03: ♦ 9 Core Java Concepts you can't afford to not know

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

Q1. How will you go about explaining the following Java concepts to a beginner who is starting to learn Java?

1. Process Vs Threads
2. Heap versus Stack
3. Local variables versus instance variables
4. How do threads communicate with each other?
5. Are Java methods reentrant?
6. Does Java support recursive method calls?
7. Object creation and Garbage collection
8. Can you garbage collect objects that have a circular

reference?
9. Producer-consumer design pattern.

A1. he best way to do this is to write some basic code and then have a pictorial representation as to how the objects are created, how do threads work and communicate with each other, and what is stored in a heap and what is stored in a stack, etc.

Here is an example of a producer thread (thread-0) producing by incrementing the counter from 0, and the consumer thread (i.e. thread-1) consumes by decrementing the counter. These two user created threads are spawned by the main thread, which is created by the JVM and is always there by default. The **ConsumerProducer** is the shared object with synchronized methods that communicate with each other via wait( ) and notifyAll methods. Only one of the two synchronized methods can be executed at a time. The wait( ) call in consume( ) relinquishes the lock to the produce( ) method, and once the produce method has incremented the count, it notifies all threads and one of the waiting threads will resume. In this example, there is only one waiting consumer (i.e. Thread-1) thread. So, both threads will be communicating with each other via the wait( ) and notifyAll( ) calls in the shared object **ConsumerProducer**. This is an example of the producer-consumer design pattern.

Firstly, look at the code and then the diagram. The diagram is simplified to get an understanding and should not be construed as exactly what happens in the JVM.

```
1  public class ConsumerProducer {
2
3    private int count;
4
5    public synchronized void consume() {
6       while (count == 0) {  // keep waiting if no
7          try {
8             wait(); // give up lock and wait
9          } catch (InterruptedException e) {
10            // keep trying
11         }
12      }
13
14      count--;              // consume
15      System.out.println(Thread.currentThread().g
```

```
16    }
17
18    public synchronized void produce() {
19        count++;                //produce
20        System.out.println(Thread.currentThread().g
21        notifyAll();            // notify waiting thre
22    }
23
24 }
```

The main thread spawns a consumer and a producer thread. The ConsumerProducer is shared between two threads. The boolean flag is used to signal if it is a consumer thread or a producer thread to invoke the relevant methods.

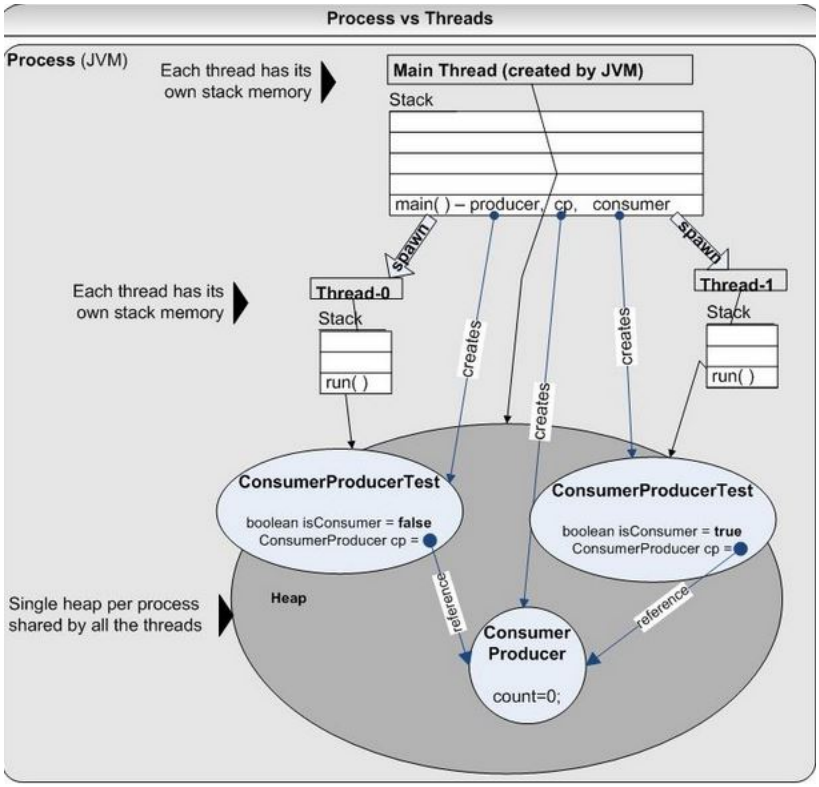
```
1  public class ConsumerProducerTest implements Run
2
3    boolean isConsumer;
4    ConsumerProducer cp;
5
6    public ConsumerProducerTest(boolean isConsumer
7        this.isConsumer = isConsumer;
8        this.cp = cp;
9    }
10
11   public static void main(String[] args) {
12       ConsumerProducer cp = new ConsumerProducer
13
14       Thread producer = new Thread(new ConsumerP
15       Thread consumer = new Thread(new ConsumerP
16
17       producer.start();
18       consumer.start();
19   }
20
21   @Override
22   public void run() {
23     for (int i = 1; i <= 10; i++) {
24        if (!isConsumer) {
25           cp.produce();
26        } else {
27           cp.consume();
28        }
29     }
30
31     //try with introducing a sleep for 100ms.
32   }
33 }
34
```

The output will vary, but the last thing consumed will be 0.

```
1  Thread-0 after producing 1
2  Thread-0 after producing 2
3  Thread-0 after producing 3
4  Thread-0 after producing 4
5  Thread-0 after producing 5
```

```
 6  Thread-0 after producing 6
 7  Thread-0 after producing 7
 8  Thread-0 after producing 8
 9  Thread-0 after producing 9
10  Thread-0 after producing 10
11  Thread-1 after consuming 9
12  Thread-1 after consuming 8
13  Thread-1 after consuming 7
14  Thread-1 after consuming 6
15  Thread-1 after consuming 5
16  Thread-1 after consuming 4
17  Thread-1 after consuming 3
18  Thread-1 after consuming 2
19  Thread-1 after consuming 1
20  Thread-1 after consuming 0
21
```

The above code can be diagrammatically represented as shown below.



Java beginner concepts

- The JVM is a process. As you could see in the diagram above that **each thread has its own stack**, but **share the same heap space**. One heap space per process.

- The **local variables** like producer, cp, and consumer are **stored in the stack** along with the method calls

like main( ), run( ), etc.

- The **objects are created in the heap**. For example, 2 instances of *ConsumerProducerTest* and a single instance of *ConsumerProducer*. Unless the methods of these instances are properly managed, multiple threads can concurrently access these intsnaces to cause thread-safety issues.

- There are scenarios where the threads want to commuincate with each other. For example, a worker thread produces something and a consumer thread consumes what the producer thread consumed. This can be achieved through thread coordination with the wait and notify( ) / notifyAll( ) methods. In the above example, since both the produce( ) and consume( ) methods of the object *ConsumerProducer* are synchronized, only one thread can execute either one of the methods. If the count == 0, the consumer thread relinquishes the lock by invoking the wait( ) method and waits for it to be notified again to resume. The producer thread will increment the counter and then will notify all the waiting threads via the notifyAll( ) method. One of the waiting threads will then be able to resume from its waiting state.

- The local variable **isConsumer** is used to differentiate between a consumer thread and a producer thread.

- The **count** is an instance variable shared by both the consumer and producer thread in a thread-safe manner.

Here are the points that answers the above questions.

- Each time an object is created in Java it goes into the area of memory known as heap.
- The primitive variables like int, double, etc are allocated in the stack if they are local variables and in

the heap if they are instance variables (i.e. fields of a class).

- Java is a stack based language and local variables are pushed into stack when a method is invoked and stack pointer is decremented when a method call is completed.
- In a multi-threaded application, each thread will have its own stack but will share the same heap. This is why care should be taken in your code to avoid any concurrent access issues in the heap space.
- The stack is thread-safe because each thread will have its own stack with say 1MB RAM allocated for each thread but the heap is not thread-safe unless guarded with synchronization through your code. The stack space can be increased with the –Xss option.
- All Java methods are automatically **re-entrant**. It means that several threads can be executing the same method at once, each with its own copy of the local variables.
- A Java method may call itself without needing any special declarations. This is known as a **recursive** method call. Given enough stack space, recursive method calls are perfectly valid in Java though it is tough to debug. Recursive methods are useful in removing iterations from many sorts of algorithms. All recursive functions are re-entrant but not all re-entrant functions are recursive.
- **Idempotent** methods are methods, which are written in such a way that repeated calls to the same method with the same arguments yield same results.
- Each time an object is created in Java, it goes into the area of memory known as heap. The Java heap is called the **garbage collectable heap**. The **garbage collection cannot be forced.** The garbage collector runs in low memory situations. When it runs, it releases the memory allocated by an unreachable object. The garbage collector runs on a low priority daemon (i.e. background thread). You can nicely ask the garbage collector to collect garbage by calling System.gc( ) but you can't force it.

- An object's life has no meaning unless something has reference to it. If you can't reach it then you can't ask it to do anything. Then the object becomes unreachable and the garbage collector will figure it out. Java automatically collects all the unreachable objects periodically and releases the memory consumed by those unreachable objects to be used by the future reachable objects.
- If two objects have **circular reference** with each other in the heap, but none of them are reachable from any thread, then those circular referenced objects can be garbage collected.

Q3. What will be the output of the following code snippet?

```
1  public class MethodOverrideVsOverload {
2
3     public boolean equals( MethodOverrideVsOverlo
4        System.out.println("MethodOverrideVsOverlo
5        return true;
6     }
7
8     public static void main(String[] args) {
9       Object o1 = new MethodOverrideVsOverload();
10      Object o2 = new MethodOverrideVsOverload();
11
12      MethodOverrideVsOverload o3 = new MethodOve
13      MethodOverrideVsOverload o4 = new MethodOve
14
15      if(o1.equals(o2)){
16        System.out.println("objects o1 and o2 are
17      }
18
19      if(o3.equals(o4)){
20        System.out.println("objects o3 and o4 are
21      }
22    }
23 }
24
```

A3. The output will be

```
1 MethodOverrideVsOverload equals method reached
2 objects o3 and o4 are equal
3
```

What concepts does this question try to test?

- In Java, a class can only extend a single class (i.e. single inheritance), and when it does not explicitly extend a class, it **implicitly** extends the class **Object**. So, *MethodOverrideVsOverload* implicitly extends the class Object.
- The majority of the non final Object class methods are meant to be overridden by the sub classes.

```
1 public boolean equals(Object obj); // make
2 public int hashCode();
3 public String toString();
```

- The method overloading takes place at **compile time** (i.e. static binding) and method overriding takes place at **runtime** (i.e. dynamic binding). Static binding means the JVM decides, which class or method to call during compile time. Dynamic binding means, the JVM decides, which class or method to call during runtime. The polymorphism is possible because of dynamic binding.
- The method overriding must adhere to the following rules

Arguments	Must not change.
Return type	Can't change except for covariant (subtype) returns.
Exceptions	The extending class can eliminate or call fewer exceptions than its parent, but must not throw new or broader checked exceptions.
Access	Must not be more restrictive than the class it extends. Can be less restrictive.
Invocation	Which method to call is based on object type, at runtime time (i.e. dynamic binding).

Now, if you look at the above code
The "*equals(MethodOverrideVsOverload other)"* method in class *MethodOverrideVsOverload* **does not** actually override the *Object* class's "*public boolean equals(Object obj)"* method. This is because it fails to adhere to the "Arguments" rule as both methods have different arguments as one is of type "*MethodOverrideVsOverload* " and the other of type "*Object*" . So, the two methods are **overloaded** (happens at compile time) and not **overridden**.

So, w**hen o1.equals(o2) is invoked**, the *public boolean equals(Object obj) method* of the object class is invoked because during compile time, the o1 and o2 are of type Object.The Object class's equals(…) method returns false as it compares the memory address (e.g. Object@235f56 and Object@653af32) of both objects.

When o3.equals(o4) is invoked, the "*equals(MethodOverrideVsOverload other)"* of the class *MethodOverrideVsOverload* is invoked as during compile time o3 and o4 are of type *MethodOverrideVsOverload,* hence you get the above output.

What follow on questions can you expect?

Q. How will you fix the above issue?
A. In Java 5, annotations were introduced and one of the handy **compile time annotations** is the @override, which will ensure that the methods are overridden correctly. If you had this annotation, when you override it incorrectly as in the above example, a compile time error will be thrown.

So, to fix it, add the @override annotation to the "boolean equals(MethodOverrideVsOverload other)" of the *MethodOverrideVsOverload* class. This will give you a compile time error indicating that the method is not properly overridden. Now, fix the method signature by changing the argument type in the method signature from

"*MethodOverrideVsOverload*" to "*Object*" as shown below.

```
1  public class MethodOverrideVsOverload {
2
3      @Override
4      public boolean equals( Object other ) {
5          System.out.println("MethodOverrideVsOverl
6          return true;
7      }
8
9      public static void main(String[] args) {
10        Object o1 = new MethodOverrideVsOverload()
11
12        Object o2 = new MethodOverrideVsOverload()
13
14
15        MethodOverrideVsOverload o3 = new MethodOv
16
17        MethodOverrideVsOverload o4 = new MethodOv
18
19
20        if(o1.equals(o2)){
21          System.out.println("objects o1 and o2 ar
22        }
23
24        if(o3.equals(o4)){
25          System.out.println("objects o3 and o4 ar
26        }
27     }
28 }
29
```

The output will be

```
1 MethodOverrideVsOverload equals method reached
2 objects o1 and o2 are equal
3 MethodOverrideVsOverload equals method reached
4 objects o3 and o4 are equal
5
```

This is because now the methods are overridden and this happens at runtime. This is a bit tricky question, and think out loud at the interview to show that you understand the fundamentals.

Popular Posts

- ♦ 11 Spring boot interview questions & answers

  852 views

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

825 views

18 Java scenarios based interview Questions and Answers

447 views

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

400 views

♦ 7 Java debugging interview questions & answers

311 views

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

301 views

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

292 views

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

286 views

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

263 views

8 Git Source control system interview questions & answers

215 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.

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.

‹ How will you go about improving on the following Java code?

Recursion Vs Tail Recursion ›

Posted in Java Overview, member-paid

Tags: Java/JEE FAQs, Novice FAQs

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

As a Java Architect

Java architecture & design concepts interview Q&As with diagrams | What should be a typical Java EE architecture?

Senior Java developers must have a good handle on

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 Ir
- 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
- Hadoop & Spark Tuto
- JEE Tutorials (19)
- Scala Tutorials (1)
- Spring & HIbernate Tu
- Tools Tutorials (19)
- Other Tutorials (45)

Preparing for Java written & coding tests

open all | close all
- ♦ Complete the given
- Can you write code? (
- 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…to go places?

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

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

The contents in this Java-Success are copy righted. The author has the right to correct or enhance the current content without any prior notice.

These are general advice only, and one needs to take his/her own circumstances into consideration. The author will not be held liable for any damages caused or alleged to be caused either directly or indirectly by these materials and resources. Any trademarked names or labels used in this blog remain the property of their respective trademark owners. No guarantees are made regarding the accuracy or usefulness of content, though I do make an effort to be accurate. Links to external sites do not imply endorsement of the linked-to sites.

↑