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 › Interview › Pressed for time? Java/JEE Interview FAQs › FAQ Core Java Job Interview Q&A Essentials › ♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

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

Posted on February 19, 2015 by Arulkumaran Kumaraswamipillai — No Comments ↓

The focus is on Java OOP interview questions and answers.

Top 50+ Core Java Interview Questions Links:

Q01-Q10 | Q24-Q36 on classes, interfaces and generics | Q37-Q42 on GC and referencing | Q43-Q54 on Objects

Q11. What is the difference between constructors and other regular methods?
A11. Constructors must have the same name as the class name and cannot return a value. The constructors are called

600+ Full Stack Java/JEE Interview Q&As ♥Free ♦FAQs

open all | close all
Ice Breaker Interview
Core Java Interview (
JEE Interview Q&A (3
Pressed for time? Jav
Job Interview Ice B
FAQ Core Java Jol
♥♦ Q1-Q10: Top
♦ Q11-Q23: Top
♦ Q24-Q36: Top
♦ Q37-Q42: Top
♦ Q43-Q54: Top
01: ♥♦ 15 Beginr
02: ♥♦ 10+ Java
FAQ JEE Job Inter
FAQ Java Web Ser
Java Application Ar
Hibernate Job Inter
Spring Job Intervie
Java Key Area Ess
OOP & FP Essenti
Code Quality Job I

only once per creation of an object while regular methods can be called many times. E.g. for a *Pet.class*

```
// constructor
public Pet( ) {}

```

Regular methods can have any name and can be called any number of times. E.g. for a *Pet.class*.

```
// regular method has a void return type.
public void Pet(){}

```

Note: method name is shown starting with an uppercase to differentiate a constructor from a regular method. Better naming convention is to have a meaningful name starting with a lowercase like:

```
// regular method has a void return type
public void createPet( ){}

```

Q12. What happens if you do not provide a constructor?
A12. Java does not actually require an explicit constructor in the class description. If you do not include a constructor, the Java compiler will create a default constructor in the byte code with an empty argument. This default constructor is equivalent to the explicit "*Pet( ){}*". If a class includes one or more explicit constructors like "*public Pet(int id)*", the java compiler does not create the default constructor "*Pet( ){}*".

Q13. Can you call one constructor from another?
A13. Yes, by using *this( )* syntax. E.g.

```
public Pet(int id) {
    this.id = id;                                            /
}

public Pet (int id, String type) {
```

⊞ SQL, XML, UML, JSC
⊞ Hadoop & BigData Int
⊞ Java Architecture Inte
⊞ Scala Interview Q&As
⊞ Spring, Hibernate, & I
⊞ Testing & Profiling/Sa
⊞ Other Interview Q&A 1
⊞ ▶ Free Java Interviev

16 Technical Key Areas

open all | close all
⊞ Best Practice (6)
⊞ Coding (26)
⊞ Concurrency (6)
⊞ Design Concepts (7)
⊞ Design Patterns (11)
⊞ Exception Handling (3
⊞ Java Debugging (21)
⊞ Judging Experience In
⊞ Low Latency (7)
⊞ Memory Managemen
⊞ Performance (13)
⊞ QoS (8)
⊞ Scalability (4)
⊞ SDLC (6)
⊞ Security (13)
⊞ Transaction Managen

80+ step by step Java Tutorials

open all | close all
⊞ Setting up Tutorial (6)
⊞ Tutorial - Diagnosis (2
⊞ Akka Tutorial (9)
⊞ Core Java Tutorials (2

```
    this(id);   // calls constructor public Pet(i
    this.type = type;   // "this" means this obje
}

```

Q14. Can you call the superclass constructor?

A14. If a class called "*SpecialPet*" extends your "*Pet*" class, then you can use the keyword "super" to invoke the superclass's constructor. E.g.

```
public SpecialPet(int id) {
     super(id);        //must be the very first s
}

```

To call a regular method in the super class use: "*super.myMethod( );*". This can be called at any line. Some frameworks based on JUnit add their own initialization code.

```
public class DBUnitTestCase extends TestCase {
  public void setUp( ) {
    super.setUp( );
    // do my own initialization
  }
 }

 public void cleanUp( ) throws Throwable
 {
     try {
           // Do stuff here to clean up your obj
     }
     catch (Throwable t) {}
     finally{
         //clean up your parent class. Unlike co
         // super.regularMethod() can be called
         super.cleanUp( );
     }
 }

```

Q15. What are the advantages of Object Oriented Programming Languages (OOPL)?

A15. The Object Oriented Programming Languages directly represent the real life objects like *Person*, *Employee*, *Account*, *Customer*, etc. The features of the OO programming languages like **Abstraction**, **Polymorphism**, **Inheritance** and **Encapsulation** make it powerful. [Tip: remember 'a pie'

⊞ Hadoop & Spark Tuto
⊞ JEE Tutorials (19)
⊞ Scala Tutorials (1)
⊞ Spring & HIbernate Tı
⊞ Tools Tutorials (19)
⊞ Other Tutorials (45)

100+ Java pre-interview coding tests

open all | close all
⊞ Can you write code? (
⊞ ♦ Complete the given
⊞ Converting from A to I
⊞ Designing your classe
⊞ Java Data Structures
⊞ Passing the unit tests
⊞ What is wrong with th
⊞ Writing Code Home A
⊞ Written Test Core Jav
⊞ Written Test JEE (1)

How good are your .....?

open all | close all
⊞ Career Making Know-
⊞ Job Hunting & Resum

which, stands for Abstraction, Polymorphism, Inheritance and Encapsulation are the 4 pillars of **OOPL**]

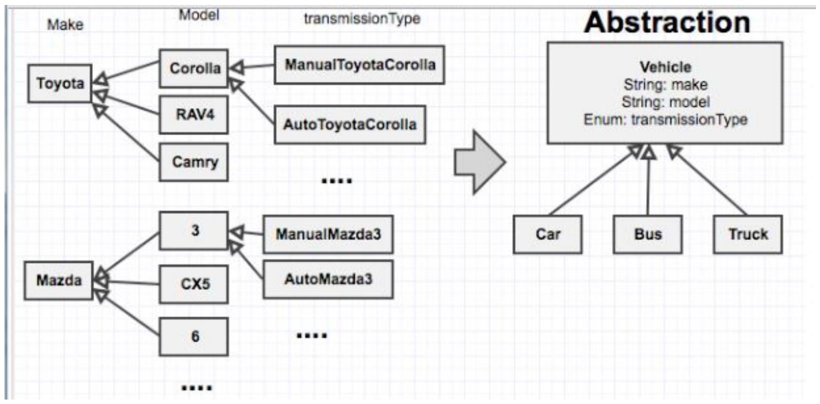
Q16. How does the Object Oriented approach improve software development?
A16. The key benefits are:

1) Re-use of previous work using implementation inheritance and object composition.
2) Real mapping of objects to the problem domain: Objects map to real world and represent vehicles, customers, products, etc with abstraction and encapsulation.
3) Modular Architecture: objects, systems, frameworks, etc are the building blocks of larger systems.

The increased quality and reduced development time are the by-products of the key benefits discussed above. If 90% of the new application consists of proven existing components, then only the remaining 10% of the code has to be tested from scratch.

Q17. Can you explain abstraction and encapsulation relating to OOP?
A17. **Abstraction** refers to hiding all the non-essential details from the user. Abstraction comes in two forms: abstracting the **behavior** and abstracting the **data**. For example, a person driving a car only needs to know about how to use a steering wheel, gear, accelerator, etc, but does not need to know about the internal details like how the engine works? how the transmission works?, how much fuel is released on acceleration?, etc. Thus, abstraction lets you focus on **what the object does instead of how it does.**

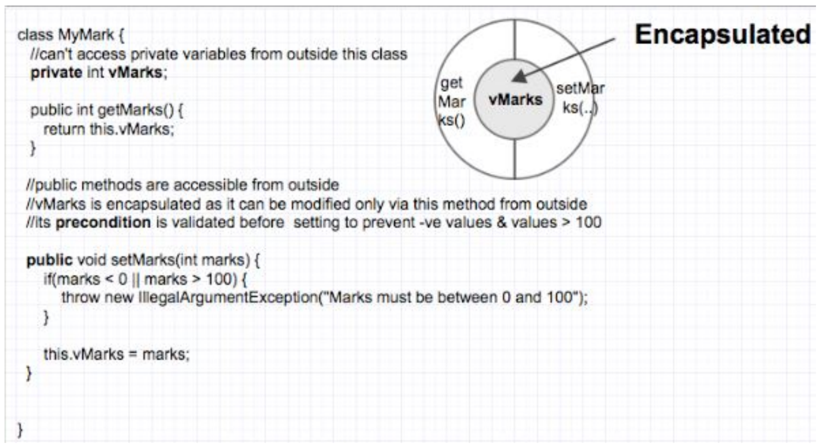
Abstraction gives you the ability to conceptualize things by ignoring the irrelevant details. If you refer to an object as a vehicle that can be used in place of an actual vehicle like a car, bus, van, etc, you are making an abstraction. You can make an abstraction at different levels by handling details at different levels. A *Vehicle* class can focus on common behaviors like forward(..), reverse(..), turn(..), etc and attributes like *make*, *model*, *transmissionType* (e.g. auto or manual), *driveType* (2 wheel, 4 wheel), etc. A set of derived classes like *Car*, *Bus*, *Truck*, etc can focus on more specific details of a vehicle. This gives you another level of abstraction by allowing you to refer to an object as a car that can be used in place of different makes like a *Toyota*, *Ford*, *Volvo*, etc and models like *Camry*, *Corolla*, etc by capturing the make and model as attributes as opposed types.



Abstraction Java example

Abstraction is all about managing complexities at package, class, interface, and method levels. Can you imagine how complex your class hierarchy will become if you represent the make and model as classes instead of attributes within a class? You may end up with thousands of classes from different make and model if not tens of thousands. Good programmers develop this essential skill to logically map a problem to its barest essential through the process of mental exercise. Both the ability to look at the big picture and eye for details of how things work are two essential traits to succeed in software development.

Encapsulation takes abstraction, which allows you to look at an object at a high level of detail a step further by forbidding access to some internal details to minimize complexity. Encapsulation makes your code modular by capturing the data and the function into a single class. Modularity is a goal to treat each class or method like a black box. It identifies the parts of an object that should be made public and those that should be made private. The data and methods that an object exposes to every other object is called the object's public interface and the parts that are exposed to its subclasses via its inheritance is called the protected interface. The access modifiers are used to restrict access to some of the internal details. Thus, encapsulation is hiding of internal details (e.g. having variables and methods with either private or protected access), and connecting with other objects through well defined narrow boundaries (e.g. well defined methods with package-private or public access) and contracts.

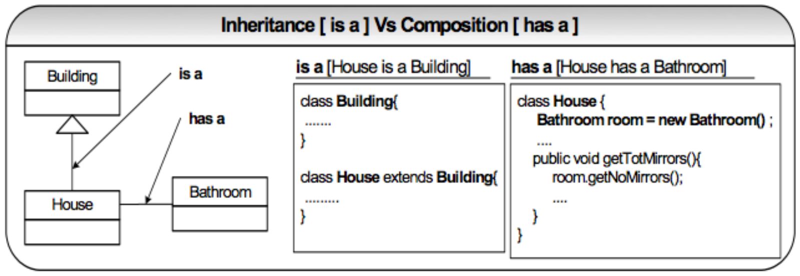


```
class MyMark {
  //can't access private variables from outside this class
  private int vMarks;

  public int getMarks() {
    return this.vMarks;
  }

 //public methods are accessible from outside
 //vMarks is encapsulated as it can be modified only via this method from outside
 //its precondition is validated before  setting to prevent -ve values & values > 100

 public void setMarks(int marks) {
    if(marks < 0 || marks > 100) {
       throw new IllegalArgumentException("Marks must be between 0 and 100");
    }

    this.vMarks = marks;
 }


}
```

Encapsulation Java example

Being able to encapsulate members of a class is important for security and integrity. We can protect variables from unacceptable values. The sample code above describes how encapsulation can be used to protect the *MyMarks* object from having negative values or values greater than 100. Any modification to member variable "*vMarks*" can only be carried out through the setter method *setMarks(int mark)*. This prevents the object "*MyMarks*" from having in correct values by throwing an exception. This also explains a design concept known as "design by contract", which states that the calling method should satisfy the contract by passing values between 0 and 100. The methods must "fail fast" by performing the pre-condition, invariant, and post-condition checks in design by contract.

Q18. How do you express an 'is a' relationship and a 'has a' relationship or explain inheritance and composition?
A18. The '**is a'** relationship is expressed with **inheritance** and '**has a**' relationship is expressed with **composition**. Both inheritance and composition allow you to place sub-objects inside your new class. Two of the main techniques for **code reuse** are class inheritance and object composition.


Inheritance [is a] Vs Composition [has a]
Building
is a
has a
House
Bathroom
is a [House is a Building]
has a [House has a Bathroom]

```
class Building{
 .......
}

class House extends Building{
 .........
}
```

```
class House {
   Bathroom room = new Bathroom() ;
   ....
  public void getTotMirrors(){
     room.getNoMirrors();
     ....
  }
}
```

Inheritance Vs Composition

Inheritance is uni-directional. For example *House* is a *Building*. But *Building* is not a *House*. Inheritance uses extends key word. In UML, it is known as the **generalization**.

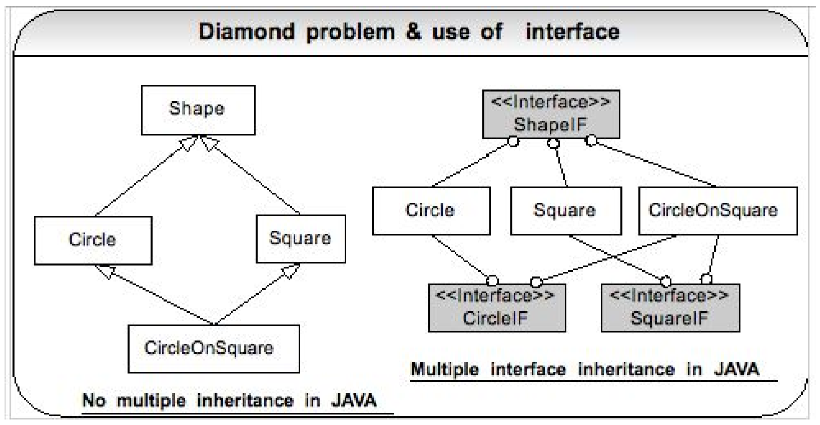
Composition: is used when *House* has a *Bathroom*. It is incorrect to say *House* is a *Bathroom*. Composition simply means using instance variables that refer to other objects. The class *House* will have an instance variable, which refers to a *Bathroom* object. In UML, you have **composition** (filled diamond) and **aggregation** (unfilled diamond indicating a weaker relationship where the contained objects (e.g. bathroom) do not take part in the full lifecycle of the containing objects (e.g. house).

Q19. Which one to favor, composition or inheritance?
A19. The guide is that inheritance should be only used when a subclass 'is a' superclass.

- Don't use inheritance just to get code reuse. If there is no 'is a' relationship then use composition for code reuse. Overuse of implementation inheritance (uses the "extends" key word) can break all the subclasses, if the superclass is modified.

- Do not use inheritance just to get polymorphism. If there is no 'is a' relationship and all you want is polymorphism then use interface inheritance, which gives you polymorphism with composition, which gives you code reuse.

Note: Java does not support **multiple implementation inheritance**, which means a class in Java can extend only one other class and not more than one. But Java supports **multiple interface inheritance**, which means a class can implement multiple interfaces.



Multiple inheritance

In UML, interface inheritance is known as **realization**. Java 8 allows default and static method implementations in interfaces. These are known as the functional interfaces, and annotated with **@FunctionalInterface**. For example, the comparator class is enhanced in Java 8 as shown below with many default and static methods.

```
@FunctionalInterface
public interface Comparator<T> {

  int compare(T o1, T o2);
  boolean equals(Object obj);

  default Comparator<T>  reversed() {
        return Collections.reverseOrder(this);
   }

     //...skipping some methods

  public static<T>  Comparator&lt;T&gt;  nullsLa
       return new Comparators.NullComparator<>(f
  }

}

```

Java 8 allowing method implementations in its interfaces is a form of multiple inheritance as you can inherit behavior from different parents. What it is missing is to inherit states, i. e., attributes. So, Java 8 on wards Java supports **multiple behavior inheritance**.

Q20. Can you differentiate compile-time inheritance and runtime inheritance? Which one does Java support?
A20. The term “inheritance” refers to a situation where behaviors and attributes are passed on from one object to another. The Java programming language natively only supports compile-time inheritance through subclassing as shown below with the keyword “extends”.

```
public class Parent {
    public String saySomething( ) {
          return “Parent is called”;
    }
}

```

```
public class Child extends Parent {
     @Override
     public String saySomething( ) {
          return super.saySomething( ) +  “, Chil
    }
}

```

A call to *saySomething( )* method on the class “*Child*” will return “*Parent* is called, *Child* is called” because the *Child* class inherits “*Parent* is called” from the class Parent. The keyword “super” is used to call the method on the “Parent” class. Runtime inheritance refers to the ability to construct the parent/child hierarchy at runtime. Java does not natively support runtime inheritance, but there is an alternative concept known as “**delegation**” or “**composition**”, which refers to constructing a hierarchy of object instances at runtime. This allows you to simulate runtime inheritance. In Java, delegation is typically achieved as shown below:

```java
public class Parent {
    public String saySomething( ) {
          return “Parent is called”;
    }
}

```

```java
public class Child  {
     public String saySomething( ) {
          return new Parent( ).saySomething( ) +
    }
}

```

The *Child* class delegates the call to the *Parent* class. Composition can be achieved as follows:

```java
public class Child  {
     private Parent parent = null;

     public Child( ){
          this.parent = new Parent( );
     }

     public String saySomething( ) {
          return this.parent.saySomething( ) +
    }
}

```

Q21. What do you understand by inheritance? What are the different types of inheritance that Java supports?

A21. Inheritance – is the inclusion of behavior (i.e. methods) and state (i.e. attributes) of a base class in a derived class so that they are accessible in that derived class. The key benefit of Inheritance is that it provides the formal mechanism for code reuse. Any shared piece of business logic can be moved from the derived class into the base class as part of refactoring process to improve maintainability of your code by avoiding code duplication. The existing class is called the superclass and the derived class is called the subclass. Inheritance can also be defined as the process whereby one object acquires characteristics from one or more other objects the same way children acquire characteristics from their parents. There are two types of inheritances:

1. Implementation inheritance (aka class inheritance): You can extend an application's functionality by reusing functionality in the parent class by inheriting all or some of the operations already implemented. In Java, you can only inherit from one superclass. Implementation inheritance promotes reusability but improper use of class inheritance can cause programming nightmares by breaking encapsulation and making future changes a problem. With implementation inheritance, the subclass becomes tightly coupled with the superclass. This will make the design fragile because if you want to change the superclass, you must know all the details of the subclasses to avoid breaking them. So when using implementation inheritance, make sure that the subclasses depend only on the behavior of the superclass, not on the actual implementation.

2. Interface inheritance (aka type inheritance): This is also known as sub typing. Interfaces provide a mechanism for specifying a relationship between otherwise unrelated classes, typically by specifying a set of common methods each implementing class must contain. Interface inheritance promotes the design concept of program to interfaces not to implementations. This also reduces the coupling or implementation dependencies between systems. In Java, you can implement any number of interfaces. This is more flexible than implementation inheritance because it won't lock you into specific implementations which make subclasses difficult to maintain. So care should be taken not to break the implementing classes by modifying the interfaces.

Which one to favor and why? Favor interface inheritance to implementation inheritance because it promotes the design concept of coding to an interface and reduces coupling. Interface inheritance can achieve code reuse with the help of object composition. If you look at Gang of Four (GoF) design patterns, you can see that it favors interface inheritance to implementation inheritance.

Q22. Why would you prefer code reuse via composition over inheritance?
A22. Both the approaches give you code reuse (in different ways) to achieve the same results but:

The advantage of class inheritance is that it is done statically at compile-time and is easy to use. The disadvantage of class inheritance is that because it is static, implementation inherited from a parent class cannot be changed at run-time. In object composition, functionality is acquired dynamically at run-time by objects collecting references to other objects. The advantage of this approach is that implementations can be replaced at run-time. This is possible because objects are accessed only through their interfaces, so one object can be replaced with another just as long as they have the same type. For example: the composed class AccountHelperImpl can be replaced by another more efficient implementation as shown below if required:

```
public class EfficientAccountHelperImpl implemen
    public void deposit(double amount) {
        System.out.println("efficient depositing
    }

    public void withdraw(double amount) {
        System.out.println("efficient withdrawin
    }
}

```

Another problem with class inheritance is that the subclass becomes dependent on the parent class implementation. This makes it harder to reuse the subclass, especially if part of the inherited implementation is no longer desirable and hence can break encapsulation. Also a change to a superclass can not only ripple down the inheritance hierarchy to subclasses, but can also ripple out to code that uses just the subclasses making the design fragile by tightly coupling the subclasses with the super class. But it is easier to change the interface/implementation of the composed class.

Due to the flexibility and power of object composition, most design patterns emphasize object composition over inheritance whenever it is possible. Many times, a design pattern shows a clever way of solving a common problem through the use of object composition rather then a standard, less flexible, inheritance based solution.

Q23. Can you explain polymorphism with an example?
A23. **Polymorphism** is the capability to invoke a method without knowing until runtime what kind of object you are dealing with. In a nutshell, polymorphism is a bottom-up method call. The benefit of polymorphism is that it is very easy to add new classes of derived objects without breaking the calling code that uses the polymorphic classes using the implementation inheritance or polymorphic interfaces using the interface inheritance. When you send a message to an object even though you don't know what specific type it is, and the right thing happens, that's called polymorphism. The process used by object-oriented programming languages to implement polymorphism is called dynamic binding. Polymorphism prevents programs to rely on low level details of object implementations. This considerably reduces the dependencies (aka coupling) between modules. Less dependencies means more maintainable and easier to modify programs.

Polymorphism in Java comes in 2 forms through method overriding (aka runtime polymorphism) and method overloading (aka compile-time polymorphism). The power of OOP using Java is centered on runtime polymorphism using class inheritance or interface inheritance with method overriding. The fact that the decision as to which version of the method to invoke cannot be made at compile time and must be deferred and made at runtime is sometimes referred to as late binding.

```java
public interface Soundable {
   abstract void createSound();
}

```

```java
public class Cat implements Soundable {

 @Override
 public void createSound() {
    System.out.println("Meow Meow");
 }

}
```

```java
public class Dog implements Soundable {

  @Override
  public void createSound() {
    System.out.println("Wow Wow");
  }

}
```

```java
//demonstrate polymorphism

public class Test {

 public static void main(String[] args) {
   //code to interface
   Soundable s1 = new Cat( );
   Soundable s2 = new Dog( );

   //The output depends on what type of object
   //is stored on s1 &amp; s2 at runtime, and no
   s1.createSound( );  //prints Meow Meow
   s2.createSound( );  //prints Wow Wow
 }
}
```

Top 50+ Core Java Interview Questions Links:

Q01-Q10 | Q24-Q36 on classes, interfaces and generics | Q37-Q42 on GC and referencing | Q43-Q54 on Objects

Popular Posts

♦ 11 Spring boot interview questions & answers

825 views

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

766 views

18 Java scenarios based interview Questions and Answers

400 views

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

388 views

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

295 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

285 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

239 views

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

201 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**

‹ ♥ 30+ Java Code Review Checklist Items

♦ Q01-Q28: Top 50+ EE Java interview questions & answers ›

Posted in FAQ Core Java Job Interview Q&A Essentials**,** member-paid**,** Top 50+ FAQ Core Java Interview Q&A

Tags: Core Java FAQs**,** Novice FAQs**,** TopX

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