Design Patterns Interview Questions

1) What is a software design pattern?

A design **pattern** is a solution to a general software **problem** within a particular **context**.

- **Context :** A recurring set of situations where the pattern applies.
- **Problem :** A system of forces (goals and constraints) that occur repeatedly in this context.
- **Solution :** A description of communicating objects and classes (collaboration) that can be applied to resolve those forces.

Design patterns capture solutions that have evolved over time as developers strive for greater flexibility in their software. Whereas class libraries are reusable source code, and components are reusable packaged objects, patterns are generic, reusable design descriptions that are customized to solve a specific problem. The study of design patterns provides a common vocabulary for communication and documentation, and it provides a framework for evolution and improvement of existing patterns.

2) Why is the study of patterns important?

As initial software designs are implemented and deployed, programmers often discover improvements which make the designs more adaptable to change. Design patterns capture solutions that have evolved over time as developers strive for greater flexibility in their software, and they document the solutions in a way which facilitates their reuse in other, possibly unrelated systems. Design patterns allow us to reuse the knowledge of experienced software designers.

Moreover, the study of design patterns provides a common vocabulary for communication and documentation, and it provides a framework for evolution and improvement of existing patterns. As an analogy, consider that during a discussion among programmers, the words "stack" and "tree" can be used freely without explanation. Software developers understand fundamental data structures such as a "stack" because these data structures are well-documented in textbooks and are taught in computer science courses. The study of design patterns will have a similar (but more profound) effect by allowing designers to say "composite pattern" or "observer pattern" in a particular design context, without having to describe all classes, relationships, and collaborations which make up the pattern. Patterns raise the level of abstraction when discussing and documenting software designs.

3) How do I document a design pattern?

A pattern description must address the following major points:

- **Pattern Name and Classification :** A short, meaningful name for the pattern, usually only one or two words. Names provide a vocabulary for patterns, and they have implied semantics – choose names carefully. Following the GoF book, we can also group patterns into higher level classifications such as creational, structural, and behavioral patterns.
- **Problem :** A general description of the problem context and the goals and constraints that occur repeatedly in that context. A concrete motivational scenario can be used to help describe the problem. The problem description should provide guidance to assist others in recognizing situations where the pattern can be applied.

- **Solution :** The classes and/or objects that participate in the design pattern, their structure (e.g., in terms of a UML class diagram), their responsibilities, and their collaborations. The solution provides an abstract description that can be applied in many different situations. Sample Code in an object-oriented language can be used to illustrate a concrete realization of the pattern.
- **Consequences :** A discussion of the results and tradeoffs of applying the pattern. Variations and language-dependent alternatives should also be addressed.
- **Known Uses :** Examples of the pattern in real systems. Look for applications of the pattern in language libraries and frameworks, published system descriptions, text books, etc. Not every good solution represents a pattern. A general rule of thumb is that a candidate pattern (also called a "proto-pattern") should be discovered in a minimum of three existing systems before it can rightfully be called a pattern.

The following quote by Robert Martin highlights the importance of providing pattern descriptions: "The revolutionary concept of the GoF book is not the fact that there are patterns; it is the way in which those patterns are documented. ... Prior to the GoF book, the only good way to learn patterns was to discover them in design documentation, or (more probably) code."

4) Where can I learn more about design patterns?

The best place to start is the seminal work by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (collectively known as the "Gang of Four" or simply "GoF") entitled *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).
Warning: This book is not light reading. From the Preface: "Don't worry if you don't understand this book completely on the first reading. We didn't understand it all on the first writing."
It is, however, a book which wears well over time, and it is definitely worth the effort required to work through it.
Beyond the GoF book, consider the list of references in the Design Patterns section of this bibliography on object technology, plus the following web links:

- Design Patterns Home Page
- Huston Design Patterns
- Brad Appleton's Software Patterns Links
- Cetus Patterns Links

5) What is an example of a design pattern?

Following the lead of the "Gang of Four" (GoF), design pattern descriptions usually contain multiple sections including

- Intent
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

A complete discussion of even a small pattern is beyond the scope of a simple FAQ entry, but it is possible to get the idea by examining an abbreviated discussion of one of the

simplest and most easily understood patterns. Consider the Singleton pattern, whose intent reads as follows:
Intent: Ensure that a class has one instance, and provide a global point of access to it.
Almost every programmer has encountered this problem and formulated an approach for solving it in a general way – some solutions are better than others. The solution offered by the GoF would look something like the following when coded in Java.

```
public class Singleton
{
   private static Singleton instance = null;
   public static Singleton getInstance()
   {
     if (instance == null)
       instance = new Singleton();
     return instance;
   }

   protected Singleton() { ... }
   // possibly another constructor form

   public void someMethod() { ... }

   //... other methods
 }
```

The programmer would access the single instance of this class by writing something similar to

```
Singleton.getInstance().someMethod()
```

or similar to

```
Singleton s = Singleton.getInstance();
s.method1();
...
s.method2();
...
```

For a more complete discussion of the Singleton pattern, see the chapter "Singleton" in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by the "Gang of Four" (Addison-Wesley, 1995), or the chapter "Singleton" in the book *Patterns in Java, Volume 1* by Mark Grand (John Wiley & Sons, 1998). For information about variations on the Singleton Pattern, see the chapter entitled "To Kill a Singleton" in the book *Pattern Hatching: Design Patterns Applied* by John Vlissides or the article *"Implementing the Singleton Pattern in Java"* by Rod Waldhoff.

6) Calendar is an abstract class. The getInstance() method tries to instantiate GregorianCalendar() i.e., parent instantiating a derived class. This looks Non-OO? Ex: Calendar a=Calendar.getInstance(); Can somebody explain why is it so?

The Calender class is an abstact class, true, however,the point you missed is that the getInstance() returns the " Calendar using the default timezone and locale. " , in your case, the GregorianCalender a class that IS a Calender (a Suzuki IS a car, a 747 IS a plane..the standard OO terminology. So what you get is a class that does some specialized work based on the default locale. Other methods

```
public static synchronized Calendar getInstance(TimeZone zone,Locale aLocale)
public static synchronized Calendar getInstance(TimeZone zone)
```

return Calenders for specific timezones and locales. The closest parallel is possibly the Factory Method design pattern.

7) What major patterns do the Java APIs utilize?

Design patterns are used and supported extensively throughout the Java APIs. Here are some examples:

- The Model-View-Controller design pattern is used extensively throughout the Swing API.
- The getInstance() method in java.util.Calendar is an example of a simple form of the Factory Method design pattern.
- The classes java.lang.System and java.sql.DriverManager are examples of the Singleton pattern, although they are not implemented using the approach recommended in the GoF book but with static methods.
- The Prototype pattern is supported in Java through the clone() method defined in class Object and the use of java.lang.Cloneable interface to grant permission for cloning.
- The Java Swing classes support the Command pattern by providing an Action interface and an AbstractAction class.
- The Java 1.1 event model is based on the observer pattern. In addition, the interface java.util.Observable and the class java.util.Observer provide support for this pattern.
- The Adapter pattern is used extensively by the adapter classes in java.awt.event.
- The Proxy pattern is used extensively in the implementation of Java's Remote Method Invocation (RMI) and Interface Definition Language (IDL) features.
- The structure of Component and Container classes in java.awt provide a good example of the Composite pattern.
- The Bridge pattern can be found in the separation of the components in java.awt (e.g., Button and List), and their counterparts in java.awt.peer.

8) How can I make sure at most one instance of my class is ever created?

This is an instance where the Singleton design pattern would be used. You need to make the constructor private (so nobody can create an instance) and provide a static method to get the sole instance, where the first time the instance is retrieved it is created:

```
public class Mine {
   private static Mine singleton;
   private Mine() {
   }
   public static synchronized Mine getInstance() {
     if (singleton == null) {
       singleton = new Mine();
     }
     return singleton;
   }
   // other stuff...
 }
```

9) When would I use the delegation pattern instead of inheritence to extend a class's behavior?

Both delegation and inheritance are important concepts in object-oriented software design, but not everyone would label them as patterns. In particular, the seminal book on design patterns by the "Gang of Four" contains a discussion of inheritance and delegation, but the authors do not treat these topics as specific patterns. It is reasonable to think of them as design concepts which are more general than specific design patterns.
Inheritance is a relationship between two classes where one class, called a subclass in this context, inherits the attributes and operations of another class, called its superclass. Inheritance can be a powerful design/reuse technique, especially when it is used in the context of the Liskov Substitution Principle. (The article by Robert Martin at http://www.objectmentor.com/publications/lsp.pdf provides an excellent explanation of the ideas behind Barbara Liskov's original paper on using inheritance correctly.) The primary advantages of inheritance are

1. it is directly supported by object-oriented languages, and
2. it provides the context for polymorphism in strongly-typed object-oriented languages such as C++ and Java.

But since the inheritance relationship is defined at compile-time, a class can't change its superclass dynamically during program execution. Moreover, modifications to a superclass automatically propagate to the subclass, providing a two-edged sword for software maintenance and reuse. In summary, inheritance creates a strong, static coupling between a superclass and its subclasses.
Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its delegate. Delegation can also a powerful design/reuse technique. The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time. But unlike inheritance, delegation is not directly supported by most popular object-oriented languages, and it doesn't facilitate dynamic polymorphism.
As a simple example, consider the relationship between a Rectangle class and a Window class. With inheritance, a Window class would inherit its rectangular properties from class

Rectangle. With delegation, a Window object would maintain a reference or pointer to a Rectangle object, and calls to rectangle-like methods of the Window object would be delegated to corresponding methods of the Rectangle object.
Now let's consider a slightly more complex example. Suppose employees can classified based on how they are paid; e.g., hourly or salaried. Using inheritance, we might design three classes: an Employee class which encapsulates the functionality common to all employees, and two subclasses HourlyEmployee and SalariedEmployee which encapsulates pay-specific details. While this design might be suitable for some applications, we would encounter problems in a scenario where a person changes, say from hourly to salaried. The class of an object and the inheritance relationship are both static, and objects can't change their class easily (but see the State pattern for tips on how to fake it).
A more flexible design would involve delegation – an Employee object could delegate pay-related method calls to an object whose responsibilities focused solely on how the employee is paid. In fact, we might still use inheritance here in a slightly different manner by creating an abstract class (or interface) called PayClassification with two subclasses HourlyPayClassification and SalariedPayClassification which implement classification-specific computations. Using delegation as shown, it would be much easier to change the pay classification of an existing Employee object.
This second example illustrates an important point: In implementing delegation, we often want the capability to replace the delegate with another object, possibly of a different class. Therefore delegation will often use inheritance and polymorphism, with classes of potential delegates being subclasses of an abstract class which encapsulates general delegate responsibilities.
One final point. Sometimes, the choice between delegation and inheritance is driven by external factors such as programming language support for multiple inheritance or design constraints requiring polymorphism. Consider threads in Java. You can associate a class with a thread in one of two ways: either by extending (inheriting) directly from class Thread, or by implementing the Runnable interface and then delegating to a Thread object. Often the approach taken is based on the restriction in Java that a class can only extend one class (i.e., Java does not support multiple inheritance). If the class you want to associate with a thread already extends some other class in the design, then you would have to use delegation; otherwise, extending class Thread would usually be the simpler approach.

10) Which patterns were used by Sun in designing the Enterprise JavaBeans model?

Many design patterns were used in EJB, and some of them are clearly identifiable by their naming convention. Here are several:

1. **Factory Method:** Define a interface for creating classes, let a subclass (or a helper class) decide which class to instantiate.
   This is used in EJB creation model. EJBHome defines an interface for creating the EJBObject implementations. They are actually created by a generated container class. See InitialContextFactory interface that returns an InitialContext based on a properties hashtable.
2. **Singleton:** Ensure a class has only one instance, and provide a global point of access to it.
   There are many such classes. One example is javax.naming.NamingManager
3. **Abstract Factory:** Provide an interface for creating families of relegated or dependent objects without specifying their concrete classes.
   We have interfaces called InitialContext, InitialContextFactory. InitialContextFactory has methods to get InitialContext.

4. **Builder:** Separate the construction of a complex factory from its representation so that the same construction process can create different representations. InitialContextFactoryBuilder can create a InitialContextFactory.
5. **Adapter:** Convert the interface of a class into another interface clients expect. In the EJB implementation model, we implement an EJB in a class that extends SessionBean or a EntityBean. We don't directly implement the EJBObject/home interfaces. EJB container generates a class that adapts the EJBObject interface by forwarding the calls to the enterprise bean class and provides declarative transaction, persistence support.
6. **Proxy:** Provide a surrogate for other object to control access to it. We have remote RMI-CORBA proxies for the EJB's.
7. **Memento:** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later. Certainly this pattern is used in activating/passivating the enterprise beans by the container/server.

11) What is an analysis pattern?

An analysis pattern is a software pattern not related to a language or implementation problem, but to a business domain, such as accounting or health care. For example, in health care, the patient visit activity would be subject to a number of patterns.
There is a good overview from an OOPSLA '96 presentation at
http://www.jeffsutherland.org/oopsla96/fowler.html
A good text would be: Martin Fowler's Martin *Analysis Patterns : Reusable Object Models*, ISBN: 0201895420, published by Addison-Wesley.
In summary, analysis patterns are useful for discovering and capturing business processes.

12) What are the differences between analysis patterns and design patterns?

Analysis pattern are for domain architecture, and design pattern are for implementation mechanism for some aspect of the domain architecture. In brief, analysis pattern are more high level and more (end-user) functional oriented.

13) How does "Extreme Programming" (XP) fit with patterns?

Extreme Programming has a large emphasis on the concept of refactoring: Writing code once and only once.
Patterns, particularly the structural patterns mentioned by the Gang of Four, can give good pointers about how to acheive that goal.
(XP states when and where factoring should happen, patterns can show you how.)

14) What is the disadvantage of using the Singleton pattern? It is enticing to use this pattern for all the classes as it makes it easy to get the reference of the singleton object.

The intent of the Singleton pattern is to ensure a class has only one instance and to provide a global point of access to it. True, the second part about providing a global point of access is enticing, but the primary disadvantage of using the Singleton pattern for all classes is related to the first part of the intent; i.e., that it allows only one instance of the class. For most classes in an application, you will need to create multiple instances. What purpose would a Customer class serve if you could create only one Customer object?

15) How do you write a Thread-Safe Singleton?

I have written plenty of non-thread-safe Singletons but it wasn't until recently when I tracked it down that I realized that thread-safety could be a big problem.
The problem is that in the typical Singleton implementation (at least the ones I've seen) there is the ability to create multiple versions of the single *instance*...I know, "But How?". Well, in the `getInstance()` call the instance is checked for null, and then immediately constructed if it is null, and then the instance is returned.
The problem is that the thread (Ta) making the call could swap-out immediately after checking for a null. A subsequent thread (Tb) could then make a call to get the instance and construct an instance of the Singleton. When the original thread (Ta) is then swapped back in, it would construct and return a completely separate object. BAD KITTY!
The following code snippet shows an example of a thread-safe Singleton.

```
package com.jgk.patterns.singleton;
public class JGKSingleton {

   /* Here is the instance of the Singleton */
   private static JGKSingleton instance_;
   /* Need the following object to synchronize */
   /* a block */
   private static Object syncObject_;
   /* Prevent direct access to the constructor
   private JGKSingleton() {
     super();
   }

   public static JGKSingleton getInstance() {
     /* in a non-thread-safe version of a Singleton */
     /* the following line could be executed, and the */
     /* thread could be immediately swapped out */
     if (instance_ == null) {
         synchronized(syncObject_) {
           if (instance_ == null) {
             instance_ = new JGKSingleton();
           }
         }
     }
     return instance_;
```

```
        }
    }
```

NOTE: The 2nd check for `if (instance_ == null)` *is needed to avoid making another unnecessary construct.*
Don't let this byte you! ;-)

16) What is the Reactor pattern?

The new book "Pattern-oriented Software Architecture Volume 2" ISBN 0471606952 has a chapter on the Reactor pattern. It falls under the general category of "Event Handling Patterns". To quote the leading bit of the chapter,
"The Reactor architectural pattern allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients"
It is used in a synchronous manner, so that if the callback you delegate the event to takes a while to complete you will run into problems with scalability.

17) What are Process Patterns?

Basically process patterns define a collection of best practices, techniques, methods for developing object-oriented software.
A good reference site is by Scott Ambler. He also has two books on the topic plus a white paper on the subject you can download.
http://www.ambysoft.com/processPatternsPage.html.

18) How and where did the concept of design patterns get started?

Work on patterns has been influenced by the works of Christopher Alexander who published on topics related to urban planning and building architecture in the late 1970s. The history of patterns for software design began in the late 1980s and reached an important milestone with the publishing of the first book fully dedicated to this subject by the "Gang of Four", Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software). In conjunction, the emergence of object-oriented software development fostered the work on other topics related to design patterns such as application frameworks, analysis patterns, language idioms, and so on.

19) Where can I find good examples of the Prototype pattern?

The prototype pattern is actually quite simple to implement in Java.
Recall that the idea of prototype is that you are passed an object and use that object as a template to create a new object. Because you might not know the implementation details of the object, you cannot create a new instance of the object and copy all of its data. (Some of the data may not be accessible via methods). So you ask the object *itself* to give you a copy of itself.
Java provides a simple interface named Cloneable that provides an implementation of the Prototype pattern. If you have an object that is Cloneable, you can call its clone() method to create a new instance of the object with the same values.

The trick here is that you must override the clone() method to increase its visibility, and just call super.clone(). This is because the implementation of clone(), defined in java.lang.object, is protected. For example:

```
public class CopyMe implements Cloneable {
   public Object clone() {
     return super.clone();
   }
}
```

Note that Cloneable is an empty interface! It merely acts as a tag to state that you really want instance of the class to be cloned. If you don't implement Cloneable, the super.clone() implementation will throw a CloneNotSupportedException.
The Object implementation of clone() performs a *shallow copy* of the object in question. That is, it copies the values of the fields in the object, but not any actual objects that may be pointed to. In other words, the new object will point to the same objects the old object pointed to.
As an example of using the cloning:

```
CopyMe thing = new Copyme();
CopyMe anotherThing = (Copyme)thing.clone();
```

This example is pretty trivial, but the real power comes when you don't know what you're actually cloning.
For example, suppose you define an interface that represents a customer:

```
public interface Customer extends Cloneable {
   public Object clone(); // require making it public!
   public String getName();
   public void setName(String name);
   ...
}
```

You might have several different implementations of this interface, possibly storing data in a file, database, or using EJB Entity beans. If a shallow copy of the data is sufficient to represent a copy of the object, Java's clone() method works great.
You might have a method that needs to make a copy of the data to store it in a Hashtable, for example:

```
public void storeCustomer(Customer customer) {
   Customer copy = (Customer)customer.clone();
   dataStore.put(copy);
}
```

Note that this method knows *nothing* about what type of customer we're getting. This pattern will work for *any* actual type of Customer, no matter how the data is stored. For example:

```
FileBasedCustomer c1 = new FileBasedCustomer(...);
RDBMSBasedCustomer c2 = new RDBMSBasedCustomer(...);
EJBBasedCustomer c3 = new EJBBasedCustomer(...);

manager.storeCustomer(c1);
```

```
manager.storeCustomer(c2);
manager.storeCustomer(c3);
```

20) What are Anti-Patterns?

There isn't really a "clean-cut" definition out there just yet, unlike Design Patterns. Basically, as Design Patterns (and more particularly, Process Patterns) try to codify a standard vocabulary for working solutions to problems that reappear frequently, Anti-Patterns represent an effort to define and classify reoccuring non-solutions, i.e., things that lots of projects do that fail to yield a solution, or actually prevent a project from working or being finished.
The most basic example I can think of is "The Hammer", inspired by the old addage, "If your only tool is a hammer, everything looks like a nail" (or the variation, "If your only tool is a hammer, everything looks like your left thumb." Hammer describes a regular, reoccuring problem in inexperienced engineers (particularly those who've only used one language for the bulk of their carreer so far), that of trying to force-feed all problems into the solutions they already know.
http://www.antipatterns.com/ has more information.

21) What patterns are particularly useful in building networked applications?

I suggest starting with Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects (POSA2). POSA2 specifically brings together 17 interrelated patterns addressing *Service Access and Configuration, Event Handling, Synchronization, and Concurrency*.
The patterns and many of the examples in POSA2 come primarily from the design and implementation of the ACE framework.

22) Are there any good Java-specific patterns books available?

The Java-specific patterns books are:

- Java Design Patterns: A Tutorial
- Patterns in Java, Volume 1
- Patterns in Java, Volume 2
- Concurrent Programming in Java , Second Edition: Design Principles and Patterns
- SanFrancisco Design Patterns

As far as the good part of the question.... Doug Lea's *Concurrent Programming* book is probably the best of the bunch. However, its patterns are specific to concurrent programming. Many people don't like the quality of Mark Grand's two *Patterns in Java* books. [They are rated 3 and 2 stars at Amazon, respectively]. The first printing of the Cooper tutorial book was riddled with errors. If you get that, be sure to get at least the second printing. [Look on last line of page before TOC.] The SanFrancisco book is definitely good, but I'm not sure how good if you aren't using SF.

23) What are Collaboration Patterns?

Collaboration Patterns are repeatable techniques used by teams of people to help them work together (collaborate). Ellen Gottesdiener of EBG Consulting has created these patterns in order to help facilitate good teamwork. These patterns really have nothing to do with object-oriented development or Java, besides the fact that they can help with requirements gathering or CRC design sessions. In a nutshell, Collaboration Patterns are techniques to help make meetings useful.

24) Is it correct from a design point of view to make an object both an Observer and Observable at the same time?

Yes, and this can be the preferred pattern in some cases.
For example, suppose you were writing a supply chain management system for a retail chain. Each store object in your system generates item-sold events; when the chain generates enough of these for a particular product, a buyer object generates a purchase order for more of the product. However, the buyer object has no particular interest in individual item sold events. Instead, the buyer (Observer) registers to receive out-of-stock events from the warehouse (Observable); the warehouse, as Observer, registers with the individual stores (Observables) to receive item-sold events. Thus, the warehouse is both Observer and Observable. (Please note that this is a synthetic example, and probably not the way to organize a supply chain.)
Another reason to use one or more central Observer-Observable object in between ultimate Observers and Observables is to fully decouple the two. In some cases, Observer and Observable may exist on different machines, and may rely on the central Observer-Observable to hide this complication.
A good source for more details is the Publisher-Subscriber section of Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns.*

25) How can I maintain a single instance of an object in an applet?

In start(), instead of always creating a new object, return the existing one if it exists or create a new one if it doesn't.

26) What is the best way to generate a universally unique object ID? Do I need to use an external resource like a file or database, or can I do it all in memory?

I need to generate unique id's that will be used for node 'ID' attribute values within XML documents. This id must be unique system-wide. The generator must be available to a number of servlets that add various node structures to my XML docs as a service. What is the best way to tackle this? The 'possible' ways I can see:

- Keep the maximum ID value in a flat-file where the service would read it upon start-up and increment it. Upon shutdown or failure, it would write the latest max id to the file.
- Calculate the max id by searching the XML itself. This will be tougher since XML requires an alpha-numeric value (not strictly numeric).

- Use a database (MySQL) with a two-field table where one field is the incremental counter.

I just have this feeling that none of the above are the most efficient ways of doing this. Regards, -Andy]

There is an additional way to do that that doesn't rely on an external file (or database) like the one you have presentred. If has been presented in the EJB Design Patterns book, written by Floyd Marinescu, and available in a pdf format for free from the given link.
The suggested solution is based on the UUID for EJB pattern, that comes out from this question:
How can universally unique primary keys can be generated in menory without requiring a database or a singleton?
Without enetring in the specifics (you can fully check out the pattern by reading the appropriate chapter), the solution is to generate a 32 digit key, encoded in hexadecimal composed as follows:

1. Unique down to the millisecond. Digits 1-8 are are the hex encoded lower 32 bits of the System.currentTimeMillis() call.
2. Unique across a cluster. Digits 9-16 are the encoded representation of the 32 bit integer of the underlying IP address.
3. Unique down to the object in a JVM. Digits 17-24 are the hex representation of the call to System.identityHashCode(), which is guaranteed to return distinct integers for distinct objects within a JVM.
4. Unique within an object within a millisecond. Finally digits 25-32 represent a random 32 bit integer generated on every method call using the cryptographically strong java.security.SecureRandom class.

27) Is there some kind of Design pattern which would make it possible to use the Same code base in EJB and non EJB context?

A good suggestion would be using Delegation

```
class PieceOfCode {
   public Object myMethod() {}
}
class EJBImpl ... {
   PieceOfCode poc = new PieceOfCode();
   public Object myMethod() {
     return poc.myMethod();
   }
}
```

This should not be a violation of EJB specs, since EJBs can use simple java classes for their use. Think about *Dependant Objects* and so on.

28) What is session facade?

Session facade is one design pattern that is often used while developing enterprise applications.
It is implemented as a higher level component (i.e.: Session EJB), and it contains all the iteractions between low level components (i.e.: Entity EJB). It then provides a single

interface for the functionality of an application or part of it, and it decouples lower level components simplifying the design.
Think of a bank situation, where you have someone that would like to transfer money from one account to another.
In this type of scenario, the client has to check that the user is authorized, get the status of the two accounts, check that there are enough money on the first one, and then call the transfer. The entire transfer has to be done in a single transaction otherwise is something goes south, the situation has to be restored.
As you can see, multiple server-side objects need to be accessed and possibly modified. Multiple fine-grained invocations of Entity (or even Session) Beans add the overhead of network calls, even multiple transaction. In other words, the risk is to have a solution that has a high network overhead, high coupling, poor reusability and mantainability.
The best solution is then to wrap all the calls inside a Session Bean, so the clients will have a single point to access (that is the session bean) that will take care of handling all the rest. Obviously you need to be very careful when writing Session Facades, to avoid the abusing of it (often called "God-Bean").
For a detailed description of this pattern, check this page: Core J2EE Patterns - Session Facade or get Floyd Marinescu's EJB Design Patterns, in PDF format.

29) How is JDO different from VO ?

JDO is a persistence technology that competes against entity beans in enterprise application development. It allows you to create POJOs (plain old java objects) and persist them to the database - letting JDO take care of the storage.
Value objects, on the other hand, represent an abstract design pattern used in conjuction with entity beans, jdbc, and possibly even JDO to overcome commonly found isolation and transactional problems in enterprise apps. Value objects alone do not allow you to persist objects - they are simple data holders used to transfer data from the database to the client and back to the database.
Side note: I know that many books out there still refer to these data holders as *value objects* but the correct term is *DTO: data transfer objects*. Value objects refer to objects that hold a value. A good example of this java.lang.Integer object which holds an int.

30) How can I implement the MVC design pattern using JSP?

The MVC (Model View Controller) design pattern is a pattern/architecture that can be used by GUI's. It seperates the application's data, user interface and control logic into three separate entities. This ensures the system will be more maintainable in the future as changes to one component will not have an affect on the others.
The MVC pattern also conforms with the JSP Model 2 architecture.
The MVC pattern can be easily implemented in web applications using JSTL core, JSP, Servlets and JavaBeans.
JSTL makes it easy for HTML designers to use dynamic data in their pages without having to learn in depth java. The tags are in a recognizable HTML like format meaning a smaller learning curve than taking on the JSP specification after taking on the Java specification.
JSTL also makes it easy for web developers who are developing all aspects of the application in helping you keep the content separate from the display by keeping your JSP clean from any Java code. Since JSTL and its EL (expression Language) are really only suited for accessing data, it forces you to use a MVC pattern so long as you keep all JSP and Java syntax/code out of the JSP pages.

A common scenario might look like this, a user sends a request to a server. The request is handled by a Servlet (the controller) which will initialize any JavaBeans (the model) required to fulfill the user's request. The Servlet (the controller) will then forward the request, which contains the JavaBeans (the model), to a JSP (the view) page which contains only HTML and JSTL syntax. The JSTL will format the data into a human readable format before being sent back to the user. Thus completing the full MVC process.
JSTL also has a tag library for XML processing, this could be used in an MVC environment as you could replace the JavaBeans with an XML document to describe your data, so long as the XML is still constructed in a controller such as a Servlet before being passed to the JSP (the view).
JSTL also has a tag library for database access, using this in your JSP (the view) pages would NOT comply to the MVC pattern as there will be no separation between the model and the view.