♦ Q43-Q54: Top 50+ Core on Java Objects Interview Questions & Answers

Posted on February 13, 2015 by Arulkumaran Kumaraswamipillai — No Comments ↓

FAQ interview questions answered on Java objects.

Top 50+ Core Java Interview Questions Links:

Q01-Q10 | Q11-Q23 on OOP | Q24-Q36 on classes, interfaces and generics | Q37-Q42 on GC and referencing

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

open all | close all

- Ice Breaker Interview
- Core Java Interview C
- JEE Interview Q&A (3
- Pressed for time? Jav
  - Job Interview Ice B
  - FAQ Core Java Job
    - ♥♦ 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 Essentia
  - Code Quality Job I

Q43. What can you tell about the performance of a *HashMap* compared to a *TreeMap*? Which one would you prefer?
A43. A balanced tree does have O (log n) performance. The *TreeMap* class in Java maintains key/value objects in a sorted order by using a red-black tree. A red-black tree is a balanced binary tree. Keeping the binary tree balanced ensures the fast insertion, removal, and lookup time of O (log n). This is not as fast as a *HashMap*, which is O(1) , but the *TreeMap* has the advantage of that the keys are in sorted order which opens up a lot of other capabilities.

Which one to choose?

The decision as to using an unordered collection like a *HashSet* or *HasMap* versus using a sorted data structure like a *TreeSet* or *TreeMap* depends mainly on the usage pattern, and to some extent on the data size and the environment you run it on. The practical reason for keeping the elements in sorted order is for frequent and faster retrieval of sorted data if the inserts and updates are frequent. If the need for a sorted result is infrequent like prior to producing a report or running a batch process, then maintaining an unordered collection and sorting them only when it is really required with *Collections.sort(…)* could sometimes be more efficient than maintaining the ordered elements. This is only an opinion, and no one can offer you a correct answer. Even the complexity theories like Big-O notation O(n) assume possibly large values of n. In practice, a O(n) algorithm can be much faster than a O(log n) algorithm, provided the data set that is handled is sufficiently small. So, always conduct a performance testing with the real life data to tune your code.

Q44. When providing a user defined key class for storing objects in the *HashMaps*, what methods do you have to provide or override (i.e. method overriding)?
A44. You should override the *equals( )* and *hashCode( )* methods from the *Object* class. The default implementation of the *equals( )* and *hashcode( )*, which are inherited from the *java.lang.Object* uses an object instance's memory location (e.g. Car@6c60f2ea). This can cause problems when two instances of the car objects have the same color but the

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

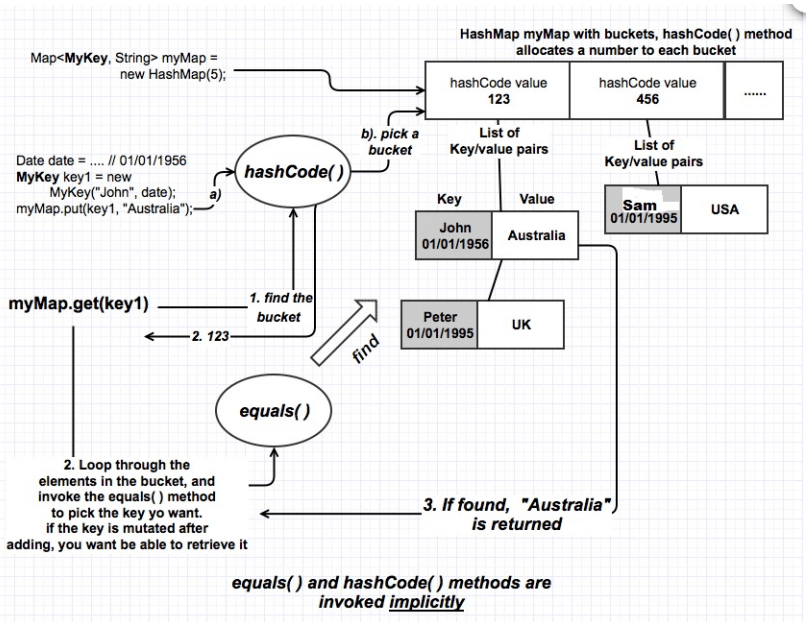
16 Technical Key Areas

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

inherited *equals( )* will return false because it uses the memory location, which is different for the two instances. Also, the *toString( )* method can be overridden to provide a proper string representation of your object.



Java equals vs hashCode

Note: Java *hashCode( )* and *equals( )* method have to be properly implemented. The map and set interfaces also use the *containsKey(Object key)* and *contains (Object o)* methods use the *equals( )* method to determine the return value – true/false. In the class defined by yourself, if you don't explicitly override these methods, it will have a default implementation. It returns true if and only if two objects refer to the same object, i.e., x == y is true.

Q45. Is the following statement true?

If you modify or add *equals* method then you must modify or add *hashCode* method as well.

A45. Yes. The contract between equals(…) and hashCode( ) can be summarized as shown below.

Hadoop & Spark Tuto
JEE Tutorials (19)
Scala Tutorials (1)
Spring & HIbernate Tu
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

- If a class overrides *equals(…)*, it must override *hashCode( )*.
- If 2 objects are equal, then their *hashCode* values must be same as well. The reverse is not true. If 2 objects have the same *hashCode* does not mean that those objects are equal as well. As per the above diagram, more than one object can result in the same hash code value say 345678965 and occupy the same bucket. These objects may or may not be equal. But if 2 objects are equal, they must occupy the same bucket with the same hash code value.
- If a field is not used in *equals(…)*, then it must not be used in *hashCode( )*.

Q46. When defining a user defined key class, what other consideration apart from overriding the *equals(..)* and *hashCode( )* method you should think of?
A46. Implement the user defined key class as an **immutable object**.

As per the code snippet shown below if you use a mutable user defined class "*UserKey*" as a *HashMap* key and subsequently if you mutate (i.e. modify via setter method e.g. *key.setName("Sam")*) the key after the object has been added to the *HashMap* then you will not be able to access the object later on. The original key object will still be in the *HashMap* (i.e. you can iterate through your *HashMap* and print it – both prints as "Sam" as opposed to "John" & Sam), but you cannot access it with *map.get(key)* or querying it with *map.containsKey(key)* will return false because the key "John" becomes "Sam" in the "List of keys" at the key index "345678965" if you mutate the key after adding. These types of errors can be very hard to trace and fix.

```
1  Map myMap = new HashMap(10);
2
3  //add the key "John"
4  UserKey key = new UserKey("John");  //Assume Use
5  myMap.put(key, "Sydney");
6
7  // same key object is mutated instead of creatin
8  // This line modifies the key value "John" to "S
9  // as shown in the diagram above. This means tha
10 // accessed. There will be two keys with "Sam" i
```

```
// values 345678965 and 76854676.

key.setName("Sam");

myMap.put(key, "Melbourne");

// The key cannot be accessed. The key hashes to
// 345678965 in the "Key index array" but cannot

myMap.get(new UserKey("John"));

```

Q47. What is an immutable object? Why is it a best practice to use immutable objects in Java?
A47. Immutable objects are objects whose state (i.e. the object's data) cannot change after construction. Examples of immutable objects from JDK include *String* and wrapper classes like *Integer*, *Double*, *Character*, etc.

- Immutable classes can greatly simplify programming by freely allowing you to cache and share the references to the immutable objects without having to defensively copy them or without having to worry about their values becoming stale or corrupted.
- Immutable classes are inherently thread-safe and you do not have to synchronize access to them to be used in a multi-threaded environment. So there are no chances for negative performance consequences as multiple threads can share the same instance.
- Eliminates the possibility of data becoming inaccessible when used as keys in *HashMaps* or as elements in *Sets*. These types of errors are hard to debug and fix.
- Eliminates the need for class invariant check once constructed.
- Allow h*ashCode( )* method to use lazy initialization, by caching its return value.
- Cloning is not required.
- Simpler to construct, use, and test due to its deterministic state.

Q48. How do you create an immutable type?

A48. 1) Make the class final so that it cannot be extended or use static factories and keep constructors private.

```
1 public final class MyImmutable { … }
2
```

2) Make fields private and final.

```
1 private final int[ ] myArray;
2
```

3) Don't provide any methods that can change the state of the immutable object in any way, not just *setXXX* methods, but any methods which can change the state.

4) The "this" reference is not allowed to escape during construction from the immutable class. Defensively copy references during construction.

```
1 public Person(Date birthDate) {
2  super( );
3  //defensively copy
4  this.birthDate = new Date(birthDate.getDate());
5 }
6
```

5) Don't return or expose the mutable references to the caller. This can be done by defensively copying the objects by deeply cloning them.

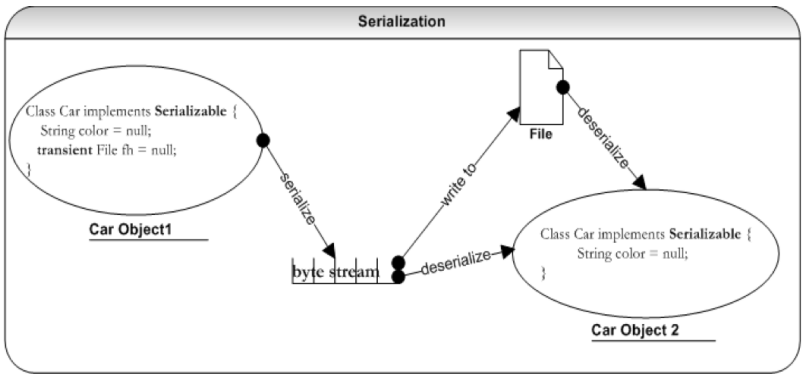
For example, when constructing a date object

```
1 //Don't let the date escape by returning a defens
2 public Date getBirthDate( ) {
3       //defensively copy so that original date do
4       return new Date(this.bithDate.getTime( ));
5 }
6
```

Another example would be when using a collection

```
1 public Collection&lt;role&gt; getRoles()
2 {
3  //returns immutable collection, so new roles can
4  return Collections.unmodifiableCollection(roles)
5 }
6
7
```

Q49. What is serialization? What rules do you need to follow?
A49. Object serialization is a process of reading or writing an object. It is a process of saving an object's state to a sequence of bytes, as well as a process of rebuilding those bytes back into a live object at some future time. An object is marked serializable by implementing the *java.io.Serializable* interface. This simply allows the serialization mechanism to verify that a class can be persisted, typically to a file. The common process of serialization is also called marshaling or deflating when an object is flattened into byte streams. The flattened byte streams can be unmarshaled or inflated back to an object.



Serialization

Rule #1: The object to be persisted must implement the Serializable interface or inherit that interface from its object hierarchy. Alternatively, you can use an Externalizable interface to have full control over your serialization process. For example, to construct an object from a pdf file.

Rule #2: The object to be persisted must mark all non-serializable fields as transient. For example, file handles, sockets, threads, etc.

Rule #3: You should make sure that all the included objects are also serializable. If any of the objects is not serializable, then it throws a NotSerializableException.

Rule #4: Base or parent class fields are only handled if the base class itself is serializable.

Rule #5: Serialization ignores static fields, because they are not part of any particular state.

Q51. How do you exclude a field of a class from serialization?
A51. By marking it as **transient**. The fields marked as transient in a serializable object will not be transmitted in the byte stream. An example would be a file handle, a database connection, a system thread, etc. Such objects are only meaningful locally. So they should be marked as transient in a serializable class.

Q52. What happens to static fields during serialization?
A52. Static fields are not serialized. Serialization persists only the state of a single object. Static fields are not part of the state of an object as they are effectively the state of the class shared by many other instances.

Q53. What are the benefits of serialization?
A53. 1) Allows you to persist objects with state to a text file on a disk, and re-assemble them by reading the file back. Application servers can do this to conserve memory. For example, stateful EJBs can be activated and passivated using serialization. The objects stored in an HTTP session should be serializable to support in-memory replication of sessions to achieve scalability.

2) Allows you to send objects from one Java process to another using sockets, RMI, RPC, etc. In other words passing objects between processes.

Allows you to deeply clone any arbitrary object graph.

3) Allows you to deeply clone any arbitrary object graph.

Q54. What is a serial version id?
A54. Say you create a "Pet" class, and instantiate it to "*myPet*", and write it out to an object stream. This flattened "*myPet*" object sits in the file system for some time. Meanwhile, if the "*Pet*" class is modified by adding a new field, and later on, when you try to read (i.e. deserialize or inflate) the flattened "*Pet*" object, you get the *java.io.InvalidClassException* – because all serializable classes are automatically given a unique identifier. This exception is thrown when the identifier of the class is not equal to the identifier of the flattened object. If you really think about it, the exception is thrown because of the addition of the new field. You can avoid this exception being thrown by controlling the versioning yourself by declaring an explicit serialVersionUID. There is also a small performance benefit in explicitly declaring your serialVersionUID because it does not have to be calculated.

So, it is a best practice to add your own serialVersionUID to your Serializable classes as soon as you define them. If no serialVersionUID is declared, JVM will use its own algorithm to generate a default SerialVersionUID. The default serialVersionUID computation is highly sensitive to class details and may vary from different JVM implementation, and result in an unexpected *InvalidClassExceptions* during deserialization process.

Top 50+ Core Java Interview Questions Links:

Q01-Q10 | Q11-Q23 on OOP | Q24-Q36 on classes, interfaces and generics | Q37-Q42 on GC and referencing

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

240 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**

‹ ♦ Q37-Q42: Top 50+ Core on Java Garbage Collection Interview Questions & Answers

Understanding Java locks, multi-threading, and synchronized keyword ›

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

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