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

Posted on February 13, 2015 by Arulkumaran Kumaraswamipillai — 1 Comment ↓

Focus is on Java classes, interfaces and generics the interview questions and answers style. Java classes and interfaces are the building blocks.

Top 50+ Core Java Interview Questions Links:

Q01-Q10 | Q11-Q23 on OOP | Q37-Q42 on GC and referencing | Q43-Q54 on Objects

Q24. What happens when a parent and a child class have the same variable name?

A24. When both a parent class and its subclass have a field with the same name, this technique is called **variable shadowing** or **variable hiding**. The variable shadowing depends on the static type of the variable in which the object's reference is stored at compile time and NOT based on the dynamic type of the actual object stored at runtime as demonstrated in polymorphism via method overriding.

Q25. What happens when the parent and the child class has the same non-static method with the same signature?

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

open all | close all

- Ice Breaker Interview Q&A (7
- Core Java Interview Q&A (11
- JEE Interview Q&A (37)
- Pressed for time? Java/JEE I
  - Job Interview Ice Breaker (
  - FAQ Core Java Job Intervi
    - ♥♦ Q1-Q10: Top 50+ Co
    - ♦ Q11-Q23: Top 50+ Cor
    - ♦ Q24-Q36: Top 50+ Cor
    - ♦ Q37-Q42: Top 50+ Cor
    - ♦ Q43-Q54: Top 50+ Cor
    - 01: ♥♦ 15 Beginner level
    - 02: ♥♦ 10+ Java multi-th
  - FAQ JEE Job Interview Q&
  - FAQ Java Web Services In
  - Java Application Architectu
  - Hibernate Job Interview Es
  - Spring Job Interview Esser
  - Java Key Area Essentials (
  - OOP & FP Essentials (3)
  - Code Quality Job Interview
- SQL, XML, UML, JSON, Reg
- Hadoop & BigData Interview
- Java Architecture Interview Q
- Scala Interview Q&As (13)
- Spring, Hibernate, & Maven I
- Testing & Profiling/Sampling
- Other Interview Q&A for Java
- Free Java Interview Videos

16 Technical Key Areas

A25. Unlike variables, when a parent class and a child class each has a non-static method (aka an instance method) with the same signature, the method of the child class **overrides** the method of the parent class. The method overriding depends on the dynamic type of the actual object being stored and NOT the static type of the variable in which the object reference is stored. The dynamic type can only be evaluated at runtime. As you can see, the rules for variable shadowing and method overriding are directly opposed. The method overriding enables polymorphic behavior.

Q26. What happens when the parent and the child class has the same static method with the same signature?
A26. The behavior of static methods will be similar to the **variable shadowing** or **variable hiding**, and not recommended. It will be invoking the static method of the referencing static object type determined at compile time, and NOT the dynamic object type being stored at runtime.

The concepts of overriding, shadowing or hiding, and overloading are explained with code and examples.

Q27. What is the difference between an abstract class and an interface and when should you use them?
A27. In design, you want the base class to present only an interface for its derived classes. This means, you don't want anyone to actually instantiate an object of the base class.

Before Java 8:

Abstract class	Interface
Can maintain state in instance and static variables.	No state.
Have executable methods and abstract methods.	Have no implementation c
Can only extend one abstract class.	A class can implement any

```
1 public ClassB extends  ClassA {....}
```

```
1 public ClassB implements InterfaceA, InterfaceB {...}
```

Java 8 onwards:

Abstract class	Interface
Can maintain state in instance and static variables.	No state.

Have executable methods and abstract methods.	Have executable default methods and static helper methods.
Can only subclass one abstract class.	A class can implement any number of interfaces.

So, from Java 8 onwards, the key difference is only one class can extend an abstract class, but a class can implement more than one interfaces.

This is a very important “MUST KNOW” question, and if you are a beginner to intermediate level Java developer, then learn more

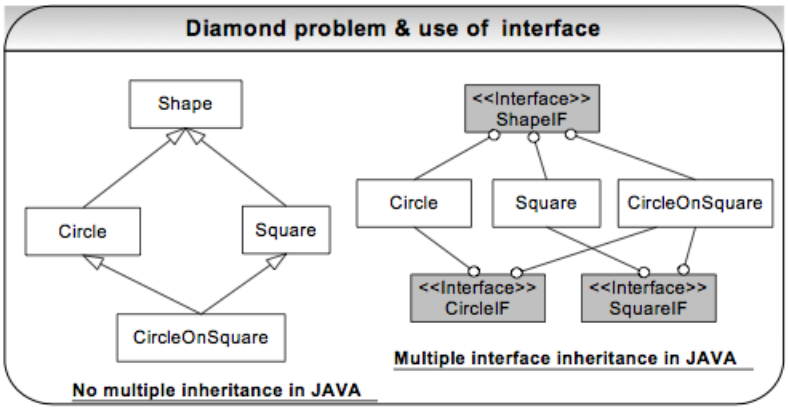
1. Java abstract classes Vs interfaces Q&A

2. Java classes and interfaces are the building blocks

3. Explain abstraction, encapsulation, Inheritance, and polymorphism with the given code?

Q28. When will you use an abstract class?
A28. In case where you want to use implementation inheritance then it is usually provided by an abstract base class. Abstract classes are excellent candidates inside of application frameworks. Abstract classes let you define some default behavior and force subclasses to provide any specific behavior. Care should be taken not to overuse implementation inheritance as discussed before. The **template method design pattern** is a good example to use an abstract class where the abstract class provides a default implementation.

Q29. When will you use an interface?
A29. For polymorphic interface inheritance, where the client wants to only deal with a type and does not care about the actual implementation, then use interfaces. If you need to change your design frequently, you should prefer using interface to abstract class. Coding to an interface reduces coupling. Another justification for using interfaces is that they solve the ‘diamond problem’ of traditional multiple inheritance as shown in the diagram. Java does not support multiple inheritance. Java only supports multiple interface inheritance. Interface will solve all the ambiguities caused by this ‘diamond problem’. Java 8 has introduced functional interfaces, and partially solves the diamond problem by allowing default and static method implementations in interfaces to inherit multiple behaviors.



Multiple inheritance and diamond issue

Strategy design pattern lets you swap new algorithms and processes into your program without altering the objects that use them.

Q30. What is a marker or a tag interface? Why there are some interfaces with no defined methods (i.e. marker interfaces) in Java?
A30. The interfaces with no defined methods act like markers. They just tell the compiler that the objects of the classes implementing the interfaces with no defined methods need to be treated differently. For example, *java.io.Serializable*, *java.lang.Cloneable*, *java.util.EventListener,* etc. Marker interfaces are also known as “tag” interfaces since they tag all the derived classes into a category based on their purpose.

Now with the introduction of annotations in Java 5, the marker interfaces make less sense from a design standpoint. Everything that can be done with a marker or tag interfaces in earlier versions of Java can now be done with annotations at runtime using reflection. One of the common problems with the marker or tag interfaces like *Serializable*, *Cloneable*, etc is that when a class implements them, all of its subclasses inherit them as well whether you want them to or not. You cannot force your subclasses to un-implement an interface. Annotations can have parameters of various kinds, and they’re much more flexible than the marker interfaces. This makes tag or marker interfaces obsolete, except for situations in which empty or tag interfaces can be checked at compile-time using the type-system in the compiler

Q31. What is a functional interface?
A31. Functional interfaces are introduced in Java 8 to allow default and static method implementations to enable functional programming (aka closures) with lambda expressions.

```
1 @FunctionalInterface
2 public interface Summable {
3       abstract int sum(int input1, int input2);
4 }
5
```

Q32. What does the @FunctionalInterface do, and how is it different from the @Override annotation?
A32. The **@Override** annotation takes advantage of the compiler checking to make sure you actually are overriding a method when you think you are. This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that you method does not actually override as you think it does. Secondly, it makes your code easier to understand because it is more obvious when methods are overwritten.

The annotation **@FunctionalInterface** acts similarly to @Override by signaling to the compiler that the interface is intended to be a functional interface. The compiler will then throw an error if the interface has multiple abstract methods.

Q33. What do you understand by the term type erasure with regards to generics?
A33. The term type erasures is used in Java generics. In the interest of backward compatibility, robustness of generics has been sacrificed through type erasure. Type erasure takes place at compile-time. So, after compiling List and List, both end up as List at runtime. They are both just lists.

```
1 List<String> myList = new ArrayList<String>( );  //in Java 6 ar
2 List<String> myList = new ArrayList<>(); // In Java 8 can use e
3
```

after compilation becomes

```
1 List myList = new ArrayList( );
2
```

Java generics differ from C++ templates. Java generics (at least until JDK 8), generate only one compiled version of a generic class or method regardless of the number of types used. During compile-time, all the parametrized type information within the angle brackets are erased and the compiled class file will look similar to code written prior to JDK 5.0. In other words, Java does not support runtime generics.

Q34. Why do you need generics?
A34. Generics was introduced in JDK 5.0, and allows you to abstract over types. Without generics, you could put heterogeneous objects into a collection. This can encourage developers to write programs that are harder to read and maintain. For example,

```
1 List list = new ArrayList( );
2 list.add(new Integer( ));
3 list.add("A String");
```

```
4 list.add(new Mango( ));
5
```

As demonstrated above, without generics you can add any type of object to a collection. This means you would not only have to use “instanceof” operator, but also have to explicitly cast any objects returned from this list. The code is also less readable. The following code with generics is not only more readable,

```
1 List<String> list1 = new ArrayList<String>( );
2 List<Integer> list2 = new ArrayList<Integer>( );
3
```

but also throws a compile time error if you try to add an Integer object to list1 or a String object to list2.

Q35. What are the differences among

– Raw or plain old collection type e.g. Collection
– Collection of unknown e.g. Collection<?>
– Collection of type object e.g. Collection<Object>

What is the super type for all the collection class?

A35. **1) The plain old Collection**: is a heterogeneous mixture or a mixed bag that contains elements of all types, for example Integer, String, Fruit, Vegetable, etc.

2) **The Collection<object>**: is also a heterogeneous mixture like the plain old Collection, but not the same and can be more restrictive than a plain old Collection discussed above. It is incorrect to think of this as the super type for a collection of any object types.

Unlike an Object class, which is a “super type” for all objects like *String*, *Integer*, *Fruit*, etc, *List<Object>* is not a super type for *List<String>*, *List<Integer>*, *List<Fruit>*, etc. So it is illegal to do the following:

```
1 List<object> list = new ArrayList<integer>( );            //il
2
```

Though *Integer* is a subtype of *Object*, *List<Integer>* is not a subtype of *List<Object>* because List of Objects is a bigger set comprising of elements of various types like Strings, Integers, Fruits, etc. A List of Integer should only contain Integers, hence the above line is illegal. If the above line was legal, then you can end up adding objects of any type to the list, violating the purpose of generics.

3) The Collection<?>: is a homogenous collection that represents a family of generic instantiations of Collection like *Collection<String>*, *Collection<Integer>*, *Collection<Fruit>*, etc.

Collection<?> is the super type for all generic collection as *Object[ ]* is the super type for all arrays.

```
1 List<?> list = new ArrayList<Integer>( );      //legal
2 List<? extends Number> list = new ArrayList<Integer>( );   //le
3
```

Note: An *Integer* is a sub type of a *Number*.

Q36. How will you go about deciding which of the following to use?

<Number>
? extends <Number>
? super <Number>

A36. Here is the guide:

1. Use the **? extends** wildcard if you need to retrieve object from a data structure. That is read only. You can't add elements to the collection.
2. Use the **? super** wildcard if you need to put objects in a data structure.
3. If you need to do both things (read and add elements), **don't use any wildcard**.

Generics are covered in detail:

1. Understanding Java generics with the help of scenarios.
2. Java generics interview questions & answers.
3. Java coding question on recursion and generics.

Top 50+ Core Java Interview Questions Links:

Q01-Q10 | Q11-Q23 on OOP | 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

767 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**

‹ Understanding Overriding, Hiding, and Overloading in Java? How does overriding give polymorphism?

♦ Q37-Q42: Top 50+ Core on Java Garbage Collection 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, Java/JEE FAQs, Novice FAQs, TopX

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

ashish.amg says:

February 8, 2016 at 4:20 pm

I need the detailed explanation of usage and when to use Abstract class and interface . Which is top most question in interview. Please elaborate it from architect perspective view.

Reply

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

↑