies within it; this can be very easily achieved by just using the new keyword and the class constructor. The problem with this approach is that it is a very hard coded approach to create objects as this creates dependency between the two classes.

So **factory pattern** solve this problem very easily by model an interface for creating an object which at creation time can let its subclasses decide which class to instantiate, Factory Pattern promotes loose coupling by eliminating the need to bind application-specific classes into the code. The **factory methods** are typically implemented as virtual methods, so this pattern is also referred to as the “**Virtual Constructor**”. These methods create the objects of the products or target classes.

### When to use Factory design pattern in Java

* Static Factory methods are common in frameworks where library code needs to create objects of types which may be sub classed by applications using the framework.
* Some or all concrete products can be created in multiple ways, or we want to leave open the option that in the future there may be new ways to create the concrete product.
* Factory method is used when Products don't need to know how they are created.
* We  can use factory pattern where we have to create an object of any one of sub-classes depending on the data provided

### Code Example of Factory Design Pattern in Java:

Let’s see an example of how factory pattern is implemented in Code.We have requirement to create multiple currency e.g. INR, SGD, USD and code should be extensible to accommodate new Currency as well. Here we have made Currency as interface and all currency would be concrete implementation of Currency interface. Factory Class will create Currency based upon country and return concrete implementation which will be stored in interface type. This makes code dynamic and extensible.

Here is complete **code example of Factory pattern in Java**:

### interface Currency {

### String getSymbol();

### }

### // Concrete Rupee Class code

### class Rupee implements Currency {

### @Override

### public String getSymbol() {

### return "Rs";

### }

### }

### // Concrete SGD class Code

### class SGDDollar implements Currency {

### @Override

### public String getSymbol() {

### return "SGD";

### }

### }

### // Concrete US Dollar code

### class USDollar implements Currency {

### @Override

### public String getSymbol() {

### return "USD";

### }

### }

### // Factroy Class code

### class CurrencyFactory {

### public static Currency createCurrency (String country) {

### if (country. equalsIgnoreCase ("India")){

### return new Rupee();

### }else if(country. equalsIgnoreCase ("Singapore")){

### return new SGDDollar();

### }else if(country. equalsIgnoreCase ("US")){

### return new USDollar();

### }

### throw new IllegalArgumentException("No such currency");

### }

### }

### // Factory client code

### public class Factory {

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

### String country = args[0];

### Currency rupee = CurrencyFactory.createCurrency(country);

### System.out.println(rupee.getSymbol());

### }

### }

### Advantage of Factory method Pattern in Java:

**Factory pattern in Java** is heavily used everywhere including JDK, open source library and other frameworks.In following are main advantages of using Factory pattern in Java:

1*) Factory method design pattern* decouples the calling class from the target class, which result in less coupled and highly cohesive code?

E.g.: JDBC is a good example for this pattern; application code doesn't need to know what database it will be used with, so it doesn't know what database-specific driver classes it should use. Instead, it uses factory methods to get Connections, Statements, and other objects to work with. Which gives you flexibility to change your back-end database without changing your DAO layer in case you are using ANSI SQL features and not coded on DBMS specific feature?

2) Factory pattern in Java enables the subclasses to provide extended version of an object, because creating an object inside factory is more flexible than creating an object directly in the client. Since client is working on interface level any time you can enhance the implementation and return from Factory.

3) Another benefit of using *Factory design pattern in Java* is that it encourages [consistency in Code](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html) since every time object is created using Factory rather than using different constructor at different client side.

4) Code written using Factory design pattern in Java is also [easy to debug](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html) and troubleshoot because you have a centralized method for object creation and every client is getting object from same place.

Some more advantages of factory method design pattern is:

1. **Static factory method** used in factory design pattern enforces use of Interface than implementation which itself a good practice. for example:

**Map** synchronizedMap = **Collections**.synchronizedMap(**new** **HashMap**());

2. Since static factory method have return type as Interface, it allows you to replace implementation with better performance version in newer release.

3. Another advantage of static factory method pattern is that they can cache frequently used object and eliminate duplicate object creation. Boolean.valueOf() method is good example which caches true and false boolean value.

4. Factory method pattern is also recommended by Joshua Bloch in Effective Java.

5 Factory method pattern offers alternative way of creating object.

6. Factory pattern can also be used to hide information related to creation of object.

That’s all on **Factory design patten in Java** for now. This is one of the most used patterns in Java library and different Java frameworks. Summary is try to use **Factory pattern** whenever you see an opportunity to encapsulate object creation code and see chance of creating different object in near future.

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

<http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html>

Inversion of Control and Dependency Injection design pattern with real world Example - Spring tutorial

Inversion of Control and Dependency Injection is a core design pattern of Spring framework. IOC and DI design pattern is also a popular [design pattern interview question in Java](http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html). As name suggest Inversion of control pattern Inverts responsibility of managing life cycle of object e.g. creating object, setting there dependency etc from application to framework, which makes writing Java application even more easy. Programmer often confused between IOC and DI, well both words used interchangeably in Java world but **Inversion of Control** is more general concept and **Dependency Injection** is a concrete design pattern. Spring framework provides two implementation of IOC container in form of [Application Context and BeanFactory](http://javarevisited.blogspot.ca/2012/11/difference-between-beanfactory-vs-applicationcontext-spring-framework.html) which manages life-cycle of bean used by Java application. As you may know necessity is mother of invention, it benefit to first understand problem solved by IOC and Dependency Injection design pattern. This makes your understanding more clear and concrete. We have touched basics of Dependency Injection and Inversion of control in our article [10 OOPS and SOLID design principles for Java programmer](http://javarevisited.blogspot.de/2012/03/10-object-oriented-design-principles.html) and this Java article tries to explain it by taking a real life example of Service based architecture popular in enterprise Java development. In this Spring or design pattern tutorial we will first see normal implementation of AutditService class, a class in this example which provides auditing in enterprise Java application and than use of dependency Injection. This will allow us to find out problems and how they are solved by *Dependency injection design pattern*. . Also there are multiple way to inject dependency in spring e.g. Setter Injection or Constructor Injection, which uses setter method and constructor for injecting dependency, see [Setter injection vs Constructor injection](http://javarevisited.blogspot.sg/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html)  to find out when to use them.

## Inversion of Control and Dependency Injection design pattern

Any way let’s back to core *concept of Inversion of Control and dependency Injection* design pattern. Look at below implementation of an AuditService whose job is to store every audit messages into database. This is one of the simplest kind of auditing Service required in Enterprise Java application.

/\*\*

 \* Java Service class which provides auditing functionality by storing

 \* auditing message into persistent.

 \*/

**public** **class** AuditServiceImpl **implements** AuditService{

**private** AuditDAO auditDao = **new** AuditDAO();

    @**Override**

**public** **boolean** audit (**String** message) {

**return** auditDao.store(message);

    }

}

In first glance this implementation looks perfect but there are three major problem with this implementation:

1) Every AuditServiceImpl has its **own copy of AuditDAO** which is an **expensive** object as it wraps a [database connection](http://javarevisited.blogspot.sg/2012/06/jdbc-database-connection-pool-in-spring.html) with in. It make no sense to create separate instances of AuditDAO, if you can share one between multiple AuditService.

2) AuditServiceImpl is **closely coupled with AuditDAO** as its creating instance of AuditDAO using new() operator. If you change the [constructor](http://javarevisited.blogspot.sg/2012/12/what-is-constructor-in-java-example-chainning-overloading.html) of AuditDAO this code will be broken. Because of this coupling its difficult to replace AuditDAO with better implementation.

3) There is **no easy way to test audit()** method which is **dependent on auditDAO**. Since you can not mock AuditDAO you have to rely on actual implementation and if AuditDAO is an environmental dependent object which it is as it connect to different database on different environment, your [Junit test](http://javarevisited.blogspot.sg/2012/08/best-practices-to-write-junit-test.html) case may pass in some environment and may fail in other environment.

## What is Dependency Injection concept:

Dependency Injection is a design pattern on which dependency of object (in this case AuditDAO is a dependency for AuditServiceImpl Object) is injected by framework rather than created by [Object](http://javarevisited.blogspot.ca/2012/12/what-is-object-in-java-or-oops-example.html) itself. Dependency Injection reduces coupling between multiple object as its dynamically injected by framework. One of the implementation of DI is Inversion of Control (IOC) on which framework like Spring controls object’s dependency. There are mainly two types of Dependency Injection: [Constructor Injection and Setter Injection](http://javarevisited.blogspot.sg/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html).

In Constructor Injection, dependency of Object is injected using [constructor](http://javarevisited.blogspot.sg/2012/12/what-is-constructor-in-java-example-chainning-overloading.html), while in Setter Injection, Dependency is provided by [setter method](http://javarevisited.blogspot.sg/2012/12/getter-and-setter-method-vs-public-modifier-field-java.html). Both has there pros and cons. Constructor DI allows object to be created in complete state and follows principle of fully functional object while Setter DI allows object to be created without its dependency. which may result in **incomplete object** if dependency is not available. This answers one of the famous [spring interview question](http://javarevisited.blogspot.sg/2011/09/spring-interview-questions-answers-j2ee.html) "when do you use Setter injection and Constructor Injection in Spring". Another benefit of Setter Dependency Injection is readability, since Spring is configured with **xml configuration file** and setter injection is provided with bean property which is much easier to read and understand than constructor injection which doesn't state the property.

**AuditServiceImpl written using Dependency Injection**

Now we will see How Dependency Injection solves all three problems we have listed with above implementation of AuditService. here is a new implementation of AuditService with setter dependency injection.

**public** **class** AuditServiceImpl **implements** AuditService{

**private** AuditDAO auditDao;

**public** **void** setAuditDao(AuditDAO AuditDao) {

**this**.AuditDao = AuditDao;

    }

    @**Override**

**public** **boolean** audit (**String** message) {

**return** auditDao.store(message);

    }

}

1. Since AuditDAO is injected here its possible to share single AuditDAO (an expensive object) between multiple AuditService.

2. Since AuditServiceImpl is not creating instance of AuditDAO its no more coupled with AuditDAO and work with any implementation of AuditDAO, thanks to another famous object oriented design principle [“program for interface than implementation"](http://javarevisited.blogspot.de/2012/03/10-object-oriented-design-principles.html).

3. Because AuditDAO is injected by DI at runtime its easy to test audit() method by providing a mock AuditDAO class. This not only makes testing easier but also independent of environmental changes as you are not using actual implementation of AuditService.

This was the exact way I learn **Dependency Injection** and **Inversion Of Control design principles**. It always help first to understand problem and than solution to related each other. From above learning we can easily derive *advantages or benefits of Dependency Injection* in Java application:

**1) Reduce coupling**

both constructor and setter dependency injection reduce coupling. like in above example coupling between AuditService and AuditDAO is reduced by using Dependency Injection.

**2) Improves testability**

Dependency Injection allows to replace actual object with mock object which improves testability by writing simple JUnit tests which uses mock object.

**3) Flexibility**

This is another advantage which comes as side benefit of reduced coupling, because of DI you can replace non performance implementation with better one later.

That’s all on **What is Inversion of control and Dependency Injection design pattern**. We have tried to learn this pattern with a real life example and compares a class which is written using principle of IOC and DI and without that. IOC and DI easily bring quality in coding. We have seen clear benefits in terms of reduce coupling, improved testability and Flexibility to change implementation. It’s always good to write code which follows principle of Inversion of Control and dependency Injection and Spring framework by default ensures that.

Other **design pattern and Spring tutorials** from Javarevisited

[Decorator design pattern in Java with real life example](http://javarevisited.blogspot.sg/2011/11/decorator-design-pattern-java-example.html)

[How to implement LDAP authentication in Java using Spring security](http://javarevisited.blogspot.de/2011/11/ldap-authentication-active-directory.html)

[When to use Factory method design pattern in Java](http://javarevisited.blogspot.sg/2011/12/factory-design-pattern-java-example.html)

[What is Builder design pattern in Java with real world example](http://javarevisited.blogspot.ca/2012/06/builder-design-pattern-in-java-example.html)

[What is Observer design pattern in Java with real life example](http://javarevisited.blogspot.sg/2011/12/observer-design-pattern-java-example.html)

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

Question 9. What is the difference between factory and abstract factory pattern?

This Java interview question is from my list of 20 Java design pattern interview question and its open for all of you to answer.

@Raj suggested

Abstract Factory provides one more level of abstraction. Consider different factories each extended from an Abstract Factory and responsible for creation of different hierarchies of objects based on the type of factory. E.g. AbstractFactory extended by AutomobileFactory, UserFactory, RoleFactory etc. Each individual factory would be responsible for creation of objects in that genre. See here for detailed answer of this question.

<http://javarevisited.blogspot.sg/2013/01/difference-between-factory-and-abstract-factory-design-pattern-java.html>

Difference between Factory and Abstract Factory design pattern in Java

Both Abstract Factory and Factory design pattern are creational design pattern and use to decouple clients from creating object they need, But there is a significant difference between Factory and Abstract Factory design pattern, [Factory design pattern](http://javarevisited.blogspot.it/2011/12/factory-design-pattern-java-example.html) produces implementation of Products e.g. Garment Factory produce different kinds of clothes, On the other hand Abstract Factory design pattern adds another layer of [abstraction](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html) over Factory Pattern and Abstract Factory implementation itself e.g. Abstract Factory will allow you to choose a particular Factory implementation based upon need which will then produce different kinds of products.

In short

1) Abstract Factory design pattern creates Factory

2) Factory design pattern creates Products

## Difference between Abstract Factory and Factory design pattern in Java

Let see another example of Abstract Factory and Factory design pattern in Java from JDK itself to get a better understanding. If you have done some XML work in Java e.g. [reading XML files using DOM parser](http://javarevisited.blogspot.sg/2011/12/parse-xml-file-in-java-example-tutorial.html), you may be familiar with DocumentBuilderFactory class which is an example abstract factory design pattern because it returns a factory called DocumentBuilder which then used to create Document.

*//Example of Abstract Factory and Factory design pattern  in Java*

**DocumentBuilderFactory** abstractFactory = **DocumentBuilderFactory**.newInstance();

**DocumentBuilder** factory = abstractFactory.newDocumentBuilder();

**Document** doc = factory.parse(stocks)

In this example DocumentBuilderFactory (Abstract Factory) creates DocumentBuilder (Factory) which creates Documents (Products).

Let's see some more difference between Abstract Factory and Factory design pattern in Java in point form :

1) One more *difference between Abstract Factory and Factory design pattern* is that AbstractFactory pattern uses composition to delegate responsibility of creating object to another class while Factory design pattern uses [inheritance](http://javarevisited.blogspot.sg/2012/10/what-is-inheritance-in-java-and-oops-programming.html) and relies on derived class or sub class to create [object](http://javarevisited.blogspot.sg/2012/12/what-is-object-in-java-or-oops-example.html).

2) Abstract Factory may use Factory design pattern for creating objects but they are not just limited to that they can also use [Builder design pattern](http://javarevisited.blogspot.ro/2012/06/builder-design-pattern-in-java-example.html) to build object by doing series of steps or Prototype pattern to build object by copying or customizing prototype of that object. It completely depends upon your implementation whether to use Factory pattern or Builder pattern for creating products.

**When to use Abstract Factory and Factory method design pattern in Java**

Factory method design pattern are modern way of creating objects. It offers some notable advantages over new() operator to create Objects e.g. By using Factory method design pattern client is completely decoupled with object creation code, which enforces [Encapsulation](http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html) and result is loosely coupled and highly cohesive system. Any change e.g. a new product from Factory requires almost no change in existing clients. See [When to use Factory method design pattern in Java](http://javarevisited.blogspot.it/2011/12/factory-design-pattern-java-example.html) for more scenarios. On the other hand if you need an additional level of abstraction over your Factory pattern than *Abstract Factory* is the right design pattern to use. Abstract Factory allows you to use different Factory implementation for different purpose. Abstract Factory pattern can be implemented using Factory method and [Singleton design pattern in Java](http://javarevisited.blogspot.sg/2012/12/how-to-create-thread-safe-singleton-in-java-example.html). One of the *best example of Abstract Factory and Factory pattern in Java* is DocumentBuilderFactory and DocumentBuilder javax.xml.parsers package.

That's all on **difference between Abstract Factory and Factory design pattern in Java**. In short Abstract Factory design pattern provides abstraction over Factory pattern itself while Factory design pattern provides abstraction over products.

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

<http://javarevisited.blogspot.sg/2012/06/builder-design-pattern-in-java-example.html>

Builder Design pattern in Java - Example Tutorial

**Builder design pattern in Java** is a creational pattern i.e. used to create objects, similar to [factory method design pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) which is also creational design pattern. Before learning any design pattern I suggest find out the problem a particular design pattern solves. Its been well said **necessity is mother on invention**. learning design pattern without facing problem is not that effective, Instead if you have already faced issues than its much easier to understand design pattern and learn how its solve the issue. In this Java design pattern tutorial we will first see what problem *Builder design pattern* solves which will give some insight on **when to use builder design pattern in Java,** which is also a [popular design pattern interview question](http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html) and then we will see example of Builder design pattern and pros and cons of using Builder pattern in Java.

## What problem Builder pattern solves in Java

As I said earlier *Builder pattern* is a creational design pattern it means its solves problem related to object creation. Constructors in Java are used to create object and can take parameters required to create object. Problem starts when an Object can be created with **lot of parameters**, some of them may be **mandatory** and others may be **optional**. Consider a class which is used to create Cake, now you need number of item like egg, milk, flour to create cake. many of them are mandatory and some  of them are optional like cherry, fruits etc. If we are going to have [overloaded constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) for different kind of cake then there will be many constructor and even worst they will accept many parameter.

**Problems:**

1) too many constructors to maintain.

2) error prone because many fields has same type e.g. sugar and and butter are in cups so instead of 2 cup sugar if you pass 2 cup butter, your compiler will not complain but will get a buttery cake with almost no sugar with high cost of wasting butter.

You can partially solve this problem by creating Cake and then adding ingredients but that will impose another problem of **leaving Object on inconsistent state during building**, ideally cake should not be available until its created. Both of these problem can be solved by using Builder design pattern in Java. Builder design pattern not only improves readability but also reduces chance of error by adding ingredients explicitly and making object available once fully constructed.

By the way there are many design pattern tutorial already there in Javarevisited like [Decorator pattern tutorial](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html) and  [Observer pattern in Java](http://javarevisited.blogspot.com/2011/12/observer-design-pattern-java-example.html). If you haven’t read them already then its worth looking.

## Example of Builder Design pattern in Java

We will use same example of creating Cake using Builder design pattern in Java. here we have [static nested builder class](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) inside Cake which is used to create object.

**Guidelines for Builder design pattern in Java**

1) Make a static nested class called Builder inside the class whose object will be build by Builder. In this example its Cake.

2) Builder class will have exactly same set of fields as original class.

3) Builder class will expose method for adding ingredients e.g. sugar() in this example. each method will return same Builder object. Builder will be enriched with each method call.

4) Builder.build() method will copy all builder field values into actual class and return object of Item class.

5) Item class (class for which we are creating Builder) should have [private constructor](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html) to create its object from build() method and prevent outsider to access its constructor.

**public** **class** BuilderPatternExample {

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

*//Creating object using Builder pattern in java*

        Cake whiteCake = **new** Cake.Builder().sugar(1).butter(0.5).  eggs(2).vanila(2).flour(1.5). bakingpowder(0.75).milk(0.5).build();

*//Cake is ready to eat :)*

**System**.out.println(whiteCake);

    }

}

**class** Cake {

**private** **final** **double** sugar;   *//cup*

**private** **final** **double** butter;  *//cup*

**private** **final** **int** eggs;

**private** **final** **int** vanila;     *//spoon*

**private** **final** **double** flour;   *//cup*

**private** **final** **double** bakingpowder; *//spoon*

**private** **final** **double** milk;  *//cup*

**private** **final** **int** cherry;

**public** **static** **class** Builder {

**private** **double** sugar;   *//cup*

**private** **double** butter;  *//cup*

**private** **int** eggs;

**private** **int** vanila;     *//spoon*

**private** **double** flour;   *//cup*

**private** **double** bakingpowder; *//spoon*

**private** **double** milk;  *//cup*

**private** **int** cherry;

*//builder methods for setting property*

**public** Builder sugar(**double** cup){**this**.sugar = cup; **return** **this**; }

**public** Builder butter(**double** cup){**this**.butter = cup; **return** **this**; }

**public** Builder eggs(**int** number){**this**.eggs = number; **return** **this**; }

**public** Builder vanila(**int** spoon){**this**.vanila = spoon; **return** **this**; }

**public** Builder flour(**double** cup){**this**.flour = cup; **return** **this**; }

**public** Builder bakingpowder(**double** spoon){**this**.sugar = spoon; **return** **this**; }

**public** Builder milk(**double** cup){**this**.milk = cup; **return** **this**; }

**public** Builder cherry(**int** number){**this**.cherry = number; **return** **this**; }

*//return fully build object*

**public** Cake build() {

**return** **new** Cake(**this**);

        }

    }

*//private constructor to enforce object creation through builder*

**private** Cake(Builder builder) {

**this**.sugar = builder.sugar;

**this**.butter = builder.butter;

**this**.eggs = builder.eggs;

**this**.vanila = builder.vanila;

**this**.flour = builder.flour;

**this**.bakingpowder = builder.bakingpowder;

**this**.milk = builder.milk;

**this**.cherry = builder.cherry;

    }

    @**Override**

**public** **String** toString() {

**return** "Cake{" + "sugar=" + sugar + ", butter=" + butter + ", eggs=" + eggs + ", vanila=" + vanila + ", flour=" + flour + ", bakingpowder=" + bakingpowder + ", milk=" + milk + ", cherry=" + cherry + '}';

    }

  }

**Output:**

Cake{sugar=0.75, butter=0.5, eggs=2, vanila=2, flour=1.5, bakingpowder=0.0, milk=0.5, cherry=0}

## Builder design pattern in Java – Pros and Cons

Live everything Builder pattern also has some disadvantages, but if you look at below, advantages clearly outnumber disadvantages of Builder design pattern. Any way here are few advantages and disadvantage of Builder design pattern for creating objects in Java.

**Advantages:**

1) more maintainable if number of fields required to create object is more than 4 or 5.

2) less error-prone as user will know what they are passing because of explicit method call.

3) more robust as only fully constructed object will be available to client.

**Disadvantages:**

1) verbose and code duplication as Builder needs to copy all fields from Original or Item class.

### When to use Builder Design pattern in Java

Builder Design pattern is a creational pattern and should be used when number of parameter required in constructor is more than manageable usually 4 or at most 5. Don't confuse with **Builder and Factory pattern** there is an obvious difference between Builder and Factory pattern, as Factory can be used to create different implementation of same interface but Builder is tied up with its Container class and only returns object of Outer class.

That's all on **Builder design pattern in Java**. we have seen why we need Builder pattern , what problem it solves, Example of builder design pattern in Java and finally when to use Builder patter with pros and cons. So if you are not using telescoping constructor pattern or have a choice not to use it than Builder pattern is way to go.

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

<http://java67.blogspot.sg/2012/08/difference-between-abstraction-and-encapsulation-java-oops.html>

Difference between Abstraction and Encapsulation in Java – OOPS

**Abstraction vs Encapsulation – Java OOPS**

Abstraction and Encapsulation in Java are two important [Object oriented programming concept](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html) and they are completely different to each other. Only similarity between Abstraction and Encapsulation is that they are OOPS concept, other than that they mean two different things. Abstraction represent taking out the behavior from How exactly its implemented, one example of [abstraction in Java](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html) is interface while Encapsulation means hiding details of implementation from outside world so that when things change no body gets affected. One example of [Encapsulation in Java](http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html) is private methods; clients don't care about it, You can change, amend or even remove that method if that method is not encapsulated and it were public all your clients would have been affected. Apart from this main difference in behavior here are couple of more *differences between Abstraction and Encapsulation in Java*.

## Abstraction vs Encapsulation in Java

Here are some of the main differences between Abstraction vs Encapsulation in Java and OOPS(Object Oriented programming) concept. Abstraction and Encapsulation along with [Inheritance](http://java67.blogspot.sg/2012/08/what-is-inheritance-in-java-oops-programming-example.html) and [polymorphism](http://javarevisited.blogspot.sg/2011/08/what-is-polymorphism-in-java-example.html) forms basis of Object oriented programming in Java.

1) First difference between Abstraction and Encapsulation is that, Abstraction is implemented in Java using [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) and abstract class while Encapsulation is implemented using [private](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html), package-private and protected access modifier.

2) Encapsulation is also called data hiding.

3) Design principles "[programming for interface than implementation](http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html)" is based on abstraction and "encapsulate whatever changes" is based upon Encapsulation.

That's all from my side on differences between Abstraction and Encapsulation in Java. Correct understanding of Encapsulation and Abstraction is must for any Java developer. Head first Object oriented Analysis and design is a great book to learn more about Abstraction, Encapsulation and other OOPS concept.

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

<http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html>

What is Factory method Design Pattern in Java with Example – Tutorial

**Factory design pattern in Java** one of the core design pattern which is used heavily not only in JDK but also in various Open Source framework such as Spring, Struts and Apache along with decorator design pattern in Java. Factory Design pattern is based on [**Encapsulation**](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html)object oriented concept. Factory method is used to create different object from factory often refereed as Item and it encapsulate the creation code. So instead of having object creation code on client side we encapsulate inside **Factory method in Java**. One of the best examples of factory pattern in Java is BorderFactory Class of Swing API. In this Design pattern tutorial we will see **What is Factory method design pattern in Java**, What are main *advantages of factory pattern in Java* , Code example of Factory design pattern and What problem **Factory pattern** solves in Java or when to use Factory design pattern.  This article is in continuation of my design pattern article as [10 OOPS and SOLID design principles java programmer should know](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html) and How to use Observer pattern in Java

## 

## What is static factory method or factory design pattern

Factory design pattern is used to create objects or [Class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html) and it provides loose coupling and high cohesion. Factory pattern encapsulate object creation logic which makes it easy to change it later when you change how object gets created or you can even introduce new object with just change in one class. In GOF pattern list Factory pattern is listed as Creation design pattern. Factory should be an interface and clients first either creates factory or get factory which later used to create objects.

## Example of static factory method in JDK

 Best Example of Factory method design pattern is valueOf() method which is there in String and wrapper classes like Integer and Boolean and used for type conversion i.e. from converting String to Integer or String to double in java..

Some more examples of factory method design pattern from JDK is :

valueOf() method which returns object created by factory equivalent to value of parameter passed.

getInstance() method which creates instance of Singleton class.

newInstance() method which is used to create and return new instance from factory method every time called.

getType() and newType() equivalent of getInstance() and newInstance() factory method but used when factory method resides in separate class.

### Problem which is solved by Factory method Pattern in Java

Whenever we talk about **object oriented language** it will based upon some concept like [abstraction](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html), [polymorphism](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html) etc and on that [encapsulation](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html) and delegation are important concept any design will be called good if task are delegated to different object and some kind of encapsulation is there.

Some time our application or framework will not know that what kind of object it has to create at run-time it knows only the interface or abstract class and as we know we can not create object of interface or abstract class so main problem is frame work knows **when** it has to create but don’t know **what kind** of object.

Whenever we create object using new() we violate **principle of programming for interface rather than implementation** which eventually result in inflexible code and difficult to change in maintenance. By using Factory design pattern in Java we get rid of this problem.

Another problem we can face is class needs to contain objects of other classes or class hierarchies within it; this can be very easily achieved by just using the new keyword and the class constructor. The problem with this approach is that it is a very hard coded approach to create objects as this creates dependency between the two classes.

So **factory pattern** solve this problem very easily by model an interface for creating an object which at creation time can let its subclasses decide which class to instantiate, Factory Pattern promotes loose coupling by eliminating the need to bind application-specific classes into the code. The **factory methods** are typically implemented as virtual methods, so this pattern is also referred to as the “**Virtual Constructor**”. These methods create the objects of the products or target classes.

### When to use Factory design pattern in Java

* Static Factory methods are common in frameworks where library code needs to create objects of types which may be sub classed by applications using the framework.
* Some or all concrete products can be created in multiple ways, or we want to leave open the option that in the future there may be new ways to create the concrete product.
* Factory method is used when Products don't need to know how they are created.
* We  can use factory pattern where we have to create an object of any one of sub-classes depending on the data provided

### Code Example of Factory Design Pattern in Java:

Let’s see an example of how factory pattern is implemented in Code.We have requirement to create multiple currency e.g. INR, SGD, USD and code should be extensible to accommodate new Currency as well. Here we have made Currency as interface and all currency would be concrete implementation of Currency interface. Factory Class will create Currency based upon country and return concrete implementation which will be stored in interface type. This makes code dynamic and extensible.

Here is complete **code example of Factory pattern in Java**:

interface Currency {

String getSymbol();

}

// Concrete Rupee Class code

class Rupee implements Currency {

@Override

public String getSymbol() {

return "Rs";

}

}

// Concrete SGD class Code

class SGDDollar implements Currency {

@Override

public String getSymbol() {

return "SGD";

}

}

// Concrete US Dollar code

class USDollar implements Currency {

@Override

public String getSymbol() {

return "USD";

}

}

// Factroy Class code

class CurrencyFactory {

public static Currency createCurrency (String country) {

if (country. equalsIgnoreCase ("India")){

return new Rupee();

}else if(country. equalsIgnoreCase ("Singapore")){

return new SGDDollar();

}else if(country. equalsIgnoreCase ("US")){

return new USDollar();

}

throw new IllegalArgumentException("No such currency");

}

}

// Factory client code

public class Factory {

public static void main(String args[]) {

String country = args[0];

Currency rupee = CurrencyFactory.createCurrency(country);

System.out.println(rupee.getSymbol());

}

}

### Advantage of Factory method Pattern in Java:

**Factory pattern in Java** is heavily used everywhere including JDK, open source library and other frameworks.In following are main advantages of using Factory pattern in Java:

1*) Factory method design pattern* decouples the calling class from the target class, which result in less coupled and highly cohesive code?

E.g.: JDBC is a good example for this pattern; application code doesn't need to know what database it will be used with, so it doesn't know what database-specific driver classes it should use. Instead, it uses factory methods to get Connections, Statements, and other objects to work with. Which gives you flexibility to change your back-end database without changing your DAO layer in case you are using ANSI SQL features and not coded on DBMS specific feature?

2) Factory pattern in Java enables the subclasses to provide extended version of an object, because creating an object inside factory is more flexible than creating an object directly in the client. Since client is working on interface level any time you can enhance the implementation and return from Factory.

3) Another benefit of using *Factory design pattern in Java* is that it encourages [consistency in Code](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html) since every time object is created using Factory rather than using different constructor at different client side.

4) Code written using Factory design pattern in Java is also [easy to debug](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html) and troubleshoot because you have a centralized method for object creation and every client is getting object from same place.

Some more advantages of factory method design pattern is:

1. **Static factory method** used in factory design pattern enforces use of Interface than implementation which itself a good practice. for example:

**Map** synchronizedMap = **Collections**.synchronizedMap(**new** **HashMap**());

2. Since static factory method have return type as Interface, it allows you to replace implementation with better performance version in newer release.

3. Another advantage of static factory method pattern is that they can cache frequently used object and eliminate duplicate object creation. Boolean.valueOf() method is good example which caches true and false boolean value.

4. Factory method pattern is also recommended by Joshua Bloch in Effective Java.

5 Factory method pattern offers alternative way of creating object.

6. Factory pattern can also be used to hide information related to creation of object.

That’s all on **Factory design patten in Java** for now. This is one of the most used patterns in Java library and different Java frameworks. Summary is try to use **Factory pattern** whenever you see an opportunity to encapsulate object creation code and see chance of creating different object in near future.

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

<http://javarevisited.blogspot.com.by/2012/06/builder-design-pattern-in-java-example.html>

Builder Design pattern in Java - Example Tutorial

**Builder design pattern in Java** is a creational pattern i.e. used to create objects, similar to [factory method design pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) which is also creational design pattern. Before learning any design pattern I suggest find out the problem a particular design pattern solves. Its been well said **necessity is mother on invention**. learning design pattern without facing problem is not that effective, Instead if you have already faced issues than its much easier to understand design pattern and learn how its solve the issue. In this Java design pattern tutorial we will first see what problem *Builder design pattern* solves which will give some insight on **when to use builder design pattern in Java,** which is also a [popular design pattern interview question](http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html) and then we will see example of Builder design pattern and pros and cons of using Builder pattern in Java.

## What problem Builder pattern solves in Java

[What is builder design pattern in Java with example](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)As I said earlier *Builder pattern* is a creational design pattern it means its solves problem related to object creation. Constructors in Java are used to create object and can take parameters required to create object. Problem starts when an Object can be created with **lot of parameters**, some of them may be **mandatory** and others may be **optional**. Consider a class which is used to create Cake, now you need number of item like egg, milk, flour to create cake. many of them are mandatory and some  of them are optional like cherry, fruits etc. If we are going to have [overloaded constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) for different kind of cake then there will be many constructor and even worst they will accept many parameter.

**Problems:**

1) too many constructors to maintain.

2) error prone because many fields has same type e.g. sugar and and butter are in cups so instead of 2 cup sugar if you pass 2 cup butter, your compiler will not complain but will get a buttery cake with almost no sugar with high cost of wasting butter.

You can partially solve this problem by creating Cake and then adding ingredients but that will impose another problem of **leaving Object on inconsistent state during building**, ideally cake should not be available until its created. Both of these problem can be solved by using Builder design pattern in Java. Builder design pattern not only improves readability but also reduces chance of error by adding ingredients explicitly and making object available once fully constructed.

By the way there are many design pattern tutorial already there in Javarevisited like [Decorator pattern tutorial](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html) and  [Observer pattern in Java](http://javarevisited.blogspot.com/2011/12/observer-design-pattern-java-example.html). If you haven’t read them already then its worth looking.

## Example of Builder Design pattern in Java

We will use same example of creating Cake using Builder design pattern in Java. here we have [static nested builder class](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) inside Cake which is used to create object.

**Guidelines for Builder design pattern in Java**

1) Make a static nested class called Builder inside the class whose object will be build by Builder. In this example its Cake.

2) Builder class will have exactly same set of fields as original class.

3) Builder class will expose method for adding ingredients e.g. sugar() in this example. each method will return same Builder object. Builder will be enriched with each method call.

4) Builder.build() method will copy all builder field values into actual class and return object of Item class.

5) Item class (class for which we are creating Builder) should have [private constructor](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html) to create its object from build() method and prevent outsider to access its constructor.

**public** **class** BuilderPatternExample {  
    
    **public** **static** **void** main(**String** args[]) {  
        
        *//Creating object using Builder pattern in java*  
        Cake whiteCake = **new** Cake.Builder().sugar(1).butter(0.5).  eggs(2).vanila(2).flour(1.5). bakingpowder(0.75).milk(0.5).build();  
        
        *//Cake is ready to eat :)*  
        **System**.out.println(whiteCake);  
    }  
}  
  
**class** Cake {  
  
    **private** **final** **double** sugar;   *//cup*  
    **private** **final** **double** butter;  *//cup*  
    **private** **final** **int** eggs;  
    **private** **final** **int** vanila;     *//spoon*  
    **private** **final** **double** flour;   *//cup*  
    **private** **final** **double** bakingpowder; *//spoon*  
    **private** **final** **double** milk;  *//cup*  
    **private** **final** **int** cherry;  
  
    **public** **static** **class** Builder {  
  
        **private** **double** sugar;   *//cup*  
        **private** **double** butter;  *//cup*  
        **private** **int** eggs;  
        **private** **int** vanila;     *//spoon*  
        **private** **double** flour;   *//cup*  
        **private** **double** bakingpowder; *//spoon*  
        **private** **double** milk;  *//cup*  
        **private** **int** cherry;  
  
        *//builder methods for setting property*  
        **public** Builder sugar(**double** cup){**this**.sugar = cup; **return** **this**; }  
        **public** Builder butter(**double** cup){**this**.butter = cup; **return** **this**; }  
        **public** Builder eggs(**int** number){**this**.eggs = number; **return** **this**; }  
        **public** Builder vanila(**int** spoon){**this**.vanila = spoon; **return** **this**; }  
        **public** Builder flour(**double** cup){**this**.flour = cup; **return** **this**; }  
        **public** Builder bakingpowder(**double** spoon){**this**.sugar = spoon; **return** **this**; }  
        **public** Builder milk(**double** cup){**this**.milk = cup; **return** **this**; }  
        **public** Builder cherry(**int** number){**this**.cherry = number; **return** **this**; }  
        
        
        *//return fully build object*  
        **public** Cake build() {  
            **return** **new** Cake(**this**);  
        }  
    }  
  
    *//private constructor to enforce object creation through builder*  
    **private** Cake(Builder builder) {  
        **this**.sugar = builder.sugar;  
        **this**.butter = builder.butter;  
        **this**.eggs = builder.eggs;  
        **this**.vanila = builder.vanila;  
        **this**.flour = builder.flour;  
        **this**.bakingpowder = builder.bakingpowder;  
        **this**.milk = builder.milk;  
        **this**.cherry = builder.cherry;         
    }  
  
    @**Override**  
    **public** **String** toString() {  
        **return** "Cake{" + "sugar=" + sugar + ", butter=" + butter + ", eggs=" + eggs + ", vanila=" + vanila + ", flour=" + flour + ", bakingpowder=" + bakingpowder + ", milk=" + milk + ", cherry=" + cherry + '}';  
  
    }   
    
}  
  
**Output:**  
Cake{sugar=0.75, butter=0.5, eggs=2, vanila=2, flour=1.5, bakingpowder=0.0, milk=0.5, cherry=0}

## Builder design pattern in Java – Pros and Cons

Live everything Builder pattern also has some disadvantages, but if you look at below, advantages clearly outnumber disadvantages of Builder design pattern. Any way here are few advantages and disadvantage of Builder design pattern for creating objects in Java.

**Advantages:**

1) more maintainable if number of fields required to create object is more than 4 or 5.

2) less error-prone as user will know what they are passing because of explicit method call.

3) more robust as only fully constructed object will be available to client.

**Disadvantages:**

1) verbose and code duplication as Builder needs to copy all fields from Original or Item class.

### When to use Builder Design pattern in Java

Builder Design pattern is a creational pattern and should be used when number of parameter required in constructor is more than manageable usually 4 or at most 5. Don't confuse with **Builder and Factory pattern** there is an obvious difference between Builder and Factory pattern, as Factory can be used to create different implementation of same interface but Builder is tied up with its Container class and only returns object of Outer class.

That's all on **Builder design pattern in Java**. we have seen why we need Builder pattern , what problem it solves, Example of builder design pattern in Java and finally when to use Builder patter with pros and cons. So if you are not using telescoping constructor pattern or have a choice not to use it than Builder pattern is way to go.

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

<http://java67.blogspot.sg/2013/07/decorator-design-pattern-in-java-real-life-example-tutorial.html>

Decorator Design Pattern in Java - Real life Example

Decorator Pattern is one of the famous Gang of Four (GOF) structural design pattern, which provides an alternative way of extending an object's functionality. It's different than traditional way of adding new functionality into object using Inheritance, since it's based on Composition and provides additional functionality at run time, as opposite to Inheritance, which adds new functionality at compile time. Decorator design pattern is introduced by famous [Gang of Four design pattern book](http://www.amazon.com/dp/0201633612/?tag=javamysqlanta-20), almost 2 decades ago. It's a time tested way of adding new functionalities into object. In this Java design pattern tutorial, we will learn Decorator design pattern by using it in a Java example. This is a best way of learning design pattern, followed you try it yourself to apply in similar scenarios. Decorator pattern is one of the popular design pattern along with Factory method pattern and [Singleton Pattern](http://java67.blogspot.sg/2012/08/what-is-singleton-pattern-in-java.html), and you can see it's usage even in JDK itself. Couple of classes from java.io package e.g. BufferedInputStream, LineNumberInputStream are good example of Decorator design pattern in Java.

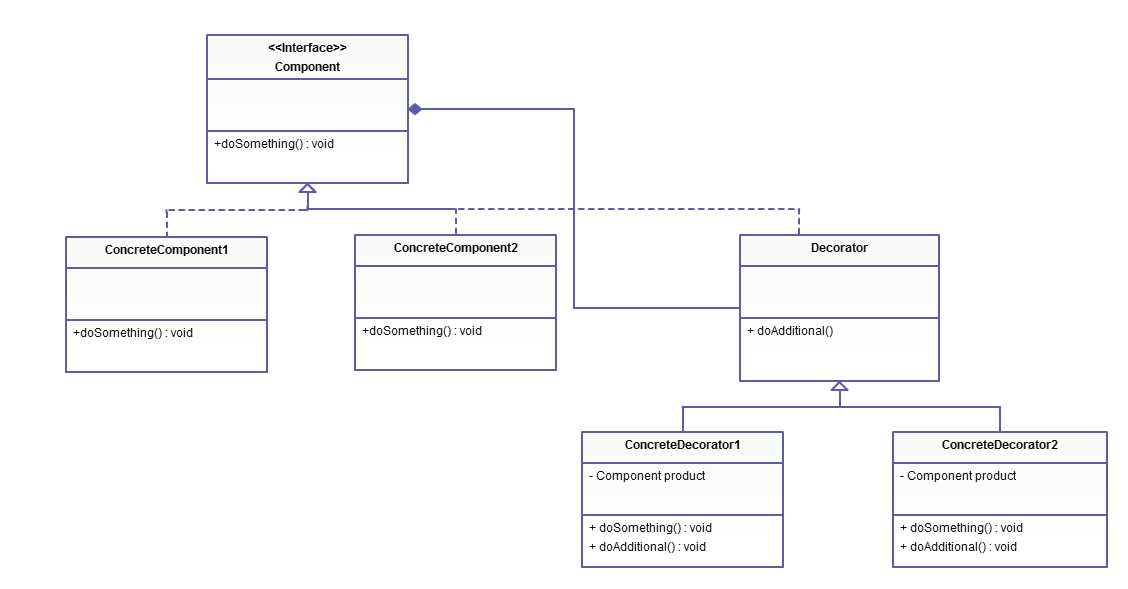
## Decorator design pattern in Java

[Real life example of Decorator design pattern in Java](http://3.bp.blogspot.com/-1lzFJzIgaHk/UF2Ci6kY5pI/AAAAAAAAAes/OYiM7r-DHzc/s1600/17.jpg)In order to show you, how to to implement Decorator pattern, let me first explain requirements. We need to create software for calculating price for a Sandwich, yummy... no? Since customer can customize sandwich by asking extra cheese or extra fillings, you also need to include cost of those items in final price of Sandwich. Since this customization can vary a lot among different customers and offering from a shop, creating classes for different types of Sandwich with different fillings or extras e.g. BrownBreadSandWithCheese or WhiteBreaSandwitchWithCheeseAndTomato will just clutter code with lots of endless small classes. Now this problem looks a natural fit for applying Decorator pattern, because we have a base object Sandwich, which can be decorated with extra cheese and fillings. By using [Decorator pattern](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html), you can extend functionality of Sandwich at runtime, based upon customer's request, which is impossible with Inheritance until you have a specific class for every possible customer request. This is one of the reason Why Composition is preferred over Inheritance in Object oriented design and particularly in Java. Now, let's see our class structure, We have an abstract class Sandwich, with abstract method price() and a concrete implementation class WhiteBreadSandwich, which cost $3.0. Now, in order to provide extra cheese, which obviously incur extra cost, we are going to use *Decorator design pattern*. We have a Decorator abstract class, which will act as base for Decorators called SandwichDecorator, and a concrete implementation of this as CheeseDecorator.

SandwichDecorator extends Sandwich, to be of same type as original object, which is getting decorated. This is a critical requirement of Decorator pattern, so that a decorated object can stand in place of original object i.e. it can be passed when a method expect original object. Decorator adds functionality before or after delegating task to original object, which means in this example price of a WhilteBreadSandwich with Cheese will be calculated by first calculating price of WhiteBreadSandwich and then price of Cheese. Finally, I have a class called SandwichMaker, which will make delicious sandwich for you :) I mean it will test whole program and demonstrate how Decorator pattern adds new functionality on the fly at runtime. I have also used BigDecimal to represent money instead of double primitive to follow best practices suggested in [Effective Java](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20). If you don't know why, read this Java mistake about using double to represent price

### UML diagram of Decorator Pattern

Here is the UML class diagram of decorator design pattern. You can see that we have a Component interface, which is used to create ConcreteComponents. In our real world example, Sandwich is component interface and WhiteBreadSandwich is concrete component. Now if we want to add additional feature to Component interface, we create another interface called Decorator, which inherit Component interface, this is very important, because it allows you to pass instance of Decorator to any method which accepts instance of Component interface. Now you can create implementation of decorator, which has both functionality provided by original interface and new feature provided by decorator. This way you can add new features in existing class hierarchy without modifying tried and tested code. This is another reason programmers says [composition is better than Inheritance](http://javarevisited.blogspot.com/2013/06/why-favor-composition-over-inheritance-java-oops-design.html). 



### Sample Code of Decorator Design Pattern in Java

Here is complete Java program to demonstrate how you can implement decorator pattern in Java. You can use this sample code to add more features and create new classes. If you are using Eclipse IDE, just create a Java project, select that project in package explorer and copy the code their, it will automatically create right packages and Java classes.

**Sandwich.java**

**import** **java.math.BigDecimal**;

/\*\*

\* Base class for all types of Sandwich, cost method is abstract because

\* different sandwiches has different price.

\*

\* @author Javin Paul

\*/

**public** **abstract** **class** **Sandwich** {

**protected** String description = "Sandwich";

**public** String **getDescription**(){

**return** description;

}

**public** **abstract** BigDecimal **price**();

}

**WhiteBreadSandWich.java**

**import** **java.math.BigDecimal**;

/\*\*

\* A Concrete implementation of abstract Sandwich class, which represent a WhiteBread

\* Sandwich, whose price is 3.0$.

\*

\* @author Javin Paul

\*/

**public** **class** **WhiteBreadSandWich** **extends** Sandwich {

**public** **WhiteBreadSandWich**(String desc){

description = desc;

}

**@Override**

**public** BigDecimal **price**() {

**return** **new** **BigDecimal**("3.0");

}

}

**SandWichDecorator.java**

/\*\*

\* Base class for Decorators, this class inherit from Sandwich, so that

\* it can be of same type, which is required to pass decorators where

\* original object is expected. Later, this class will also come handy

\* to provide common functionalities to Decorators.

\*

\* @author

\*/

**public** **abstract** **class** **SandWichDecorator** **extends** Sandwich {

**@Override**

**public** **abstract** BigDecimal **price**();

}

**CheeseDecorator.java**

**import** **java.math.BigDecimal**;

/\*\*

\* A Decorator class, which adds cheese (new functionality) into Sandwich object.

\* This Decorator class modifies price() and getDescritption() method to implement

\* new behaviour.

\*

\* @author

\*/

**public** **class** **CheeseDecorator** **extends** SandWichDecorator{

Sandwich currentSandwich;

**public** **CheeseDecorator**(Sandwich sw){

currentSandwich = sw;

}

**@Override**

**public** String **getDescription**(){

**return** currentSandwich.getDescription() + ", Cheese";

}

**@Override**

**public** BigDecimal **price**() {

**return** currentSandwich.price().add(**new** BigDecimal("0.50"));

}

}

**SandwichMaker.java**

/\*\*

\* Test class to demonstrate How Decorator Pattern in Java work together. This class

\* first creates a Sandwich and decorates it with extra cheese. This is nice example

\* of how to provide new functionalities to an object at runtime using Decorator Pattern.

\*

\* @author Javain Paul

\*/

**public** **class** **SandwichMaker** {

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

Sandwich mySandwich = **new** WhiteBreadSandWich("White bread Sandwich");

System.out.printf("Price of %s is $%.2f %n", mySandwich.getDescription(),

mySandwich.price());

//adding extra cheese using Decorator Pattter

mySandwich = **new** CheeseDecorator(mySandwich);

System.out.printf("Price of %s is $%.2f %n", mySandwich.getDescription(),

mySandwich.price());

}

}

**Output:**

Price of White bread Sandwich is $3.**00**

Price of White bread Sandwich, Cheese is $3.**50**

.

### Key things about Decorator Design Pattern

Now, we have seen an example of decorator pattern in Java, we can quickly summarize few important things which are worth remembering while implementing or applying decorator pattern or even to answers design pattern questions like *When to use Decorator design pattern in Java*.

1) Decorator must be of same type of object, which they are decorating. This can be achieved either by implementing interface of object, or by extending abstract class of original class.

2) Decorator is based on Composition, which means it needs original object to decorate it. This is achieved by creating constructor on decorator class which accept a base type of original object. e.g. in this example constructor of CheeseDecorator accepts Sandwich object. Decorator pattern is also a good example of [Open Closed design principle](http://javarevisited.blogspot.com/2011/11/great-example-of-open-closed-design.html), which is one of the key principle from Uncle Bob's SOLID design principles.

3) Decorator class adds new functionality before or after delegating task to original object. In this example, price of Decorator i.e. cheese is included after calculating price of White Bread Sandwich.

4) Remember, Decorator design pattern only affects objects at runtime, it doesn't affect class. You should use DECORATOR PATTERN when your intent is to add new functionality at runtime (i.e. a customer order, where you only know about order details, one it placed).

5) There is *one disadvantage of Decorator pattern* as well, it adds lots of small classes in code base, remember overwhelming number of classes in java.io package. Though, once you know that which classes are main classes, and which are decorators, you tend to get better understanding of overall structure. UML diagrams certain helps in this case.

That's all on **Decorator design pattern in Java and Object oriented design**. I must say, this is one of the must know design patterns for a senior Java developers, it's general purpose and has lot's of use cases as well.

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

[How to use Decorator pattern in Java with Example](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html)

[Benefits of Observer Pattern in Java with Example](http://javarevisited.blogspot.com/2011/12/observer-design-pattern-java-example.html)

[10 best practices to follow while writing code comments](http://javarevisited.blogspot.com/2011/08/code-comments-java-best-practices.html)

[10 tips to debug java program in Eclipse](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html)

[Code review checklist for programmer](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html)

[How to write production quality code](http://javarevisited.blogspot.com/2011/09/how-to-write-production-quality-code.html)

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