Java-Success.com

Register | Login | Logout | Contact Us

Industrial strength Java/JEE Career Companion to open more doors

search here … Go

Home | Java FAQs | 600+ Java Q&As | Career | Tutorials | Member | Why?

Can u Debug? | Java 8 ready? | Top X | Productivity Tools | Judging Experience?

Home › Tech Key Areas › 13 Technical Key Areas Interview Q&A › Design Patterns › GoF Patterns › ♦ 12 Java design patterns interview questions & answers

♦ 12 Java design patterns interview questions & answers

Posted on September 9, 2014 by Arulkumaran Kumaraswamipillai — No Comments ↓

0 Like Share 0 G+1 Share

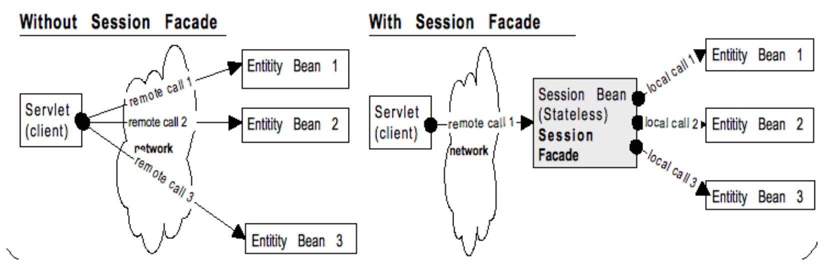
Java Design Patterns Interview Questions Links:

Why do Proxy, Decorator, Adapter, Bridge, and Facade design patterns look very similar? What are the differences? | Builder pattern and immutability in Java | Flyweight pattern and improve memory usage & performance | Java ExecutorService with strategy design pattern | Java dynamic proxy for service retry

Q1. Why use design patterns?

A1.

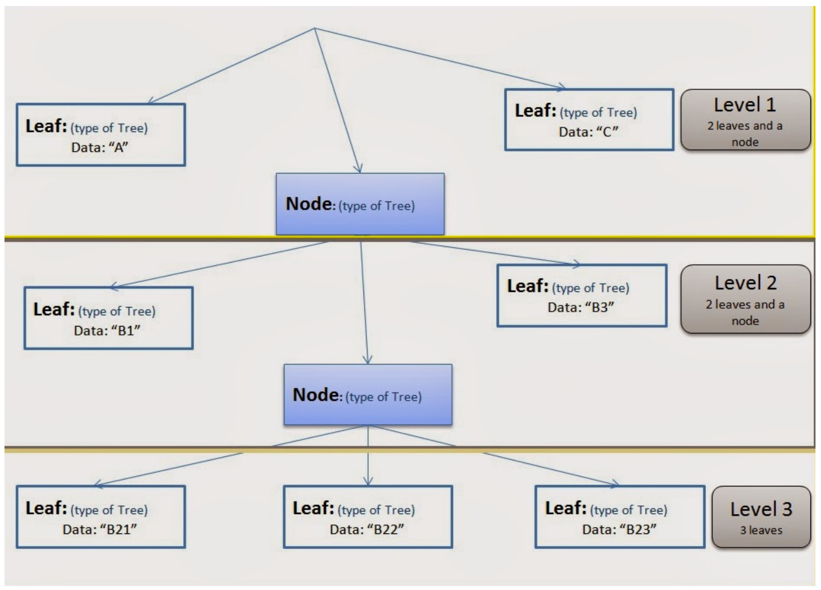
1) **Capture design experience from the past**: E.g. Facade and value object patterns evolved from performance problems experienced due to multiple remote calls.



Facade

2) **Promote reuse without having to reinvent the wheel**: E.g. The flyweight pattern improves application performance through object reuse, which minimizes the overhead such as memory allocation and garbage collection.

3) **Define the system structure better**: E.g. The composite design pattern treats nodes and leafs uniformly. The Leaf contains the data, and the Node contains Leaves. Both the Leaf and Node are treated as a Tree. This is the power of composite design pattern.


Leaf: (type of Tree) Data: "A"
Leaf: (type of Tree) Data: "C"
Level 1 2 leaves and a node
Node: (type of Tree)
Leaf: (type of Tree) Data: "B1"
Leaf: (type of Tree) Data: "B3"
Level 2 2 leaves and a node
Node: (type of Tree)
Leaf: (type of Tree) Data: "B21"
Leaf: (type of Tree) Data: "B22"
Leaf: (type of Tree) Data: "B23"
Level 3 3 leaves

Composite design pattern

The builder pattern makes your code more elegant. For example

```
NetAsset.Builder builder = new NetAsset.Builder()
NetAsset na = builder.setGrossAssetValueBeforeTax
        .setGrossAssetValueAfterTax(BigDecimal.val
        .setTotalAssets(BigDecimal.valueOf("3500.0
        .setTotalReceivables(BigDecimal.valueOf("2

```

Instead of

```
NetAsset na = new NetAsset();
na.setGrossAssetValueBeforeTax(BigDecimal.valueOf
na.setGrossAssetValueAfterTax(BigDecimal.valueOf(
na.setTotalAssets(BigDecimal.valueOf("3500.00"));
na.setTotalReceivables(BigDecimal.valueOf("2500.0

```

4. **Provide a common design vocabulary**: Should we use a Data Access Object (DAO)? How about using a Business Delegate? How about a bridge to minimize the explosion of class hierarchies? How about an adapter to work with the third party libraries? How about a singleton to provide a global point of access? E.g. A DataSource should have only a single instance where it will supply multiple connections from its single DataSource pool.

Q2. Why use a factory pattern?

A2. **Factory pattern** returns an instance of several (product hierarchy) subclasses (like *Circle*, *Square* etc), but the calling code is unaware of the actual implementation class. The calling code invokes the method on the interface for example *Shape* and using polymorphism the correct *draw()* method gets invoked.

So, as you can see, the factory pattern **reduces the coupling** or the dependencies between the calling code and called objects like Circle, Square etc. This is a very powerful and common feature in many frameworks. You do not have to create a new Circle or a new Square on each invocation.

16 Technical Key Areas

80+ step by step Java Tutorials

In future, to conserve memory you can decide to cache objects or reuse objects in your factory with no changes required to your calling code. In other words, you are applying the **flyweight design pattern** within your **factory** to conserve memory.

You can also load objects in your factory based on attribute(s) read from an external properties file or some other condition. Another benefit going for the factory is that unlike calling constructors directly, factory patterns have more meaningful names like getShape(…), getInstance(…) etc, which may make calling code more clear.

Q3. How does a builder pattern differ from a factory pattern?
A3. The subtle difference between the builder pattern and the factory pattern is that in **builder pattern, the user is given the choice to create the type of object he/she wants** but the construction process is the same. But with the **factory method pattern the factory decides how to create one of several possible classes** based on data provided to it.

The builder design pattern builds an object over several steps. It holds the needed state for the target item at each intermediate step. The ***StringBuilder*** is a good example that goes through to produce a final string. Here is a custom class example.

```
import java.math.BigDecimal;

public class NetAsset
{
    private final BigDecimal grossAssetValueAfte
    private final BigDecimal grossAssetValueBefo

    private NetAsset(Builder builder) //private
    {
        this.grossAssetValueAfterTax = builder.g
        this.grossAssetValueBeforeTax = builder.
    }

    public BigDecimal getGrossAssetValueAfterTax
    {
        return grossAssetValueAfterTax;
    }

    public BigDecimal getGrossAssetValueBeforeTa
    {
        return grossAssetValueBeforeTax;
```

100+ Java pre-interview coding tests

How good are your .....?

```
    }

    //inner builder class
    public static class Builder
    {
        private BigDecimal grossAssetValueAfterT
        private BigDecimal grossAssetValueBefore

        public Builder setGrossAssetValueAfterTa
        {
            this.grossAssetValueAfterTax = gross
            return this;
        }

        public Builder setGrossAssetValueBeforeT
        {
            this.grossAssetValueBeforeTax = gros
            return this;
        }

        //return the built NetAsset
        public NetAsset build()
        {
            return new NetAsset(this);
        }
    }
}
```

The builder can be used as shown below:

```
NetAsset.Builder builder = new NetAsset.Builder()
NetAsset na = builder.setGrossAssetValueBeforeTax
        .setGrossAssetValueAfterTax(BigDecimal.val
```

Q4. How does a builder pattern differ from a flyweight pattern?

A4. The **Builder pattern** is used to create many objects, whereby the Flyweight pattern is about sharing such a collection of objects. The **flyweight design pattern** is a structural pattern used to improve memory usage and performance (i.e. due to shorter and less frequent garbage collections). Here, instead of creating a large number of objects, we reuse the objects that are already created. With fewer objects, your application could fly.

Example 1: In Java, String objects are managed as flyweight. Java puts all fixed String literals into a literal pool. For redundant literals, Java keeps only one copy in the pool.

```
String author = "Little brown fox";
String authorCopy = "Little brown fox";
```

```

if(author == authorCopy) {
    System.out.println("referencing the same obje
}
```

Example 2: The Wrapper classes like *Integer*, *Float*, *Decimal*, *Boolean*, and many other classes having the *valueOf* **static factory method** applies the flyweight design pattern to conserve memory by reusing the objects.

```
public static void main(String[] args) {
    Integer value1 = Integer.valueOf(5);
    Integer value2 = Integer.valueOf(5);

    if (value1 == value2) {
       System.out.println("referencing the same o
    }
}
```

Q5. What is a decorator design pattern?
A5. By implementing the **decorator pattern** you construct a wrapper around an object by extending its behavior. The wrapper will do its job before or after and delegate the call to the wrapped instance. The decoration happens at **run-time via object composition**. A good example is the **Java I/O classes** as shown below. Each reader or writer will decorate the other to extend or modify the behavior.

```
String inputText = "Some text to read";
ByteArrayInputStream bais = new ByteArrayInputStr
Reader isr = new InputStreamReader(bais);
BufferedReader br = new BufferedReader(isr);
br.readLine();

```

Q6. How does a decorator design pattern differ from a proxy design pattern?
A6. In **Proxy pattern**, you have a proxy and a **real subject**. The relationship between a proxy and the real subject is typically set at **compile time**, whereas decorators can be recursively constructed at run time. The Decorator Pattern is also known as the **Wrapper pattern**. The Proxy Pattern is also known as the **Surrogate pattern**. The purpose of decorator pattern is to add additional responsibilities to an object. These responsibilities can of course be added through

inheritance, but composition provides better flexibility as explained above via the Java I/O classes. The purpose of the proxy pattern is to add an intermediate between the client and the target object. This intermediate shares the same interface as the target object. Here are some scenarios in which a proxy pattern can be applied.

— A **remote proxy** provides a local representative for an object in a different address space.Providing interface for remote resources such as web service resources, EJBs or RMI (Stub and Skeleton).

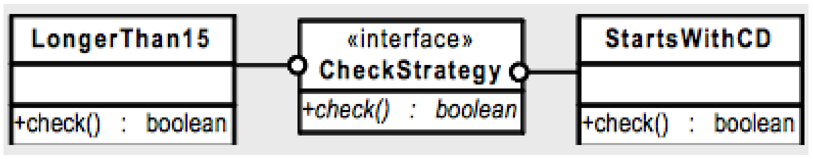
— A **virtual proxy** creates expensive object on demand. E.g. Hibernate lazy loaded proxy objects.

— A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, adding a thread-safe feature to an existing class without changing the existing class's code. This is useful when you do not have the freedom to fix thread-safety issues in a third-party library.

Q7. What is a strategy design pattern?
A7. The **Strategy pattern** lets you build software as a loosely coupled collection of interchangeable parts, in contrast to a monolithic, tightly coupled system. Loose coupling makes your software much more extensible, maintainable, and reusable. The main attribute of this pattern is that **each strategy encapsulates algorithms**.

Example 1: You can draw borders around almost all Swing components, including panels, buttons, lists, and so on. Swing provides numerous border types for its components: bevel, etched, line, titled, and even compound. The various borders are drawn using the strategy pattern.

Example 2: Strategies to check if a given description is longer than 15 characters, starts with "CD", etc.



Strategy pattern

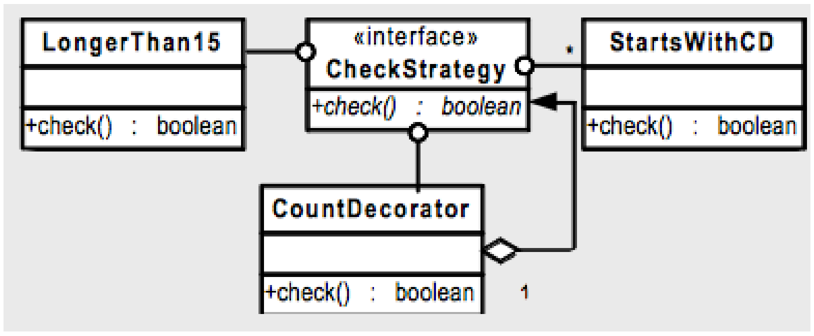
```
public interface CheckStrategy {
   public boolean check(String word);
}
```

```
public class LongerThan15 implements CheckStrate
   public static final int LENGTH = 15; //consta

   public boolean check(String description) {
    if (description == null)
        return false;
    else
        return description.length() > LENGTH;
   }
}
```

```
public class StartsWithCD implements CheckStrate
  public static final String STARTS_WITH = "cd";

  public boolean check(String description) {
     String s = description.toLowerCase();
     if (description == null || description.leng
        return false;
     else
        return s.startsWith(STARTS_WITH);
  }
}
```

Q8. How will you go about counting the occurrences of characters greater than 15 or description starting with “cd”?

A8. Have a decorator nor wrapper to perform the count first, and then forward the request to the strategy class.



Decorator

```
public class CountDecorator implements CheckStra

  private CheckStrategy cs = null;
  private int count = 0;

  public CountDecorator(CheckStrategy cs) {
    this.cs = cs;
  }

  public boolean check(String description) {
    boolean isFound = cs.check(description);
    if (isFound){
        this.count++;
    }
    return isFound;
  }

  public int count() {
      return this.count;
  }

  public void reset() {
       this.count = 0;
  }
}

```

Q9. What is the difference between a strategy and a command pattern?

A9. Firstly, some examples

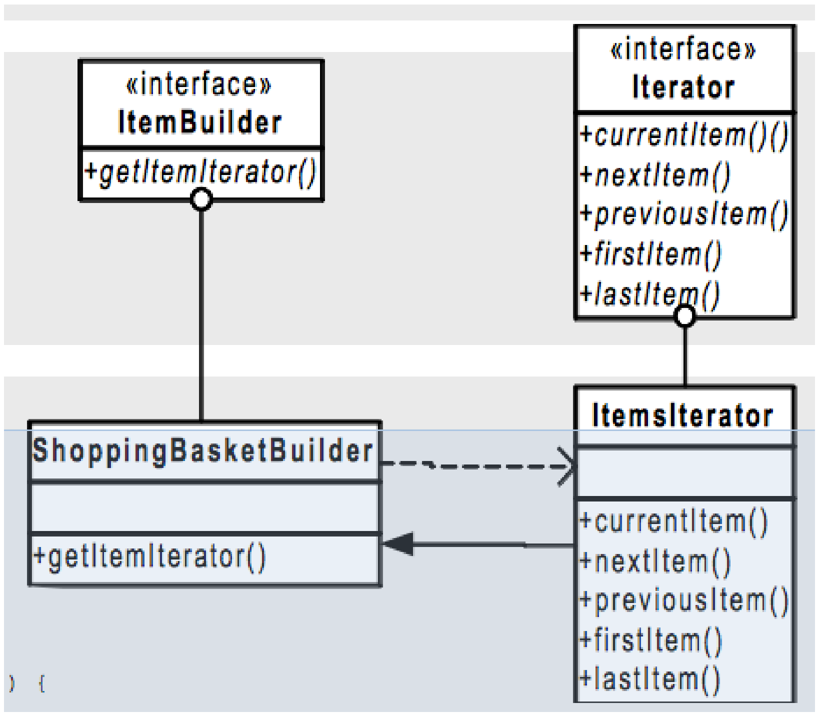
Strategy – quicksort or mergesort, simple vs compound interest calculations, etc
Command – Open or Close actions, redo or undo actions, etc. **You need to know the states undo**.

```
public interface AbstractCommand {
     abstract void execute();
}
```

```
public class ConcreteCommand implements Abstract

    private Object arg; //state

    public ConcreteCommand(Object arg) {
        this.arg = arg;
    }

    @Override
    public void execute() {
        // Work with own state.
    }

}

```

Strategy handles how something should be done by taking the supplied arguments in the execute(….) method. Command creates an object out of what needs to be done (i.e. **hold state**) so that these command objects can be passed around between other classes. The actions the command represent can be undone or redone by maintaining the state.

Q10. What is the purpose of an iterator pattern?
A10. Iterator provides a way to access the elements of an aggregate object without exposing its underlying implementation.



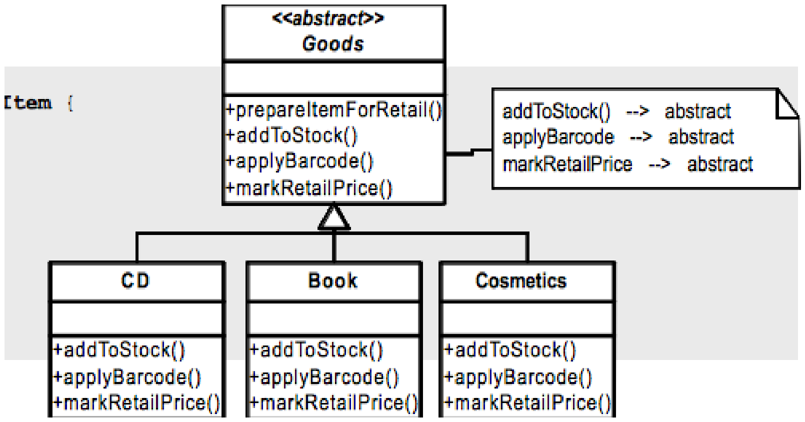
Iterator pattern

```
public interface Iterator {
    public Item nextItem();
    public Item previousItem();
    public Item currentItem();
    public Item firstItem();
    public Item lastItem();
    public boolean isDone();
    public void setStep(int step);
}
```

```
public class ShoppingBasketBuilder implements It

  private List listItems = null;

  public Iterator getIterator() {
     return listItems.iterator();
  }

  public com.item.Iterator getItemIterator() {
    return new ItemsIterator();
  }

  /**
   * inner class which iterates over basket of i
   */
  class ItemsIterator implements Iterator {
     private int current = 0;
     private int step = 1;

     public Item nextItem() {
       Item item = null;
       current += step;
       if (!isDone()) {
         item = (Item) listItems.get(current);
       }
       return item;
     }

    public Item previousItem() {
      Item item = null;
      current -= step;
      if (!isDone()) {
        item = (Item) listItems.get(current);
      }
      return item;
    }

    public Item firstItem() {
      current = 0;
      return (Item) listItems.get(current);
    }

    public Item lastItem() {
      current = listItems.size() - 1;
      return (Item) listItems.get(current);
    }

    public boolean isDone() {
      return current >= listItems.size() ? true
    }

    public Item currentItem() {
      if (!isDone()) {
        return (Item) listItems.get(current);
      } else {
        return null;
      }
    }

    public void setStep(int step) {
        this.step = step;
    }
  }
}
```

Q11. When you have a sequence of steps to be processed within a method and you want to defer some of the steps to its subclass, what design pattern will you use?

A11. **Template method** design pattern. Good example of this is the process() method in the **Struts *RequestProcessor*** class, which executes a sequence of *processXXXX(…)* methods allowing the subclass to override some of the methods when required.



Template method

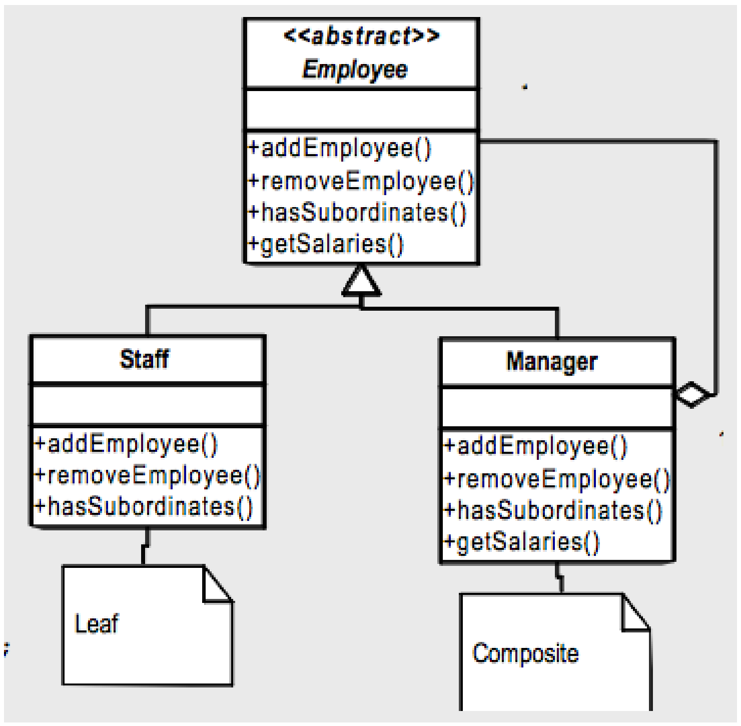
prepareItemForRetail() has 3 steps like **1)** add the item to stock **2)** apply the bar code and **3)** mark a retail price. The abstract base class will implement all the default behavior. The specific item classes like Boo, CD, etc can either use the default behvior, or override one or more of the step behavior(s).

```
public abstract class Goods implements Item {

  /**
   * The template method
   */
  public void prepareItemForRetail() {
      addToStock();
      applyBarcode();
      markRetailPrice();
  }

  public void addToStock(){
     //..default impl
  };

  public void applyBarcode(){
     //..default impl
  };

  public void markRetailPrice(){
     //..default impl
  };
}

```

```
public class Book extends Goods {

  @Override
  public void addToStock() {
     //override default behavior
     //... special logic
  }
}

```

Q12. What is a composite design pattern? Can you explain it with a class diagram and an example?

A12. The composite design pattern composes objects into tree structures where individual objects like sales staff and composite objects like managers are handled uniformly.



Composite design pattern

```
public abstract class Employee {
    private String name;
    private double salary;

   public Employee(String name, double salary) {
       this.name = name;
        this.salary = salary;
   }

  public String getName() {
       return name;
  }

  public double getSalaries() {
       return salary;
  }

  public abstract boolean addEmployee(Employee e
  public abstract boolean removeEmployee(Employe
  protected abstract boolean hasSubordinates();
}
```

```
public class Manager extends Employee {

  List<Employee> subordinates = null;

  public Manager(String name, double salary) {
      super(name, salary);
  }

  public boolean addEmployee(Employee emp) {
    if (subordinates == null) {
         subordinates = new ArrayList(10);
    }
    return subordinates.add(emp);
  }

  public boolean removeEmployee(Employee emp) {
    if (subordinates == null) {
         subordinates = new ArrayList(10);
    }
    return subordinates.remove(emp);
  }

  /**
   * Recursive method call to calculate the sum
   * means sum of salary of a manager on whom th
   * themselves will have any subordinates and s
   */
  public double getSalaries() {
      double sum = super.getSalaries(); //this o

      if (this.hasSubordinates()) {
       for (int i = 0; i < subordinates.size();
             sum += ((Employee) subordinates.get
       }
    }
    return sum;
  }

  public boolean hasSubordinates() {
     boolean hasSubOrdinates = false;
     if (subordinates != null && subordinates.si
           hasSubOrdinates = true;
    }
    return hasSubOrdinates;
  }
}

```

```
public class Staff extends Employee {

  public Staff(String name, double salary) {
      super(name, salary);
  }

  public boolean addEmployee(Employee emp) {
      throw new RuntimeException("Improper use o
  }

  public boolean removeEmployee(Employee emp) {
       throw new RuntimeException("Improper use
  }

  protected boolean hasSubordinates() {
       return false;
  }
}
```

Java Design Patterns Interview Questions Links:

Why do Proxy, Decorator, Adapter, Bridge, and Facade design patterns look very similar? What are the differences? | Builder pattern and immutability in Java | Flyweight pattern and improve memory usage & performance | Java ExecutorService with strategy design pattern | Java dynamic proxy for service retry

Popular Posts

♦ 11 Spring boot interview questions & answers

827 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

767 views

18 Java scenarios based interview Questions and Answers

400 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

389 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

296 views

♦ 7 Java debugging interview questions & answers

293 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

286 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

279 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

240 views

001B: ♦ Java architecture & design concepts interview questions & answers

202 views

Bio | **Latest Posts**

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

‹ 15 Security key area interview Q&A for Java developers

Java BDD (Behavior Driven Development) interview Q&A ›

Posted in GoF Patterns, Java Key Area Essentials, member-paid

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category ▼

© Disclaimer