Core Java Interview Questions-Part1

Notebook:	Core Java		
Created:	23-06-2018 16:38	**Updated:**	23-06-2018 17:20
URL:	https://www.java-success.com/hashmap-and-how-it-works/		

8-06: ♥♦ HashMap & HashSet and how do they internally work? What is a hashing function? | Java-Success.com

SourceURL: https://www.java-success.com/hashmap-and-how-it-works/

06: ♥♦ HashMap & HashSet and how do they internally work? What is a hashing function?

Posted on February 9, 2016 by  Arulkumaran Kumaraswamipillai Posted in 11 - FAQs Free, Collection and Data structures, Objects, Understanding Core Java

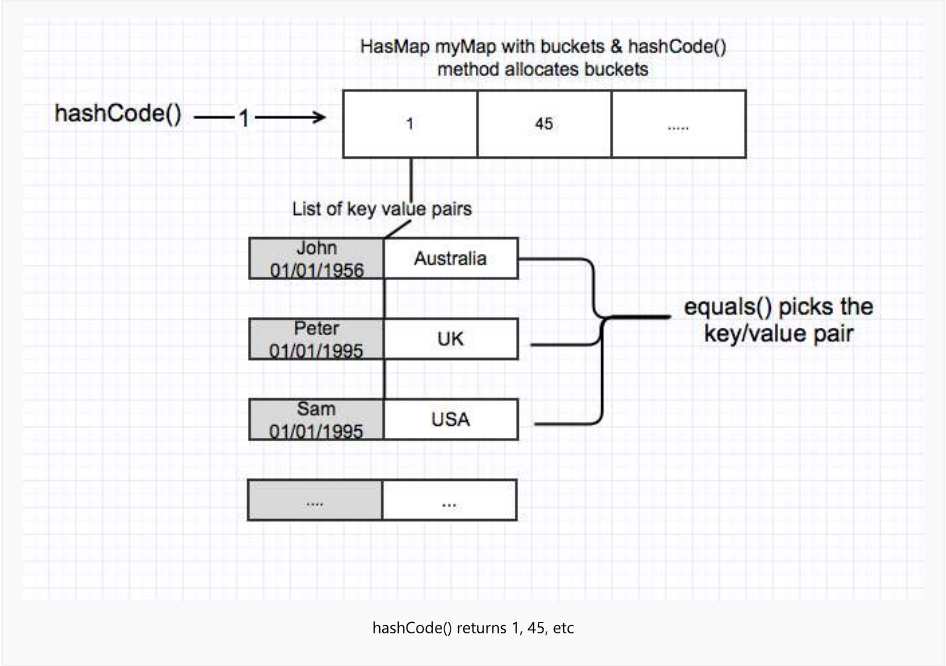
HashMap & **HashSet** are not only one of the frequently used data structures, but also one of the popular interview topics.

Q1. How does a HashMap store data?
A1. As key/value pairs. You can store and retrieve values with the keys.

Q2. What is the HashMap lookup time in Big O notation?
A2. It is O(**n**) = O(k * n). On average it is O(1) if the hashCode() method spreads out the buckets as discussed below.

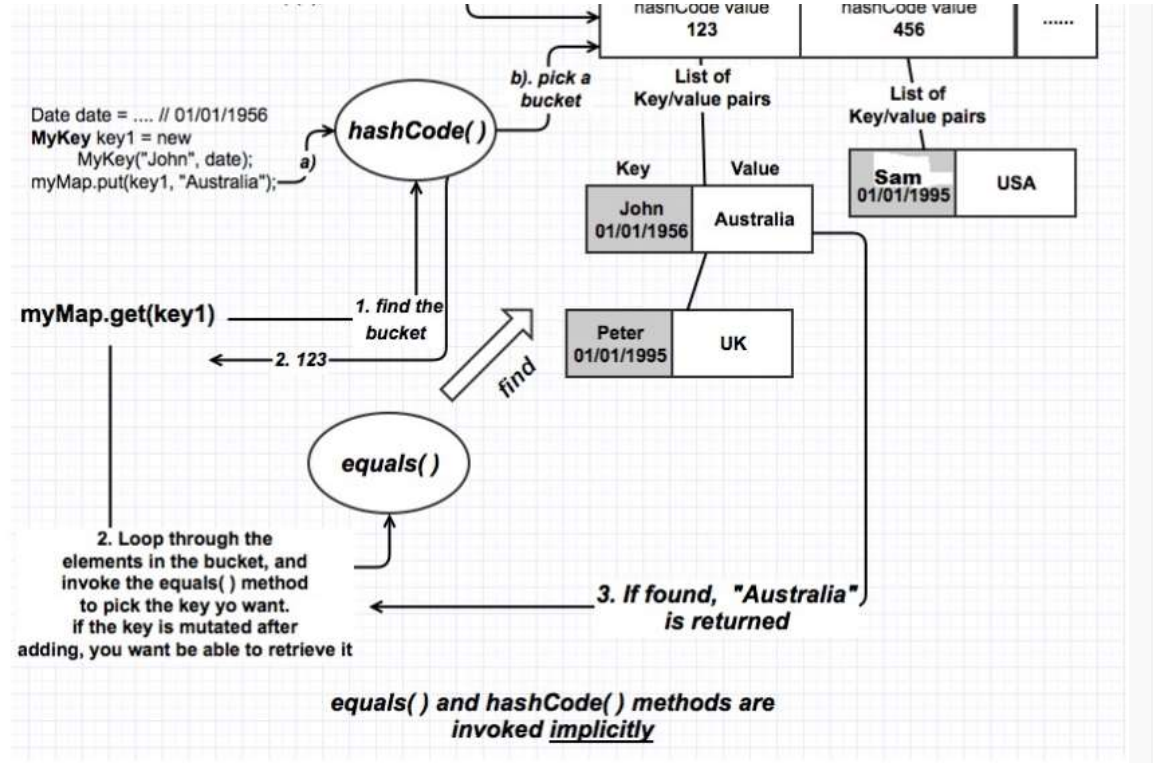
Q3 How does a HashMap internally store data?
A3. A backing array for the buckets and a **linked list** (until Java 8) and a "**binary tree**" (Java 8 onwards) to store the values..

Backing Array with buckets: as shown below.



hashCode() returns 1, 45, etc

1) When you put an object into a map with a key and a value, **hashCode()** method is implicitly invoked, and hash code value say 123 is returned. Two different keys can return the same hash value. A good hashing algorithm spreads out the numbers. In the above example, let's assume that ("John",01/01/1956) key and ("Peter", 01/01/1995) key return the same hash value of **123**.



Java equals vs hashCode

2) Now when a hashCode value of say 123 is returned and the initial HashMap capacity is say 10, how does it know which index of the backing array to store it in? This is where the HashMap internally invokes the **hash(int)** & **indexFor(int h, int length)** methods. This is known as the **hashing** function. This function in a very simple explanation is like

```
hashCode() % capacity

123 % 10 = 3
456 % 10 = 6
```

This means the "hashcode = 123" gets stored in index 3 in the backing array. So you get numbers between **0** and **9** for the capacity of 10. Once the HashMap reaches its 75% of the capacity indicated by the default hash factor of 0.75, the backing array's capacity is doubled and the **rehashing** takes place to reallocate the buckets for the new capacity of 20.

```
hashCode() % capacity

123 % 20 = 3
456 % 20 = 16
```

Now the above modulus operator approach of rehashing has a flaw. What if the hashCode is a negative number? You don't want a negative index. Hence, an improved hashing formula would be to shift out the sign bit and then calculate the remainder with the modulus (i.e. %) operator.

```
(123  & 0x7FFFFFF) % 20 = 3
(456 & 0x7FFFFFF) % 20 = 16
```

This ensures that you get a positive index value. If you look at the Java 8 HashMap source code, its implementation uses

a). Protect against poorer hashes by only extracting the important lower bits.

```
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
```

```
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

b). Determine the index based on the **hashCode** and **capacity**.

```
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

Actual name value pairs are stored as a key/value pair LinkedList

As shown in the diagram, the key/value pairs are stored as linked list. It is important to understand that two different keys can result in the same hashcode say 123, and get stored in the same bucket. For example, "John, 01/01/1956" & "Peter, 01/01/1995" in the above example. So, how do you retrieve just "John, 01/01/1956"? This is where you key class's **equals()** method gets invoked. It loops through the items in the LinkedList bucket "123" to retrieve the key "John, 01/01/1956" by using the equals() method to find a match. This is why it is important to implement the **hashCode()** & **equals()** method in your key class. If you are using an existing wrapper class like Integer or the String as a key, it is already implemented. If you write your own key class like "MyKey" with name and DOB as the attributes as in "John, 01/01/1956", you are responsible for properly implementing these methods.

Q5. Why is it a good practice to appropriately set the initial capacity of a HashMap?
A5. To minimize the occurrences of **rehashing**.

Q6. How does a HashSet internally store data?
A6. A HashSet internally uses a HashMap. It stores the element as both key and value.

Q7. What is the effect of implementing a poor hashcode() for an object?
A7. The invocation of hashCode() method should return different values for different objects. If it returns same values for different objects then this results in more key/value pairs getting stored against the same bucket. This **adversely impacts performance** of HashMaps & HashSets.

Relevant topics

1) 5 Java Object class methods interview questions & answers.

2) HashMap from scratch Java example

3) Stack from scratch Java example

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ 03: Java GC tuning for low latency applications

Huffman coding in Java ›

Tags: Free Content, Free FAQs

8-06: ♥♦ HashMap & HashSet and how do they internally work? What is a hashing function? | Java-Success.com

SourceURL: https://www.java-success.com/hashmap-and-how-it-works/

06: ♥♦ HashMap & HashSet and how do they internally work? What is a hashing function?

Posted on February 9, 2016 by Arulkumaran Kumaraswamipillai Posted in 11 - FAQs Free, Collection and Data structures, Objects, Understanding Core Java

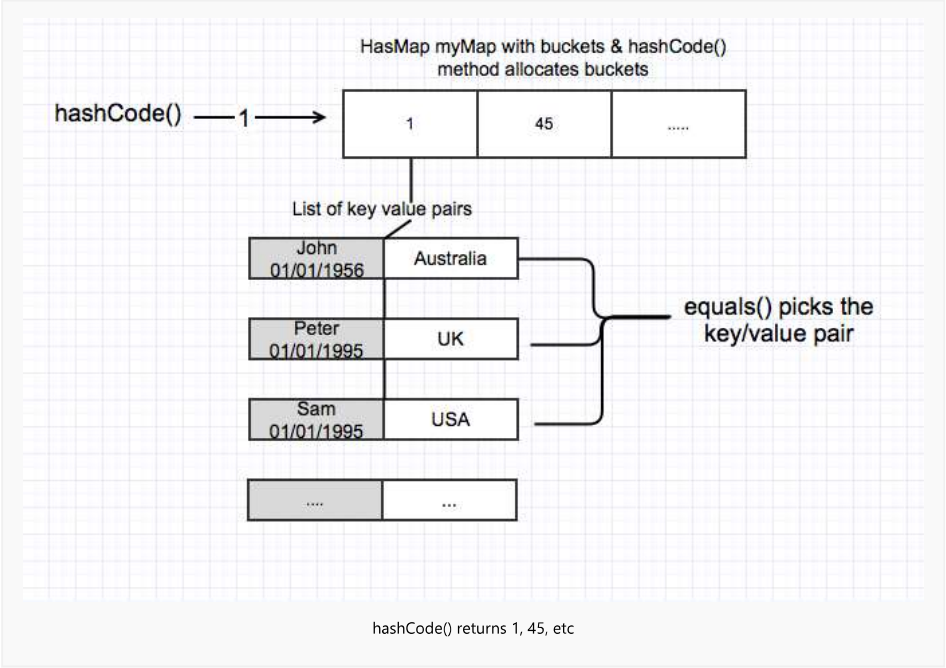
HashMap & **HashSet** are not only one of the frequently used data structures, but also one of the popular interview topics.

Q1. How does a HashMap store data?
A1. As key/value pairs. You can store and retrieve values with the keys.

Q2. What is the HashMap lookup time in Big O notation?
A2. It is O(**n**) = O(k * n). On average it is O(1) if the hashCode() method spreads out the buckets as discussed below.

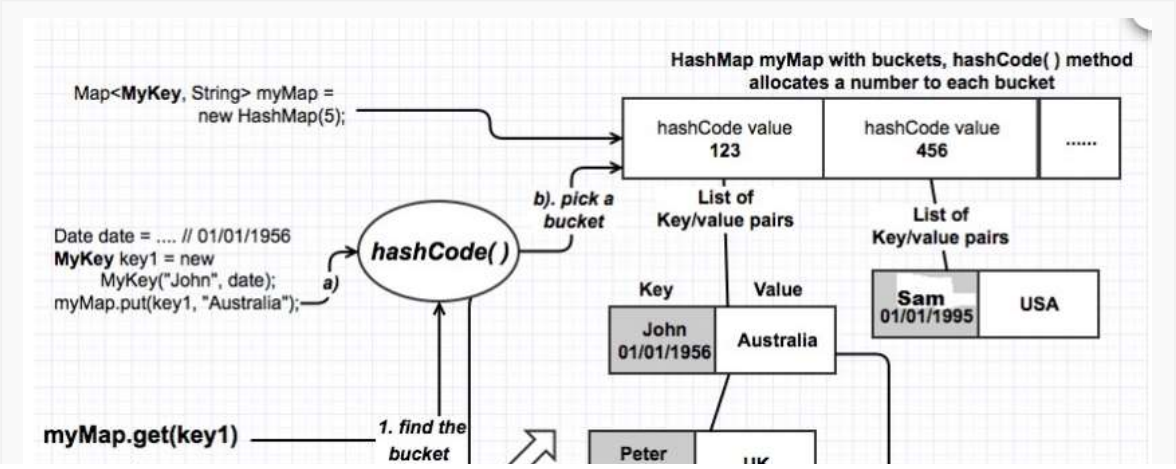
Q3 How does a HashMap internally store data?
A3. A backing array for the buckets and a **linked list** (until Java 8) and a "**binary tree**" (Java 8 onwards) to store the values..

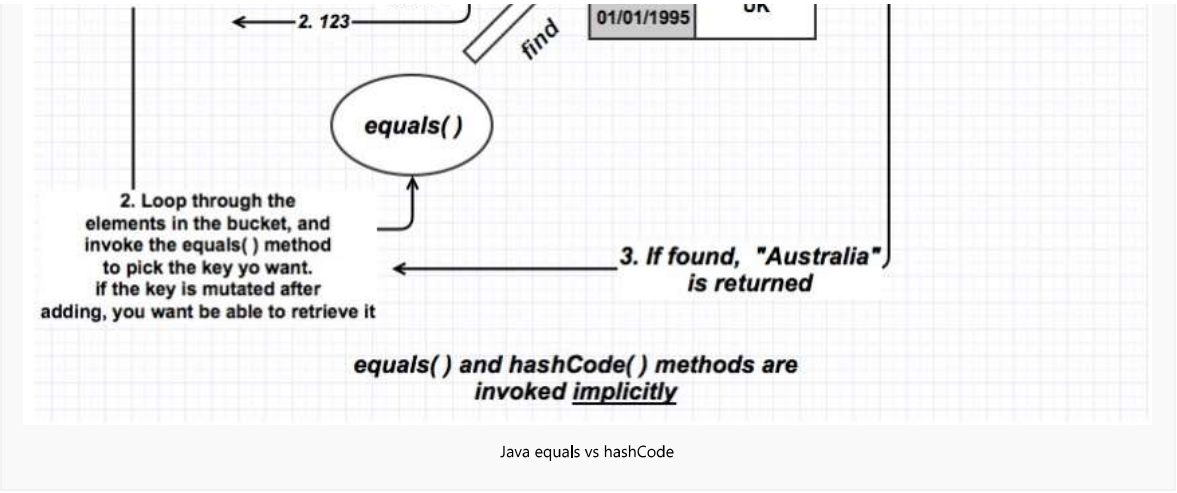
Backing Array with buckets: as shown below.



hashCode() returns 1, 45, etc

1) When you put an object into a map with a key and a value, **hashCode()** method is implicitly invoked, and hash code value say 123 is returned. Two different keys can return the same hash value. A good hashing algorithm spreads out the numbers. In the above example, let's assume that ("John",01/01/1956) key and ("Peter", 01/01/1995) key return the same hash value of **123**.





Java equals vs hashCode

2) Now when a hashCode value of say 123 is returned and the initial HashMap capacity is say 10, how does it know which index of the backing array to store it in? This is where the HashMap internally invokes the **hash(int)** & **indexFor(int h, int length)** methods. This is known as the **hashing** function. This function in a very simple explanation is like

```
hashCode() % capacity

123 % 10 = 3
456 % 10 = 6
```

This means the "hashcode = 123" gets stored in index 3 in the backing array. So you get numbers between **0** and **9** for the capacity of 10. Once the HashMap reaches its 75% of the capacity indicated by the default hash factor of 0.75, the backing array's capacity is doubled and the **rehashing** takes place to reallocate the buckets for the new capacity of 20.

```
hashCode() % capacity

123 % 20 = 3
456 % 20 = 16
```

Now the above modulus operator approach of rehashing has a flaw. What if the hashCode is a negative number? You don't want a negative index. Hence, an improved hashing formula would be to shift out the sign bit and then calculate the remainder with the modulus (i.e. %) operator.

```
(123  & 0x7FFFFFF) % 20 = 3
(456 & 0x7FFFFFF) % 20 = 16
```

This ensures that you get a positive index value. If you look at the Java 8 HashMap source code, its implementation uses

a). Protect against poorer hashes by only extracting the important lower bits.

```
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

b). Determine the index based on the **hashCode** and **capacity**.

```
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

Actual name value pairs are stored as a key/value pair LinkedList

As shown in the diagram, the key/value pairs are stored as linked list. It is important to understand that two different keys can result in the same hashcode say 123, and get stored in the same bucket. For example, "John, 01/01/1956" & "Peter, 01/01/1995" in the above example. So, how do you retrieve just "John, 01/01/1956"? This is where you key class's **equals()** method gets invoked. It loops through the items in the LinkedList bucket "123" to retrieve the key "John, 01/01/1956" by using the equals() method to find a match. This is why it is important to implement the **hashCode()** & **equals()** method in your key class. If you are using an existing wrapper class like Integer or the String as a key, it is already implemented. If you write your own key class like "MyKey" with name and DOB as the attributes as in "John, 01/01/1956", you are responsible for properly implementing these methods.

Q5. Why is it a good practice to appropriately set the initial capacity of a HashMap?
A5. To minimize the occurrences of **rehashing**.

Q6. How does a HashSet internally store data?
A6. A HashSet internally uses a HashMap. It stores the element as both key and value.

Q7. What is the effect of implementing a poor hashcode() for an object?
A7. The invocation of hashCode() method should return different values for different objects. If it returns same values for different objects then this results in more key/value pairs getting stored against the same bucket. This **adversely impacts performance** of HashMaps & HashSets.

Relevant topics

1) 5 Java Object class methods interview questions & answers.

2) HashMap from scratch Java example

3) Stack from scratch Java example

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ 03: Java GC tuning for low latency applications

Huffman coding in Java ›

Tags: Free Content, Free FAQs

5-Understanding compile time vs runtime relating to Java

SourceURL: https://www.java-success.com/compile-time-vs-run-time-basics-interview-qa/

02: ♥♦ Java Compile-time Vs Run-time Interview Q&As

Posted on September 17, 2014 by Arulkumaran Kumaraswamipillai Posted in 11 - FAQs Free, Java Overview

During development and design, one needs to think in terms of **compile-time, run-time**, and **build-time**. It will also help you understand the fundamentals better. These are beginner to intermediate level questions.

Q1. What is the difference between line A & line B in the following code snippet?

```
1 public class ConstantFolding {
```

```
public class ConstantFolding {
    static final  int number1 = 5;
    static final  int number2 = 6;
    static int number3 = 5;
    static int number4= 6;

    public static void main(String[ ] args) {
          int product1 = number1 * number2;             //line A
          int product2 = number3 * number4;             //line B
    }
}
```

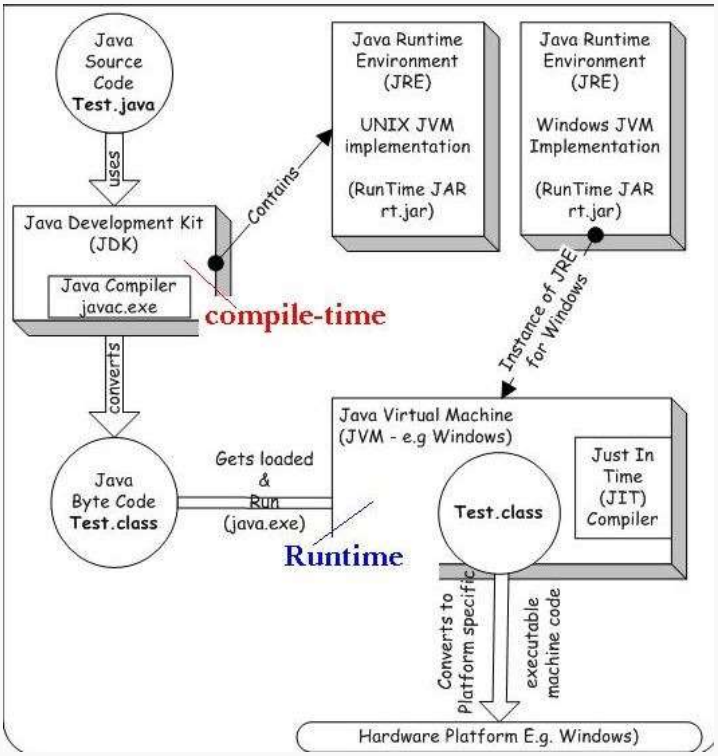
A1. Line A, evaluates the product at **compile-time**, and Line B evaluates the product at **runtime**. If you use a Java Decompiler (e.g. jd-gui), and decompile the compiled *ConstantFolding.class* file, you will see whyas shown below.

```
public class ConstantFolding
{
  static final int number1 = 5;
  static final int number2 = 6;
  static int number3 = 5;
  static int number4 = 6;

  public static void main(String[ ] args)
  {
      int product1 = 30;
      int product2 = number3 * number4;
  }
}
```

Constant folding is an optimization technique used by the Java compiler. Since final variables cannot change, they can be optimized. Java Decompiler and javap command are handy tool for inspecting the compiled (i.e. byte code) code.

Q2. Can you think of other scenarios other than code optimization, where inspecting a compiled code is useful?
A2. Generics in Java are compile-time constructs, and it is very handy to inspect a compiled class file to understand and troubleshoot generics.

Q3. Does this happen during compile-time, runtime, or both?
A3.



compile-time Vs run-time

Method overloading: This happens at compile-time. This is also called compile-time polymorphism because the compiler must decide how to select which method to run based on the data types of the arguments.

```
public class {
     public static void evaluate(String param1);  // method #1
     public static void evaluate(int param1);     // method #2
}
```

If the compiler were to compile the statement:

```
evaluate("My Test Argument passed to param1");
```

it could see that the argument was a string literal, and generate byte code that called method #1.

Method overriding: This happens at runtime. This is also called runtime polymorphism because the compiler does not and cannot know which method to call. Instead, the JVM must make the determination while the code is running.

```
public class A {
   public int compute(int input) {          //method #3
        return 3 * input;
   }
}

public class B extends A {
   @Override
   public int compute(int input) {          //method #4
        return 4 * input;
   }
}

```

The method compute(..) in subclass "B" overrides the method compute(..) in super class "A". If the compiler has to compile the following method,

```
public int evaluate(A reference, int arg2)  {
     int result = reference.compute(arg2);
}
```

The compiler would not know whether the input argument 'reference' is of type "A" or type "B". This must be determined during runtime whether to call method #3 or method #4 depending on what type of object (i.e. instance of Class A or instance of Class B) is assigned to input variable "reference".

Generics (aka type checking): This happens at compile-time. The compiler checks for the type correctness of the program and translates or rewrites the code that uses generics into non-generic code that can be executed in the current JVM. This technique is known as "type erasure". In other words, the compiler erases all generic type information contained within the angle brackets to achieve backward compatibility with JRE 1.4.0 or earlier editions.

```
List<String> myList = new ArrayList<String>(10);
```

after compilation becomes:

```
List myList = new ArrayList(10);
```

Annotations: You can have either run-time or compile-time annotations.

```
public class B extends A {
   @Override
    public int compute(int input){       //method #4
        return 4 * input;
    }
}
```

@Override is a simple compile-time annotation to catch little mistakes like typing tostring( ) instead of toString( ) in a subclass. User defined annotations can be processed at compile-time using the Annotation Processing Tool (APT) that comes with Java 5. In Java 6, this is included as part of the compiler itself.

```
public class MyTest{
    @Test
     public void testEmptyness( ){
         org.junit.Assert.assertTrue(getList( ).isEmpty( ));
     }
```

```
6
7       private List getList( ){
8          //implemenation goes here
9       }
10  }
```

@Test is an annotation that JUnit framework uses at runtime with the help of reflection to determine which method(s) to execute within a test class.

```
1  @Test (timeout=100)
2  public void testTimeout( ) {
3      while(true);   //infinite loop
4  }
```

The above test fails if it takes more than 100ms to execute at runtime.

```
1  @Test (expected=IndexOutOfBoundsException.class)
2  public void testOutOfBounds( ) {
3          new ArrayList<Object>( ).get(1);
4  }
```

The above code fails if it does not throw IndexOutOfBoundsException or if it throws a different exception at runtime. User defined annotations can be processed at runtime using the new AnnotatedElement and "Annotation" element interfaces added to the Java reflection API.

Exceptions: You can have either runtime or compile-time exceptions.

RuntimeException is also known as the unchecked exception indicating not required to be checked by the compiler. RuntimeException is the superclass of those exceptions that can be thrown during the execution of a program within the JVM. A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of a method but not caught.

Example: *NullPointerException*, *ArrayIndexOutOfBoundsException*, etc

Checked exceptions are verified by the compiler at compile-time that a program contains handlers like throws clause or try{} catch{} blocks for handling the checked exceptions, by analyzing which checked exceptions can result from execution of a method or constructor.

Aspect Oriented Programming (AOP): Aspects can be weaved at compile-time, post-compile time, load-time or runtime.

- **Compile-time:** weaving is the simplest approach. When you have the source code for an application, the AOP compiler (e.g. ajc – AspectJ Compiler) will compile from source and produce woven class files as output. The invocation of the weaver is integral to the AOP compilation process. The aspects themselves may be in source or binary form. If the aspects are required for the affected classes to compile, then you must weave at compile-time. ?
- **Post-compile:** weaving is also sometimes called binary weaving, and is used to weave existing class files and JAR files. As with compile-time weaving, the aspects used for weaving may be in source or binary form, and may themselves be woven by aspects.?
- **Load-time:** weaving is simply binary weaving deferred until the point that a class loader loads a class file and defines the class to the JVM. To support this, one or more "weaving class loaders", either provided explicitly by the run-time environment or enabled through a "weaving agent" are required.
- **Runtime:** weaving of classes that have already been loaded to the JVM.

Inheritance – happens at compile-time, hence is static.
Delegation or composition – happens at run-time, hence is dynamic and more flexible.

Q4. Have you heard the term "**composition should be favored over inheritance**"? If yes, what do you understand by this phrase?
A4. Inheritance is a polymorphic tool and is not a code reuse tool. Some developers tend to use inheritance for code reuse when there is no polymorphic relationship. The guide is that inheritance should be only used when a subclass 'is a' super class.

- 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 super class is modified. This is due to **tight coupling** occurring between the parent and the child classes happening at **compile time**.
- 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 with composition, which gives you code reuse and **runtime** flexibility.

This is the reason why the GoF (Gang of Four) design patterns favor composition over inheritance. The interviewer will be looking for the key terms — "**coupling**", "**static versus dynamic**" and "**happens at compile-time vs runtime**" in your answers. The runtime flexibility is achieved in composition as the classes can be composed **dynamically** at runtime either conditionally based on an outcome or unconditionally.

Whereas an inheritance is **static**, as Java does not allow this natively. There are a number of projects and technologies available that will enable you to modify the byte code of a class after compilation, but they really aren't intended to use for runtime inheritance.

Q5. Can you differentiate compile-time inheritance and runtime delegation/composition with examples and specify which Java supports?
A5.

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( ) +  ", Child is called";
    }
}
```

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/composition is typically achieved as shown below:

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

```
public class Child  {
     public String saySomething( ) {
          return new Parent( ).saySomething( ) +  ", Child is called";
    }
}
```

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

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

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

     public String saySomething( ) {
          return this.parent.saySomething( ) +  ", Child is called";
    }
}
```

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group

2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ QoS interview Q&As on Availability, Serviceability & Disaster Recoverability

04: 7 Java Collection Quizzes ›

Tags: Free FAQs

4-17 beginner Java interview questions and answers

SourceURL: https://www.java-success.com/why-use-java-and-java-overview-interview-questions-and-answers/

01: ♦♥ 17 beginner Java interview questions and answers

Posted on August 17, 2014 by  Arulkumaran Kumaraswamipillai Posted in Java Overview

These 17 beginner Java interview questions & answers are not only for the entry level job interviews, but also get a good grasp of the beginner level Java concepts.

Q1. Why use Java?
A1. One needs to use the best tool for the job, whether that tool is Java or not. When choosing a technology to solve your business problems, you need to consider many factors like development cost, infrastructure cost, ongoing support cost, robustness, flexibility, security, performance, etc.

— Java provides **client technologies**, **server technologies**, and **integration technologies** to solve small scale to very large scale business problems.

— Firstly, Java is a **proven** and **matured** technology used in many mission critical projects, and there are millions of developers world wide, thousands of frameworks, tools, and libraries for it, and millions of sites, blogs, and books to find relevant information.

— The emergence of **open-source technologies** has truly made Java a powerful competitor in the server and integration technology space. You can always find a proven framework that best solves your business problems.

— The platform also comes with a rich set of APIs. This means developers spend less time writing support libraries, and more time developing content for their applications.

— Built-in support for multi-threading, socket communication, and automatic memory management (i.e. automatic garbage collection).

Q2. Is Java a 100% Object Oriented (OO) language? if yes why? and if no, why not?
A2. I would say Java is not 100% object oriented, but it embodies practical OO concepts. There are 6 qualities to make a programming language to be pure object oriented. They are:

1. Encapsulation – data hiding and modularity.
2. Inheritance – you define new classes and behavior based on existing classes to obtain code reuse.
3. Polymorphism – the same message sent to different objects results in behavior that's dependent on the nature of the object receiving the message.
4. All predefined **types are objects**.
5. All operations are performed by **sending messages to objects**.
6. All user defined **types are objects**.

points 1- 3, stands for PIE (**P**olymorphism, **I**nheritance, and **E**ncapsulation).

Reason #1: The main reason why Java cannot be considered 100% OO is due to its existence of 8 primitive variables (i.e. violates point number 4) like int, long, char, float, etc. These data types have been excused from being objects for simplicity and to improve performance. Since primitive data types are not objects, they don't have any qualities such as inheritance or polymorphism. Even though Java provides immutable wrapper classes like Integer, Long, Character, etc representing corresponding primitive data as objects, the fact that it allowed non object oriented primitive variables to exist, makes it not fully OO.

Reason #2: Another reason why Java is considered not full OO is due to its existence of static methods and variables (i.e. violates point number 5). Since

Reason #2: Another reason why Java is considered not full OO is due to its existence of static methods and variables (i.e. violates point number 5). Since static methods can be invoked without instantiating an object, we could say that it breaks the rules of encapsulation.

Reason #3: Java does not support multiple class inheritance to solve the diamond problem because different classes may have different variables with same name that may be contradicted and can cause confusions and result in errors. In Java, any class can extend only one other class, but can implement multiple interfaces.

We could also argue that Java is not 100% OO according to this point of view. But Java realizes some of the key benefits of multiple inheritance through its support for multiple interface inheritance and in **Java 8, you can have multiple behavior (not state) inheritance** as you can have default methods in interfaces.

Reason #4: Operator overloading is not possible in Java except for string concatenation and addition operations. String concatenation and addition example,

```
System.out.println(1 + 2 + "3");//outputs 33
System.out.println("1" + 2 + 3); //outputs 123

```

Since this is a kind of polymorphism for other operators like * (multiplication), / (division), or – (subtraction), and Java does not support this, hence one could debate that Java is not 100% OO. Working with a primitive in Java is more elegant than working with an object like BigDecimal. For example,

```
int a,b, c;
//without operator overloading
a = b - c * d
```

What happens in Java when you have to deal with large decimal numbers that must be accurate and of unlimited size and precision? You must use a BigDecimal. BigDecimal looks verbose for larger calculations without the operator overloading as shown below:

```
BigDecimal b = new BigDecimal("25.24");
BigDecimal c = new BigDecimal("3.99");
BigDecimal d = new BigDecimal("2.78");
BigDecimal a = b.subtract(c).multiply(d); //verbose and wrong

```

Also, the last line above is wrong. The rules of precedence have changed. With chained method calls like this, evaluation is strictly left-to-right. Instead of subtracting the product of c and d from b, we are multiplying the difference between b and c by d. We would have to rewrite the last line as shown below:

```
BigDecimal a = b.subtract(c.multiply(d)); //correct

```

So, it is error prone as well. Another point is that the BigDecimal class is immutable and, as such, each of the "operator" methods returns a new instance. In future Java versions, you may have operator overloading for BigDecimal, and it would make your code more readable as shown below.

```
BigDecimal a = b - (c * d); //much better

```

Q3. What is the difference between C++ and Java?
A3. Both C++ and Java use similar syntax and are Object Oriented, but:

1) Java does not support pointers. Pointers are inherently tricky to use and troublesome.

2) Java does not support multiple inheritances because it causes more problems than it solves. Instead Java supports multiple interface behavior inheritance (In Java 8, you can have interfaces with default & static methods, but can't have state), which allows an object to inherit many method signatures from different interfaces with the condition that the inheriting object must implement those inherited methods.

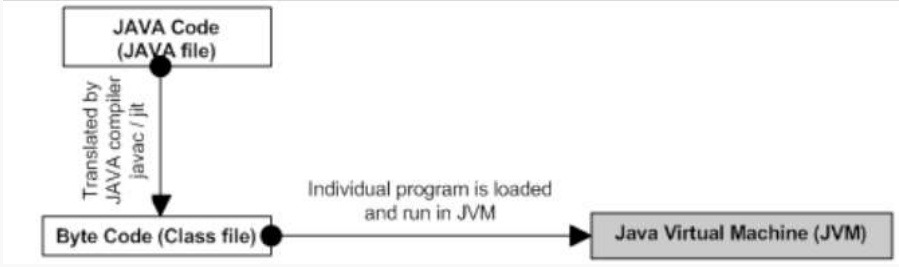
3) Java does not support destructors but rather adds a finalize() method. Finalize methods are invoked by the garbage collector prior to reclaiming the memory occupied by the object, which has the finalize() method. This means you do not know when the objects are going to be finalized. Avoid using finalize() method to release non-memory resources like file handles, sockets, database connections etc because Java has only a finite number of these resources and you do not know when the garbage collection is going to kick in to release these resources through the finalize() method.

4) Java does not include structures or unions. Java make use of the Java Collection framework.

5) All the code in Java program is encapsulated within classes therefore Java does not have global variables or functions.

6) C++ requires explicit memory management, while Java includes automatic garbage collection.

Q4. What is the main difference between the Java platform and the other software platforms?
A4. Java platform is a software-only platform, which runs on top of other hardware-based platforms like Unix, Windows, Mac OS, etc.



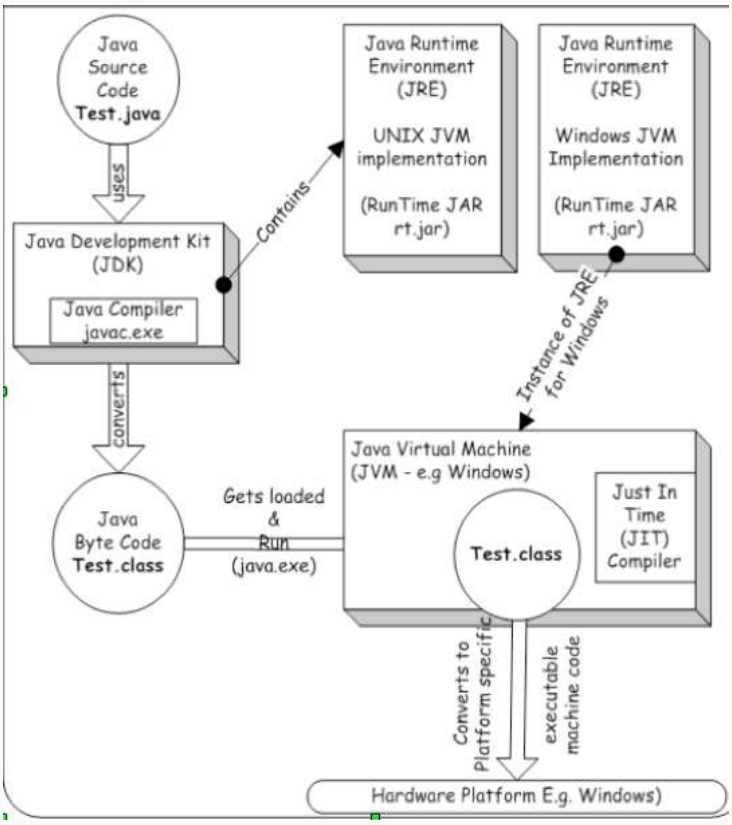
Java platform

The Java platform has 2 components:

1) Java Virtual Machine (JVM) – 'JVM' is a software that can be ported onto various hardware platforms. Byte codes are the machine language of the JVM.

2) Java Application Programming Interface (Java API) – is nothing but a set of classes and interfaces that come with the JDK. All these classes are written using the Java language and contains a library of methods for common programming tasks like manipulating strings and data structures, networking, file transfer, etc. The source *.java files are in the src.zip archive and the executable *.class files are in the rt.jar archive.

Q5. How would you differentiate JDK, JRE, JVM, and JIT?
A5. There is no better way to get the big picture than a diagram.



JDK, JRE, JVM, and JIT

1) **JDK:** You can download a copy of the Java Development Kit (JDK) for your operating system like Unix, Windows, etc.

2) **JRE:** Java Runtime Environment is an implementation of the JVM. The JDK typically includes the Java Runtime Environment (JRE) which contains the virtual machine and other dependencies to run Java applications.

3) **JIT:** A JIT is a code generator that converts Java byte code into native machine code. Java programs invoked with a JIT generally run much faster than when the byte code is executed by the interpreter. The JIT compiler is a standard tool that is part of the JVM and invoked whenever you use the Java interpreter command. You can disable the JIT compiler using the -Djava.compiler=NONE option to the Java VM. You might want to disable the JIT compiler if you are running the Java VM in remote debug mode, or if you want to see source line numbers instead of the label (Compiled Code) in your Java stack traces.

Q6. Is it possible to convert byte code into source code?
A6. Yes. A Java decompiler is a computer program capable of reversing the work done by a compiler. In essence, it can convert back the byte code (i.e. the .class file) into the source code (i.e the .java file). There are many decompilers that exist today, but the most widely used JD – Java Decompiler is available both as stand-alone GUI program and as an eclipse plug-in.

Q7. When would you use a decompiler?
A7.

1) When you have *.class files and you do not have access to the source code (*.java files). For example, some vendors do not ship the source code for java class files or you accidentally lost (e.g deleted) your source code, in which case you can use the Java decompiler to reconstruct the source file.

2) Another scenario is that if you generated your .class files from another language like a groovy script, using the groovyc command, you may want to use a Java decompiler to inspect the Java source code for the groovy generated class files to debug or get a better understanding of groovy integration with Java.

3) To ensure that your code is adequately obfuscated before releasing it into the public domain.

4) Fixing and debugging .class files when developers are slow to respond to questions that need immediate answers. To learn both Java and how the Java VM works.

5) Learn and debug how code with generics has been converted after compilation.

Q8. Is it possible to prevent the conversion from byte code into source code?
A8. If you want to protect your Java class files from being decompiled, you can take a look at a Java obfuscator tool like yGuard or ProGuard, otherwise you will have to kiss your intellectual property good bye.

Q9. What are the two flavors of JVM?
A9. Client mode and server mode.

Client mode is suited for short lived programs like stand-alone GUI applications and applets. Specially tuned to reduce application start-up time and memory footprint, making it well suited for client applications. For example: c:\> java -client MyProgram

Server mode is suited for long running server applications, which can be active for weeks or months at a time. Specially tuned to maximize peak operating speed. The fastest possible operating speed is more important than fast start-up time or smaller runtime memory footprint. c:\> java -server MyProgram

Q10. How do you know in which mode your JVM is running?
A10. c:\> java -version

Q11. What are the two different bits of JVM? What is the major limitation of 32 bit JVM?
A11. JVMs are available in 32 bits (-d32 JVM argument) and 64 bits (-d64 JVM argument). 64-bit JVMs are typically only available from JDK 5.0 onwards. It is recommended that the 32-bit be used as the default JVM and 64-bit used if more memory is needed than what can be allocated to a 32-bit JVM. The Oracle Java VM cannot grow beyond ~2GB on a 32bit server machine even if you install more than 2GB of RAM into your server. It is recommended that a 64bit OS with larger memory hardware is used when larger heap sizes are required. For example, >4GB is assigned to the JVM and used for deployments of >250 concurrent or >2500 casual users.

Q12. What are some of the JVM arguments you have used in your projects?
A12. To set a system property that can be retrieved using System.getPropety("name");

```
1 $java -Dname=value MyApp
```

To set the classpath: -cp or -classpath

```
$java -cp library.jar MyApp
```

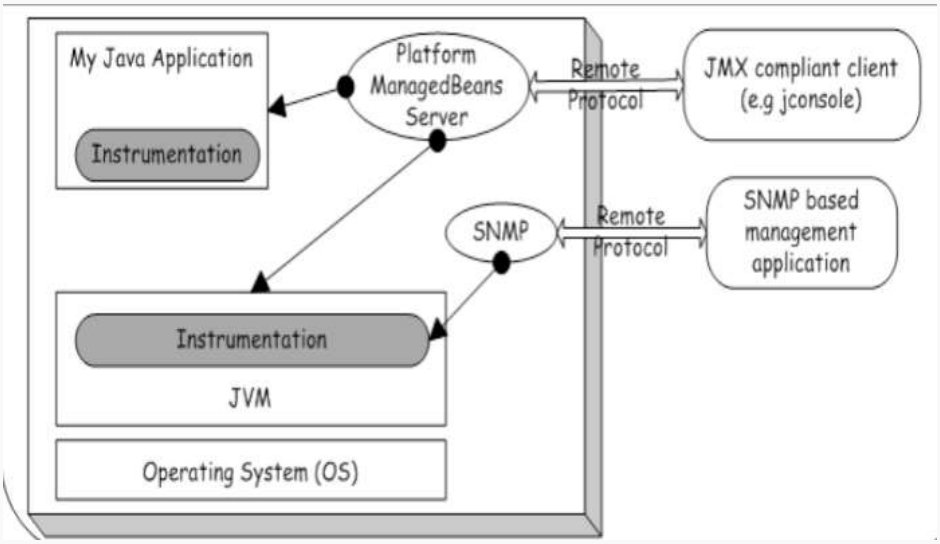
To set minimum and maximum heap sizes:

```
$java -Xms1024 -Xmx1024 MyApp
```

To set garbage collection options:

```
$ java -Xincgc MyApp
```

Q13. How do you monitor the JVMs?
A13. Since Java SE 5.0, the JRE provides a mean to manage and monitor the Java Virtual Machine. It comes in two flavors:



JVM Monitoring

1) The JVM has built-in instrumentation that enables you to monitor and manage it using **Java Management eXtension (JMX)**. You can also monitor instrumented applications with JMX. To start a Java application with the management agent for local monitoring, set the following JVM argument when you run your application.

```
$JAVA_HOME/bin/java -Dcom.sun.management.jmxremote  MyApp

```

To start the JConsole:

```
$JAVA_HOME/bin/jconsole
```

2) The other is a **Simple Network Management Protocol (SNMP)** agent that builds upon the management and monitoring API of the Java SE platform, and exposes the same information through SNMP. SNMP events can be sent **Splunk**. Nagios, Zenoss, OpenNMS, and netsnmp are some of the popular SNMP tools.

Q14. What is a jar file? How does it differ from a zip file?
A14. The jar stands for Java ARchive. A jar file usually has a file name extension .jar. It mainly contains Java class files but any types of files can be included. For example, XML files, properties files, HTML files, image files, binary files, etc. You can use the "jar" application utility bundled inside /JDK1.6.0/jre/bin to create, extract, and view its contents. You can also use any zip file utility program to view its contents. **A jar file cannot contain other jar files.**, whereas a war or ear file used in Java EE.

Basically, a jar file is same as a zip file, except that it contains a META-INF directory to store meta data or attributes. The most known file is META-

Basically, a jar file is same as a zip file, except that it contains a META-INF directory to store meta data or attributes. The most known file is META-INF/MANIFEST.MF. When you create a JAR file, it automatically receives a default manifest file. There can be only one manifest file in an archive. Most uses of JAR files beyond simple archiving and compression require special information to be in the manifest file. For example,

— If you have an application bundled in a JAR file, you need some way to indicate which class within the JAR file is your application's entry point. The entry point is the class having a method with signature public static void main(String[ ] args). For example, Main-Class: Test.class

— A package within a JAR file can be optionally sealed, which means that all classes defined in that package must be archived in the same JAR file. You might want to seal a package, for example, to ensure version consistency among the classes in your software or as a security measure.

Name: myCompany/myPackage/
Sealed: true

Q15. What do you need to develop and run Java programs? How would you go about getting started?
A15. **Step 1:** Download and install the Java Development Kit (JDK) SE (Standard Edition) for your operating system (e.g. Windows, Linux, etc) and processor (32 bit, 64 bit, etc) .

Step 2: Configure your environment. The first environment variable you need to set is the **JAVA_HOME**.

```
JAVA_HOME=C:\DEV\java\jdk-1.6.0_11 #on windows
export JAVA_HOME=/usr/java/jdk-1.6.0_11  #on Unix
```

The second environment variable is **PATH**.

```
PATH=%JAVA_HOME%\bin; #on windows
export PATH=$PATH:$JAVA_HOME/bin #on Unix
```

Step 3: Verify your configurations with the following commands:

```
echo %JAVA_HOME% #windows
echo %PATH%  #windows
$ echo $JAVA_HOME #Unix
$ echo $PATH   #Unix
```

Step 5: Verify your installation with

```
$ javac -version
$ java -version
```

Q16. How do you create a class under a package in Java? What is the first statement in Java?
A16. ou can create a class under a package as follows with the package keyword, which is the first keyword in any Java program followed by the import statements. The java.lang package is imported implicitly by default and all the other packages must be explicitly imported. The core Java packages like java.lang.*, java.net.*, java.io*, etc and its class files are distributed in the archive file named rt.jar.

```
package com.xyz.client ;
import  java.io.File;
import  java.net.URL;
```

Q17. What do you need to do to run a class with a main( ) method in a package? For example: Say, you have a class named Pet in a project folder C:\projects\Test\src and package named com.xyz.client, will you be able to compile and run it as it is?

A17. **Step 1:** Write source code "Pet.java" as shown below

```
package com.xyz.client;

public class Pet {
    public static void main(String[ ] args) {
         System.out.println("I am found in the classpath");
    }
```

```
}
```

Step 2: The source code can be compiled into byte code i.e. Pet.class file as shown below:

```
C:\projects\Test\src>javac -d ../bin com/xyz/client/Pet.java
```

Note: The compiled byte code file Pet.class will be saved in the folder C:\projects\Test\bin\com\xyz\client.

Step 3: If you run it inside where the Pet.class is stored, the answer is yes.

```
C:\projects\Test\bin>java com/xyz/client/Pet
```

The Pet.class file will be found since com.xyz.client.Pet class file is in the projects\Test\bin folder. If you run it in any other folder say c:\

```
C:\> java com.xyz.client.Pet
```

The answer is no, and you will get the following exception: Exception in thread "main" **java.lang.NoClassDefFoundError:** com/xyz/client/Pet. To fix this, you need to tell how to find the Pet.class by setting or providing the **classpath**. How can you do that? One of the following ways:

1) Set the operating system CLASSPATH environment variable to have the project folder c:\projects\Test\bin.

```
CLASSPATH=C:/projects/Test2/bin
```

2) Set the operating system CLASSPATH environment variable to have a jar file c:/projects/Test/pet.jar, which has the Pet.class file in it

```
CLASSPATH=c:/projects/Test/pet.jar
```

3) Run it with -cp or -classpath option as shown below.

```
C:\>java -cp projects\Test\bin com/xyz/client/Pet
C:\>java -classpath c:/projects/Test/pet.jar    com.xyz.client.Pet
C:\projects\Test\src>java -cp ../bin com/xyz/client/Pet

```

You may also like

12 Java String class interview Q&A | Web Services interview Q&A | Java EE Overview interview Q&A | Multithreading scenarios in Java applications interview Q&A

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ ♥♦ Object equals Vs == and pass by reference Vs value

♦ 15 Database design interview Questions & Answers ›

Tags: Free Content, Free FAQs, Java/JEE FAQs, Novice FAQs

3-Java OOPs interview questions & answers with lots of diagrams

01: ♦Q1 – Q10 Java OOPs interview questions & answers

Posted on August 11, 2014 by Arulkumaran Kumaraswamipillai Posted in 01 - FAQs Core Java, 09 - FAQs OOP & FP, OOP

If you don't get Java OOPs interview questions & answers right in the job interviews you can say **OOPS !!!!** to your Java interview success.

Q1. Is Java a 100% Object Oriented (OO) language? if yes why? and if no, why not?
A1. I would say Java is not 100% object oriented, but it embodies practical OO concepts. Strictly speaking not 100% object orieneted due to:

1) its existence of 8 primitive variables like int, long, char, float, etc. These data types have been excused from being objects for simplicity and to improve performance.

2) its existence of static methods and variables. Static methods are be invoked without instantiating an object.

3) not supporting multiple class inheritance to solve the diamond problem because different classes may have different variables with same name that may be contradicted and can cause confusions and result in errors.

4) not supporting operator overloading except for string concatenation and addition operations.

[**More detailed answer: Q2** at 17 Java overview interview questions and answers.]

Q2. How to create a well designed Java application?
A2. A software application is built by **coupling** various **classes & interfaces**, **modules** (i.e. made of classes & interfaces), and **components** (i.e. made up of modules). Without coupling, you can't build a software system. But, the software applications are always subject to changes and enhancements. So, you need to build your applications such a way that they can not only **adapt** to growing requirements, but also are easy to **maintain** and **understand**. This is what OOP achieves.

[**Detailed example**: How to create a well designed Java application?]

Q3. What do you achieve through good class and interface design with OOP?
A3. OOP is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields aka attributes, and functionality, in the form of methods. Objects couple with each other by invoking methods on each other. Classes & interfaces are the building blocks to create objects. If the classes & interfaces are not structured properly, the resulting code becomes harder to maintain or extend due to increased complexity & coupling.

Each object has its own responsibility and couple or collaborate with the other objects to get the task done.

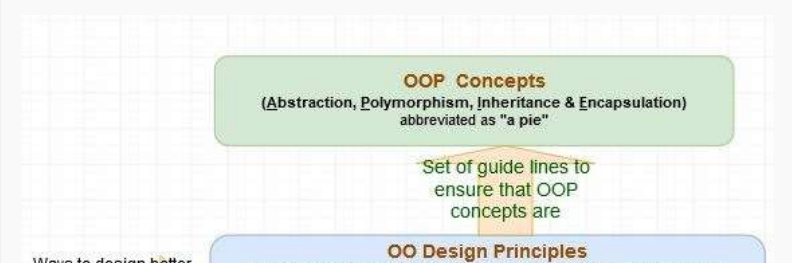
CircuService –> handleLion() –> LionHandler –> help() –> LionHelper

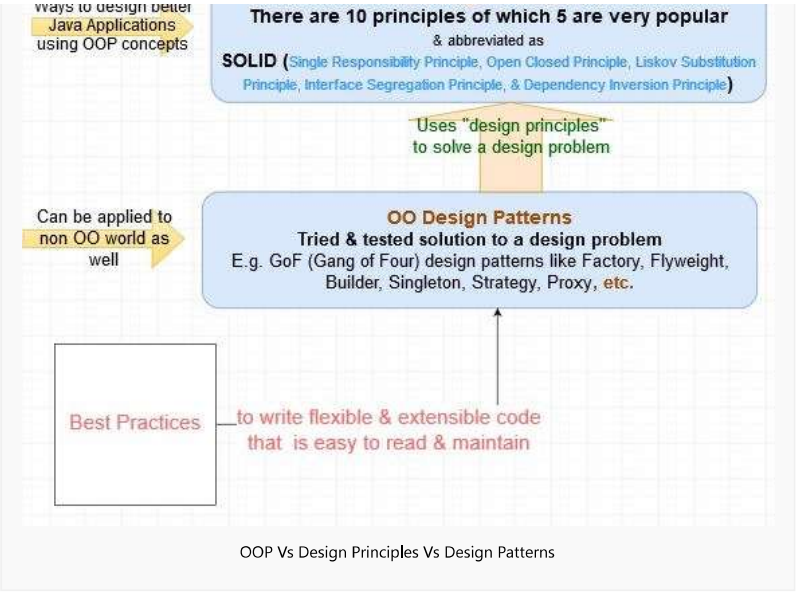
OrderApp –> placeOrder(...) –> OrderService –> saveOrder(...) –> OrderDao

OOP concepts, OO design principles, and OO design patterns help you create

1. Loosely coupled classes, objects, and components enable your application to easily grow and adapt to changes without being rigid or fragile.

2. Less complex and **reusable** code that increases maintainability, extendability and testability.

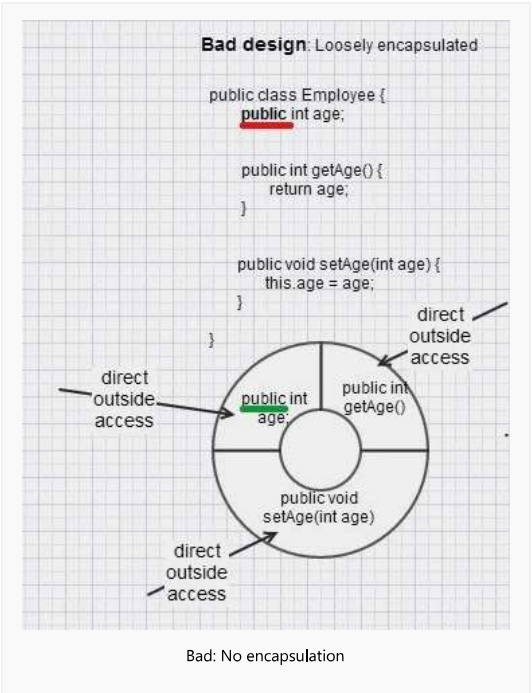




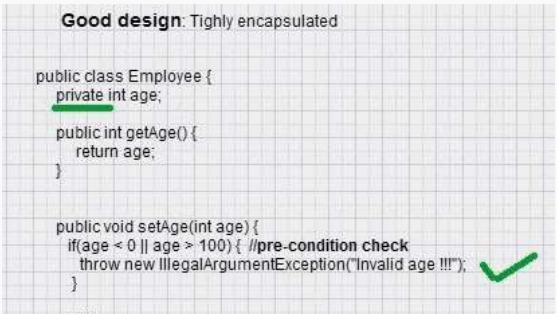
OOP Vs Design Principles Vs Design Patterns

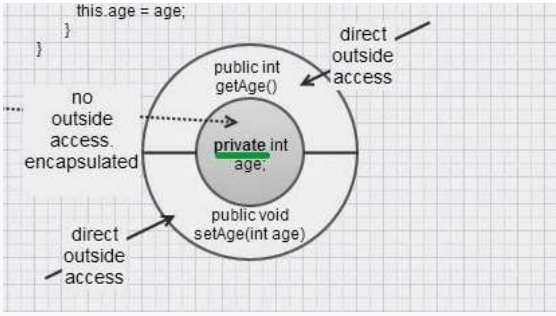
Q4. What are the 4 main concepts of OOP?
A4. Encapsulation, polymorphism, and inheritance are the 3 main concepts or pillars of an object oriented programming. Abstraction is another important concept that can be applied to both object oriented and non object oriented programming. [Remember: "**a pie**" abstraction, polymorphism, inheritance, and encapsulation.]

Q5. What problem(s) does abstraction and encapsulation solve?
A5. Both abstraction and encapsulation solve same problem of **complexity** in different dimensions. Encapsulation **exposes only the required details** of an object to the caller by forbidding access to certain members,



Bad: No encapsulation



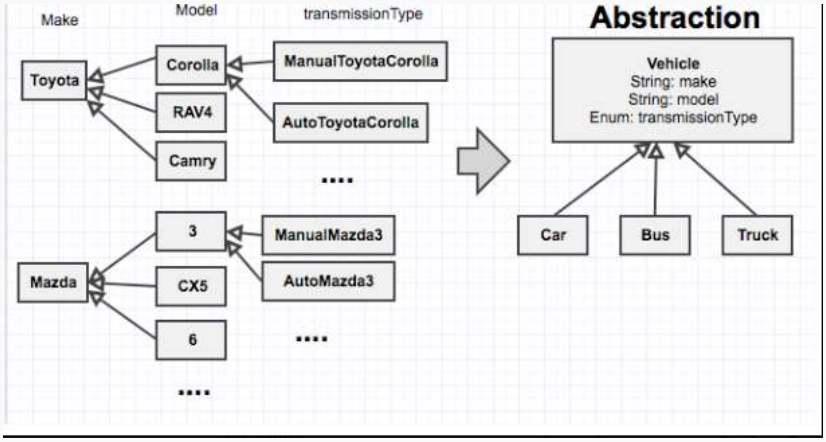


Good: Encapsulated

Abstraction allows us to represent complex real world in simplest manner by

1) Hiding non-essential implementation details by letting you focus on **what** the object does instead of **how** it does it. For example, expose only a handful of methods to be accessed via an "Interface" or "Abstract" class. When you drive a car, you know **what** a steering wheel does but you may not know the underlying implementation details as to **how** it does.

2) providing a basis for your application to grow and change over a period of time with the help of **generalization**. For example, if you generalize out the "make" and "model" of a vehicle as **class attributes** as opposed to as individual classes like "Toyota.java", "ToyotaCamry.java", "ToyotaCorolla.java", etc, you can easily incorporate new types of cars at runtime by creating a new car object with the relevant "make" and "model" as arguments as opposed to having to declare a new set of classes.



Java Abstraction

A good OO design should **hide** non-essential implementation details through abstraction and information details through encapsulation.

Q. What is the difference between Abstraction & Generalization?
A. Abstraction and generalization are often used together. Abstracts are generalized through variables, parameterization, generics and polymorphism. Generalization places the emphasis on the similarities among objects. It helps to manage complexity by collecting individuals into groups.

For example, say you want to capture employment type like part-time, full-time, casual, semi-casual, and so on, it is a bad practice to define them as classes as shown below.

Bad non abstracted example:

```
1 package com.oo;
2
3 public class PartTimeEmployee extends Employee {
4
5 }
```

```
1 package com.oo;
2
3 public class FullTimeEmployee extends Employee {
4
```

```
}
```

Why is it bad? You will end up creating a new class for each employee type. This will make your code more rigid and **tightly coupled**. The better approach is to abstract out the *employmentType* as a class attribute. This way, instead of creating new classes for every employment type, you will be just creating new objects at **runtime** with different *employmentTypes*.

```
public class Employee {

 enum EmploymentType {PART_TIME, FULL_TIME, CASUAL, SEMI_CASUAL}

 private String name;
 private int age;
 private BigDecimal salary;
 private EmploymentType employmentType;

    // ... getters and setters for external access

}
```

The above class is both properly abstracted and encapsulated.

Q6. What problem(s) does inheritance & composition solve?
A6. **Reusability**. How can logic be easily used in two places? In object-oriented language, there are four primary ways to accomplish this:

- **Copy and Paste –** Bad as it is hard to maintain. Any changes original logic need to be applied across all the pasted locations.
- **Inheritance –** Ok. Takes place at compile-time. So, can be fragile.
- **Composition –** Good, and favored over inheritance. Happens at runtime.
- **Mixins –** Good, but not supported in Java and can be abused and consequently increase complexity. The mixins are kind of composable abstract classes. They are used in a multi-inheritance context to add services to a class. Java 8 has a naive emulation of mixins with virtual extension or default methods.

There is a post dedicated to why favor composition over inheritance?: Why favor composition over inheritance? Java OOP Interview Q&As

Q7. Can you explain the Java concepts like overloading, overriding, hiding & obscuring? Which one of these concepts lead to polymorphism?
A7. **Overriding** is the means by which you achieve polymorphism. Overloading, overriding, hiding & obscuring are related concepts explained in detail at Java Polymorphism vs Overriding vs Overloading.

Q8. Given the following code snippets, can you explain the OOP concepts – abstraction, encapsulation, Inheritance, and polymorphism ?

```
List<String> list = new ArrayList<>();
list.add("Java");
list.add("JEE");
```

You can elaborate with additional code examples using java.util.List methods.

A8. This is explained in detail at Explain abstraction, encapsulation, Inheritance, and polymorphism with the given Java code?

Q9. What is the relationship among "**OOP**", "**Design Principles**", and "**Design Patterns**" ?
A9. All these 3 lead to "best practices" for building a quality application with Java classes & interfaces, which are the building blocks. You can't use these building blocks the any way you like. There are overarching **principles** & tried and tested **patterns** as depicted above.

7 Design principles interview questions & answers for Java developers | 8+ Java design patterns interview questions & answers

Q10. What is the difference between OOP & Functional Programming (i.e. **FP**)?
A10. FP is another programming **paradigm** like OOP. In FP "Functions" are the first class citizens as "Objects" are in OOP. Both OOP & FP compliment each other and you can use both in your next Java application if using Java 8. You like it or not, you will be using functional programming in Java, and interviewers are going to quiz you on functional programming.

Top 6 tips to transforming your thinking from OOP to FP with examples

Q. If you are mentoring a junior developer, what tips would you give him/her on OOP?
A. An open-ended question to judge your experience & know how.Top 5 OOPs tips for Java developers

Q. How would you go about writing loosely coupled Java application?

A.Top 6 tips to go about writing loosely coupled Java applications

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/EE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ ♥♦ 6 Java Modifiers every interviewer likes to quiz

02: ♦ Q11 – Q15 Why favor composition over inheritance? ›

Tags: Core Java FAQs

2-15+ Java multithreading interview questions & answers

SourceURL: https://www.java-success.com/java-beginner-multithreading-interview-questions-and-answers/

01: ♦15+ Beginner Java multithreading interview questions & answers

Posted on August 22, 2014 by Arulkumaran Kumaraswamipillai Posted in 01 - FAQs Core Java, member-paid, Multithreading

Beginner Java multithreading interview questions answered with lots of diagrams & code for Java developers to standout in beginner to intermediate Java job interviews. Even the experienced Java developers dread Java multithreading interview questions & coding exercises.

Q1. What is a thread?
A1. It is a thread of execution in a program. The JVM allows an application to have multiple threads of execution running concurrently. In the Hotspot JVM there is a direct mapping between a Java Thread and a native operating system (i.e. OS) Thread. The native OS thread is created after preparing the state for a Java thread involving thread-local storage, allocation of buffers, creating the synchronization objects, stacks and the program counter.

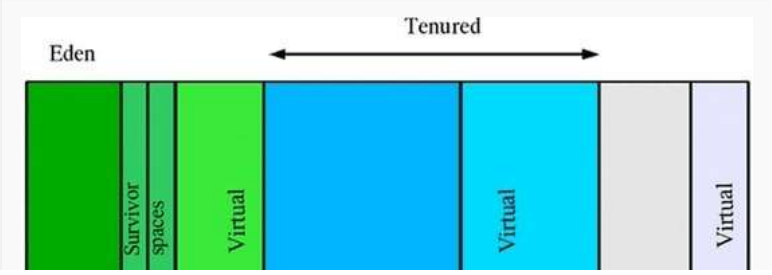
The **OS** is responsible for scheduling all threads and dispatching them to any available CPU. In a multi-core CPU, you will get real **parallelism**. When you have more threads than the number of CPU cores, the CPU switches from executing one thread to executing another, and it needs to save the local data, program pointer, etc. of the current thread, and load the local data, program pointer, etc. of the next thread to execute. This switch is called **context switching**, which is not cheap, and you should NOT context switch more than necessary.

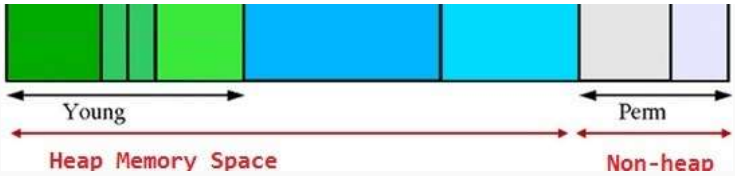
When the thread terminates, all resources for both the native and Java thread are released.

Q2. What are the JVM or system created threads?
A2. The **main thread** and a number of **background threads**.

1) main thread, which is created as part of invoking "public static void main(String[ ] args) { .... }".

2) VM background thread to perform major GC, thread dumps (e.g. kill -QUIT in Unix), thread suspension, etc. Major GC evicts the objects that are promoted to the "**Tenured**" part of the heap.





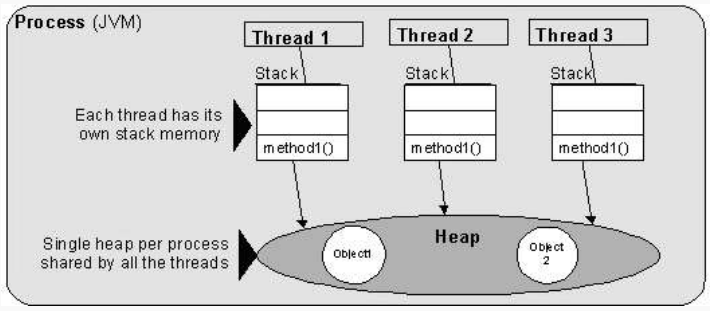
Java heap sections

3) Garbage Collection low priority background thread for minor GC activities. Minor GC evicts the objects in the "**Eden**" and "**Survivor**" part of the heap.

4) Compiler background thread to compile byte code to native code at run-time.

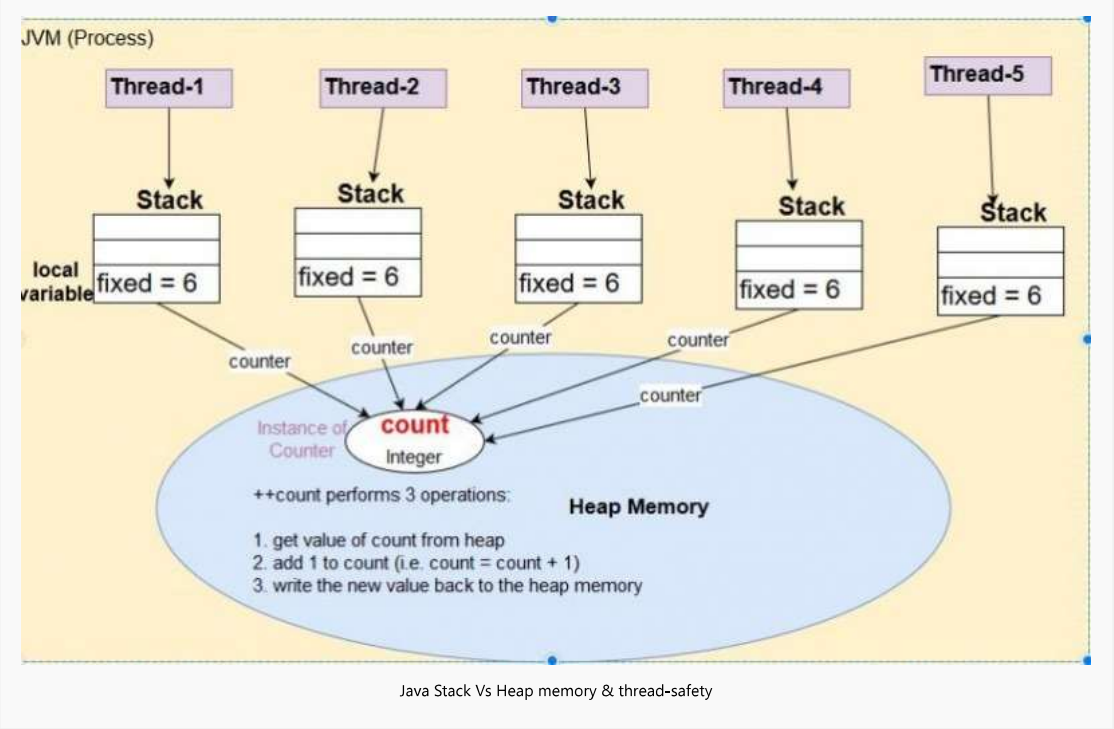
5) Other background threads such as signal dispatcher thread and periodic task thread.

Q3. What is the difference between processes and threads?
A3. A process is an execution of a program but a thread is a single execution sequence within the process. A process can contain multiple threads. A thread is sometimes called a lightweight process.



Process Vs Threads

A JVM runs in a single process and threads in a JVM share the heap belonging to that process. That is why several threads may access the same object. Threads share the **heap** and have their own **stack** space. This is how one thread's invocation of a method and its local variables are kept thread safe from other threads. But the heap is not thread-safe and must be synchronized for thread safety, which is depicted below:



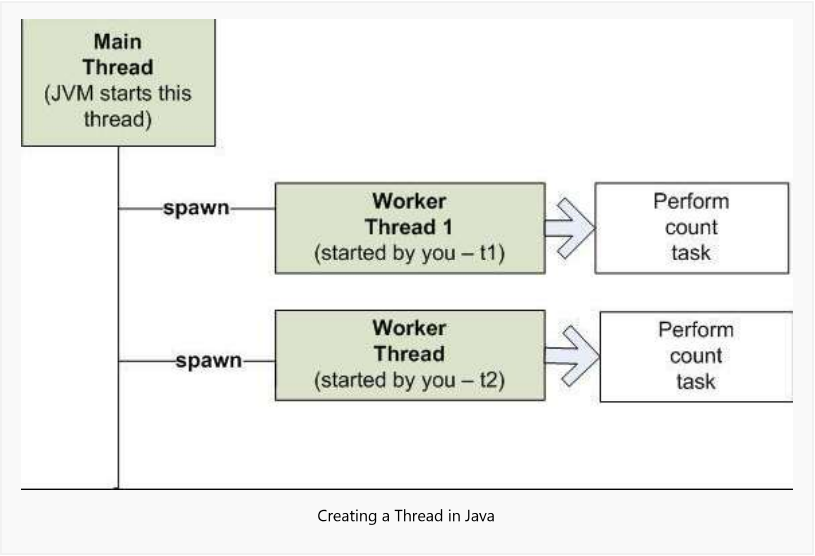
Java Stack Vs Heap memory & thread-safety

and learn more about 3 Java Multithreading basics – Heap Vs Stack, Thread-safety & Synchronization with Java code.

If you are an experienced Java developer?

More advanced & practical Java multi-threading interview Q&As

Q4. Explain different ways of creating a thread?
A4. In addition to the JVM created threads, application developers can create new threads in Java. Threads can be created in a number of different ways.



Creating a Thread in Java

1) Extending the java.lang.Thread class.

2) Implementing the java.lang.Runnable interface.

3) Implementing the java.util.concurrent.Callable interface with the java.util.concurrent.Executor framework to pool the threads. The java.util.concurrent package was added in Java 5. [7 basic Java Executor framework Interview Q&As with Future & CompletableFuture]

4) Using the Fork/Join Pool. Java 7 fork and join tutorial with a diagram and an example.

5) The actor model, which is also known as Reactive Programming using frameworks like **Akka**. Simple Akka tutorial in Java step by step

Note: Learn more about ExecutorService Vs Fork/Join & Future Vs CompletableFuture Interview Q&As.

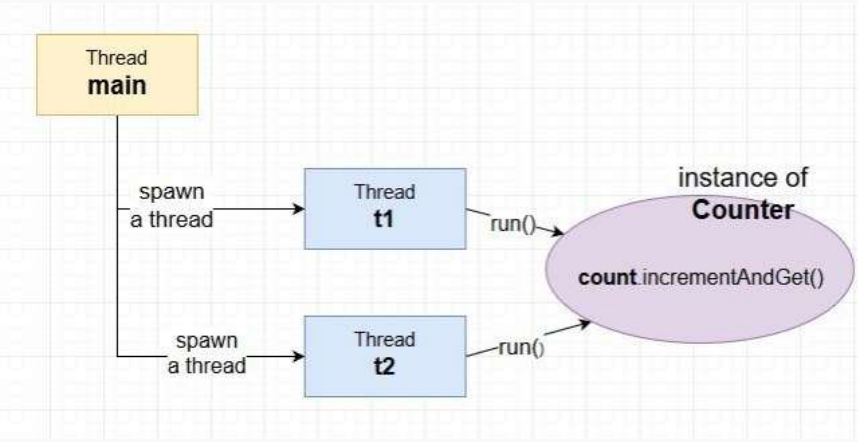
1. Extending the **java.lang.Thread** class

```
import java.util.concurrent.atomic.AtomicInteger;

class Counter extends Thread {

    AtomicInteger count = new AtomicInteger();

    // method where the thread execution will start
    public void run() {
        // logic to execute in a thread
        // e.g. performing a count task

        System.out.println(Thread.currentThread().getName() + " is executing..." + count.incrementAndGet());

    }

    // let's see how to start the threads
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + " is executing..." );
        Counter counter = new Counter();
        Thread t1 = new Thread(counter);
        Thread t2 = new Thread(counter);
        t1.start(); // start the thread. This calls the run() method.
```

```
        t2.start(); // start the thread. This calls the run() method.

    }
}

```

Output:

```

main is executing...
Thread-1 is executing...1
Thread-2 is executing...2

```



Pictorial depiction what the code above and below do.

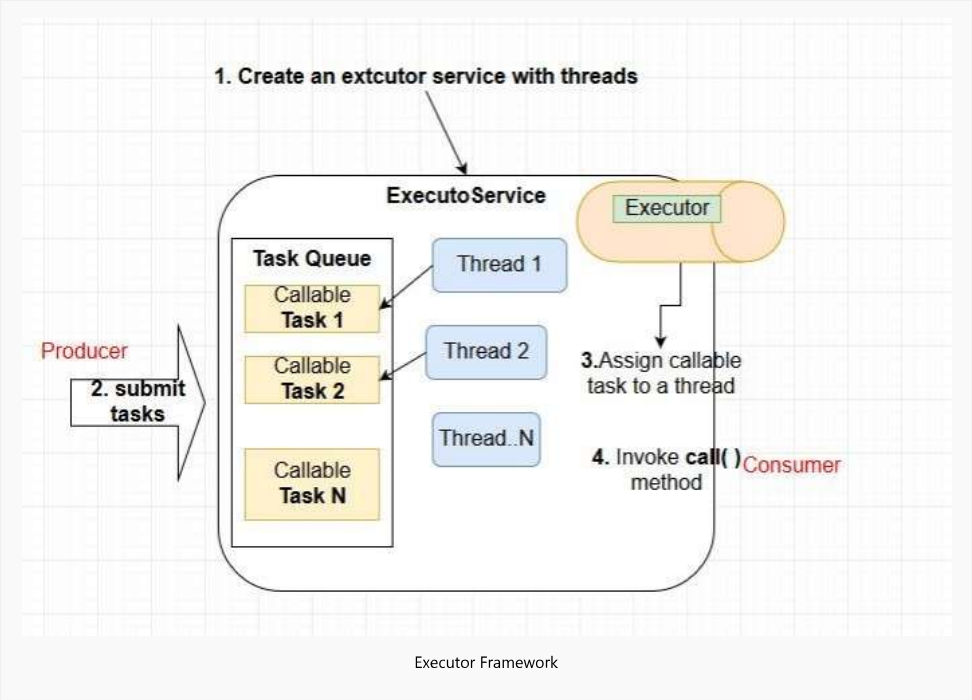
2. Implementing the **java.lang.Runnable** interface. The Thread class takes a runnable object as a constructor argument.

```
import java.util.concurrent.atomic.AtomicInteger;

class Counter implements Runnable {

    AtomicInteger count = new AtomicInteger();

    // method where the thread execution will start
    public void run() {
        // logic to execute in a thread
        // e.g. performing a count task

        System.out.println(Thread.currentThread().getName() + " is executing..." + count.incrementAndGet());

    }

    // let's see how to start the threads
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + " is executing..." );
        Counter counter = new Counter();
        Thread t1 = new Thread(counter);
        Thread t2 = new Thread(counter);
        t1.start(); // start the thread. This calls the run() method.
        t2.start(); // start the thread. This calls the run() method.

    }
}

```

Output:

```
main is executing...
Thread-0 is executing...1
Thread-1 is executing...2
```

3. Implementing the **java.util.concurrent.Callable** interface with the executor service framework.



Executor Framework

```java
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.atomic.AtomicInteger;

class Counter implements Callable<String> {

    private static final int THREAD_POOL_SIZE = 2;
    private AtomicInteger count = new AtomicInteger();

    // method where the thread execution takes place
    public String call() {
        return Thread.currentThread().getName() + " executing ..." + count.incrementAndGet(); //Consumer
    }

    public static void main(String[] args) throws InterruptedException,
            ExecutionException {
        // create a pool of 2 threads
        ExecutorService executor = Executors
                .newFixedThreadPool(THREAD_POOL_SIZE);

        Counter counter = new Counter();

        Future<String> future1 = executor.submit(counter);    //Producer
        Future<String> future2 = executor.submit(counter);    //Producer

        System.out.println(Thread.currentThread().getName() + " executing ...");

        //asynchronously get from the worker threads
        System.out.println(future1.get());
```

```
        System.out.println(future2.get());

    }
}
```

Output:

```
main executing ...
pool-1-thread-1 executing ...1
pool-1-thread-2 executing ...2
```

Q5. Which approach would you favor and why?
A5. Favor Callable interface with the Executor framework for thread pooling.

1) The thread pool is more efficient. Even though the threads are light-weighted than creating a process, creating them utilizes a lot of resources. Also, creating a new thread for each task will consume more stack memory as each thread will have its own stack and also the CPU will spend more time in context switching. Creating a lot many threads with no bounds to the maximum threshold can cause application to run out of heap memory. So, creating a Thread Pool is a better solution as a finite number of threads can be pooled and reused. The runnable or callable tasks will be placed in a queue, and the finite number of threads in the pool will take turns to process the tasks in the queue.

2) The Runnable or Callable interface is preferred over extending the Thread class, as it does not require your object to inherit a thread because when you need multiple inheritance, only interfaces can help you. Java class can extend only one class, but can implement many interfaces.

3. The Runnable interface's void run( ) method has no way of returning any result back to the main thread. The executor framework introduced the Callable interface that returns a value from its call( ) method. This means the asynchronous task will be able to return a value once it is done executing.

Q6. What design pattern does the executor framework use?
A6. The java.util.concurrent.**Executor** is based on the producer-consumer design pattern, where threads that submit tasks are producers and the threads that execute tasks are consumers. In the above examples, the main thread is the producer as it loops through and submits tasks to the worker threads. The "Counter" is the consumer that executes the tasks submitted by the main thread.

Q7. What is the difference between yield and sleep? What is the difference between the methods sleep( ) and wait( )?
A7. When a task invokes yield( ), it changes from running state to runnable state. When a task invokes sleep ( ), it changes from running state to waiting/sleeping state.

The method wait(1000) causes the current thread to wait up to one second for a signal (i.e. notify()/notifyAll()) from other threads. A thread could wait less than 1 second if it receives the notify( ) or notifyAll( ) method call. The call to sleep(1000) causes the current thread to sleep for t least 1 second.

Threads performing tasks by talking to each other

Q8. Why is locking of a method or block of code for thread safety is called "**synchronized**" and not "lock" or "locked"?
A8. When a method or block of code is locked with the reserved "synchronized" key word in Java, the memory (i.e. heap) where the shared data is kept is synchronized. This means,

When a synchronized block or method is entered after the lock has been acquired by a thread, it first reads (i.e. **synchronizes**) any changes to the locked object from the main heap memory to ensure that the thread that has the lock has the current info before start executing.

After the synchronized block has completed and the thread is ready to relinquish the lock, all the changes that were made to the object that was locked is written or flushed back (i.e. **synchronized**) to the main heap memory so that the other threads that acquire the lock next has the current info.

This is why it is called "synchronized" and not "locked". This is also the reason why the immutable objects are inherently thread-safe and does not require any synchronization. Once created, the immutable objects cannot be modified.

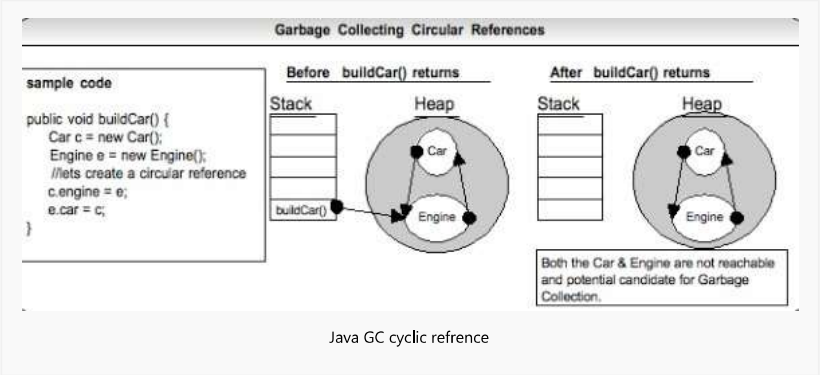
Learn more about the memory model & the synchronization at: 10+ Atomicity, Visibility, and Ordering interview Q&A in Java multi-threading

Q9. Can you explain what an intrinsic lock or monitor is?
A9. 7 Things you must know about Java locks and synchronized key word.

Q10. What does re-entrancy mean regarding intrinsic or explicit locks?
A10. Re-entrancy means that locks are acquired on a **per-thread** rather than **per-invocation** basis. In Java, both intrinsic and explicit locks are **re-entrant**.

```
1  public synchronized void method1(){
2    //intrinsic lock is acquired
3    operation1(); //ok to enter this synchronized method
4                  //as locks are on per thread basis
5
6    operation2(); //ok to enter this synchronized method
7                  //as locks are on per thread basis
8    //intrinsic lock is released
9  }
10
11 public synchronized void operation1(){
12     //process 1
13 }
14
15 public synchronized void operation2(){
16     //process 1
17 }
18
19
```

Q11. If you have a circular reference of objects, but you no longer reference it from an execution thread, will this object be a potential candidate for garbage collection?
A11. Yes. Refer diagram below.



Java GC cyclic refrence

Q12. When you have automatic memory management in Java via GC, why do you still get memory leaks in Java?
A12. In Java, memory leak can occur due to

1) Long living objects having reference to short living objects, causing the memory to slowly grow. For example, singleton classes referring to short lived objects. This prevents short-lived objects being garbage collected.

2) Improper use of thread-local variables. The thread-local variables will not be removed by the garbage collector as long as the thread itself is alive. So, when threads are pooled and kept alive forever, the object might never be removed by the garbage collector.

3) Using mutable static fields to hold data caches, and not explicitly clearing them. The mutable static fields and collections need to be explicitly cleared.

4) Objects with circular references from a thread. GC uses "**reference counting**". Whenever a reference to an object is added its reference count is increased by 1. Whenever a reference to an object is removed, the reference count is decreased by 1. If "A" references object B and B references object A, then both of their reference counts can never be less than 1, which means they will never get collected.

5) JNI (Java Native Interface) memory leaks.

Q13. Why is synchronization important?
A13. Without synchronization, it is possible for one thread to modify a shared object while another thread is in the process of using or updating that object's value. This often causes dirty data and leads to significant errors. 7 Things you must know about Java locks and synchronized key word.

Q14. What is the disadvantage of synchronization?
A14. The disadvantage of synchronization is that it can cause deadlocks when two threads are waiting on each other to do something. Also, synchronized code has the overhead of acquiring lock, and preventing concurrent access, which can adversely affect performance.

Inter thread communication & thread dead-lock explained Q&As & tutorial style

Q15. How does thread synchronization occurs inside a monitor? What levels of synchronization can you apply? What is the difference between synchronized method and synchronized block?
A15. In Java programming, each object has a lock. A thread can acquire the lock for an object by using the synchronized keyword. The synchronized keyword can be applied in method level (coarse grained lock – can affect performance adversely) or block level of code (fine grained lock). Often using a lock on a method level is too coarse. Why lock up a piece of code that does not access any shared resources by locking up an entire method. Since each object has a lock, dummy objects can be created to implement block level synchronization. The block level is more efficient because it does not lock the whole method.

```
class MethodLevel {
    //shared among threads
    SharedResource x, y ;

    pubic void synchronized method1() {
       //multiple threads can't access
    }

    pubic void synchronized method2() {
       //multiple threads can't access
    }

    public void method3() {
      //not synchronized
      //multiple threads can access
    }
}
```

```
class BlockLevel {
    //shared among threads
    SharedResource x, y ;
    //dummy objects for locking
    Object xLock = new Object(), yLock = new Object();

    pubic void method1() {
       synchronized(xLock){
          //access x here. thread safe
       }

       //do something here but don't use SharedResource x, y
       // because will not be thread-safe

        synchronized(xLock) {
            synchronized(yLock) {
               //access x,y here. thread safe
            }
        }

        //do something here but don't use SharedResource x, y
        //because will not be thread-safe
    }//end of method1
}
```

coarse grained Vs fine grained locks

The JVM uses locks in conjunction with monitors. A monitor is basically a guardian who watches over a sequence of synchronized code and making sure only one thread at a time executes a synchronized piece of code. Each monitor is associated with an object reference. When a thread arrives at the first instruction in a block of code it must obtain a lock on the referenced object. The thread is not allowed to execute the code until it obtains the lock. Once it has obtained the lock, the thread enters the block of protected code. When the thread leaves the block, no matter how it leaves the block, it releases the lock on the associated object. For static methods, you acquire a class level lock.

Q16. How will you go about writing a thread-safe & lazily initialized singleton class?
A16. Using a "Double checked locking" pattern. Explained in detail at Singleton design pattern in Java & 5 key follow up Interview Q&As

Q17. Why is ThreadLocal useful, and what is the consequence of abusing it?
A17. Allows you to create **per-thread-singleton**. Many frameworks use ThreadLocal to maintain some context related to the current thread.

Having said this, you must be very careful when using ThreadLocal as they are per thread static/global variable, and can cause **memory leaks**.

Learn more at Java ThreadLocal Interview questions & answers

Why multithreading questions are very popular?

It is because NOT many developers have a good grasp on multi-threading.

Java multithreading interview Q&As

Java multi threading scenarios for intermediate to experienced developers.

Comprehensive list of Java multi-threading Q&As.

Coding exercises in Java multithreading

Multithreaded Java code with issues & how will you fix it?

Multi-Threading – Create a simple framework where work items can be submitted

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ ♦ 12 UML interview Questions & Answers

04: ♦ 6 popular Java multithreading interview Q&As for the experienced ›

Tags: Core Java FAQs, Free FAQs, Novice FAQs

1-50+ Core Java Interview Questions and Answers with code & diagrams

SourceURL: https://www.java-success.com/a-q10-top-50-core-java-interview-questions/

Q1-Q10: ♥♦ Top 50+ FAQ Core Java Interview Questions and Answers

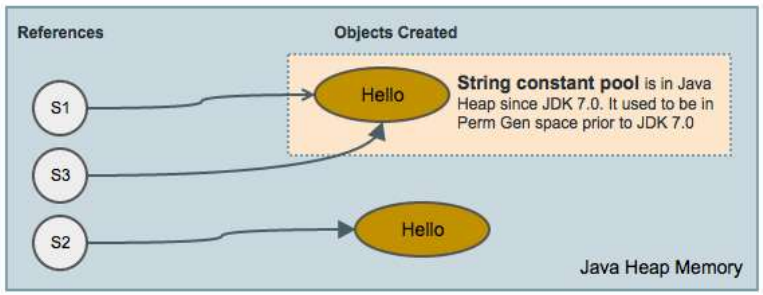
Posted on March 18, 2015 by  Arulkumaran Kumaraswamipillai Posted in 01 - FAQs Core Java, 11 - FAQs Free, Top 50+ FAQs Core Java

50+ Core Java interview questions answered for freshers to experienced Java developers. These FAQs have drill down links to more must know Core Java Interview Q&As.

Q1. What is the difference between "==" and "equals(...)" in comparing Java String objects?
A1. When you use "==" (i.e. shallow comparison), you are actually comparing the two object references to see if they point to the same object. When you use "equals(...)", which is a "deep comparison" that compares the actual string values. For example:

```
1  public class StringEquals {
2   public static void main(String[ ] args) {
3     String s1 = "Hello";
4     String s2 = new String(s1);
5     String s3 = "Hello";
6
7     System.out.println(s1 + " equals " + s2 + "--> " +  s1.equals(s2)); //true
8
9     System.out.println(s1 + " == " + s2 + " --> " + (s1 == s2)); //false
10    System.out.println(s1 + " == " + s3+ " --> " + (s1 == s3)); //true
11  }
12 }
13
```

The variable **s1** refers to the String instance created by "Hello". The object referred to by **s2** is created with s1 as an initializer, thus the contents of the two String objects are identical, but they are 2 distinct objects having 2 distinct references s1 and s2. This means that s1 and s2 do not refer to the same object and are, therefore, not ==, but *equals( )* as they have the same value "Hello". The s1 == s3 is true, as they both point to the same object due to internal caching. The references s1 and s3 are **interned** and points to the same object in the string pool.



Create a String object as a literal without the "new" keyword for caching

In Java 6 — all interned strings were stored in the **PermGen –** the fixed size part of heap mainly used for storing loaded classes and string pool.

In Java 7 – the string pool was relocated to the **heap**. So, you are not restricted by the limited size.

How about comparing the other objects like Integer, Boolean, and custom objects like "Pet"? Object **equals** Vs "==", and pass by **reference** Vs **value**.

Q2. Can you explain how Strings are interned in Java?
A2. String class is designed with the **Flyweight design pattern** in mind. Flyweight is all about re-usability without having to create too many objects in memory.

[**Further Reading:** Flyweight design pattern to improve memory usage & performance in Java]

A pool of Strings is maintained by the String class. When the *intern( )* method is invoked, equals(..) method is invoked to determine if the String already exist in the pool. If it does then the String from the pool is returned instead of creating a new object. If not already in the string pool, a new String object is added to the pool and a reference to this object is returned. For any two given strings s1 & s2, *s1.intern( ) == s2.intern( )* only if *s1.equals(s2)* is true.

Two String objects are created by the code shown below. Hence s1 == s2 returns false.

```
//Two new objects are created. Not interned and not recommended.
String s1 = new String("A");
String s2 = new String("A");

```

s1.intern() == s2.intern() returns **true**, but you have to remember to make sure that you actually do intern() all of the strings that you're going to compare. It's easy to forget to intern() all strings and then you can get confusingly incorrect results. Also, why unnecessarily create more objects?

Instead use string literals as shown below to intern automatically:

```
String s1 = "A";
String s2 = "A";

```

s1 and s2 point to the same String object in the pool. Hence s1 == s2 returns true.

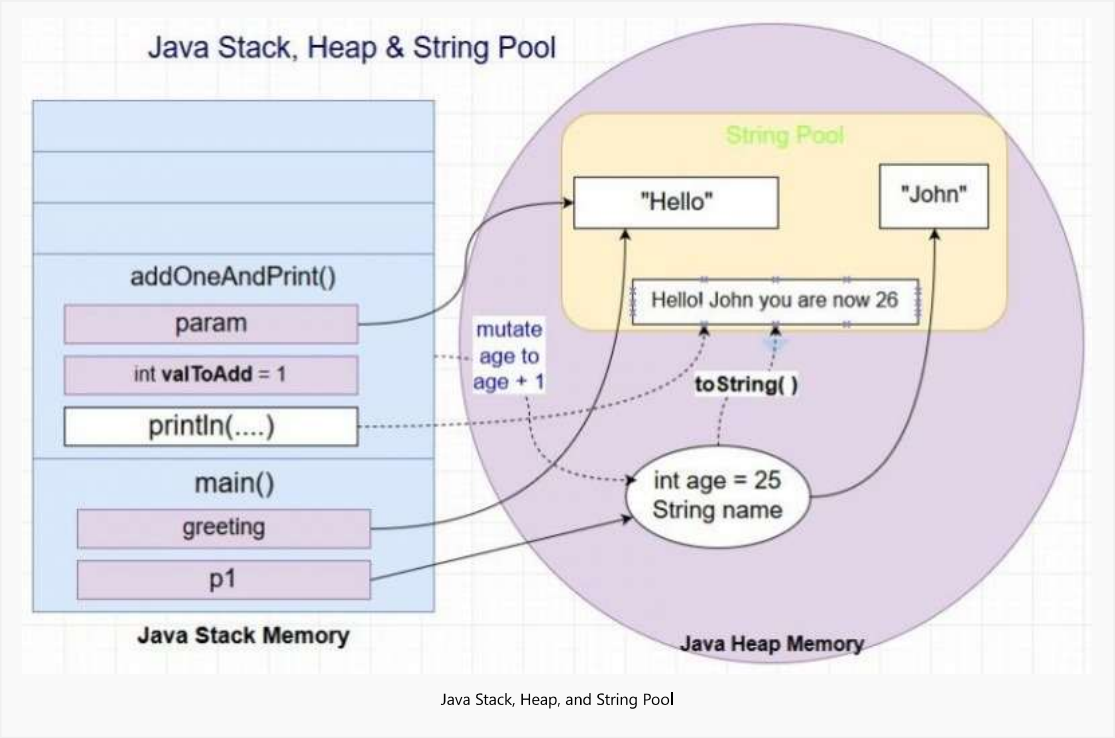
Since interning is automatic for String literals String s1 = "A", the *intern( )* method is to be used on Strings constructed with new String("A").

Q3. Can you describe what the following code does and what parts of memory the local variables, objects, and references to the objects occupy in Java?

```

public class Person {

    //instance variables
    private String name;
    private int age;

    //constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
13
14      public static void main(String[] args) {
15          Person p1 = new Person("John", 25); // "p1" is object refrence in stack pointing to the object in the heap
16          String greeting = "Hello";          // "greeting" is a reference in stack pointing to an Object in Heap String pool
17          p1.addOneAndPrint(greeting);
18      }
19
20      public void addOneAndPrint(String param) {
21          int valToAdd = 1;                       // local primitive variable in stack
22          this.age  = this.age + valToAdd;        // instance variable in heap is mutated to 26 via "param" reference in stack
23          System.out.println(param + "! " + this); // calls toString() method
24      }
25
26      @Override
27      public String toString() {
28          return name + " you are now " + age;
29      }
30  }
31
32
```

A3. The above code outputs "Hello! John you are now 26". The following diagram depicts how the different variable references & actual objects get stored.



Java Stack, Heap, and String Pool

Stack: is where local variables both primitives like int, float, etc & references to objects in the heap and method parameters are stored as shown in the above diagram.

Heap: is where objects are stored. For example, an instance of "Person" with name="John" and age=25. Strings will be stored in **String Pool** within the heap.

Q4. Why String class has been made immutable in Java?
A4. For performance & thread-safety.

1. Performance: **Immutable** objects are ideal for representing values of abstract data (i.e. value objects) types like numbers, enumerated types, etc. If you need a different value, create a different object. In Java, *Integer*, *Long*, *Float*, *Character*, *BigInteger* and *BigDecimal* are all immutable objects. Optimization strategies like caching of hashcode, caching of objects, object pooling, etc can be easily applied to improve performance. If Strings were made mutable, string pooling would not be possible as changing the string with one reference will lead to the wrong value for the other references.

[**Further reading:** Java immutable objects interview Q&As | Builder pattern and immutability in Java]

2. Thread safety as immutable objects are inherently thread safe as they cannot be modified once created. They can only be used as read only objects. They can easily be shared among multiple threads for better scalability.

Q. Why is a char array i.e char[] preferred over String to store a password?
A. String is immutable in Java and stored in the String pool. Once it is created it stays in the pool until garbage collected. This has greater risk of **1)**

someone producing a memory dump to find the password **2)** the application inadvertently logging password as a readable string.

If you use a char[] instead, you can override it with some dummy values once done with it, and also logging the char[] like "[C@5829428e" is not as bad as logging it as String "password123".

Q. How will you go about debugging thread-safety issues in Java?
A. Here are 5 ways to debug thread safety issues in Java.

Shouldn't there be more FAQs on Java strings? : 10 FAQ Java String class interview questions and answers as it is one of the very frequently used data types.

Let's move on to the next set of FAQ Core Java interview questions and answers focusing on the Java modifiers final, finally and finalize.

Q5. In Java, what purpose does the key words **final**, **finally**, and **finalize** fulfill?
A5. '**final**' makes a variable reference not changeable, makes a method not overridable, and makes a class not inheritable. Learn more about the drill down question on 6 Java modifiers that interviewers like to quiz you on...

'**finally**' is used in a try/catch statement to almost always execute the code. Even when an exception is thrown, the finally block is executed. This is used to close non-memory resources like file handles, sockets, database connections, etc till Java 7. This is is no longer true in Java 7.

Java 7 has introduced the **AutoCloseable** interface to avoid the unsightly try/catch/finally(within finally try/catch) blocks to close a resource. It also prevents potential resource leaks due to not properly closing a resource.

//Pre Java 7

```
1  BufferedReader br = null;
2  try {
3    File f = new File("c://temp/simple.txt");
4    InputStream is = new FileInputStream(f);
5    InputStreamReader isr = new InputStreamReader(is);
6    br = new BufferedReader(isr);
7
8    String read;
9
10   while ((read = br.readLine()) != null) {
11      System.out.println(read);
12   }
13 } catch (IOException ioe) {
14   ioe.printStackTrace();
15 } finally {
16   //Hmmm another try catch. unsightly
17   try {
18     if (br != null) {
19        br.close();
20 }
21   } catch (IOException ex) {
22     ex.printStackTrace();
23  }
24 }
25
```

Java 7 – try can have AutoCloseble types. *InputStream* and *OutputStream* classes now implements the Autocloseable interface.

```
1  try (InputStream is = new FileInputStream(new File("c://temp/simple.txt"));
2   InputStreamReader isr = new InputStreamReader(is);
3   BufferedReader br2 = new BufferedReader(isr);) {
4     String read;
5
6     while ((read = br2.readLine()) != null) {
7        System.out.println(read);
8     }
9   }
10
11  catch (IOException ioe) {
12    ioe.printStackTrace();
13  }
14
```

try can now have multiple statements in the parenthesis and each statement should create an object which implements the new *java.lang.AutoClosable* interface. The ***AutoClosable*** interface consists of just one method. void close() throws *Exception* {}. Each *AutoClosable* resource created in the try statement will be automatically closed without requiring a finally block. If an exception is thrown in the try block and another Exception is thrown while closing the resource, the first Exception is the one eventually thrown to the caller. Think of the *close( )* method as implicitly being called as the last line in the try block. If using Java 7 or later editions, use *AutoCloseable* statements within the try block for more concise & readable code.

If using Java 7 or later editions, use AutoCloseable statements within the try block for more concise & readable code.

'**finalize**' is called when an object is garbage collected. You rarely need to override it. It should not be used to release non-memory resources like file handles, sockets, database connections, etc because Java has only a finite number of these resources and you do not know when the **Garbage Collection** (i.e. GC) is going to kick in to release these non-memory resources through the *finalize( )* method.

[Further reading: 9 Java Garbage Collection interview Q&As to ascertain your depth of Java knowledge]

So, final and finally are used very frequently in your Java code, but the key word finalize is hardly or never used.

Q6. What is wrong with the following Java code?

```
1
2  import java.util.concurrent.TimeUnit;
3
4  class Counter extends Thread {
5
6      //instance variable
7      Integer count = 0;
8
9      // method where the thread execution will start
10     public void run() {
11         int fixed = 6;  //local variable
12
13         for (int i = 0; i < 3; i++) {
14             System.out.println(Thread.currentThread().getName() + ": result="
15                                + performCount(fixed));
16             try {
17                 TimeUnit.SECONDS.sleep(1);
18             } catch (InterruptedException e) {
19                 e.printStackTrace();
20             }
21         }
22     }
23
24     // let's see how to start the threads
25     public static void main(String[] args) {
26         System.out.println(Thread.currentThread().getName() + " is executing..." );
27         Counter counter = new Counter();
28
29         //5 threads
30         for (int i = 0; i < 5; i++) {
31             Thread t = new Thread(counter);
32             t.start();
33         }
34
35     }
36
37     //multiple threads can access me concurrently
38     private int performCount(int fixed) {
39         return (fixed + ++count);
40     }
41 }
42
```

A6. Above code is NOT thread-safe. This is explained in detail at Heap Vs Stack, Thread safety & Synchronized

Q7. When using Generics in Java, when will you use a wild card, and when will you mot use a wild card?
A7. It depends on what you are trying to do with a collection.

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 add objects to a data structure.

3. If you need to do both things (i.e. read and add objects), don't use any wildcard.

Learn more at Java Generics in no time "? extends" & "? super" explained with a diagram

Q8. Can you describe "method overloading" versus "method overriding"? Does it happen at **compile time** or **runtime**?
A8. Overloading deals with multiple methods in the same class with the **same name** but **different method signatures**. Both the below methods have the same method names but different method signatures, which mean the methods are overloaded.

```
1  public class {
2    public static void evaluate(String param1);     // overloaded method #1
3    public static void evaluate(int param1);        // overloaded method #2
4  }
```

This happens at compile-time. This is also called **compile-time polymorphism** because the compiler must decide which method to run based on the data types of the arguments. If the compiler were to compile the statement:

```
evaluate("My Test Argument passed to param1");
```

it could see that the argument was a "string" literal, and generates byte code that called method #1.

If the compiler were to compile the statement:

```
evaluate(5);
```

it could see that the argument was an "int", and generates byte code that called method #2.

Overloading lets you define the **same operation in different ways for different data**.

Overriding deals with two methods, one in the **parent class** and the other one in the **child class** and has the **same name** and **same signatures**. Both the below methods have the **same method names and the signatures** but the method in the subclass "B" overrides the method in the superclass (aka the parent class) "A".

Parent class

```
public class A {
 public int compute(int input) { //method #3
   return 3 * input;
 }
}
```

Child class

```
public class B extends A {
 @Override
 public int compute(int input) { //method #4
   return 4 * input;
 }
}
```

This happens at runtime. This is also called **runtime polymorphism** because the compiler does not and cannot know which method to call. Instead, the JVM must make the determination while the code is running.

The method compute(..) in subclass "B" overrides the method compute(..) in super class "A". If the compiler has to compile the following method,

```
public int evaluate(A reference, int arg2) {
  int result = reference.compute(arg2);
}
```

The compiler would not know whether the input argument '**reference**' is of type "A" or type "B". This must be determined during runtime whether to call method #3 or method #4 depending on what type of object (i.e. instance of Class A or instance of Class B) is assigned to the input variable "reference".

```
A obj1 = new B( );
A obj2 = new A( );
evaluate(obj1, 5); // 4 * 5 = 20. method #4 is invoked as stored object is of type B
evaluate(obj2, 5); // 3 * 5 = 15. method #3 is invoked as stored object is of type A
```

Overriding lets you define **the same operation in different ways for different object types**. It is determined by the "stored" object type, and NOT by the "referenced" object type.

It is important to understand **compile-time** vs **runtime** from core and enterprise Java interview questions & answers perspective. **More detailed**

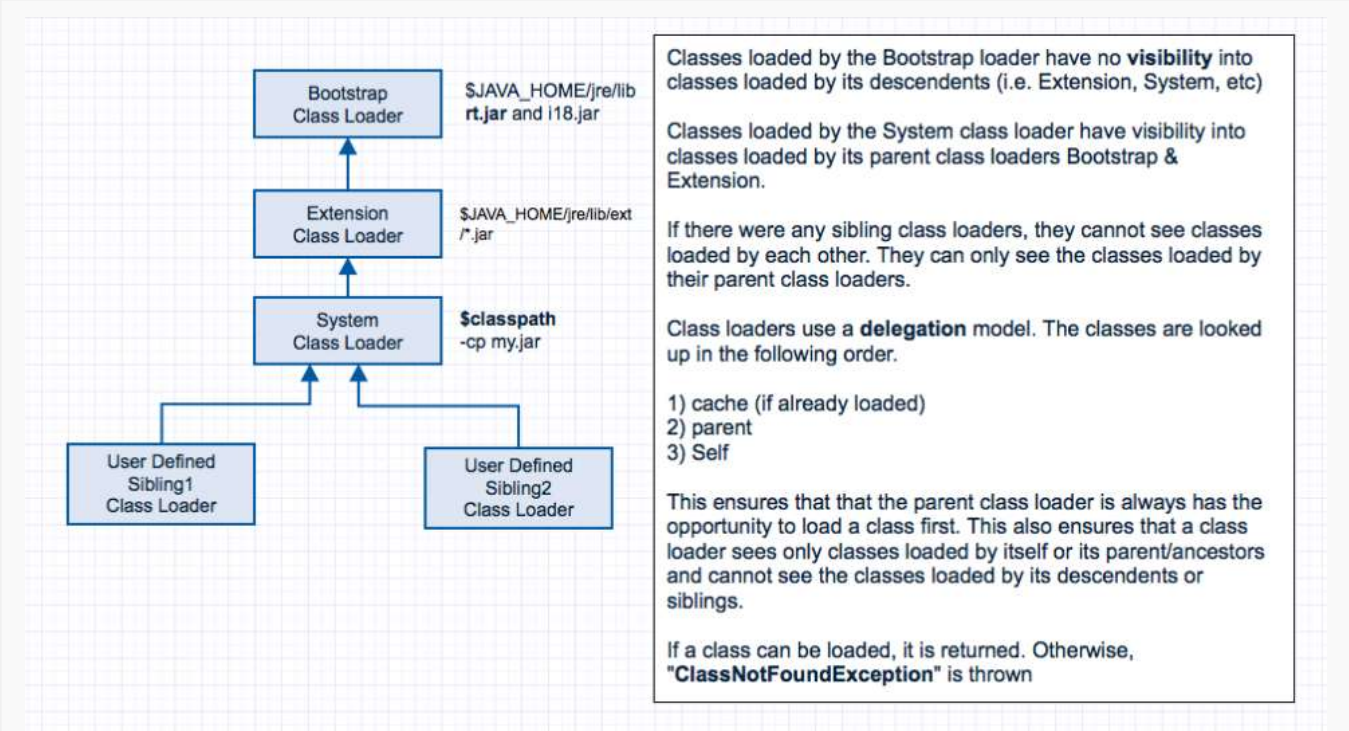
discussion: Compile-time Vs Run-time Interview Q&As

Q. Are overriding & polymorphism applicable static methods as well?
A. No. If you try to override a static method, it is known as hiding or shadowing.

More detailed discussion: Understanding Overriding, Hiding, and Overloading in Java? How does overriding give polymorphism?

Core Java Interview Questions and answers will not be complete without **Class Loaders**.

Q9. What do you know about class loading? Explain Java class loaders? If you have a class in a package, what do you need to do to run it? Explain dynamic class loading?
A9. Class loaders are hierarchical. Classes are introduced into the JVM as they are referenced by name in a class that is already running in the JVM. So, how is the very first class loaded? The very first class is specially loaded with the help of static *main( )* method declared in your class. All the subsequently loaded classes are loaded by the classes, which are already loaded and running. A class loader creates a namespace. All JVMs include at least one class loader that is embedded within the JVM called the primordial (or bootstrap) class loader. The JVM has hooks in it to allow user defined class loaders to be used in place of primordial class loader. Let us look at the class loaders created by the JVM.



Java Class Loader Basics

Class loaders are hierarchical and use a delegation model when loading a class. Class loaders request their parent to load the class first before attempting to load it themselves. When a class loader loads a class, the child class loaders in the hierarchy will never reload the class again. Hence uniqueness is maintained. Classes loaded by a child class loader have visibility into classes loaded by its parents up the hierarchy but the reverse is not true as explained in the above diagram.

Q10. Explain static vs. dynamic class loading?
A10. Classes are **statically loaded** with Java's "new" operator.

```
1 class MyClass {
2  public static void main(String args[]) {
3    Car c = new Car( );
4  }
5 }
6
```

Dynamic loading is a technique for programmatically invoking the functions of a **classloader** at **runtime**. Let us look at how to load classes dynamically.

```
1 //static method which returns a Class
```

```
Class clazz = Class.forName ("com.Car");  // The value "com.Car" can be evaluated at runtime & passed in via a variable
```

The above static method “forName” & the below instance (i.e. non-static method) “loadClass”

```
ClassLoader classLoader = MyClass.class.getClassLoader();
Class clazz = classLoader.loadClass(("com.Car"); //Non-static method that returns a Class
```

return the class object associated with the class name.Once the class is dynamically loaded the following method returns an instance of the loaded class. It’s just like creating a class object with no arguments.

```
// A non-static method, which creates an instance of a
// class (i.e. creates an object).
Car myCar = (Car) clazz.newInstance ( );
```

The string class name like “com.Car” can be supplied dynamically at runtime. Unlike the static loading, the dynamic loading will decide whether to load the class “com.Car” or the class “com.Jeep” at runtime based on a runtime condition.

```

public void process(String classNameSupplied) {
    Object vehicle = Class.forName (classNameSupplied).newInstance();
    //......
}
```

Static class loading throws NoClassDefFoundError if the class is NOT FOUND whereas the dynamic class loading throws ClassNotFoundException if the class is NOT FOUND.

Q. What is the difference between the following approaches?

```
Class.forName("com.SomeClass");
```

and

```
classLoader.loadClass("com.SomeClass");
```

A.

Class.forName(“com.SomeClass”)

— Uses the caller’s classloader and initializes the class (runs static intitializers, etc.)

classLoader.loadClass(“com.SomeClass”)

— Uses the “supplied class loader”, and initializes the class **lazily** (i.e. on first use). So, if you use this way to load a JDBC driver, it won’t get registered, and you won’t be able to use JDBC.

The “java.lang.API” has a method signature that takes a boolean flag indicating whether to initialize the class on loading or not, and a class loader reference.

```

forName(String name, boolean initialize, ClassLoader loader)
```

So, invoking

```
Class.forName("com.SomeClass")
```

is same as invoking

```
forName("com.SomeClass", true, currentClassLoader)
```

Q. What are the different ways to create a "ClassLoader" object?
A.

```

ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
ClassLoader classLoader = MyClass.class.getClassLoader();       // Assuming in class "MyClass"
ClassLoader classLoader = getClass().getClassLoader();          // works in any class

```

Q. How to load property file from classpath?
A. **getResourceAsStream()** is the method of java.lang.Class. This method finds the resource by implicitly delegating to this object's class loader.

```

final Properties properties = new Properties();
try (final InputStream stream = this.getClass().getResourceAsStream("myapp.properties")) {
    properties.load(stream);
    /* or properties.loadFromXML(...) */
}

```

Note: "Try with AutoCloseable resources" syntax introduced with Java 7 is used above.

Q. What is the benefit of loading a property file from classpath?
A. It is portable as your file is relative to the classpath. You can deploy the "jar" file containing your "property" file to **any location** where the JVM is.

Loading it from outside the classpath is NOT portable

```

final Properties properties = new Properties();
final String dir = System.getProperty("user.dir");

try (final InputStream stream =  new FileInputStream(dir + "/myapp/myapp.properties")) {
    properties.load(stream);
}

```

As the above code is NOT portable, you must document very clearly in the installation or deployment document as to where the property file is loaded from because if you deploy your "jar" file to another location, it might not already have the path "dir" and "myapp" configured.

So, loading it via the classpath is recommended as it is a portable solution.

Question to ponder

Q. What tips would you give to someone who is experiencing a class loading or "Class Not Found" exception?
A. "***ClassNotFoundException***" could be quite tricky to troubleshoot. When you get a *ClassNotFoundException*, it means the JVM has traversed the entire classpath and not found the class you've attempted to reference.

[**Further reading:** Java class loading interview Q&As to ascertain your depth of Java knowledge and Debugging LinkageErrors in Java]

Top 50+ FAQ Core Java Interview Q&As

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

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

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

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

Top 50+ FAQ EE Java Interview Q&As

5) Q01-Q28: Top 50+ EE Java interview questions & answers

6) Q29-Q53: Top 50+ JEE Interview Questions & Answers

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ Q08-Q15 written test questions and answers on core Java

05: ♦ Finding the 2nd highest number in an array ›

Tags: Core Java FAQs, Free Content, Free FAQs, Novice FAQs, TopX

Core Java Interview Questions-Part2

Notebook:	Core Java		
Created:	23-06-2018 16:48	**Updated:**	23-06-2018 17:24
URL:	https://www.java-success.com/why-favor-composition-over-inheritance-in-java-oop/		

7-Why favor composition over inheritance? Java OOP Interview Q&As

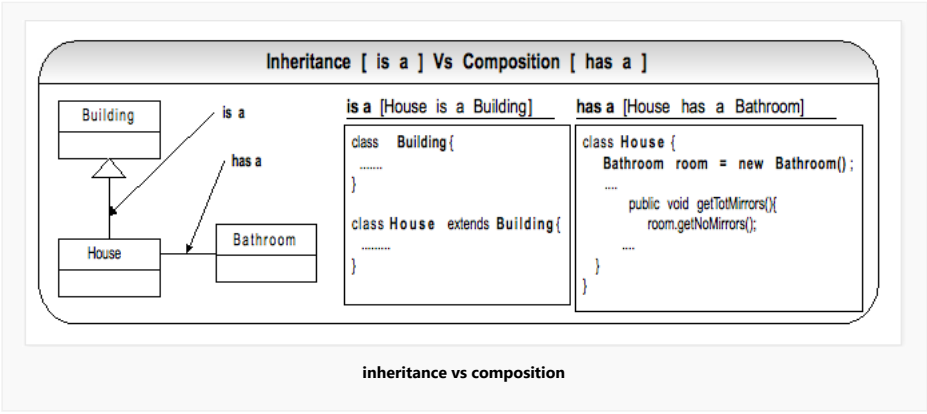
SourceURL: https://www.java-success.com/why-favor-composition-over-inheritance-in-java-oop/

02: ♦ Q11 – Q15 Why favor composition over inheritance?

Posted on August 11, 2014 by  Arulkumaran Kumaraswamipillai Posted in 09 - Design Concepts, OOP

One of the most popular Java OOP Interview Questions & Answers asked 90% of the time in job interviews is **why favor composition over inheritance?** The correct answer depends on the problem you are trying to solve, and the answer you give lead to more follow up questions to test your understanding of OOP as described below in detail.

Q11. How do you express an 'is a' relationship and a 'has a' relationship or explain inheritance and composition?
A11. 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 vs composition

Inheritance is uni-directional. For example *House* is a type of *Building*. But *Building* is not a *House*. Inheritance uses extends key word.

Composition: is used when a *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.

Q12. Which one to favor, composition or inheritance?
A12. The guide is that inheritance should be only used when subclass 'is a' super class. Don't use inheritance just to get code reuse. If there is no '**is a**' relationship then use composition for code reuse.

Reason #1: Overuse of implementation inheritance (uses the "extends" key word) can break all the subclasses, if the super class 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** with **composition**, which gives you code reuse. Interface inheritance is accomplished by implementing interfaces.

Reason #2: Composition is more flexible as it is easily achieved at runtime while inheritance provides its features at compile time. Don't confuse inheritance with polymorphism. Polymorphism happens at runtime as it states that Java chooses which overridden method to run only at runtime.

Reason #3: Composition offers better testability than Inheritance. Composition is easier to test because inheritance tends to create very coupled classes that are more fragile (i.e. fragile parent class) and harder to test in isolation. The IoC containers like Spring, make testing even easier through injecting the composed objects via constructor or setter injection.

Q13. Can you give an example of the Java API that favors composition?
A13. The Java IO classes that use composition to construct different combinations of I/O outcomes like reading from a file or System.in, buffering the streams, tracking the line numbers, piping the streams for efficiency, etc using the **decorator design pattern** at run time.

```
//construct a reader
StringReader sr = new StringReader("Some Text....");
//decorate the reader for performance
BufferedReader br = new BufferedReader(sr);
//decorate again to obtain line numbers
LineNumberReader lnr = new LineNumberReader(br);
```

The GoF design patterns like strategy, decorator, and proxy favor composition for code reuse over inheritance. Interesting read ti further your knowledge in OOP & design patterns: Why do Proxy, Decorator, Adapter, Bridge, and Facade design patterns look very similar? What are the differences?

Q14. Can you a give an example where GoF design patterns use inheritance?
A14. A typical example of using inheritance for code reuse is in frameworks where the **template method design pattern** is used.

Template Method design pattern is a good example of using an **abstract class** and this pattern is used very prevalently in application frameworks.

1. Java HTTP Servlet's doGet and doPost methods.
2. Message Driven EJB's and Spring message listener's onMessage(....) method.
3. Spring framework's JdbcTemplate, JmsTemplate, etc.
4. All non-abstract methods of *java.io.InputStream*, *java.io.OutputStream*, *java.util.AbstractList*, *java.util.AbstractMap*, *java.io.Reader*, etc

The Template Method design pattern is about providing partial implementations in the abstract base classes, and the subclasses can complete when extending the Template Method base class(es). Here is an example

```
//cannot be instantiated
public abstract class BaseTemplate {

 public void process() {
    fillHead();
    //some default logic
    fillBody();
    //some default logic
    fillFooter();
  }

  //to be overridden by sub class
  public abstract void fillBody();

  //template method
  public void fillHead() {
      System.out.println("Simple header");
  }

   //template method
   public void fillFooter() {
      System.out.println("Simple footer");
   }

   //more template methods can be defined here
}
```

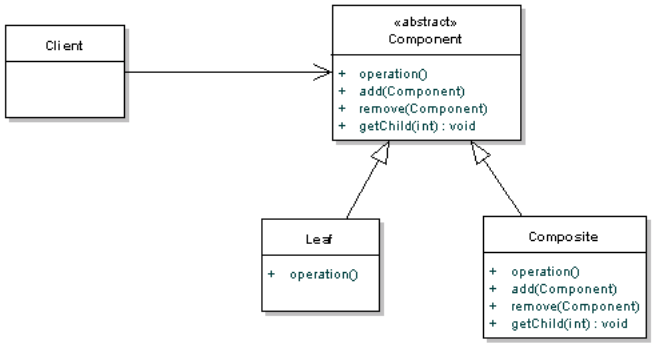
```
public class InvoiceLetterProcessor extends BaseTemplate {

 @Override
 public void fillBody() {
  System.out.println("Invoice body" );
 }

 // template method
 public void fillHead() {
  System.out.println("Invoice header");
 }
}
```

```
public class InvoiceTestMain {

   public static void main(String[] args) {
        //subclass is up cast to base class -- polymorphism
        BaseTemplate template = new InvoiceLetterProcessor();
        template.process();
```

```
    }
}
```

Another common pattern that would use inheritance is the **Composite design pattern**.

A node or a component is the parent or base class and derivatives can either be leaves (singular), or collections of other nodes, which in turn can contain leaves or collection-nodes. When an operation is performed on the parent, that operation is recursively passed down the hierarchy. An interface can be used instead of an abstract class, but an abstract class can provide some default behavior for the add(), remove() and getChild() methods.



Q15. What questions do you ask yourself to choose composition (i.e. has-a relationship) for code reuse over implementation inheritance (i.e. is-a relationship)?
A15. Do my subclasses only change the implementation and not the meaning or internal **intent** of the base class? Is every object of type *House* really "is-an" object of type *Building*? Have I checked this for "Liskov Substitution Principle"

According to **Liskov substitution principle** (LSP), a Square is not a Rectangle provided they are mutable. Mathematically a square is a rectangle, but behaviorally a rectangle needs to have both length and width, whereas a square only needs a width.

Another typical example would be an Account class having a method called *calculateInterest(..)*. You can derive two subclasses named *SavingsAccount* and *ChequeAccount* that reuse the super class method. But you cannot have another class called a *MortgageAccount* to subclass the above Account class. This will break the Liskov substitution principle because the **intent** is different. The savings and cheque accounts calculate the interest due to the customer, but the mortgage or home loan accounts calculate the interest due to the bank.

Violation of LSP results in all kinds of mess like failing unit tests, unexpected or strange behavior, and violation of **open closed principle** (OCP) as you end up having if-else or switch statements to resolve the correct subclass. For example,

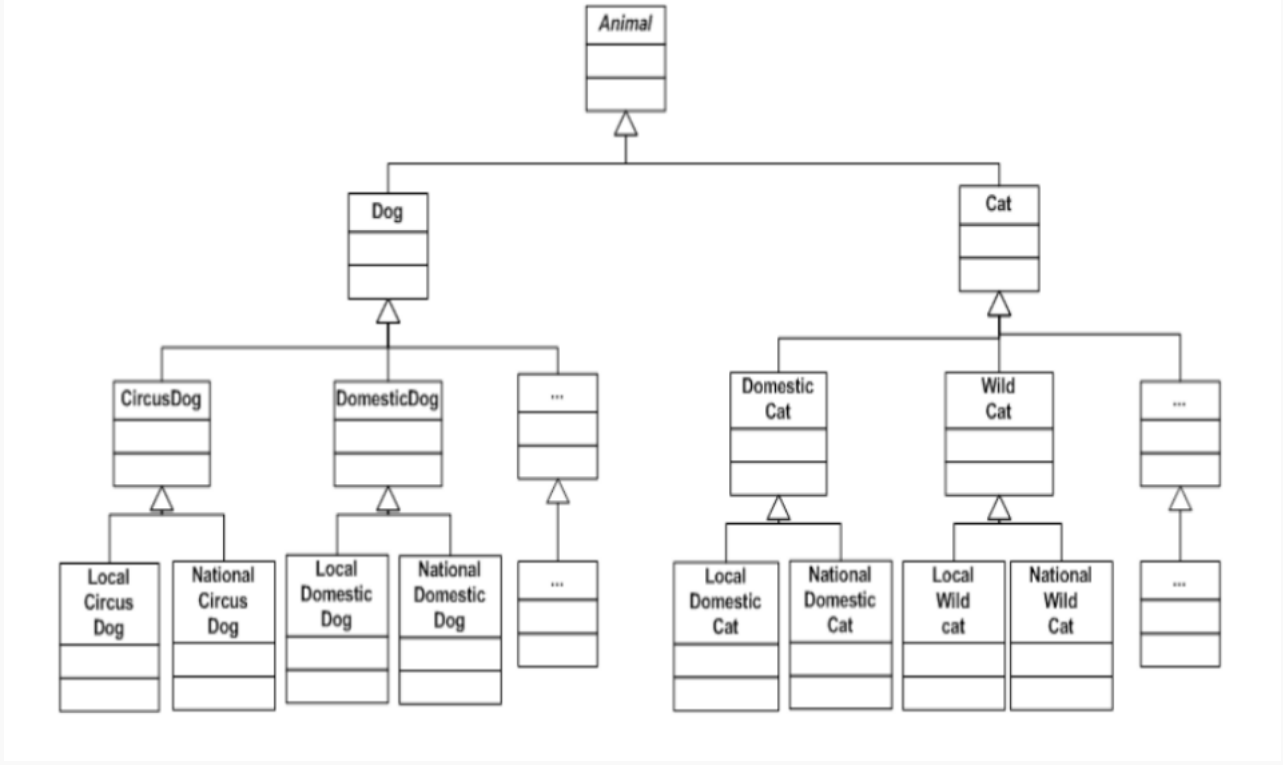
```
if(shape instanceof Square){
     //....
}
else if (shape instanceof Rectangle){
    //...
}
```

If you cannot truthfully answer yes to the above questions, then favor using "has-a" relationship (i.e. composition). Don't use "is-a" relationship for just convenience. If you try to force an "is-a" relationship, your code may become inflexible, post-conditions and invariants may become weaker or violated, your code may behave unexpectedly, and the API may become very confusing. LSP is the reason it is hard to create deep class hierarchies.

Learn more about **SOLID** OOP design principles.

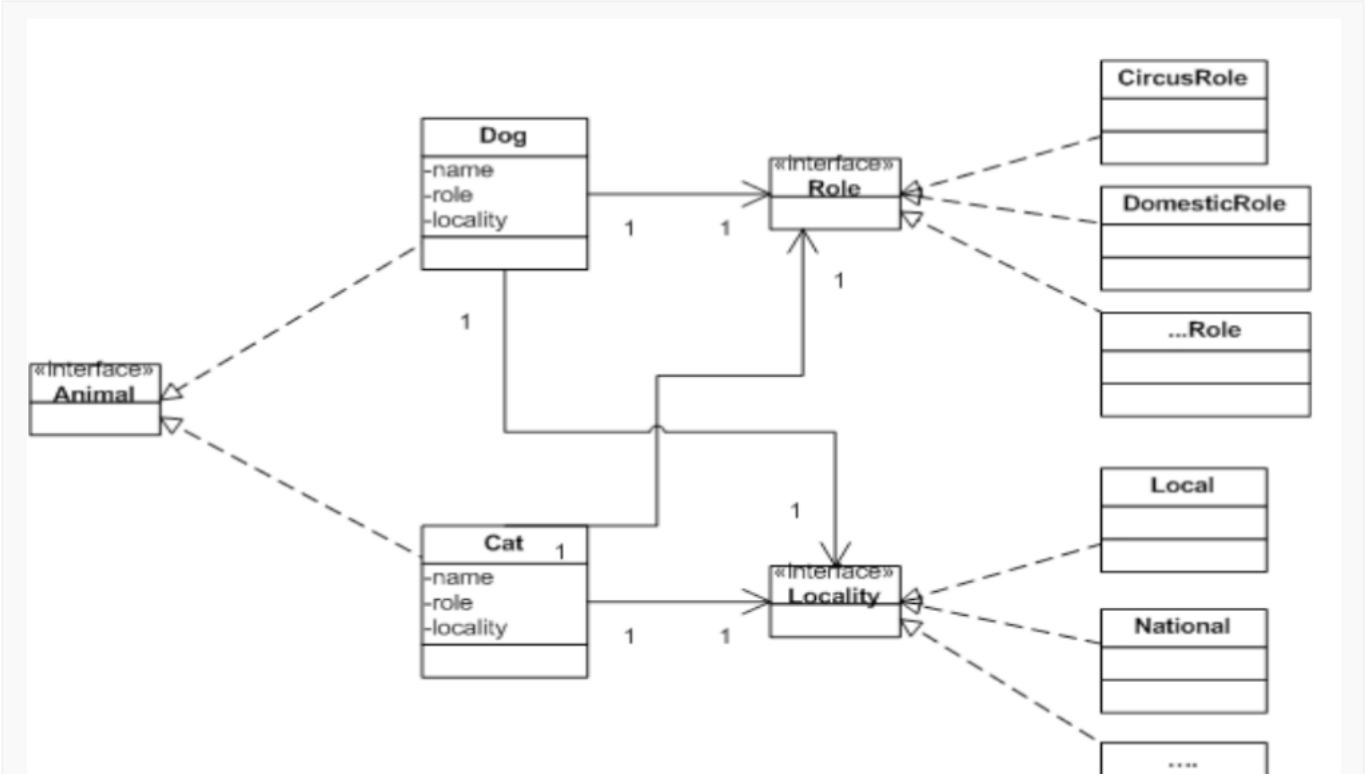
Always ask yourself, can this be modeled with a "**has-a**" relationship to make it more flexible?

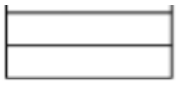
For example, If you want to model a circus dog, will it be better to model it with "is a" relationship as in a *CircusDog* "is a" Dog or model it as a role that a dog plays? If you implement it with implementation inheritance, you will end up with sub classes like *CircusDog*, *DomesticDog*, *GuideDog*, *SnifferDog*, and *StrayDog*. In future, if the dogs are differentiated by locality like local, national, international, etc, you may have another level of hierarchy like *LocalCircusDog*, *NationalCicusDog*, *InternationalCircusDog*, etc extending the class *CircusDog*. So you may end up having 1 animal x 1 dog x 5 roles x 3 localities = 15 dog related classes. If you were to have similar differentiation for cats, you will end up having similar cat hierarchy like *WildCat*, *DomesticCat*, *LocalWildCat*, *NationalWildCat*, etc. **This will make your classes strongly coupled.**



Explosion of classes due to inheritance

If you implement it with interface inheritance, and composition for code reuse, you can think of circus dog as a role that a dog plays. These roles provide an abstraction to be used with any other animals like cat, horse, donkey, etc, and not just dogs. The role becomes a "has a" relationship. There will be an attribute of interface type Role defined in the Dog class as a composition that can take on different subtypes (using interface inheritance) such as *CircusRole*, *DomesticRole*, *GuideRole*, *SnifferRole*, and *StrayRole* at **runtime**. The locality can also be modeled similar to the role as a composition. This will enable different combinations of roles and localities to be constructed at runtime with 1 dog + 5 roles + 3 localities = 9 classes and 3 interfaces (i.e. *Animal*, *Role* and *Locality*). As the number of roles, localities, and types of animals increases, the gap widens between the two approaches. You will get a better abstraction with looser coupling with this approach as **composition is dynamic and takes place at run time compared to implementation inheritance, which is static**.





Composition to the rescue

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ 01: ♦Q1 – Q10 Java OOPs interview questions & answers

05: ♦Q19- Q24 How to create a well designed Java application with OOP? ›

Tags: Core Java FAQs

6-Java Collection interview questions on differences between X and Y

SourceURL: https://www.java-success.com/java-collection-interview-questions/

01: ♦ 17 FAQ Java Collection interview questions answered on differences between X & Y

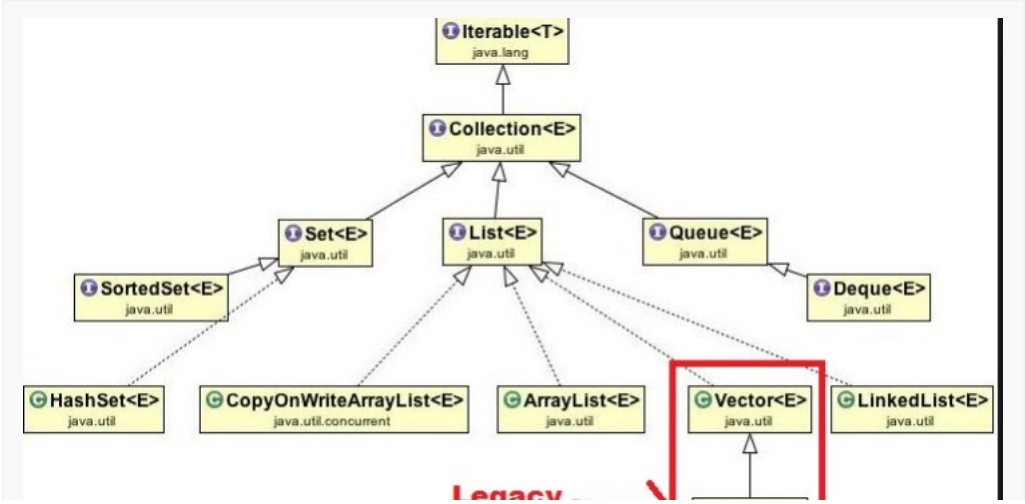
Posted on April 1, 2015 by  Arulkumaran Kumaraswamipillai Posted in 01 - FAQs Core Java, 03 - FAQs Code Quality, Collection and Data structures, Differences Between X & Y, member-paid

Java Collections interview questions and answers and Java String class interview Q&As are must know as you can't write any decent Java application without these 2 APIs. This post focuses on Java Collection interview Q&As on differences between X and Y based on the Java Collection Framework (i.e. **JCF**).

Q1. What are the differences between legacy and non-legacy collection classes in the java.util package?
A1. Early version of Java defined several classes and one interface to store objects. These old classes are known as legacy classes. The legacy classes like Vector, Hashtable, and Stack still exist in the API to provide forwards compatibility to code written during early days of Java.

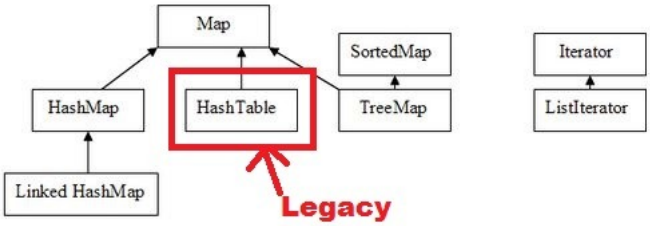
These classes are now re-engineered with the "Java Collections Framework (JCF)" that looks as shown below with interfaces & classes.

List, Set, and Queue



Legacy classes & JCF

Map



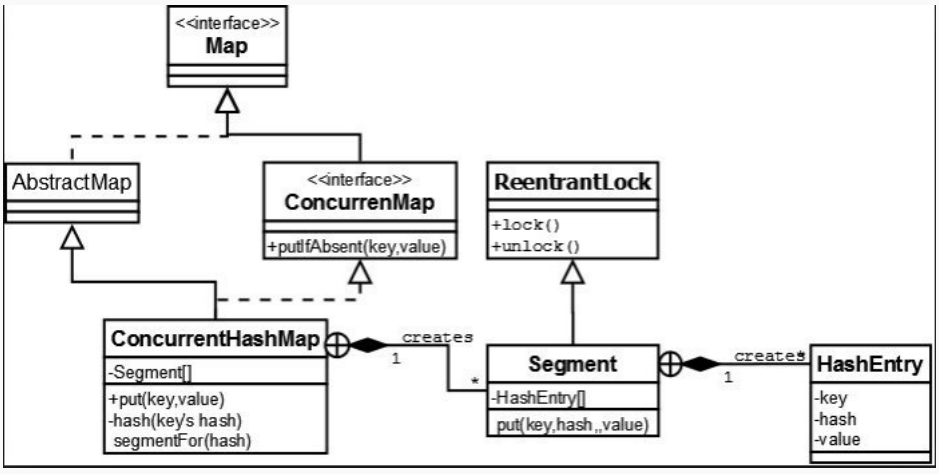
Legacy Hashtable & JCF

All the methods of the legacy classes are **synchronized** (i.e. locks the whole collection), hence can lead to performance issues. The methods of the re-engineered Collections framework classes like ArrayList, HashSet, BlockingQueue, HashMap, etc are NOT synchronized (i.e. they don't lock). If you need thread-safety you have two choices:

1) Using the java.util.Collections API methods provide a basic conditionally thread-safe implementation of Map and List.

```
1
2 Collections.synchronizedMap(aMap)
3 Collections.synchronizedList(aList)
4
```

2) Using the java.util.**concurrent** package classes like ConcurrentHashMap, CopyOnWriteArrayList, etc that use better concurrency techniques. For example, the ConcurrentHashMap uses a technique known as the "**lock striping**", which divides the whole map into several segments and locks only the relevant segments, which allows multiple threads to access other segments of same ConcurrentHashMap without locking. So, concurrent reads are possible.



Reference: http://www.javarticles.com/2012/06/concurrenthashmap.html

Similarly, CopyOnWriteArrayList & CopyOnWriteArraySet allow **concurrent reads** by multiple threads without requiring any locking and when a write happens it copies the whole ArrayList or HashSet and swap with a newer one.

Q2. What are the differences between Enumeration and Iterator?
A2. Enumeration is old and it's there from JDK1.0 whereas iterator is newer. The key difference between Enumeration and iterator is that **"Iterator has a remove() method"** whereas Enumeration doesn't. Enumeration acts as Read-only interface, whilst an Iterator can manipulate the objects like adding and removing. Enumeration can only be used with legacy collection classes whereas an Iterator can be used with both legacy & non-legacy classes.

Q3. What is the difference between an **Iterable** & an **Iterator**?
A3. Java Iterable Vs Iterator differences and know how.

Q4. What are the differences between fail-fast and fail-safe iterators?
A4. Iterators returned by most of pre JDK1.5 collection classes like Vector, ArrayList, HashSet, etc are **fail-fast iterators**. Iterators returned by JDK 1.5+ ConcurrentHashMap and CopyOnWriteArrayList classes are **fail-safe iterators**.

Use copy-on-write List/Set and concurrent maps from the java.util.**concurrent** package to prevent **ConcurrentModificationException** being thrown whilst preserving thread safety. These classes provide **fail-safe iteration** as opposed to non-concurrent classes like ArrayList, HashSet, etc use **fail-fast iteration** leading to **ConcurrentModificationException** if you try to remove an element while iterating over a collection.

[**Further readings:** Beginner what is wrong with this Java code? | Top 5 Core Java Exceptions and best practices]

You need to choose the right data structure based on its **usage/access patterns:** More reads Vs more writes, FIFO (First-In-First-Out) Vs LIFO (Last-In-First-Out), random reads, inserts in the middle vs end, does ordering matter?, duplicates allowed?, concurrent access possible? etc.

Q5. What are the differences between ArrayList and LinkedList?
A5.

1. Insertions and deletions are faster in **LinkedList** compared to an ArrayList as LinkedList uses links (i.e. before and next reference) as opposed to an ArrayList, which uses an array under the covers, and may need to resize the array if the array gets full. Adding to an ArrayList has a worst case scenario of O(n) whilst LinkedList has O(1).

2. LinkedList has more memory footprint than **ArrayList**. An ArrayList only holds actual object whereas LinkedList holds both data and reference of next and previous node.

3. Random access has the worst case scenario of O(n) in LinkedList as to access 6th element in a LinkedList with 8 elements, you need to traverse through 1 to 5th element before you can get to the 6th element, whereas in an **ArrayList**, you can get the 6th element with O(1) with list.get(5).

4. Add or remove from the head of the list is in favor of **LinkedList** with O(1) whereas O(n) for ArrayList.

5. Iterating over either kind of List is the same. ArrayList is technically faster, but the difference is small unless you're doing something really performance-sensitive.

What are these O(n), O(log n), O(1), etc? Learn more about understanding Big O notations through Java examples.

Q6. Does a HasMap internally use a LinkedList or an ArrayList?
A6. Until Java 8: LinkedList. **Java 8 onwards:** a "binary tree" because in case of **high collision** the lookup is reduced to O(log n) from O(n) by using binary trees. Learn more at HashMap & HashSet and how do they internally work?

Also when hash keys arrive from untrusted sources like HTTP headers, the resulting keys will have the same hashcode, which not only causes high collisions, but also when you perform lots of look-ups, you will experience the **Denial of Service** (i.e. DoS) attacks.

Q7. What is the difference between an "unmodifiable" & an "immutable" collection?
A7. An unmodifiable collection is often a **wrapper** around a modifiable collection. "unmodifiableList" will throw "java.lang.UnsupportedOperationException" if you try to add/remove an element from it, but other code may still have access to "modifiableList", which is a back door. So, you can't rely on the contents not changing.

```
1
2 Collection<String> unmodifiableList = Collections.unmodifiableCollection(modifiableList);
3
```

An immutable collection ensures that the collection itself cannot be altered by **deeply cloning or copying** the collection. It does not let its reference escape via its constructors & getter methods as shown below:

```
1
2 import java.util.ArrayList;
3 import java.util.List;
4
5 public final class MyImmutableList {
6
7     private final List<String> myList;
8
9     public MyImmutableList(List<String> list) {
```

```
        List<String> clone = new ArrayList<String>(list.size());
        for (String item : list){
            clone.add(item);
        }

        this.myList = clone; //cloned list is assigned
                             //original list cannot be mutated from outside this class
    }

    public List<String> getMyList() {
        List<String> clone = new ArrayList<String>(myList.size());
        for (String item : myList){
            clone.add(item);
        }
        return clone; // cloned list is returned
                      // original list cannot be mutated from outside this class
    }

    // ....equals(), hashCode(), etc

    @Override
    public String toString() {
        return "MyImmutableList [myList=" + myList + "]";
    }
}
```

Q8. What are the differences among HashSet, ArrayList, synchronized list/set, CopyOnWriteArraySet and CopyOnWriteArrayList?
A8. Compared to a list interface, a set interface does not allow duplicates. HashSet and ArrayList are not thread-safe and you need to provide your own synchronization with locks or use the java.util.**Collections** class that provides utility methods like

```
Collections.synchronizedList(aList);
Collections.synchronizedSet(aSet);
Collections.synchronizedCollection(aSetOrList);
Collections.synchronizedSortedSet(aSortedSet)
```

The above synchronizedXXX lock the whole collection like the legacy Vector whereas CopyOnWriteArraySet and CopyOnWriteArrayList are not only thread-safe, but also **1)** more efficient as they allow **concurrent multiple reads and single write**. This concurrent read and write behavior is accomplished by making a brand new copy of the list every time it is altered.

The CopyOnWriteArrayList's iterator **2)** never throws ConcurrentModificationException while Collections.synchronizedList's iterator may throw it.

It is also imperative to note that as per the Java API for the "Collections" class's synchronizedXXX methods, the user must manually synchronize on the returned list when iterating over it as shown below.

```
  List list = Collections.synchronizedList(new ArrayList());
      ...
  synchronized (list) {
      Iterator i = list.iterator(); // Must be in synchronized block
      while (i.hasNext())
          foo(i.next());
  }
```

When to favor a CopyOnWriteXXX structure? Write operations for a CopyOnWriteXXX can potentially be very slow as they involve copying the entire list. The CopyOnWriteXXX is favored when the number of reads & traversals via an iterator are significantly more than the number of writes. It also gives you **fail safe** iteration when you want to add/remove elements during iteration.

Q9. What are differences between a HasMap and a TreeMap?
A9. TreeMap is an implementation of a SortedMap, where the **order** of the keys can be sorted, and when iterating over the keys, you can expect that keys will be in order. HashMap on the other hand, makes no such guarantee on the **order**.

Q10. What are differences between a HashMap and a ConcurrentHashMap?
A10. HashMap is not thread-safe and you need to provide your own synchronization with Collections.synchornizedMap(hashMap), which will return a collection which is almost equivalent to the legacy Hashtable, where every modification operation on Map is locked.

What is wrong with this code? HashMap race condition example & how ConcurrentHashMap fixes it?

As the name implies, ConcurrentHashMap provides thread-safety by dividing the whole map into different segments based upon concurrency level and **locking only particular segment instead of locking the whole map**. In the ConcurrentHashMap API, you will find the following constants are used

```
static final int DEFAULT_INITIAL_CAPACITY = 16;
static final int DEFAULT_CONCURRENCY_LEVEL = 16;
```

and the constructor takes "concurrencyLevel" as an argument.

```
public ConcurrentHashMap (int initialCapacity, float loadFactor, int concurrencyLevel)
```

Instead of a map wide lock, ConcurrentHashMap maintains a list of 16 locks by default, which means each to lock on a single segment of the Map. Each segment can have multiple buckets. This indicates that 16 threads can modify the ConcurrentHashMap at the same time as long as each thread works on a different segment. The reads don't block at all.

ConcurrentHashMap does not allow NULL key values, whereas HashMap can hold only one null key. This is because if map.get(key) returns a null, you can't distinguish even with map.contains(key) call whether the value is null or the key itself is not present as the map might have changed between the map.get(key) and map.contains(key) calls due its concurrent read/write feature.

What is wrong with this code? ConcurrentHashMap atomic operations issue & how to fix?

Q11. What are differences between a HashMap and a LinkedHashMap?
A11. LinkedHashMap will iterate in the **order** in which the entries were put into the map. HashMap does not provide any guarantees about the iteration order.

Q12. What are differences between a Queue and a BlockingQueue?
A12. BlockingQueue is a Queue that supports additional operations that **wait** for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element. The main advantage is that a BlockingQueue is that it provides a correct, thread-safe implementation with throttling.

- The producers are throttled to add elements if the consumers get too far behind.
- If the Queue capacity is limited, the memory consumption will be limited as well.

Q13. What are differences between an Array and a List?
A13.

- Array is a fixed length data structure whilst a List is a variable length Collection class. List allows you to add and subtract elements even it is an O(n) operation in worst case scenario.
- An array can use primitive data types or objects, but the List classes can only use objects.
- Arrays are inflexible and do not have the expressive power of generic types.
- List gives you the data abstraction as you can swap ArrayList, LinkedList, CopyOnWriteArrayList, etc depending on the requirements.

Q14. What is the difference between a Comparable and a Comparator?
A14. The Comparable interface provides a compareTo(..) method to be called while sorting **naturally** (i.e.by default). You can define your own ordering (i.e. **custom**) logic through the compare(...) method by implementing the Comparator interface.

[**Further readings:** 4 Sorting objects in Java interview Q&As | Different ways to sort a collection of objects in pre and post Java 8]

Q15. What is the difference between ArrayList and Vector?
A15. Vector, Stack, and Hashtable are legacy data structures and must not to be used. All methods in these classes are synchronized (i.e. coarse grained lock), hence not efficient. Favor the concurrent data structures for concurrent reads and single write.

Java 8

Q16. What is the difference between "with and without **lambdas**" to the collections API?
A16. Lambdas introduced in Java 8 would be worthless if we didn't have any means for applying lambdas to the JCF. So, "**default methods**" were introduced to Java interfaces, which has the benefit that default methods don't break the implementations. In other words, interfaces in Java 8 onwards can now implement **behavior** via the default methods. So, default methods filter, map, reduce, forEach, etc are now added to the "java.util.stream.**Stream**" interface.

The Iterable interface with the Default method is shown below,

```
public interface Iterable<T> {

    public default void forEach(Consumer<? super T> consumer) {

        for (T t : this) {
            consumer.accept(t);
        }
    }
}
```

The same mechanism has been used to add Stream without breaking the implementing classes. Java 8 Streams, lambdas, intermediate vs terminal ops, and lazy loading with simple examples.

Q17. What is the difference between having the package java.util.stream and not having java.util.stream?
A17. The new java.util.stream package has been added to Java 8 to allow us to perform filter, map, and reduce operations with the help of **lambda expressions** on the collection classes. For example,

```
List<Person> persons = constructPersons(...);
Stream<Person> personsOver16 = persons.stream().filter(p -> p.getAge() > 16);
```

Questions to ponder

Q. If Java did not have a stack or map, how would you go about writing your own?
A. It is a best practice NOT to reinvent the wheel, and also writing your own API or framework is NOT a trivial task. The interviewer is trying to assess your depth of knowledge with the data structures with this question. Writing your own HashSet and HashMap example with code & diagrams.

Don't worry, if these Q&As are not detailed

This post is for a quick brush-up on the differences. These Q&As are discussed in detail with code, examples & diagrams:

1. When to use which Java collection or data structure? and why?

2. Sorting objects in Java interview Q&A.

3. 60+ Java Collection API Interview Q&As.

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/EE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ ♦ Part 4: Badly designed classes & interfaces

♦ Finding the perfect number ›

Tags: Core Java FAQs

13-Java 8 transforming your thinking from OOP to FP

SourceURL: https://www.java-success.com/java-8-transforming-thinking-oop-fop-java-8-examples/

04: ♦ 5 FAQs on transforming your thinking from OOP to

Posted on November 8, 2014 by Arulkumaran Kumaraswamipillai Posted in 09 - FAQs OOP & FP, FP, Java 8

One needs to get used to the transformation from **imperative programming** to **functional programming**. You like it or not, you will be using functional proc interviewers are going to quiz you on functional programming. Fortunately, Java is not a fully functional programming language, and hence one does not requ programming. Java 8 supports both imperative and functional programming approaches.

Q1. What is the difference between imperative and declarative programming paradigms?
A1. **Imperative** (or procedural) programming: is about defining the computation how to do something in terms of statements and state changes, and as a resu will happen.

Declarative programming: is about declaratively telling what you would like to happen, and let the library or functions figure out how to do it. SQL, XSLT and declarative languages.

Q2. Does functional programming use imperative or declarative approach?
A2. Functional programming is a form of declarative programming, where **functions are composed of other functions** — g(f(x)) where g and f are functions. becomes the input for the composing function. A typical example of functional programming, which you may have used is transforming an XML document usi composable and isolated XSL style sheets are used for transformation.

Q3. Where would you use FP?
A3. As IT professionals, we need to use the right paradigm or tool for the right job. In Java, OOP, FP, and AOP can co-exist. These paradigms compliment each very powerful at

1) Solving problems that require lots of concurrency & parallelism. FP favors immutability, hence easier to parallelize. FP shines in efficient processing of BigDa

2) Applying math heavy algorithms or a series of transformations. For example, solving a problem that can be decomposed into a series of mathematical opera would be to take a list of numbers, filter out only the odd numbers, double the resulting odd numbers, compute the sum, and so on.

3) Creating Domain Specific Languages (i.e. DSLs). A DSL is a computer language targeted at solving a particular kind of problem and it is not planned to solve domain. DSLs have become a valuable part of the Groovy language.

Q4. What are the differences between OOP and FP?
A4.

	OOP	FOP
focus	To solve problems, OOP developers design class hierarchies, focus on proper encapsulation, and think in terms of class contracts (i.e. design by contract). The behavior and state of object types are of paramount importance, and language features, such as classes, interfaces, inheritance, and polymorphism, are provided to address these concerns.	To solve computational problems by evaluating pure functions to transform a collection of data. Functional programming avoids state and mutable data, and instead emphasizes on the application and composition of functions.
flow control	loops, conditions, and method calls	function calls and recursion
state changes	Necessary	No state changes. **Effectively final variables** are used in Java
order of execution	High importance	low importance

In OOP, x = x+ 5 makes sense, but in mathematics or functional programming, you can't say x = x + 5 because if x were to be 2, you can't say that 2 = 2 + 5. Ir you need to say f(x) -> x + 5.

#1. All about

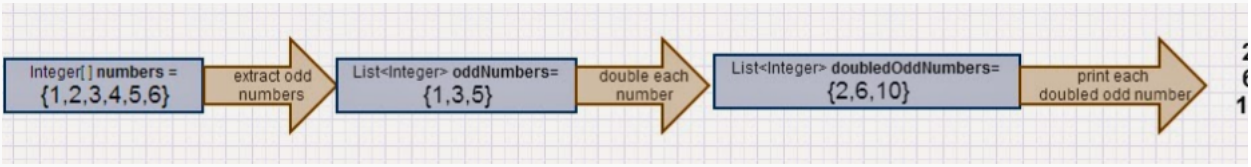
OOP is all about objects. Objects can be stored in variables & collections, objects can be passed around in method arguments, objects can be composed, etc. T citizens.

FP is all about functions. Functions can be stored in a variables & collections, functions can be passed as arguments to functions which can return a function, e other functions are called **higher-order** functions.

#2. Focus:

OOP focuses on solving business problems by designing classes, interfaces, and contracts. The behavior and state are very important to OOP, and applies OO c polymorphism, inheritance, and encapsulation.

FP focuses on computational problems by evaluating functions to transform a collection of data by focusing on the composition and application of functions.



Java functional programming focusing on transforming data

#3. Flow Control:

OOP uses loops, conditions, and method calls. Order of execution is very important.

FP does not like loops as the loops are for state change constructs. FP uses function calls and recursion. Order of execution is less important.

#4. Mutability:

OOP uses loops to mutate state. For example, increases the employee salary by 5% as shown below.

```
Employee[] employees = { new Employee("John", 45000), new Employee("Peter", 65000.00), new Employee("Sam", 85000.00)};

List<Employee> inputList = Arrays.asList(employees);

//state change
for (Employee employee : inputList) {
    employee.setSalary(employee.getSalary() * 1.05);
}

//original list is mutated
System.out.println(inputList); //has side-effects

```

FP favours immutability. Pure functions must always return the same output for a given input. This is called **idempotency** and **referential transparency**. In a re alter values by taking some data as input, and output some results. FP avoids mutation by returning **new data** instead.

```
Employee[] employees = { new Employee("John", 45000), new Employee("Peter", 65000.00), new Employee("Sam", 85000.00)};

List<Employee> inputList = Arrays.asList(employees);
//apply 5% salary increase to the given input data
List<Employee> newMutatedList = inputList.stream()
                                .map(x -> new Employee(x.getName(), x.getSalary() * 1.05)) // returns a new list
                                        .collect(Collectors.toList());

System.out.println(inputList); //[Employee [name=John, salary=45000.0], Employee [name=Peter, salary=65000.0], Employee [name=Sam
System.out.println(newMutatedList); //[Employee [name=John, salary=47250.0], Employee [name=Peter, salary=68250.0], Employee [nam

```

The above code might seem inefficient, but there are techniques like lazy evaluation to optimize this practice of duplicating data. The real benefit of immutabil problems that require concurrency (aka parallelism) as shown below.

```
Employee[] employees = { new Employee("John", 45000), new Employee("Peter", 65000.00), new Employee("Sam", 85000.00)};

List<Employee> inputList = Arrays.asList(employees);

List<Employee> newMutatedList = inputList.stream().parallel()
                                .map(x -> new Employee(x.getName(), x.getSalary() * 1.05)) // returns a new list
                                        .collect(Collectors.toList());

System.out.println(inputList); //[Employee [name=John, salary=45000.0], Employee [name=Peter, salary=65000.0], Employee [name=Sam
System.out.println(newMutatedList); //[Employee [name=John, salary=47250.0], Employee [name=Peter, salary=68250.0], Employee [nam

```

A pragmatic FP as opposed to a purist FP can tolerate functions with side-effects, but avoid them as much as possible. For example, "forEach" function is a func loop. It should be used only to produce side-effects for each item. For example, issue a RESTful request, serialize an item, create new elements in a DOM, etc. T code can be coded with side-effects as shown below:

```
Employee[] employees = { new Employee("John", 45000), new Employee("Peter", 65000.00), new Employee("Sam", 85000.00)};

```

```
List<Employee> inputList = Arrays.asList(employees);

inputList.stream().parallel()
            .forEach(x -> x.setSalary(x.getSalary() * 1.05));

//original list is mutated
System.out.println(inputList); [Employee [name=John, salary=47250.0], Employee [name=Peter, salary=68250.0], Employee [name=Sam,
```

Document any side effects and isolate them to specific modules.

Q5. Where does functional programming shine?
A5.**Example:** Here is an example written functional programming to extract odd numbers from a given list of numbers and then double each odd number and

Functional programming using the Java 8 lambda expressions.

Functional programming with good set of libraries can cut down lot of fluff and focus on just transformations. In other words, just tell what you would like to d

```

package com.java8.examples;

import java.util.Arrays;

public class NumberTest {

 public static void main(String[] args) {
  Integer[] numbers = {1,2,3,4,5,6};

  Arrays.asList(numbers)    //convert to least
        .stream()           //stream
        .filter((e) -> (e % 2 != 0))    // extract only odd numbers 1, 3, 5
        .map((e) -> (e * 2))            // double the odd numbers 2, 6, 10
        .forEach(System.out::println); // print each doubled number.
 }
}


```

Output:

```

2
6
10

```

Shining moment 1: The FP has much improved readability and maintainability because each function is designed to accomplish a specific task for given argun programming to accomplish the same will require for loops and will be more verbose.

Shining moment 2: A functional program can be easily made to run in parallel using the “fork and join” feature added in Java 7. To improve performance of th to do is use .parallelStream( ) instead of .stream( ).

```

import java.util.Arrays;

public class NumberTest {

   public static void main(String[] args) {
     Integer[] numbers = {1,2,3,4,5,6};

     Arrays.asList(numbers)
        .parallelStream( )      //use fork and join thread pool introduced in Java 7
        .filter(NumberTest::isOddNumber)
        .map(NumberTest::doubleIt)
        .peek(x -> System.out.println(Thread.currentThread().getName() + " processed " + x))
        .count(); // a terminal operation is required as the intermediate operations are lazily evaluated.
                  // if count() is omitted, nothing gets processed
   }

   private static boolean isOddNumber(int input) {
       return input % 2 != 0;
   }

   private static int doubleIt(int input) {
       return input * 2;
   }
}


```

```
ForkJoinPool.commonPool-worker-2 processed 10
ForkJoinPool.commonPool-worker-1 processed 6
ForkJoinPool.commonPool-worker-3 processed 2
```

Note: Also, the Java 8 **CompletableFuture** is enabled for functional programming to write more elegant asynchronous code in Java. Look at ForkJoinPool, Exe CompletableFuture Q&A.

Learn more about terminal vs intermediate operations with diagrams.

Shining moment 3: The code is also easier to refactor as shown below. If you want to switch the logic to double it first and then extract the odd numbers, all y **.filter** call with **.map** call.

```
Arrays.asList(numbers)
        .parallelStream()
        .map(NumberTest::doubleIt)
        .filter(NumberTest::isOddNumber)
        .forEach(System.out::println);
```

Shining moment 4: Easier testing and debugging. Because pure functions can more easily be tested in isolation, you can write test code that calls the pure fur valid edge cases, and invalid edge cases. In the above example, I was using the Java 8 library that was well tested with confidence. I will only have to provide ur functions "boolean isOddNumber(int number)" and "int doubleIt(int number)" that provide the logic.

Having said this, OO programming and functional programming can co-exist. Both have their strengths and weaknesses. So, both compliment each other.

Q5. What are the characteristics of functional programming you need to be familiar with?
A5.

1. A focus on what is to be computed rather than how to compute it.
2. Function Closure Support
3. Higher-order functions
4. Use of recursion as a mechanism for flow control
5. Referential transparency
6. No side-effects

Let's look at these one by one:

1. A focus on what is to be computed rather than how to compute it

```
Integer[] numbers = {1,2,3,4,5,6};

//focusses on what to do as opposed to how to do it
//no fluff like for loops, mutations, etc
Arrays.asList(numbers)
        .stream()
        .filter((e) -> (e % 2 != 0))
        .map((e) -> (e * 2))
        .forEach(System.out::println);
```

Extract odd numbers, and multiply each by 2, and the print the result.

2. Function closure support

In order to create closures, you need a language where the function type is a 1st class citizen, where a function can be assigned to a variable, and then passed variables like a string, int or boolean. closure is basically a snapshot of the stack at the point that the lambda function is created. Then when the function is re-e stack is restored to that state before executing the function.

The **java.util.function** package provides a number of functional interfaces like Consumer, Function, etc to define closures or you can define your own function

```
package com.java8.examples;

import java.util.function.Function;

public class ClosureTest {

 public static void main(String[] args) {

  //closure 1 that adds 5 to a given number
  Function<Integer, Integer> plus5 = (i) -> (i+5);
  //closure 2 that times by 2 a given number
  Function<Integer, Integer> times2 = (i) -> (i*2);
  //closure 3 that adds 5 and then multiply the result by 2
  Function<Integer, Integer> plus5AndThenTimes2 = plus5.andThen(times2);
  //closure 4 that times by 2 and then adds 5
  Function<Integer, Integer> times2AndThenplus5 = times2.andThen(plus5);

        //callback or execute closure
  //functions plus5, times2, etc can be passed as arguments
  System.out.println("9+5=" + execute(plus5, 9));
  System.out.println("9*2=" + execute(times2, 9));
  System.out.println("(9+5)*2=" + execute(plus5AndThenTimes2, 9));
  System.out.println("9*2+5=" + execute(times2AndThenplus5, 9));

 }

 //functions can be used as method parameters
 private static Integer execute(Function<Integer, Integer> function, Integer number){
  return function.apply(number); //execute the function
 }
}

```

Output:

9+5=14
9*2=18
(9+5)*2=28
9*2+5=23

In pre Java 8, you can use a**nonymous inner classes** to define closures. In Java 8, lambda operators like (i) -> (i+5) are used to denote anonymous functions.

Is currying possible in Java 8?

Currying (named after Haskell Curry) is the fact of evaluating function arguments one by one, producing a new function with one argument less on each step.

Java 8 still does not have first class functions, but currying is “practically” possible with verbose type signatures, which is less than ideal. Here is a very simple ex

```
package com.java8.examples;

import java.util.function.Function;

public class CurryTest {

 public static void main(String[] args) {
  Function<Integer,Function<Integer,Integer>> add = (a) -> (b) -> a + b;

  Function<Integer,Integer> addOne = add.apply(1);
  Function<Integer,Integer> addFive = add.apply(5);
  Function<Integer,Integer> addTen = add.apply(10);

  Integer result1 = addOne.apply(2); // returns 3
  Integer result2 = addFive.apply(2); // returns 7
  Integer result3 = addTen.apply(2); // returns 12

  System.out.println("result1 = "  + result1);
  System.out.println("result2 = "  + result2);
  System.out.println("result3 = "  + result3);

 }
}

```

The **output** is

result1 = 3
result2 = 7
result3 = 12

3. Higher order functions

In mathematics and computer science, a **higher-order function** (aka functional form) is a function that does at least one of the following:

1. takes one or more functions as an input — for example g(f(x)), where f and g are functions, and function g composes the function f.
2. **outputs a function** — for example, in the code above ***plus5*** and *plus2* outputs a function

```
Function<Integer, Integer> plus5 = (i) -> (i+5);
 //closure 2 that times by 2 a given number
 Function<Integer, Integer> times2 = (i) -> (i*2);

```

also

```
Function<integer integer> plus5AndThenTimes2 = plus5.andThen(times2);

```

outputs another function, where the **plus5** function takes **plus2** function as an input.

4. Use of recursion as a mechanism for flow control

Java is a stack based language that supports **reentrant (**a method can call itself) methods. This means recursion is possible in Java. Using recursion you don't n
the problems can be solved in a much simpler fashion compared to using an iterative approach with a loop.

```
package com.java8.examples;

import java.util.function.Function;
import java.util.function.IntToDoubleFunction;

public class RecursionTest {

    static class Recursive<I> {
        public I func;
    }

    static Function<Integer, Double> factorial = x -> {
        Recursive<IntToDoubleFunction> recursive = new Recursive<IntToDoubleFunction>();
        recursive.func = n -> (n == 0) ? 1 : n
                * recursive.func.applyAsDouble(n - 1);

        return recursive.func.applyAsDouble(x);
    };

    public static void main(String[] args) {

        Double result = factorial.apply(3);
        System.out.println("factorial of 3 = " + result);
    }
}

```

Recursion is used in the Lambda expression to calculate the factorial. This is not very straight-forward, and here are the key points to understand the above coc

1) In java 8, you have Consumer<T> and Function<T, R> interfaces where a **Consumer** takes an object input and returns nothing and a **Function** takes an Obj
of type object.

"factorial" is a "**Function**" that takes an **input** of type "Integer" and result of type "Double". The output needs to be a double because the factorials can get to v
and long are not appropriate as this will lead to data overflow.

2) In lambda, you can't specify a recursion as shown below as you will get a compile error of "**The method factorial(int) is undefined**"

```

static Function<Integer, Double> factorial = x -> {
    return (x == 0) ? 1.0 : x * factorial(x-1);
};

```

The lambda expression and anonymous classes capture local variables by value when they are created. Therefore, it is impossible for them to refer to themselve
variable as the value for pointing to itself does not exist yet at the time it is being created.

3) So, to overcome the above problem, let's create a generic helper class that wraps the variable of the functional interface type. The functional interface type i which converts an int to double.

4) The "applyAsDouble" method applies **this** function to the given argument.

5. Referential Transparency

Referential transparency is a term commonly used in **functional programming**, which means that given a function and an input value, you will always receive external state used in the function. In other words, a referentially transparent function is one which only depends on its input. The ***plus5*** and ***times2*** functions transparent.

A function that reads from a text file and prints the output is not referentially transparent. The external text file could change at any time so the function would

6. No side-effects

One way to do programming without side effects is to use only immutable classes. In real world, you try to minimize the mutations, but can't eliminate them. L variables that are not declared final but are never modified. This is known as "**effectively final**".
When you are using methods like **reduce** on a stream, you need to be aware of the side effects due to parallelism. For example, the following code will have nc serial mode.

```
public static void main(String[] args) {
 Integer[] numbers = {10, 6};
 Integer result = Arrays.asList(numbers).stream()
                      .reduce(20, (a,b) -> (a - b));

 System.out.println(result);
}

```

Outputs:

4

It starts with 20, and then subtracts 10 and 6. **20 – 10 – 6 = 4**; But if you run it again in parallel mode as shown below

```
public static void main(String[] args) {
 Integer[] numbers = {10, 6};
 Integer result = Arrays.asList(numbers).parallelStream()
                        .reduce(20, (a,b) -> (a - b));

 System.out.println(result);
}

```

The **output** will be:

-4

What happened here? 10 – (20 – 6) = -4;

This means the **reduce** method on a stream produce side effects when run in parallel for **non-associative** operations.

Q. What is an associative property?
A. Associative operations are operations where the order in which the operations were performed does not matter. For example.

Associative:

3 + (2 + 1) = (3+ 2) + 1
3 * (2 * 1) = (3 * 2) * 1

Non-associative:

3 – (2 – 1) != (3 – 2) – 1
(4/2)/2 != 4/(2/2)

Pure functions are functions with no side effects. In computers, **idempotence** means that applying an operation once or applying it multiple times has the sam

Idempotent operations

- Multiplying a number by zero. No matter how many times you do it, the result is still zero.
- Setting a boolean flag. No matter how many times you do it, the flag stays set.
- Deleting a row from a database with a given ID. If you try it again, the row is still gone.
- Removing an element from a collection.

In real life where you have concurrent operations going on, you may find that operations you thought were idempotent cease to be so. For example, another t of the boolean flag.

Java 8 functional programming examples

1. 7 Java FP (lambda expressions) real life examples in wrangling normal & big data

2. 19 Java 8 Functional Programming (i.e. FP) interview Q&As with examples

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. A interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdatec Java training. Join my LinkedIn group.

‹ 03: ♦ Functional interfaces and Lambda expressions Q&A

Top 6 Java 8 featu

Tags: TopX

12-03: ♥♦ Java autoboxing & unboxing benefits & caveats interview Q&As | Java-Success.com

SourceURL: https://www.java-success.com/%E2%99%A5-java-autoboxing-unboxing-benefits-caveats-interview-qa/

03: ♥♦ Java autoboxing & unboxing benefits & caveats interview Q&As

Posted on June 29, 2015 by Arulkumaran Kumaraswamipillai Posted in 11 - FAQs Free, Data types, Understanding Core Java

Q1. What do you understand by the terms “autoboxing” and “autounboxing” in Java?
A1. Java automatically converts a primitive type like “int” into corresponding wrapper object class Integer. This is known as the **autoboxing**. When it converts a wrapper object class Integer back to its primitive type “int”, it is know as “**autounboxing**“.

Example 1:

```
1  package com.autoboxing;
2
3  public class AutoBoxUnbox {
4
5      public static void main(String[] args) {
6
7          int i = 5;
8          Integer objI = i; //autoboxing takes place by invoking Integer.valueOf(i);
9
10         if(objI != null)
```

```
        {
            int result = objI + 3; //auto unboxing takes place by  "objI.intValue() + 3"
            System.out.println(result);
        }
    }
}
```

This can be applied to one of 8 primitivies in Java to convert from primitive to wrapper via autoboxing and from wrapper to primitive via autounboxing. Autoboxing and unboxing can happen anywhere where an object is expected and primitive type is available

Example 2:

```
package com.autoboxing;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class AutoBoxUnbox {

    public static void main(String[] args) {

        List<Character> characters = new ArrayList<>();
        characters.add('C'); //autoboxed to Character object and then added to the list

        Map<Long, Double> myMap = new HashMap<>();
        myMap.put(5L, 12.50); //autoboxed to Long.valueOf(5L), Double.valueOf(12.50)

        char myChar = characters.get(0); //unboxed
        System.out.println(myChar);

        double myDouble = myMap.get(5L); //unboxed
        System.out.println(myDouble);
    }

}
```

Q2. What are the benefits of autoboxing?
A2. Less code to write, and the code looks cleaner.

For example, you don't have to do as shown below:

```
list.add(Integer.valueOf(6));
```

More readable with autoboxing

```
list.add(6);
```

Q3. What are some of the pitfalls of autoboxing?
A3. It is very convenient to havae autoboxing, but it can cause issues and many beginners fall into it caveats.

1. Unnecessary Object creation due to Autoboxing

```
package com.autoboxing;

public class AutoBoxUnbox {

    public static void main(String[] args) throws InterruptedException {

        Integer sum = 0;
        for (int i = 1000; i < 500000; i++) {
            sum += i;
            Thread.sleep(100);
        }
    }
}
```

Q. How do you know unnecessary objects are being created?
A. jmap to the rescue.

Step 1: Run the above code.

Step 2: Open a DOS or Unix command prompt and run the following commands. "jps" to find the process id, and then "jmap" to print the object graph

```
C:\>jps
8148
8420 Jps
3832 JConsole
8896 AutoBoxUnbox
10300 JConsole
10948 JConsole

C:\>jmap -histo:live 8896 > mem.txt


```

Step 3: Inspect the mem.txt file

```

 num     #instances         #bytes  class name
----------------------------------------------

   7:          1318          21088  java.lang.Integer

```

after some time

```

 num     #instances         #bytes  class name
----------------------------------------------
   7:          1704          27264  java.lang.Integer

```

You can see the growing instances and bytes.

Now try the samething after fixing the code as shown below.

```
package com.autoboxing;

public class AutoBoxUnbox {

    public static void main(String[] args) throws InterruptedException {

        int sum = 0;  //FIX. change to primitive type
        for (int i = 1000; i < 500000; i++) {
            sum += i;
            Thread.sleep(100);
        }
    }
}
```

```

 num     #instances         #bytes  class name
----------------------------------------------
   8:           256           4096  java.lang.Integer


```

after some time

```

 num     #instances         #bytes  class name
----------------------------------------------
  8:           256           4096  java.lang.Integer


```

The improved code does not create unnecessary Integer objects. You may also like the detailed "javap, jps, jmap, and jvisualvm tutorial – analyzing the heap histogram"

2. GC overhead

Unnecessarily creating too many objects and then discarding them will increase the Garbage Collection overhead. This may cause performance impact due to more frequent garbage collection.

3. java.lang.NullPointerException

Especially when mixing object and primitive in equality and relational operator.

```
package com.autoboxing;

public class AutoBoxUnbox {

    public static void main(String[] args) throws InterruptedException {
       Integer i = null;

       if(i > 6) { // tries to do i.intValue(); i is null so  java.lang.NullPointerException is thrown here
           System.out.println("I am in here");
       }
    }
}
```

Conditional operators can cause **NullPointerException**.

```
package com.autoboxing;

public class AutoBoxUnbox {

    public static void main(String[] args) throws InterruptedException {
        boolean b = false;
        double d1 = 0d;
        Double d2 = null;
        Double d = b ? d1 : d2; //NullPointerException when "d2.doubleValue()" is evaluated.
    }
}

```

Since d1 is primitive, d2 is implicitly tried to auto unbox. To fix it, you need to change d1 to wrapper object type "**Double**". This way auto unboxing won't take place.

4. Overloading

Q. What will be the output of the following code?

```
package com.autoboxing;

public class AutoBoxUnbox {

    public static void main(String[] args) throws InterruptedException {
        Integer value = 0;
        new AutoBoxUnbox().eval(value);
    }

    void eval(long val) {
        System.out.println(1);
    }

     void eval(Long value) {
        System.out.println(2);
    }
}
```

A. The result is **1**, because there is no direct conversion from Integer to Long, so the "conversion" from Integer to long is used.

Q4. How will you go about debugging auto boxing and unboxing error?
A4.

1) Being aware of the potential auto boxing and unboxing caveats discussed above.

2) Configuring your IDE to pick up auto boxing and unboxing error. For example, in **eclipse**

```
Java --> Compiler --> Errors/Warnings --> "Potential programming problems" --> "Boxing and unboxing conversions"
```

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group

2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ 02: ♥♦ Java Generics in no time "? extends" & "? super" explained with a diagram

javap, jps, jmap, and jvisualvm tutorial – analyzing the heap histogram ›

Tags: Free Content, Free FAQs

10-02: ♥♦ Java Generics in no time "? extends" & "? super" explained with a diagram | Java-Success.com

SourceURL: https://www.java-success.com/java-generics-and-wildcards-extends-super-explained-with-a-diagram/

02: ♥♦ Java Generics in no time "? extends" & "? super" with a diagram

Posted on June 25, 2015 by Arulkumaran Kumaraswamipillai Posted in 11 - FAQs Free, Generics, Understanding Core Java

Generics in Java can be be a bit tricky to get your head around. Hope the explanation below enhances your understanding of generics. This complements 5 Jav examples.

Plain old List, List <Object>, and List<?>

The plain old List: is a heterogeneous mixture or a mixed bag that contains elements of all types, for example Integer, String, Pet, Dog, etc.

The List<Object>: is also a heterogeneous mixture like the plain old List, but not the same and can be more restrictive than a plain old List. It is incorrect to th a collection of any object types.

The List<?>: is a homogenous collection that represents a family of generic instantiations of List like List<String>, List<Integer>, List<Pet>, List<Dog>, etc.

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

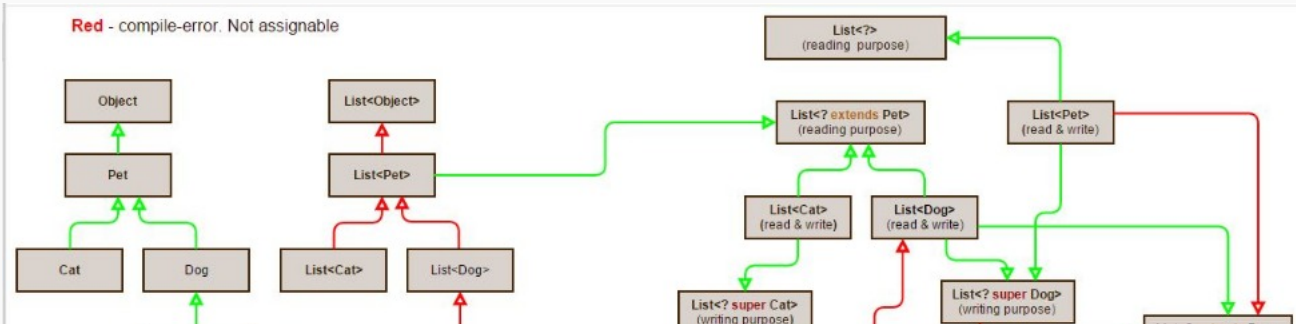
What can I assign to? What can I add to the Collection?

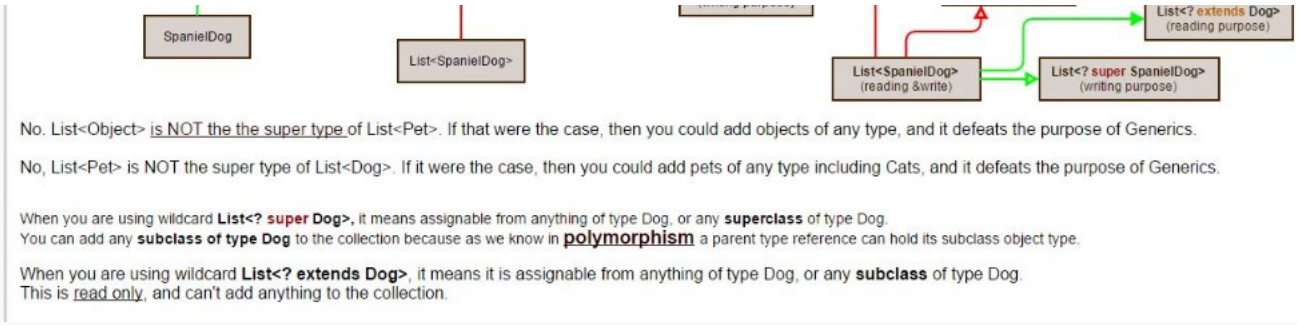
When working with Collection & Generics, you need to ask 4 important questions.

1) Can the RHS be assigned to the LHS?
2) What types of objects can I add to the collection?
3) Is it a read only or read & write collection?
4) When to use which wild card ("? extends", "? super") ?

If you do the wrong thing, you will get "compile-time" errors. Also, note that you can't use wildcards on the RHS when assigning and Java 8 supports empty "<

Understanding Generics and assign-abilities





Java Generics Overview for assignability

Note: "**?**" is a wild card meaning anything. "? extends Pet" means "**anything** that extends a Pet". Two key points to remember when using the wild card charact

Key point 1: you can add

When you are using wildcard List<? super Dog>, means **assignable** from anything of type Dog, or any superclasses of type Dog. You can add any subclasses o polymorphism where a parent type reference can hold its subclass types.

Key point 2: read only

When you are using wildcard List<? extends Dog>, means **assignable** from anything of type Dog, or any subclasses of type Dog. This is read only, and can't ac

Now let's see a code example based on the above diagram. Click on the diagram to expand.

```
1  package com.generics;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class GenericsAssignable {
7
8      public static void main(String[] args) {
9          new GenericsAssignable().create();
10     }
11
12     public void create() {
13
14         List<Object> objectsBad = new ArrayList<Pet>(); //1. COMPILE ERROR
15         List<Pet> petsBad = new ArrayList<Dog>();    //2. COMPILE ERROR
16
17         //===== "?" and "? extends" - read only =====
18         List<?> petsOk = new ArrayList<Pet>(); //read only
19         List<? extends Pet> petsOk2 = new ArrayList<Dog>(); //read only
20         List<? extends Pet> petsOk3 = new ArrayList<Pet>(); //read only
21         List<? extends Dog> petsOk4 =  new ArrayList<Dog>(); //read only
22         List<? extends Dog> petsOk5 =  new ArrayList<SpanielDog>(); //read only
23
24         List<? extends Dog> petsBad2 =  new ArrayList<Pet>(); //3. COMPILE ERROR - read only
25
26         //====  "? super" - can add objects to collection  ====
27         List<? super Dog> petsOk6 = new ArrayList<Dog>();
28         List<? super Dog> petsOk7 = new ArrayList<Pet>();
29         List<? super Dog> petsOk8 = new ArrayList<Object>();
30
31         //can add Dog or any subclass of Dog
32         petsOk6.add(new Dog());
33         petsOk6.add(new SpanielDog());//polymorphic
34
35         petsOk6.add(new Pet()); //4. COMPILE ERROR
36
37         List<? super Dog> petsBad3 = new ArrayList<SpanielDog>(); //5. COMPILE ERROR
38
39     }
40 }
```

5 Compile Errors marked in the above code reasoning

#1. List<Object> is NOT the super type of List<Pet>. If it were the case, then you could add pets of any type including Cats, and it defeats the purpose of Gene **List<Pet>**. But read only. Can't add any objects.

#2. Same as **#1**. If were not illegal, you could add a Cat to a Dog collection. Defeating the purpose of Generics.

#3. List<? extends Dog> means assignable from any objects that are of type **Dog** or **subclasses** of Dog. Pet is a **superclass** of Dog.

#4. You can only add objects of type **Dog** or **subclasses** of Dog. Subclasses are possible because of **polymorphism** , where a parent type reference can hold it can add objects of type Dog or its subclasses like SpanielDog, but NOT its super classes.

#5. List<? super Dog> means assignable from any objects that are of type **Dog** or **superclasses** of type Dog. Pet, Object, etc are valid superclasses. But Spanie

So, how to decide when to use a wild card, and when not to?

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 add objects to a data structure.

3. If you need to do both things (i.e. read and add objects), don't use any wildcard.

More code examples to validate your understanding

```
1  package com.generics;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class GenericsByExample {
7
8      public static void main(String[] args) {
9          new GenericsByExample().create();
10     }
11
12     public void create() {
13
14         // dogs only
15         List<Dog> dogs = new ArrayList<Dog>();
16         dogs.add(new Dog());
17         dogs.add(new SpanielDog()); //polymorphism
18         dogs.add(new Pet()); // compile error
19         dogs.add(new Cat()); // compile error
20
21         // cats only
22         List<Cat> cats = new ArrayList<Cat>();
23         cats.add(new Cat());
24         // cats.add(new Dog()); //No can't add dogs to list of cats
25
26         // any pets can be added
27         List<Pet> pets = new ArrayList<Pet>();
28         pets.add(new Dog()); //polymorphism
29         pets.add(new Cat()); //polymorphism
30         pets.add(new Pet());
31
32         // so, wrong to say List<Pet> is a super type of List<Dog>
33         // defeats the purpose of having Generics (i.e type safety)
34
35         // RHS allows Pet, Dog, Cat, SpanielDog, but read only
36         List<? extends Pet> petsOnlyForReading = new ArrayList<Dog>();
37
38         // RHS allows Dog and SpaniellDog, but read only
39         List<? extends Dog> dogsOnlyForReadingy = new ArrayList<SpanielDog>();
40
41         petsOnlyForReading.add(new Dog()); // compile error
42         dogsOnlyForReadingy.add(new Cat()); // compile error
43
44         List<Dog> dogsOnly = new ArrayList<Pet>(); // compile error, see dogsOnly 1, 2 , and 3
45
46         List<? super Dog> dogsOnly1 = new ArrayList<Dog>();
47         dogsOnly1.add(new Dog());
48         dogsOnly1.add(new SpanielDog()); // polymorphism
49         dogsOnly1.add(new Pet()); // compile error
50
51         List<? super Dog> dogsOnly2 = new ArrayList<Pet>();
52         dogsOnly2.add(new Dog());
53         dogsOnly2.add(new SpanielDog()); // polymorphism
54         dogsOnly2.add(new Pet()); // compile error
55
56         List<? super Dog> dogsOnly3 = new ArrayList<Object>();
57         dogsOnly3.add(new Dog());
58         dogsOnly3.add(new SpanielDog()); // polymorphism
59         dogsOnly3.add(new Pet()); // compile error
60         dogsOnly3.add(new Object()); // compile error
61
62         List<? super SpanielDog> spanielsOrSubclassesOnly = new ArrayList<SpanielDog>();
63         spanielsOrSubclassesOnly.add(new Dog()); // compile error
64         spanielsOrSubclassesOnly.add(new SpanielDog());
65         spanielsOrSubclassesOnly.add(new Pet()); // compile error
66     }
67 }
```

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. A and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replac Join my LinkedIn group.

‹ Remote debugging in Java with Java Debug Wire Protocol (JDWP)

03: ♥♦ Java autoboxing & unboxing ben

Tags: Free Content, Free FAQs

9-02: ♥♦ What is wrong with this code? Heap Vs Stack, Thread safety & Synchronized | Java-Success.com

SourceURL: https://www.java-success.com/02-%E2%99%A5%E2%99%A6-3-java-multithreading-basics-heap-vs-stack-thread-safety-synchronization/

02: ♥♦ What is wrong with this code? Heap Vs Stack, Thread safety & Synchronized

Posted on April 7, 2017 by  Arulkumaran Kumaraswamipillai Posted in 11 - FAQs Free, Multithreading, What is wrong with this code?

This post covers must know Java Multithreading basics – Heap Vs Stack, Thread-safety & Synchronization. When you have a multithreaded Java application, you need to code in a thread-safe manner. Java interviewers may ask you to detect thread-safety issues as discussed in "What is wrong with this code?".

1.What is wrong with the following code?

A very simple code that should print numbers from 7 to 21. But does it?

```
1
2  import java.util.concurrent.TimeUnit;
3
4  class Counter extends Thread {
5
6      //instance variable
7      Integer count = 0;
8
9      // method where the thread execution will start
10     public void run() {
11         int fixed = 6;  //local variable
12
13         for (int i = 0; i < 3; i++) {
14             System.out.println(Thread.currentThread().getName() + ": result="
15                                 + performCount(fixed));
16             try {
17                 TimeUnit.SECONDS.sleep(1);
18             } catch (InterruptedException e) {
19                 e.printStackTrace();
20             }
21         }
22     }
23
24     // let's see how to start the threads
25     public static void main(String[] args) {
26         System.out.println(Thread.currentThread().getName() + " is executing..." );
27         Counter counter = new Counter();
28
29         //5 threads
30         for (int i = 0; i < 5; i++) {
31             Thread t = new Thread(counter);
32             t.start();
33         }
34
35     }
36
37     //multiple threads can access me concurrently
38     private int performCount(int fixed) {
39         return (fixed + ++count);
40     }
```

```
}

```

2. Above code is NOT Thread-safe

If you run it multiple times, you will see that **some numbers get repeated** as shown below. You get five"15"s and three "12"s. The result will be **unpredictable** and you will get different results each time you run it.

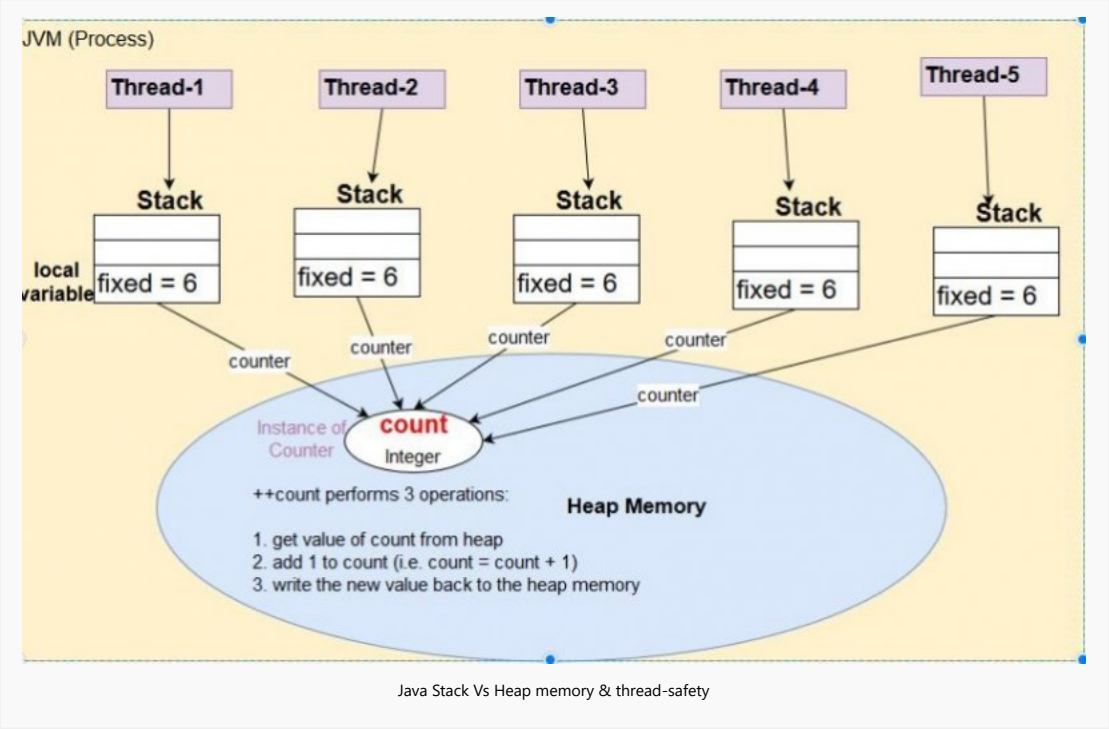
```

main is executing...
Thread-1: result=7
Thread-2: result=8
Thread-3: result=9
Thread-4: result=10
Thread-5: result=11
Thread-2: result=12
Thread-3: result=12
Thread-4: result=12
Thread-5: result=13
Thread-1: result=14
Thread-2: result=15
Thread-3: result=15
Thread-4: result=15
Thread-5: result=15
Thread-1: result=15

```

3. What is happening under the covers in terms of Heap Vs Stack memory & thread-safety

As shown below in the diagaram, the **local variable** "fixed", and the **reference** "counter" to the instance of the class "Counter" are stored in the stack. The instance of "Counter", i,e. the object itself is stored in the heap. So, it will be shared by all the threads. The "++count" operation is **not atomic** and performs 3 operations under the covers:

Step 1: get value of count from heap
Step 2: add 1 to count (i.e. count = count + 1)
Step 3: write the new value back to the heap memory

So, it is possible that 5 threads read the same value of say "count = 8" and increment them all to "9", and then when added with the fixed value of 6, resulting in five "15"s. Each time you run, you get different results. The above code is **unpredictable**.



Java Stack Vs Heap memory & thread-safety

4. How to fix the concurrency issue?

The above thread safety issue can be fixed two ways by controlling the access to the shared object "counter".

Solution 1: Synchronized i.e. a lock on the performCount() method

This will put a lock on "counter" object so that when one thread is performing the other threads have to wait for the lock.

```
import java.util.concurrent.TimeUnit;

class Counter extends Thread {

    //instance variable
    Integer count = 0;

    // method where the thread execution will start
    public void run() {
        int fixed = 6;

        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName() + ": result="
                               + performCount(fixed));
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    // let's see how to start the threads
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + " is executing..." );
        Counter counter = new Counter();

        //5 threads
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(counter);
            t.start();
        }

    }

    private synchronized int performCount(int fixed) {
        return (fixed + ++count);
    }
}


```

Output:

```
main is executing...
Thread-1: result=7
Thread-3: result=8
Thread-2: result=9
Thread-4: result=10
Thread-5: result=11
Thread-4: result=13
Thread-1: result=12
Thread-2: result=14
Thread-3: result=15
Thread-5: result=16
Thread-4: result=18
Thread-3: result=19
Thread-1: result=20
Thread-5: result=21
Thread-2: result=17

```

Why is locking of a method for thread safety is called "synchronized" and not "locked"?

When a method or block of code is locked with the reserved "synchronized" key word in Java, the memory (i.e. heap) where the shared data is kept is synchronized. This means,

When a synchronized block or method is entered after the lock has been acquired by a thread, it first reads (i.e. **synchronizes**) any changes to the locked object from the main heap memory to ensure that the thread that has the lock has the current info before start executing.

After the synchronized block has completed and the thread is ready to relinquish the lock, all the changes that were made to the object that was locked is written or flushed back (i.e. **synchronized**) to the main heap memory so that the other threads that acquire the lock next has the current info.

[**Further Reading:** 7 Things you must know about Java locks and synchronized key word]

Solution 2: AtomicInteger so that the increment operation is atomic

Solution 2: AtomicInteger so that the increment operation is atomic

The "incrementAndGet()" on AtomicInteger happens atomically so that two or more threads cannot read the same value and increment them to the same result.

```
1
2  import java.util.concurrent.TimeUnit;
3  import java.util.concurrent.atomic.AtomicInteger;
4
5  class Counter extends Thread {
6
7      //instance variable
8      AtomicInteger count = new AtomicInteger();
9
10     // method where the thread execution will start
11     public void run() {
12         int fixed = 6;
13
14         for (int i = 0; i < 3; i++) {
15             System.out.println(Thread.currentThread().getName() + ": result="
16                                 + performCount(fixed));
17             try {
18                 TimeUnit.SECONDS.sleep(1);
19             } catch (InterruptedException e) {
20                 e.printStackTrace();
21             }
22         }
23     }
24
25     // let's see how to start the threads
26     public static void main(String[] args) {
27         System.out.println(Thread.currentThread().getName() + " is executing..." );
28         Counter counter = new Counter();
29
30         //5 threads
31         for (int i = 0; i < 5; i++) {
32             Thread t = new Thread(counter);
33             t.start();
34         }
35
36     }
37
38     private  int performCount(int fixed) {
39         return (fixed + count.incrementAndGet());
40     }
41 }
42
```

Output:

```
1
2  main is executing...
3  Thread-1: result=7
4  Thread-2: result=8
5  Thread-3: result=9
6  Thread-4: result=10
7  Thread-5: result=11
8  Thread-1: result=16
9  Thread-3: result=13
10 Thread-2: result=12
11 Thread-4: result=15
12 Thread-5: result=14
13 Thread-1: result=20
14 Thread-5: result=19
15 Thread-2: result=21
16 Thread-4: result=17
17 Thread-3: result=18
18
```

What does atomicity mean?

Learn more about atomicty:

1) 10+ Atomicity, Visibility, and Ordering interview Q&As in Java multi-threading

2) 9 Java Transaction Management Interview Q&As

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government

organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ 16: Coding Scala Way – Scala concurrency styles

08: ♦ 7 basic Java Executor framework Interview Q&As with Future & CompletableFuture ›

Tags: Free Content, Free FAQs

11-12+ Java String class interview questions and answers

SourceURL: https://www.java-success.com/java-string-class-interview-questions-and-answers/

02: ♦ 10 Java String class interview questions & answers

Posted on August 10, 2014 by  Arulkumaran Kumaraswamipillai Posted in 01 - FAQs Core Java, Data types

Java Collection interview questions and answers and Java String class interview questions and answers are must know for any Java developer as these two APIs are the most frequently used in your Java application code. You can't write any decent Java application without these 2 APIs.

Q1. What will be the output of the following code snippet?

```
1 String s = " Hello ";
2 s += " World ";
3 s.trim( );
4 System.out.println(s);
```

A1. The output will be

```
1 " Hello  World "
```

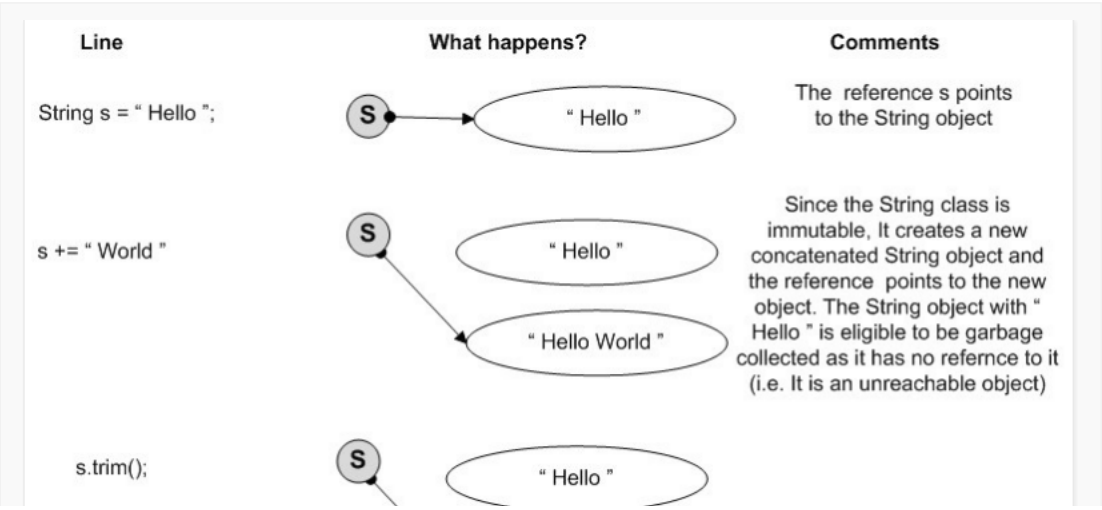
with the **leading and trailing spaces**. Some would expect a trimmed "Hello World". So, **what concepts does this question try to test?**

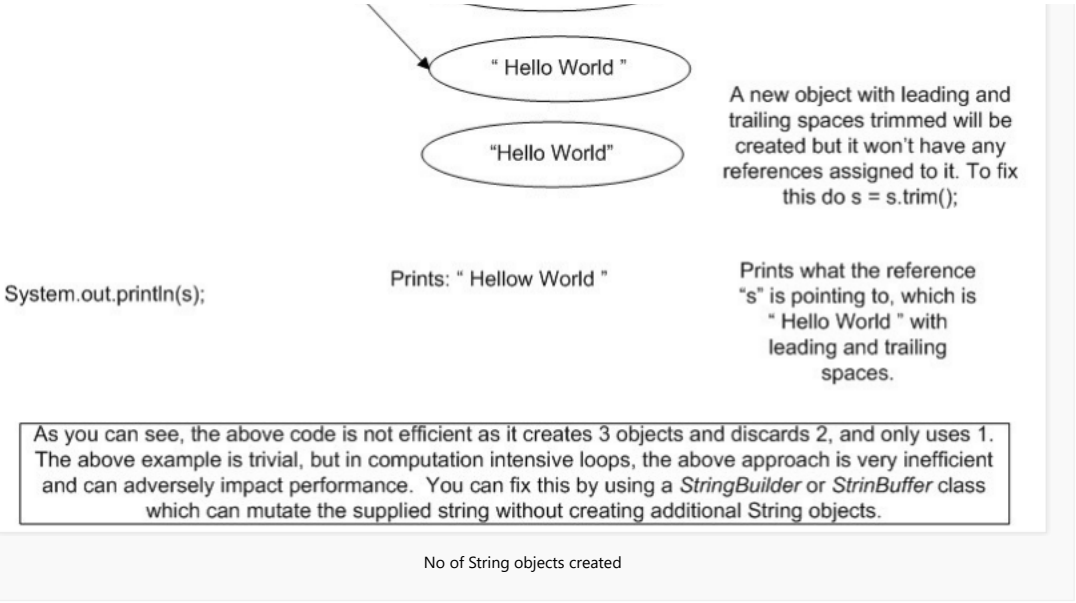
1. String objects are immutable and there is a trick in s.trim( ) line.
2. Concept of object references and unreachable objects that are eligible for garbage collection. 3 String objects are created, and 2 of them become unreachable as there are no references to them, and gets garbage collected.

What follow on questions can you expect?

1. You might get a follow on question on how many string objects are created in the above example and when will it become an unreachable object to be garbage collected.
2. You might also be asked a follow on question as to if the above code snippet is efficient.

The best way to explain this is via a self-explanatory diagram as shown below. Click on it to enlarge.





If you want the above code to output "Hello World" with leading and trailing spaces trimmed then assign the s.trim( ) to the variable "s". This will make the reference "s" to now point to the newly created trimmed String object.

The above code can be rewritten as shown below

```
StringBuilder sb = new StringBuilder(" Hello ");
sb.append(" World ");
System.out.println(sb.toString().trim( ));

```

Q2. What is the main difference between *String*, *StringBuffer*, and *StringBuilder*?
A2.

- **String** is **immutable** in Java, and this immutability gives the benefits like security and performance discussed above.
- **StringBuffer** is **mutable**, hence you can add strings to it, and when required, convert to an immutable String with the toString( ) method.
- **StringBuilder** is very similar to a *StringBuffer*, but *StringBuffer* has one disadvantage in terms of performance as all of its public methods are synchronized for thread-safety. *StringBuilder* in Java is a copy of *StringBuffer* but without synchronization to be used in local variables which are inherently thread-safe. So, if thread-safety is required, use StringBuffer, otherwise use StringBuilder.

Q3. Can you write a method that reverses a given String?
A3. A popular Java interview coding question.

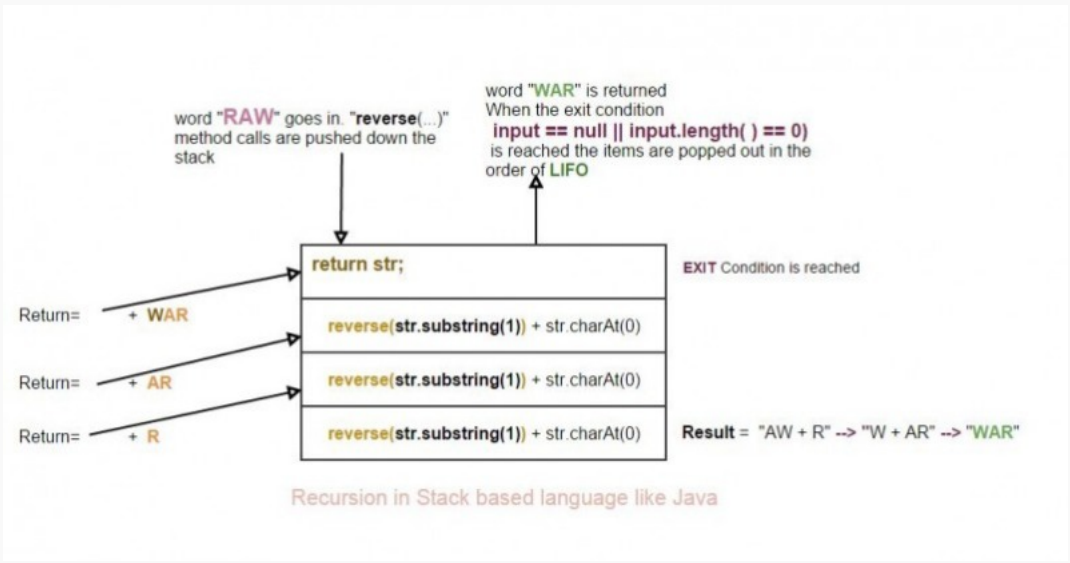
Example 1: It is always a best practice to reuse the API methods as shown below with the *StringBuilder(input).reverse( )* method as it is fast, efficient (uses bit wise operations) and knows how to handle Unicode surrogate pairs, which most other solutions ignore. The code shown below handles null and empty strings, and a StringBuilder is used as opposed to a thread-safe StringBuffer, as the StringBuilder is locally defined, and local variables are implicitly thread-safe.

```

public static String reverse(String input) {

    if(input == null || input.length( ) == 0){
        return input;
    }

 return new StringBuilder(input).reverse( ).toString( );

}

```

Example 2: Some interviewers might probe you to write other lesser elegant code using either recursion or iterative swapping. Some developers find it very difficult to handle recursion, especially to work out the termination condition. All recursive methods need to have a condition to terminate the recursion.
Recursive solution.

```
public String reverse(String str) {
    // exit or termination condition
```

```
    if ((null == str) || (str.length( )  <= 1)) {
        return str;
    }

    // put the first character (i.e. charAt(0)) to the end. String indices are 0 based.
    // and recurse with 2nd character (i.e. substring(1)) onwards
    return reverse(str.substring(1)) + str.charAt(0);
}
```



Java Recursion – String example

Step 1: reverse("RAW")

Step 2: reverse(AW) + "R"
[**Note:** charAt[0] = "R", and str.substring(1) = "AW"]

Step 3: reverse(W) + "A" + "R"
[**Note:** charAt[0] = "A", and str.substring(1) = "W"]

Step 4: return "W" + "A" + "R"
[**Exit condition is reached** when "str.length( ) <=1"]

outputs: "WAR"

Example 3: Iterative solution.

```
public String reverse(String str) {
    // validate
    if ((null == str) || (str.length( )  <= 1)) {
        return str;
    }

    char[ ] chars = str.toCharArray( );
    int rhsIdx = chars.length - 1;

    //iteratively swap until exit condition lhsIdx < rhsIdx is reached
    for (int lhsIdx = 0; lhsIdx < rhsIdx; lhsIdx++) {
        char temp = chars[lhsIdx];
        chars[lhsIdx] = chars[rhsIdx];
        chars[rhsIdx--] = temp;
    }

    return new String(chars);
}
```

Q4. Can you remember a design pattern discussed in this post?
A4. **Flyweight design pattern**. The flyweight design pattern is a structural pattern used to improve **memory usage** (i.e. due to fewer objects and object reuse) and **performance** (i.e. due to shorter and less frequent garbage collections).

Q5. Can you give some examples of the usage of the flyweight design pattern in Java?
A5.

Example 1: As discussed above, 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";

//only 1 String object is created. Both  author and authorCopy point to that
if(author == authorCopy) {
        System.out.println("referencing the same object");
}

```

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

```
public class FlyWeightWrapper {

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

    //only one object is created
    if (value1 == value2) {
        System.out.println("referencing the same object");
    }

 }

}

```

If you use new Integer(5), a new object will be created every time.

Both the above examples will print "**referencing the same object**".

Q6. What is a static factory method, and when will you use it?
A6. The factory method pattern is a way to encapsulate object creation. It has the benefits like

1. Factory can choose what to return from many subclasses or implementations of an interface. This allows the caller to specify the behavior desired via parameters, without having to know or understand a potentially complex class hierarchy. The lesser a caller knows about a callee's internal details, the more **loosely coupled** a callee is from the caller.

2. The factory can apply the fly weight design pattern to **cache objects** and return cached objects instead of creating a new object every time. In other words, objects can be pooled and reused. This is the reason why you should favor using *Integer.valuOf(6)* as opposed to *new Integer(6)*.

3. The factory methods have **more meaningful names** than the constructors. For example, *getInstance( )*, *valueOf( )*, *getConnection( )*, *deepCopy( )*, etc.

```
public static List<Car> deepCopy(List<Car> listCars) {
     List<Car> copiedList = new ArrayList<Car>(10);
     for (Car car : listCars) {                          //JDK 1.5 for each loop
         Car carCopied = new Car( );                 //instantiate a new Car object
         carCopied.setColor((car.getColor( )));
         copiedList.add(carCopied);
    }
    return copiedList;
}

```

Q7. How will you split the following string of text into individual vehicle types?

"Car,Jeep, Wagon Scooter Truck, Van"

A7. Regular expressions to the rescue.

```
public class String3 {
    public static void main(String[ ] args) {

        String pattern = "[,\\s]+";  //regex pattern - a comma or white space repeated 1 or more times

        String vehicles = "Car,Jeep, Wagon   Scooter        Truck, Van";
        String[ ] result = vehicles.split(pattern);
        for (String vehicle : result) {
            System.out.println("Vehicle = \"" + vehicle + "\"");
        }
    }
}

```

Q8. What are the different ways to concatenate strings? and which approach is most efficient?
A8.

Plus ("+") operator:

```
String s1 = "John" + "Davies";

```

Using a **StringBuilder** or **StringBuffer** class.

```
StringBuilder sb = new StringBuilder("John");
sb.append("Davies");
```

Using the **concat(...)** method.

```
"John".concat("Davies");
```

The efficiency depends on what you are concatenating and how you are concatenating it.

Concatenating constants: Plus operator is more efficient than the other two as the JVM optimizes constants.

```
String s1 = "John" + "Davies";
```

Concatenating String variables: Any one of the three methods should do the job.

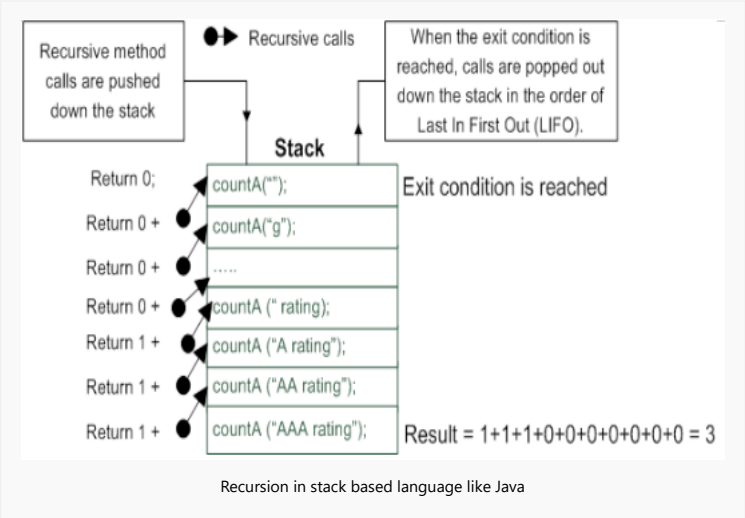
```
String s1 = s2 + s3 + s4;
String s1 =  "name=";
s1 += name;
```

Concatenating in a for/while loop: ***StringBuilder*** or ***StringBuffer*** is the most efficient. Avoid using plus operator as it is the worst offender.

```
StringBuilder sb = new StringBuilder(250);
for( int i=0; i<SIZE; i++ ) {
      sb.append("Item:" + i);
}
```

Prefer StringBuilder to StringBuffer unless multiple threads can have access to it.

Q9. Java being a stack based language, allows you to make recursive method calls. Can you write a recursion based solution to count the number of A's in string "AAA rating"?
A9. A function is recursive if it calls itself. Given enough stack space, recursive method calls are perfectly valid in Java though it is tough to debug. Recursive functions are useful in removing iterations from many sorts of algorithms.



Recursion in stack based language like Java

```
public class RecursiveCall {

    public int countA(String input) {

        // exit condition - recursive calls must have an exit condition
        if (input == null || input.length( ) == 0) {
            return 0;
        }
```

```
9
10          int count = 0;
11
12          //check first character of the input
13          if (input.substring(0, 1).equals("A")) {
14              count = 1;
15          }
16
17          //recursive call to evaluate rest of the input
18          //(i.e.  2nd character onwards)
19          return count + countA(input.substring(1));
20      }
21
22      public static void main(String[ ] args) {
23          System.out.println(new RecursiveCall( ).countA("AAA rating"));     // 3
24      }
25  }
26
```

Recursion might not be the efficient way to code, but recursive functions are shorter, simpler, and easier to read and understand. Recursive functions are very handy in working with tree structures and avoiding unsightly nested for loops.

Bonus Java String Q&A

Q10. How do you stream a string class in Java 8? ★ 🕴
A10. **chars()** method.

```
1 public static void main(String[] args) {
2     "cactus".chars().forEach(c -> System.out.println((char)c));
3  }
```

Q. Does parallel processing as shown below preserve the order?

```
1 public static void main(String[] args) {
2     "cactus".chars().parallel().forEach(c -> System.out.println((char)c));
3  }
```

A. No.

You may also like

17 Java overview interview Q&A | Web Services interview Q&A | Java EE Overview interview Q&A | Multithreading scenarios in Java applications interview Q&A

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

♥♦ 6 Java Modifiers every interviewer likes to quiz ›

Tags: Core Java FAQs, Novice FAQs

Core Java Interview Questions-Part3

Notebook:	Core Java		
Created:	23-06-2018 16:54	**Updated:**	23-06-2018 17:25
URL:	https://www.java-success.com/module-2/7-java-primitives-objects-interview-qas/		

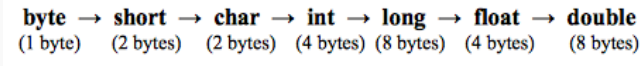
7 Java primitives & objects memory consumption Q&As | Java-Success.com

SourceURL: https://www.java-success.com/module-2/7-java-primitives-objects-interview-qas/

7 Java primitives & objects memory consumption Q&As

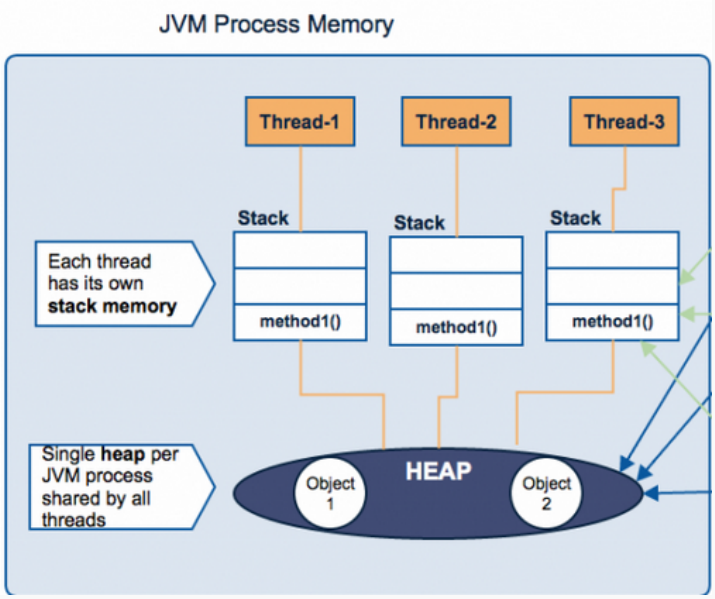
Posted on July 1, 2017 by  Arulkumaran Kumaraswamipillai

Q1. How much memory space does a primitive type **int** occupy in Java?
A1. 4 bytes.



Java primitive data types

Q2. Java objects get stored in the heap memory space, but how about the primitive variables?
A2. It depends.



Stack vs Heap

1) Primitives defined **locally** would be on the **stack**.

```
1
2  public class Primitive
3  {
4      public static void main(String[] args)
5      {
6          int number = 1; // This is on the stack.
7      }
8  }
9
```

2) If a primitive is defined as part of an **instance of an object**, that primitive would be on the **heap**

```
1
2  class MyWrapper {
3
4      int number ; // this will be on the heap.
5
6      public MyWrapper(int number) {
7          this.number = number;
8      }
9  }
10
```

Q3. How much space does java.lang.Integer object occupy?
A3. Java objects need to store **1) object metadata** information and then the **2) data** .

java.lang.Integer object metadata on a 32bit JVM

1) Class information: 32 bits = **4 bytes**.
2) Flags: array or not, hashCode, etc : 32 bits = **4 bytes**.
3) Lock information: synchronization 32 bits = **4 bytes**.

java.lang.Integer data

int is = 32 bits = **4 bytes**.

Total memory occupied on a 32bit JVM is = 128 bits = **16 bytes**. This is 4 times the space occupied by a primitive.

java.lang.Integer object metadata on a 64bit JVM

1) Class information: 64 bits = **8 bytes**.
2) Flags: array or not, hashCode, etc : 64 bits = **8 bytes**.
3) Lock information: synchronization 64 bits = **8 bytes**.

java.lang.Integer data

int is 32 bits = **4 bytes**.

Total memory occupied on a 64bit JVM is = 224 bits = **28 bytes**.

So, if you take an application that was running on a 32 bit JVM and port it to a 64 bit JVM, it is going to require more memory.

Q4. How much space does java.lang.Integer[] array with 1 element occupy on a 32 bit JVM?
A4. Very similar to an Integer object, but requires an extra object data called "**size**"

java.lang.Integer[] object metadata on a 32bit JVM

1) Class information: 32 bits = **4 bytes**.
2) Flags: array or not, hashCode, etc: 32 bits = **4 bytes**.
3) Lock information: synchronization: 32 bits = **4 bytes**.
4) Size of the array 32 bits = **4 bytes**.

Then depending on how many elements are in an array: 32 bits or 4 bytes per data.

An array of size 1 will consume = 160 bits = 20 bytes.

Q5. Can you arrange the following Collection data types in terms if their memory overheads in ascending order?

1) ArrayList
2) LinkedList
3) HashSet
4) HashMap

A5. ArrayList –> LinkedList –> HashMap –> HashSet.

1) ArrayList is the least in terms of memory overhead as it is backed by a data structure of type array. A default size of an array list is 10 entries.

2) A HashSet has the highest memory overhead and it takes more memory than a HashMap because internally a HashSet uses a HashMap to store data. So, it needs space for the HashMap + additional meta data space to wrap around a HashMap.

3) A HashMap by default creates a backing data structure (i.e. an array) with a capacity for **16 objects** regardless of you add all 16 objects or not. Hence it consumes more memory than a LinkedList as a linked list only occupies space for whatever data that is added.

4) A HashMap uses additional object entries for **key**, **value**, **next** reference (i.e. for iterating), and an **int to store hash value** whereas a LinkedList uses only **next** & **previous** references in addition the data themselves.

So, when using a collection type in Java, it is always a **trade off between memory usage & functionality**. Some collection types even though consume more memory, but functionally more efficient. For example, a HashMap lookup of elements on average is O(1). This is explained Understanding "Big O" Notation in Java with examples.

Q6. How will you go about evaluating sizeof an object in Java?
A6. Java does not have a sizeof operator like C++ does. Java uses automatic memory management known as the Garbage Collection, hence it is not that important to evaluate size of various objects. But, for the learning purpose, I have used "jvisualvm", which is a very handy & free profiling tool that gets shipped with the JDK. Step by step instructions are provided: jvisualvm to sample Java heap memory

Q7. What are the best practices with regard to conserving memory when using Java Collections?
A7. **1)** Set the initial capacity of the collection appropriately so that the space is not unnecessarily wasted. Most collections double their capacity when the current capacity is reached.

For example, to store 130 elements in a Map, initialize it to say 150, rather than using the default capacity of 16, which has to grow like 16 –> 32 –> 64 –> 128 –> 256, where 256, where 256 is a lot greater than 130.

2) Lazily initialize your collections. This means initialize it just before adding elements.

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ 8 Java data types interview Q&As

12 Java String class Interview Q&As ›

3 Abstract classes Vs interfaces interview Q&As | Java-Success.com

SourceURL: https://www.java-success.com/module-4/3-abstract-classes-vs-interfaces-interview-qas/

3 Abstract classes Vs interfaces interview Q&As

Posted on July 7, 2017 by  Arulkumaran Kumaraswamipillai

Q1. When to use an abstract class over an interface?
Q2. What is a diamond problem?
Q3. Does Java support multiple inheritance?

In design, you want the base class to present only an interface (or a contract) for its derived classes. This means, you don't want anyone to actually instantiate an object of the base class. You only want to **up cast** to it (implicit up casting, which gives you polymorphic behavior), so that its interface can be used. This is accomplished by making that class **abstract** using the abstract keyword. If anyone tries to make an object of an abstract class, the compiler prevents it with a compile-time error.

Q. What is the purpose of an abstract class?
A. The purpose of abstract classes is to function as base classes which can be extended by sub classes to create a full implementation. The base class is used for code reuse and gives **polymorphism** by up casting to it.

Q. Should it have at least one abstract member?
A. It is subjective, but it is preferred. If your intention is to prevent a class from being instantiated, then the best way to handle this is with a private or protected constructor, and not by marking it abstract.

Q.Can you give an example of an abstract class where it is used prevalently?
A. **Template Method design pattern** is a good example of using an abstract class and this pattern is used very prevalently in application frameworks. The **Template Method** design pattern is about providing partial implementations in the abstract base classes, and the subclasses can complete when extending the Template Method base class(es). Here is an example

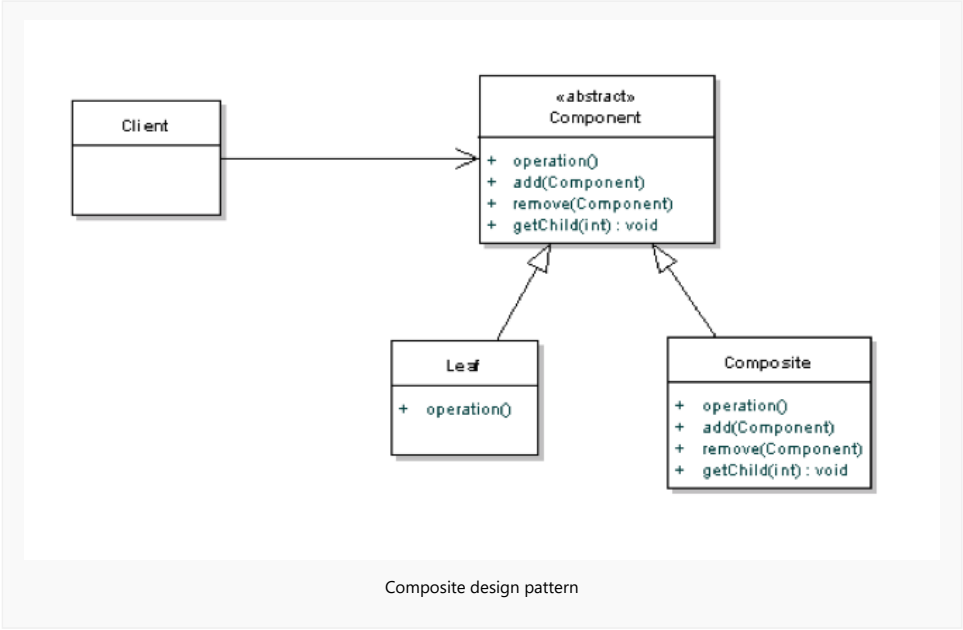
```
1  //cannot be instantiated
2  public abstract class BaseTemplate {
3
4   public void process() {
5      fillHead();
6      //some default logic
7      fillBody();
8      //some default logic
9      fillFooter();
10    }
11
12    //to be overridden by sub class
13    public abstract void fillBody();
14
15    //template method. Sub classes can override or reuse this default implementation
16    public void fillHead() {
17        System.out.println("Simple header");
18    }
19
20     //template method. Sub classes can override or reuse this default implementation
21     public void fillFooter() {
22        System.out.println("Simple footer");
23     }
24
25     //more template methods can be defined here
26 }
27
```

```
1  public class InvoiceLetterProcessor extends BaseTemplate {
2
3   @Override
4   public void fillBody() {
5    System.out.println("Invoice body" );
6   }
7
8   // template method
9   public void fillHead() {
10   System.out.println("Invoice header");
```

```
}
}
```

```
public class InvoiceTestMain {

    public static void main(String[] args) {
        //subclass is up cast to base class -- polymorphism
        BaseTemplate template = new InvoiceLetterProcessor();
        template.process();
    }

}
```

Another design pattern that makes use of abstract classes is the **composite design pattern**. A **node** or a **component** is the parent or base class and derivatives can either be leaves (singular), or collections of other nodes, which in turn can contain leaves or collection-nodes. When an operation is performed on the parent, that operation is recursively passed down the hierarchy. An interface can be used instead of an abstract class, but an abstract class can provide some default behavior for the *add()*, *remove()* and *getChild()* methods.



Composite design pattern

Java 8: If you are using Java 8 or later versions of Java, you can have **default methods** and static helper methods in Java 8 interface definition.

Q. What is the purpose of an **interface**?
A. The interface keyword takes this concept of an abstract class a step further where till Java 7, you can't have implementations in an interface, but from **Java 8** onwards, the concept of **functional interfaces** was introduced where you can implement **default methods** and **static helper methods**.

Q. What are the differences between abstract classes and interfaces?
A.

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 code. All methods are abstract.
Can only extend one abstract class.	A class can implement any number of interfaces.

```
1
2 public ClassB extends  ClassA {}
3
```

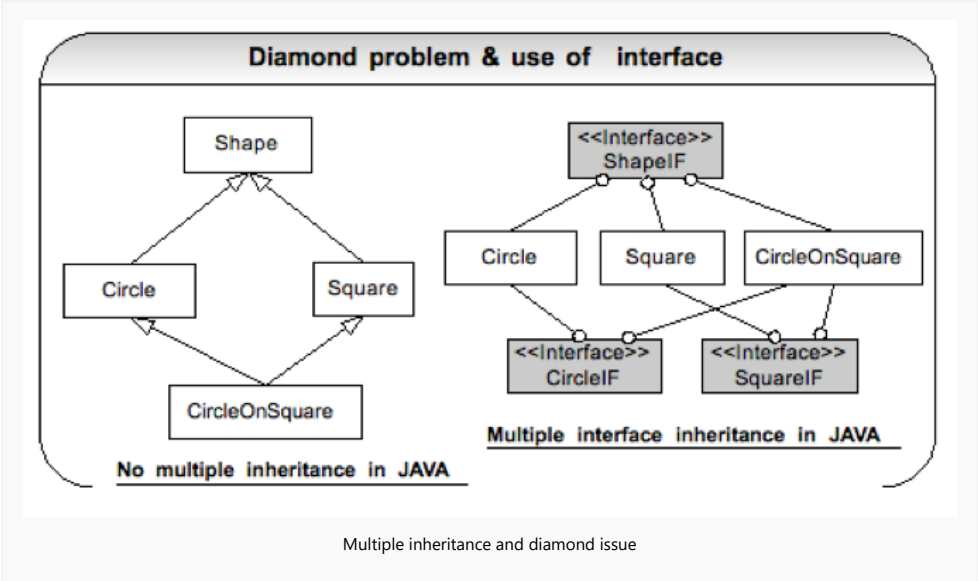
```
1
2 public ClassB implements InterfaceA, InterfaceB{}
3
```

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.

Q. What is the diamond problem?
A. The diamond problem is that in the **multiple-inheritance diagram** on the left hand side below where *CircleOnSquare* inherits **state** and **behavior** from both *Circle* and *Square*. So, when we instantiate an object of class *CircleOnSquare*, any calls to method definitions in class *Shape* will be ambiguous – because it's not sure whether to call the version of the method derived from class *Circle* or class *Square*.



Multiple inheritance and diamond issue

Java does not have **multiple inheritance**, as you can only extend one class. This means that Java is not at risk of suffering the consequences of the diamond problem. But, Java does support **multiple interface inheritance** as shown above on the right hand side of the diagram as a Java class can implement multiple interfaces. Hang on!!!!! From **Java 8** onwards, you can have functional interfaces where you can implement default and static methods. This means, Java supports **multiple behavior inheritance**, but not full multiple inheritance as state cannot be inherited because you can't define instance variables in an interface.

Before Java 8 interface example:

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

After Java 8 interface example with default and static helper methods:

```java
@FunctionalInterface
public interface Operation<Integer>  {

   Integer operate(Integer operand);

   default Operation<Integer> add(Integer o){
       return (o1) -> operate(o1) +   o;
   }

    default Operation<Integer> multiply(Integer o){
        return (o1) -> operate(o1) * o;
    }

    //ads 5 to a given number
   static Integer plus5(Integer input) {
      return input + 5 ;
  }

}

```

Now, if we have a class ***Calculator***, which implements two functional interfaces *Operation* and *Sum*, and both has a default method *add(Integer o)*. How does it know which default method to use? The one from *Operation* or the one from *Sum*. The *Calculator* class must solve the **multiple behavior inheritance ambiguity** by throwing a compile-time error

java: class Impl inherits unrelated defaults for *add(Integer o)* from types *Operation* and *Sum*.

In order to fix this class, the ***Calculator*** class needs to implement the *add(Integer o)* method to resolve the ambiguity.

Q. When to use an abstract class over an interface?
A.

Abstract classes

With the advent of Default Methods in Java 8, it seems that Interfaces and abstract Classes are same as you can implement behavior in both. However, they are still different concepts as

- An Abstract Class can **define constructor**(s).
- Abstract classes are **more structured and can have a state associated with them**. While in contrast, default method can be implemented only in the terms of invoking other Interface methods, with no reference to a particular implementation's state.

Hence, both are used for different purposes and choosing between two really depends on the scenario. Abstract methods are good for implementing template method and composite design patterns with state and behavior.

Interfaces

If you need to change your design frequently, you should prefer using interfaces to abstract classes. Coding to an interface reduces coupling and interface inheritance can achieve code reuse with the help of object composition. For example: The Spring frameworks' dependency injection promotes code to an interface principle. Another justification for using interfaces is that they solve the 'diamond problem' of traditional multiple inheritance. Java does not support multiple inheritance, but supports multiple behavior inheritance.

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

Q. What are the different ways to get **code reuse**?
A. There are 3 approaches

1. **Implementation inheritance** with **abstract** classes.
2. **Composition**.
3. **Delegation** to a helper class.

Implementation inheritance gives you **polymorphism** in addition to code reuse. You can up cast your child class to your abstract base class. If you are using composition for code reuse, then you need to use ***behavior inheritance*** with interfaces to get **polymorphism (**i.e. code to interface**)**.

The **GoF** design patterns favor **composition for code reuse** with **polymorphism** with **interfaces** over **implementation inheritance** with **abstract classes for code reuse** and **poylmorphism**.

Why?

1. Code reuse via composition happens at run time whereas code reuse via inheritance happens at compile time. Hence, composition is more flexible and less fragile.

2. It is easy to misuse inheritance when there is really no "is a" relationship, hence breaking the Lithkov's Substitution Principle (**LSP**). For example, a square is not a rectangle. Overuse of implementation inheritance (uses the "extends" key word) can break all the sub classes, if the super class 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 with composition, which gives you code reuse.

3. Composition offers better testability than Inheritance. Composition is easier to test because inheritance tends to create very coupled classes that are more fragile (i.e. fragile parent class) and harder to test in isolation. The IoC containers like Spring, make testing even easier through injecting the composed objects via constructor or setter injection.

Why favor composition over inheritance? detailed discussion.

Q. What is the difference between the default methods introduced in Java 8 and regular methods?
A. In summary, **default methods** are like regular methods, but

- the default methods come with the default modifier.
- the default methods can only access its arguments as Interfaces do not have any state.

Regular methods in Classes can use and modify method arguments as well as the variables (i.e state) of their *Class*.

The default (aka defender) methods allow you to add new methods to interfaces without breaking the existing implementations of your interfaces. This provides an added flexibility by allowing interfaces to provide default implementations in situations where concrete classes fail to provide implementations for a methods.

4 Java autoboxing & unboxing interview Q&As | Java-Success.com

SourceURL: https://www.java-success.com/module-2/4-java-autoboxing-unboxing-interview-qas/

4 Java autoboxing & unboxing interview Q&As

Posted on July 1, 2017 by  Arulkumaran Kumaraswamipillai

Q1. What do you understand by the terms "autoboxing" and "autounboxing" in Java?
A1. Java automatically converts a primitive type like "int" into corresponding wrapper object class Integer. This is known as the **autoboxing**. When it converts a wrapper object class Integer back to its primitive type "int", it is know as "**autounboxing**".

Example 1:

```
1  package com.autoboxing;
2
3  public class AutoBoxUnbox {
4
5      public static void main(String[] args) {
6
7          int i = 5;
8          Integer objI = i; //autoboxing takes place by invoking Integer.valueOf(i);
9
10         if(objI != null)
11         {
12             int result = objI + 3; //auto unboxing takes place by  "objI.intValue() + 3"
13             System.out.println(result);
14         }
15     }
16 }
```

This can be applied to one of 8 primitivies in Java to convert from primitive to wrapper via autoboxing and from wrapper to primitive via autounboxing. Autoboxing and unboxing can happen anywhere where an object is expected and primitive type is available

Example 2:

```
1  package com.autoboxing;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5  import java.util.List;
6  import java.util.Map;
7
8  public class AutoBoxUnbox {
9
10     public static void main(String[] args) {
11
12         List<Character> characters = new ArrayList<>();
13         characters.add('C'); //autoboxed to Character object and then added to the list
14
15         Map<Long, Double> myMap = new HashMap<>();
16         myMap.put(5L, 12.50); //autoboxed to Long.valueOf(5L), Double.valueOf(12.50)
17
18         char myChar = characters.get(0); //unboxed
19         System.out.println(myChar);
20
21         double myDouble = myMap.get(5L); //unboxed
22         System.out.println(myDouble);
23     }
24
25 }
```

Q2. What are the benefits of autoboxing?
A2. Less code to write, and the code looks cleaner.

For example, you don't have to do as shown below:

```
1  list.add(Integer.valueOf(6));
```

More readable with autoboxing

```
1  list.add(6)
```

```
list.add(6);
```

Q3. What are some of the pitfalls of autoboxing?
A3. It is very convenient to havae autoboxing, but it can cause issues and many beginners fall into it caveats.

1. Unnecessary Object creation due to Autoboxing

```
package com.autoboxing;

public class AutoBoxUnbox {

    public static void main(String[] args) throws InterruptedException {

        Integer sum = 0;
        for (int i = 1000; i < 500000; i++) {
            sum += i;
            Thread.sleep(100);
        }
    }
}
```

Q. How do you know unnecessary objects are being created?
A. jmap to the rescue.

Step 1: Run the above code.

Step 2: Open a DOS or Unix command prompt and run the following commands. “jps” to find the process id, and then “jmap” to print the object graph

```
C:\>jps
8148
8420 Jps
3832 JConsole
8896 AutoBoxUnbox
10300 JConsole
10948 JConsole

C:\>jmap -histo:live 8896 > mem.txt


```

Step 3: Inspect the mem.txt file

```

 num     #instances         #bytes  class name
----------------------------------------------

   7:          1318          21088  java.lang.Integer

```

after some time

```

 num     #instances         #bytes  class name
----------------------------------------------
   7:          1704          27264  java.lang.Integer

```

You can see the growing instances and bytes.

Now try the samething after fixing the code as shown below.

```
package com.autoboxing;

public class AutoBoxUnbox {

    public static void main(String[] args) throws InterruptedException {

        int sum = 0;  //FIX. change to primitive type
        for (int i = 1000; i < 500000; i++) {
```

```
            for (int i = 1000; i < 500000; i++) {
                sum += i;
                Thread.sleep(100);
            }
        }
}
```

```
 num     #instances         #bytes  class name
----------------------------------------------
   8:           256           4096  java.lang.Integer
```

after some time

```
 num     #instances         #bytes  class name
----------------------------------------------
  8:           256           4096  java.lang.Integer
```

The improved code does not create unnecessary Integer objects. You may also like the detailed "javap, jps, jmap, and jvisualvm tutorial – analyzing the heap histogram"

2. GC overhead

Unnecessarily creating too many objects and then discarding them will increase the Garbage Collection overhead. This may cause performance impact due to more frequent garbage collection.

3. java.lang.NullPointerException

Especially when mixing object and primitive in equality and relational operator.

```
package com.autoboxing;

public class AutoBoxUnbox {

    public static void main(String[] args) throws InterruptedException {
       Integer i = null;

       if(i > 6) { // tries to do i.intValue(); i is null so  java.lang.NullPointerException is thrown here
           System.out.println("I am in here");
       }
    }
}
```

Conditional operators can cause **NullPointerException**.

```
package com.autoboxing;

public class AutoBoxUnbox {

    public static void main(String[] args) throws InterruptedException {
        boolean b = false;
        double d1 = 0d;
        Double d2 = null;
        Double d = b ? d1 : d2; //NullPointerException when "d2.doubleValue()" is evaluated.
    }
}

```

Since d1 is primitive, d2 is implicitly tried to auto unbox. To fix it, you need to change d1 to wrapper object type "**Double**". This way auto unboxing won't take place.

4. Overloading

Q. What will be the output of the following code?

```java
package com.autoboxing;

public class AutoBoxUnbox {

    public static void main(String[] args) throws InterruptedException {
        Integer value = 0;
        new AutoBoxUnbox().eval(value);
    }

    void eval(long val) {
        System.out.println(1);
    }

     void eval(Long value) {
        System.out.println(2);
    }
}
```

A. The result is **1**, because there is no direct conversion from Integer to Long, so the "conversion" from Integer to long is used.

Q4. How will you go about debugging auto boxing and unboxing error?
A4.

1) Being aware of the potential auto boxing and unboxing caveats discussed above.

2) Configuring your IDE to pick up auto boxing and unboxing error. For example, in **eclipse**

```
Java --> Compiler --> Errors/Warnings --> "Potential programming problems" --> "Boxing and unboxing conversions"
```

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ 12 Java String class Interview Q&As

6 Java Modifiers every interviewer seems to like ›

12 Java classes and interfaces interview Q&As | Java-Success.com

SourceURL: https://www.java-success.com/module-4/12-java-classes-interfaces-interview-qas/

12 Java classes and interfaces interview Q&As

Posted on July 1, 2017 by  Arulkumaran Kumaraswamipillai

Q1. Which class declaration is correct if A and B are classes and C and D are interfaces?

a) class Z extends A implements C, D{}
b) class Z extends A,B implements D {}

b) class Z extends A,B implements D {}
c) class Z extends C implements A,B {}
d) class Z extends C,D implements B {}

A1. a). class Z extends A implements C, D{}

A class is a template. A class can extend only a single class (i.e. single inheritance. Java does not support multiple implementation inheritance), but can implement multiple interfaces to achieve multiple interface inheritance. An interface can also extend more than one other interfaces.

```
1
2 interface E extends C,D {  //.... }
3
```

Q2. What happens when a parent and a child class has the same variable name?
A2. 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.

Q3. What happens when the parent and the child class has the same non-static method with the same signature?
A3. 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**.

Q4. What happens when the parent and the child class has the same static method with the same signature?
A4. 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.

Q5. What is the difference between an abstract class and an interface and when should you use them?
A5. 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. You only want to upcast to it (implicit **upcasting**, which gives you polymorphic behavior), so that its interface can be used. This is accomplished by making that class abstract using the **abstract** keyword. If anyone tries to make an object of an abstract class, the compiler prevents it.

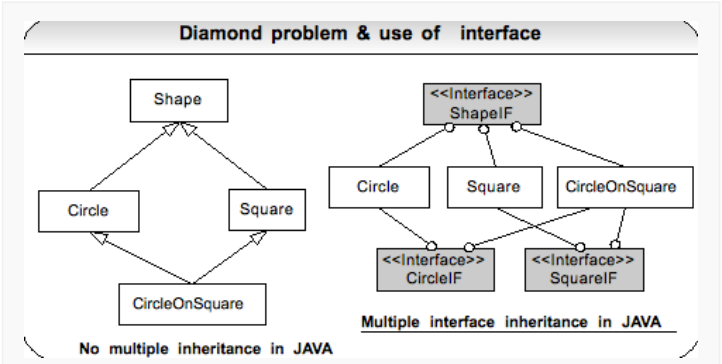
The **interface** keyword takes this concept of an abstract class a step further by preventing any method or function implementation at all. You can only declare a method or function but not provide the implementation till Java 7. The class, which is implementing the interface, should provide the actual implementation. The interface is a very useful and commonly used aspect in OO design, as it provides the separation of interface and implementation and enables you to

— Capture similarities among unrelated classes without artificially forcing a class relationship.
Declare methods that one or more classes are expected to implement.
— Reveal an object's programming interface without revealing its actual implementation.
— Model multiple interface inheritance in Java, which provides some of the benefits of full on multiple inheritances, a feature that some object-oriented languages support that allow a class to have more than one super class.

Q6 When will you use an abstract class?
A6 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.

Q7. When will you use an interface?
A7. 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.**Strategy design pattern** lets you swap new algorithms and processes into your program without altering the objects that use them.

program without altering the objects that use them.



Diamond problem and multiple inheritance in Java

Q8. What is a marker or a tag interface? Why there are some interfaces with no defined methods (i.e. marker interfaces) in Java?
A8. 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 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

Q9. What is a functional interface?
A9. **Functional interfaces** are introduced in Java 8 to allow default and static method implementations to enable functional programming (aka closures) with lambda expressions.

```
1
2  @FunctionalInterface
3  public interface Summable {
4          abstract int sum(int input1, int input2);
5  }
6
```

Q10. What does the @FunctionalInterface do, and how is it different from the @Override annotation?
A10. 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.

Q10. Does Java 8 solve the "diamond problem" discussed earlier? If yes how?
A10. Partially yes.

We know that Java does not support multiple implementation inheritance to solve the diamond problem (till Java 8). Java did only support multiple interface inheritance. That is, a class can implement multiple interfaces. By having default method implementations in interfaces, you can now have multiple behavioral inheritance in Java 8. Partially solving the diamond problem.

Q11. Does Java 8 solve the need to have a separate helper classes? For example, you have

1) *java.util. Collection* interface and a separate *java.util.Collections* helper class with static methods.
2) *java.nio.file.Path* interface and a separate *java.util.Paths* helper or utility class with static methods.

A11 Yes. In Java 8, interfaces can have static helper methods. Here is an example of a Java 8 interface with both default and static methods.

```
1
2  package com.java8.examples;
3
4  import java.util.function.BinaryOperator;
5  import java.util.function.Function;
6  import java.util.Objects;
7
8  @FunctionalInterface
9  public interface Operation<Intetger>  {
10
11    //SAM -- Single Abstract Method.
12    //identifier abstract is optional
13    Integer operate(Integer operand);
14
15     default Operation<Integer> add(Integer o){
16         r return (o1) -> operate(o1) +   o;
17     }
18
19     default Operation<Integer> multiply(Integer o){
20          return (o1) -> operate(o1) * o;
21     }
22
23     //define other default methods for divide, subtract, etc
24     default Integer getResult() {
25        return operate(0);
26     }
27
28    default void print(){
29       System.out.println("result is = " + getResult());
30     }
31
32
33    //helper -- adds 5 to a given number
34    static Integer plus5(Integer input) {
35         return input + 5 ;
36   }
37 }
38
```

Now, how to invoke the default and static methods via **Lambda expressions**:

```
1
2  package com.java8.examples;
3
4  import static java.lang.System.out;
5
6  public class OperationTest {
7
8   public static void main(String[] args) {
9
10   //plus5 is an expressive static helper method that adds 5 to a given number
11   Operation<Integer> calc = (x) -> Operation.plus5(2);
12
13   Operation complexOp = calc.add(3)
14           .multiply(4)
15           .multiply(2)
16           .multiply(2)
17           .add(4);
18
19   complexOp.print();
20
21   int result = complexOp.getResult();
22
23  }
24 }
25
```

So, if you are coding in Java 8, a separate helper class with static methods may no longer be required.

Q12. Does this mean that abstract classes may not be required as you can have default and static methods in Java 8 interfaces?
A12 No. You still can't have states in Java 8 interfaces. You can only have behavior. So, Java 8 interfaces only gives your **multiple behaviour implementation inheritance**. You still need abstract classes to maintain default states via instance or member variables.

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government

organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

Have you completed this unit? Then mark this unit as completed.

Mark as Completed

« Previous Unit | Next Unit »

‹ 10 Java initializers, constructors, regular methods and static factory methods Q&As

3 Java class loading interview Q&As to ascertain your depth of Java knowledge ›

10 Java initializers, constructors, regular methods and static factory methods Q&As | Java-Success.com

SourceURL: https://www.java-success.com/module-3/10-java-initializers-constructors-regular-methods-static-factory-methods-qas/

10 Java initializers, constructors, regular methods and static factory methods Q&As

Posted on July 1, 2017 by  Arulkumaran Kumaraswamipillai

Q1. What are "**static initializers**" or "**static blocks with no function names**" in Java?
A1. When a class is loaded, all blocks that are declared static and don't have function name (i.e. static initializers) are executed even before the constructors are executed. As the name suggests they are typically used to initialize static fields.

```
1  public class StaticInitializer {
2     public static final int A = 5;
3     public static final int B;
4     //note that since final above line cannot do --> public static final int B = null;
5
6     //Static initializer block, which is executed only once when the class is loaded.
7
8     static {
9        if(A == 5)
10          B = 10;
11       else
12         B = 5;
13    }
14
15    public StaticInitilaizer(){}  //constructor is called only after static initializer block
16 }
17
```

The following code gives an Output of A=5, B=10.

```
1 public class Test {
2     System.out.println("A =" + StaticInitilaizer.A + ", B =" + StaticInitilaizer.B);
3 }
4
```

Q2. How will you initialize an instance variable say *dueDate* to first day of next month?
A2. Like static initializers, you can use an initializer block for instance variables. Initializer blocks for instance variables look just like static initializer blocks, but without the 'static' keyword.

```
1  public class Initilization {
2
3     private Date dueDate;
```

```
    //initializer block
    {
        Calendar cal = GregorianCalendar.getInstance( );
        cal.add(Calendar.MONTH, 1);
        cal.set(Calendar.DAY_OF_MONTH, 1);
        dueDate = cal.getTime( );          //dueDate defaults to first day of next month
    }

    //...

    public static void main(String[ ] args) {
        Initilization init = new Initilization( );
        System.out.println("dueDate = " + init.dueDate);  // first day of next month
    }
}
```

Constructors Vs Regular Methods

Q3. What is the difference between constructors and other regular methods? What happens if you do not provide a constructor? Can you call one constructor from another? How do you call the superclass' constructor?
A3.

Constructors	**Regular methods**
Constructors must have the same name as the class name and cannot return a value. The constructors are called only once per creation of an object while regular methods can be called many times. E.g. for a ***Pet.class***	Regular methods can have any name and can be called any number of times. E.g. for a ***Pet.class***.

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

```
// regular method has a void or other return types.
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 Pet createPet(){}
```

Q4. What happens if you do not provide a constructor?
A4. 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)*" or "***Pet(){ }***" etc, the java compiler does not create the default constructor "***Pet( ){ }***".

Q5. Can you call one constructor from another?
A5. Yes, by using ***this( )*** syntax. E.g.

```
public Pet(int id) {
    this.id = id;                       // "this" means this object
}
public Pet (int id, String type) {
    this(id);                           // calls constructor public Pet(int id)
    this.type = type;                  // "this" means this object
}
```

Q6. How to call the superclass constructor?
A6. 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 statement in the constructor.
}

```

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, and not only do they need to remember to invoke their parent's *setUp* method, you, as a user, need to remember to invoke them after you wrote your 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 object(s).
     }
     catch (Throwable t) {}
     finally{
         super.cleanUp(); //clean up your parent class. Unlike constructors
                          // super.regularMethod() can be called at any line.
     }
}

```

Q7. Why do super(..) and this(..) calls need to be in the first line of a constructor?
A7. The parent class' constructor needs to be called before the subclass' constructor. This will ensure that if you call any methods on the parent class in your constructor, the parent class has already been set up correctly.

In cases where a parent class has a default constructor the call to super is inserted for you automatically by the compiler. Enforcing super to appear first, enforces that constructor bodies are executed in the correct order. *Object –> Pet –> SuperPet*

The compiler also forces you to declare ***this(..)*** as the first statement within a constructor, otherwise, you will get compile-time error.

Q8. Can constructors have private access modifiers? If yes, can you give an example?
A8. Yes**. Singleton** (i.e design pattern) classes use **private constructors** as shown below**.**

```
public final class MySingletonFactory {

    private static final MySingletonFactory instance = new MySingletonFactory( );

    private MySingletonFactory( ){}

    public static MySingletonFactory getInstance( ) {
        return instance;
    }
}

```

Use cases for **private constructors**:

- The classes with a private constructor cannot be extended from outside even if not declared as final.
- The classes with a private method cannot be invoked from outside. Only the factory methods within that class like *getInstance( ), deepCopy(…),* etc can access a private constructor.

Q9. What are **static factory methods** in Java?
A9. The **factory method pattern** is a way to encapsulate object creation. Without a factory method, you would simply call the class' constructor directly:

```
Pet p = new Pet( );

```

With this pattern, you would instead call the factory method:

```
Pet p = Pet.getInstance();

```

The constructors are marked private, so they cannot be called except from inside the class, and the factory method is marked as static so that it can be called without first having an object.

Java API have many factory methods like *Calendar.getInstance( )*, *Integer.valueOf(5)*, *DriverManager.getConnection( )*, *Class.forName( )*, etc.

Q10. What are the benefits of static factory methods over using constructors directly?
A10.

- Factory can choose from many subclasses (or implementations of an interface) to return. This allows the caller to specify the behavior desired via parameters, without having to know or understand a potentially complex class hierarchy.
- The factory can apply the fly weight design pattern to cache objects and return cached objects instead of creating a new object every time. In other words, objects can be pooled and reused. This is the reason why you should favor using *Integer.**valuOf**(6)* as opposed to **new** *Integer*(6).
- The factory methods have more meaningful names than the constructors. For example, *getInstance( )*, *valueOf( )*, *getConnection( )*, *deepCopy( )*, etc.

Here is a factory method example to deeply clone a list of objects.

```
public static List<Car> deepCopy(List<Car> listCars) {
     List<Car> copiedList = new ArrayList<Car>(10);
     for (Car car : listCars) {                  //JDK 1.5 for each loop
         Car carCopied = new Car();              //instantiate a new Car object
         carCopied.setColor((car.getColor()));
         copiedList.add(carCopied);
    }
    return copiedList;
}

```

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/EE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ 6 Java Modifiers every interviewer seems to like

12 Java classes and interfaces interview Q&As ›

4 Java annotation types & processing interview Q&As | Java-Success.com

SourceURL: https://www.java-success.com/module-3/4-java-annotation-types-processing-interview-qas/

4 Java annotation types & processing interview Q&As

Posted on July 2, 2017 by  Arulkumaran Kumaraswamipillai

Q1. What do you understand by the terms annotation types, annotations, and annotation processors?
A1. An **annotation type** is used for defining an annotation. Java 5 defines a number of annotation types like @Override, @SuppressWarnings, @Deprecated, etc and meta annotation types that are used by annotation (i.e. a meta meta data) type like @Target, @RetntionPolicy, @Inherited, and @Documented. For example,

```
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD,ElementType.TYPE})
public @interface ToDo {
      String comments();
}
```

As you can see the difference between an interface definition and that of an annotation is the presence of **@** before the interface keyword. Here are some of the rules for defining an annotation:

– Method declarations inside an annotation should not have any parameters.
– Method declarations should not have any throws clauses.
– Return types of the methods should be primitives, String, Class, enum, or array of the above types.

An annotation is the meta tag that you use in your applications. For example, you can use the annotation types @Override, @Suppresswarnings, @Inherited, etc included in Java and the custom annotation type @ToDo that was defined above as shown below:

```
package annotation.example;

import java.util.ArrayList;
import java.util.List;

@ToDo(comments="Not yet complete")
public class MyClass {

    @Deprecated
    public void doSomething() {
        // some logic
    }

    @SuppressWarnings(value = "unchecked")
    @ToDo(comments="Need to confirm with legacy app")
    public void doSomethingBetter() {
        List vocabulary = new ArrayList();
        vocabulary.add("deliberate");
    }

    @Override
    public String toString() {
        return super.toString();
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }
}
```

Finally, annotating your code alone is not going to give you any functionality apart from some form of documentation unless you have **processors** that process the annotations in special way to add behavior. The processors can be Java compiler itself, tools that are shipped with Java itself like Javadoc, apt (Annotation Processing Tool), Java Runtime, Java IDEs like eclipse, Net Beans, etc and frameworks like Hibernate 3.0 and Spring 3.0, JEE CDI, etc.

1) **@Override** and **@SuppressWarnings** are used by the Java compiler.
2) The annotation **@Deprecated** is used by the Java compiler and the IDEs like eclipse.
3) The custom annotation **@ToDo** can be used at runtime to produce a summary report as a to do list by querying the annotations at runtime using the Java reflection API.

```java
package annotation.example;

import java.lang.annotation.Annotation;

public class QueryAnnotation {

    public static void main(String[] args) {
        Annotation[] typeAnnotations = MyClass.class.getAnnotations();
        for (Annotation annotation : typeAnnotations) {
            if (annotation.annotationType().getSimpleName().equals(
                    ToDo.class.getSimpleName())) {
                System.out.println(" --> Comments: "
                        + ((ToDo) annotation).comments());
            }
        }
    }
}
```

Q2. Can you describe the ways in which annotations can be used at pre-compile time, compile time, post-compile time and runtime?
A2. **Pre compile-time:** You can generate additional boiler plate source code and descriptor files using tools like apt (Annotation Processing Tool) during the build process. For example, a service framework can be developed using annotations where a developer provides a delegate class say UserDelegate with the required business logic and relevant annotations. The service framework will make use of the apt tool to read this delegate class and produce additional artifacts required to expose the business functions via RMI (using EJBs) and Web Services. The apt tool can be used to generate the required source files like local interfaces, remote interfaces, wrapper implementation to expose the service as RMI and Web Service during build time (i.e. prior to compiling). The annotation processing tool is very powerful.

Compile-time: By the Java compiler and IDEs to raise errors and warnings during compiling source code into byte code (i.e. .class files) as discussed earlier.

Post Compile-time: Annotations can be scanned on byte code files (i.e. .class files) using byte code processing libraries like Javaassist or ASM. Javaassist does have reflection like API that allows you iterate over methods and fields of a class file. You can read your class files from the InputStreams from your classpath or .jar. Based on annotations, additional code can be injected or existing code can be modified. Any form of byte code manipulation can make your code harder to read or understand. Don't favor this approach unless you have a compelling reason to do so. For example, performance considerations associated with scanning for all the annotations by loading each and every class using your Class loader and Java reflection.

Runtime: By the application itself and other frameworks to check the validity of the input passed by the clients and extract program behaviors at runtime using Java reflection. The Java reflection API has been updated with facility to work with annotations since JDK 5.0.

Q3. What do you understand by annotation of annotation? How do you control when an annotation is need and where an annotation should go?
A3. JDK 5.0 provides four annotations in the java.lang.annotation package that are used only when writing annotations.

@Documented → Should the annotation be in Javadoc? Annotations on a class or method don't appear in the Javadocs by default. The @Documented is a marker annotation (i.e. accepts no parameters) that changes this behavior.

@Retention → When the annotation is needed? There are three options as listed in the RetentionPolicy enumeration.

Policy	Description
RetentionPolicy. SOURCE	Annotations are discarded during compile-time. Annotations are not written to the byte code as they would not make any sense. For example, @Override and @SuppressWarnings.
RetentionPolicy. CLASS	Annotations are written to byte code, but discarded when they are loaded into the JVM. This is useful if you want to do any byte code level manipulation.
RetentionPolicy. RUNTIME	Do not discard. The annotation should be available for reflection at runtime. For example, @Deprecated and @Documented. You might be wondering why @Documented retention policy is runtime. This is because Javadoc loads its information from class files, using a JVM.

@Target → Where the annotation can go? You have eight options listed in the ElementType enumeration to tell where a particular annotation can be applied.

ElementType.TYPE (class, interface, enum)
ElementType.FIELD (instance variable)
ElementType.METHOD
ElementType.PARAMETER
ElementType.CONSTRUCTOR
ElementType.LOCAL_VARIABLE
ElementType.ANNOTATION_TYPE (on another annotation)
ElementType.PACKAGE

@Inherited → Should subclasses get the annotation? This controls if an annotation should affect subclasses. If you look at the earlier examples MyClass base class and @ToDo annotation, the @ToDo annotation is annotated with @Inherited.

Q4. What are the different annotation types?
A4.

1) Marker annotation.
2) Single element annotation.
3) Full value or multi-value annotation.

Marker type annotations have no elements, except the annotation name itself.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

Single Element or single value type annotations provide a single piece of data only.

```
@Documented
@Inherited
@Retention(RetentionPolicy.SOURCE)
@Target({ElementType.METHOD,ElementType.TYPE})
public @interface ThreadSafe {
     //default makes it optional
    String value() default "";
}
```

Usage:

```
@ThreadSafe("not using any instance variables")
public void method1(){
   //...
}
```

Full-value or multi-value type annotations have multiple data members.

```
package javax.ejb;

import java.lang.annotation.Retention ;
import java.lang.annotation.Target ;

@Target (ElementType.TYPE)
@Retention (RetentionPolicy.RUNTIME)
public @interface Stateless {
   String  name() default "";
   String  mappedName() default "";
   String  description() default "";
}
```

Usage:

```
@Stateless(name="Charging", description="Charging Service")
@TransactionManagement
                    (TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.NEVER)
public class ChargingDAOImpl extends MediationDAO implements
                                ChargingDAO {
        // fields and methods
}
```

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ 8 Java Annotations interview Q&As

5 Inheritance Vs Composition OOP Interview Q&As ›

8 Java Annotations interview Q&As | Java-Success.com

SourceURL: https://www.java-success.com/module-3/8-java-annotations-interview-qas/

8 Java Annotations interview Q&As

Posted on July 2, 2017 by  Arulkumaran Kumaraswamipillai

Q1. Are annotations a compile time or run-time feature?
A1. You can have either compile-time or run-time annotations.

@Override is a simple **compile-time** annotation to catch little mistakes like typing tostring( ) instead of toString( ) in a subclass.

```
public class B extends A {

    private String input;

    @Override
    public String toString(){
        return "input=" + input;
    }
}

```

If you remove the toString( ) method in Class A or misspell toString() method in Class B, the compiler will warn you.

User defined annotations can be processed at compile-time using the **Annotation Processing Tool** (APT) that is included in the Java 6 itself.

@Test is an annotation that JUnit framework uses at **runtime** with the help of reflection to determine which method(s) to execute within a test class.

```
public class MyTest{
    @Test
     public void testEmptyness( ){
```

```
    public void testEmptyness( ){
        org.junit.Assert.assertTrue(getList( ).isEmpty( ));
    }

    private List getList( ){
        …
    }
}
```

The test below fails if it takes more than 100ms to execute at runtime.

```
@Test(timeout=100)
public void testTimeout( ) {
    while(true);   //infinite loop
}
```

The code shown below fails if it does not throw “IndexOutOfBoundsException” or if it throws a different exception at runtime. A negative JUnit test.

```
@Test (expected=IndexOutOfBoundsException.class)
public void testOutOfBounds( ) {
       new ArrayList<Object>( ).get(1);
}
```

Q2. Are marker or tag interfaces like Serializable, Runnable, etc obsolete with the advent of annotations (i.e. runtime annotations)?
A2. 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 unimplement 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

— In the very rare event of the profiling indicating that the runtime checks are expensive due to being accessed very frequently, and compile-time checks with interfaces is preferred.

— In the event of existing marker interfaces like Serializable, Cloneable, etc are used or Java 5 or later versions cannot be temporarily used.

Q3. What is an annotation, and why are they so popular and used in every framework like Spring, JAX-RS (i.e RESTful web service), JEE 6, and a lot more? When will you favor XML based meta data over annotations based meta data?
A3. One word to explain Annotation is Metadata. Metadata is data about data. So Annotations are metadata for code. The IDEs, compilers, frameworks, and other tools read the annotations to control the behavior of the code that are annotated.

Prior to annotation (and even after) XML were extensively used for metadata, but XML is very verbose and its maintenance was becoming troublesome. Since annotations are closely coupled with the code, they are less verbose. You can define annotations for a class, method, field, etc. XML is defined separately from the code, so it is more verbose as you have to define the class name, method name, etc.

If you want to set some application wide constants and parameters XML would be a better choice because this is not related with any specific piece of code. If you want to expose some method as a Web service, annotation would be a better choice as it needs to be tightly coupled with that method and developer of the method must be aware of this.

Annotations: let you avoid boilerplate code under many circumstances by enabling tools to generate it from annotations in the source code. This leads to “attribute oriented” (aka declarative) programming. This eliminates the need to maintain “side files” that must be kept up to date with changes in source files.

Annotations based development relieves Java developers from the pain of cumbersome configuration. Annotations provide declarative style programming where the programmer says what should be done and tools emit the code to do it. This assists rapid application development, easier maintenance, and less likely to be bug-prone.

In Spring’s world, an XML config (e.g myapp-applicationContext.xml) to scan for all annotations in the base-package can be defined as shown below.

```
<!-- Component-Scan automatically detects annotations in the Java classes.  -->
<context:component-scan base-package="com.myapp.batch" />
```

A POJO is exposed as a RESTful web service with JAX-RS annotations like @Path, @Produces, @GET, @PathParam, etc.

```
package com.mytutorial.webservice;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import com.mytutorial.pojo.User;

@Path("userservice/1.0")
@Produces("application/xml")
public class HelloUserWebServiceImpl implements HelloUserWebService {

   @GET
   @Path("/user/{userName}")
   public User greetUser(@PathParam("userName") String userName) {
      User user = new User();
      user.setName(userName);
      return user;
   }

}

```

In JEE 6 CDI (**C**ontexts and **D**ependency **I**njection) world

You need to have at least an empty "**beans.xml**" file defined under META-INF resource folder for jar files or under WEB-INF folder for war files for DI to take effect.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">

</beans>

```

and **@inject** annotation to inject dependencies.

```
import javax.inject.Inject;

public class MyApp {

   private final HelloService helloService;

   @Inject
   public MyApp(HelloBean helloBean){
      this.helloService = helloService;
   }

   //.....
}

```

```
@Default
public class HelloServiceImpl implements HelloService {

    public void sayHello() {
        System.out.println("say hello .........");
    }
}

```

Q4. How will you go about defining a custom annotation? Can you give a practical example?
A4. Here is an example where methods annotated with the following custom **@DeadlockRetry** annotation will retry the database operation. The annotation has the attributes *maxTries* and *tryIntervalMillis*.

```
package com.myapp.deadlock;

import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.METHOD)
@Inherited
public @interface DeadlockRetry
{
    int maxTries() default 10;
    int tryIntervalMillis() default 1000;
}


```

The above custom annotation is annotated with **@Retention** to tell that it is used at run-time, and **@Target** to tell that it is for methods.

Q. Now, who processes this annotation?
A. The following **dynamic proxy** class that uses reflection to see if a method is annotated with @DeadlockRetry. If does, retries the database method call.

```
package com.myapp.deadlock;

import java.lang.annotation.Annotation;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class DeadlockRetryHandler implements InvocationHandler
{
    private Object target;

    public DeadlockRetryHandler(Object target)
    {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
    {
        Annotation[] annotations = method.getAnnotations();
        DeadlockRetry deadlockRetry = (DeadlockRetry) annotations[0];

        final Integer maxTries = deadlockRetry.maxTries();
        long tryIntervalMillis = deadlockRetry.tryIntervalMillis();

        int count = 0;

        do
        {
            try
            {
                count++;
                Object result = method.invoke(target, args);    // retry
                return result;
            }
            catch (Throwable e)
            {
                if (!DeadlockUtil.isDeadLock(e))
                {
                    throw new RuntimeException(e);
                }

                if (tryIntervalMillis > 0)
                {
                    try
                    {
                        Thread.sleep(tryIntervalMillis);
                    }
                    catch (InterruptedException ie)
                    {
                        System.out.println("Deadlock retry thread interrupted", ie);
                    }
                }
            }
        }
        while (count <= maxTries);

        //gets here only when all attempts have failed
        throw new RuntimeException("DeadlockRetryMethodInterceptor failed to successfully execute target "
                + " due to deadlock in all retry attempts",
                new DeadlockDataAccessException("Created by DeadlockRetryMethodInterceptor", null));
    }
}


```

The *DeadlockUtil* class determines if the exception is related to deadlock or other SQL exception based on error code and exception type like “LockAcquisitionException”.

Now define the target object interface with the custom annotation. The maxTries = 10, tryIntervalMillis = 5000.

```java
package com.myapp.engine;

import com.myapp.DeadlockRetry;

public interface AccountServicePersistenceDelegate
{
    @DeadlockRetry(maxTries = 10, tryIntervalMillis = 5000)
    abstract Account getAccount(String accountNumber);
}

```

Q5. What are some of the JAX-RS (i.e RESTful) web service annotations?
A5. @**GET, @POST, @PUT, @DELETE** to specify what type of verb this method (or web service) will perform.

@Produces to specify the type of output this method (or web service) will produce.

@Consumes to specify the MIME media types a REST resource can consume.

@Path to specify the URL path on which this method will be invoked.

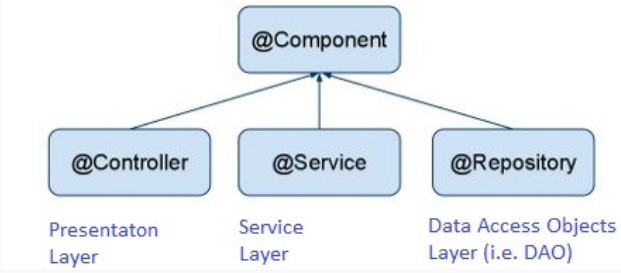
@PathParam to bind REST style parameters to method arguments.

@QueryParam to access parameters in query string (http://localhost:8080/context/accounting-services?accountName=PeterAndCo).

Another example for custom annotation would be for service retry.

Q6. What are some of the widely used Spring annotations?
A6. The Spring beans can be wired either by name or type. **@Autowired** by default is a type driven injection. @Autowired is Spring annotation, while @Inject is a JEE CDI annotation. **@Inject** is equivalent to @Autowired or @Autowired(required=true). **@Qualifier** spring annotation can be used to further fine-tune auto-wiring. There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property, in such case you can use @Qualifier annotation along with @Autowired to remove the confusion by specifying which exact bean will be wired.

The different POJO objects in different layers are annotated with one of the following key annotations.



Spring annotations

@Component is the parent annotation from which the other annotations like **@Service**, **@Resource**, **@Repository** etc are defined. Here is an example of a DAO class annotated with @Repository, and @Resource is used to inject "jdbcTemplateSybase" with which the database calls are made.

```java
@Repository(value = "myapp_Dao")
public class CashForecastDaoImpl implements CashForecastDao
{

    @Resource(name = "myapp_JdbcTemplate")
    private JdbcTemplate jdbcTemplateSybase;//configure via jdbcContext.xml

    public PortfolioSummaryVO  retrievePortfolioSummaries(MyAppPortfolioCriteria criteria) {
      //............use jdbcTemplateSybase
    }

}
```

The annotations shown above allow you to declare beans that are to be picked up by autoscanning with or **@ComponentScan**.

The **@Configuration** annotation was designed as the replacement for XML configuration files. You use @Bean annotation to wire up dependencies. Here is an example.

```
package com.myapp.bdd.stories;

import com.myapp.*;

import org.powermock.api.mockito.PowerMockito;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.context.annotation.Import;

@Configuration
//packages and components to scan and include
@ComponentScan(
        basePackages =
        {
            "com.myapp.calculation.bdd",
            "com.myapp.refdata",
            "com.myapp.dm.dao"

        },
        useDefaultFilters = false,
         //interfaces
        includeFilters =
        {
            @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, value = MyApp.class),
            @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, value = MyAppDroolsHelper.class),
            @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, value = TransactionService.class),
            @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, value = TransactionValidator.class),

            @ComponentScan.Filter(type = FilterType.ANNOTATION, value = org.springframework.stereotype.Repository.class)
        })
//import other config classes
@Import(
{
    StepsConfig.class,
    DataSourceConfig.class,
    JmsConfig.class,
    PropertiesConfig.class
})
public class StoryConfig
{
    //creates a partially mocked transaction service class
    @Bean(name = "txnService")
    public TransactionService getTransactionService()
    {
        return PowerMockito.spy(new TransactionService());
    }
}

```

Q7. What are some of the widely used JEE CDI annotations?
A7. CDI is one of the biggest promises of JEE 6.

@Default, **@Alternative**, and **@Name**: A class is @Default by default. Thus marking it a redundant annotation. Mark as @Alternative to annotate the alternative implementations that implement the same interface. Use the @Named annotation to look up by name. The **@Named** annotation is also used by JEE 6 application to make the bean accessible via the Unified EL.

The interface

```
public interface PaymentService {
    public void processPayment(BigDecimal amount, Account account);
}
```

The **default** implementation

```
@Default
public class CashPaymentServiceImpl implements PaymentService{
    public void processPayment(BigDecimal amount, Account account) {
        //.............
    }
}
```

An **alternative** implementation

```
@Alternative
public class BPayPaymentServiceImpl implements PaymentService{
    public void processPayment(BigDecimal amount, Account account) {
         //.............
    }
}
```

```
@Alternative
public class CreditCardPaymentServiceImpl implements PaymentService{
    public void processPayment(BigDecimal amount, Account account) {
         //.............
    }
}
```

An alternative can be selected via XML based deployment files in {classpath}/META-INF/beans.xml

```
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
    <alternatives>
        <class>BPayPaymentServiceImpl</class>
    </alternatives>
</beans>

```

A **named** implementation

```
@Named("cash")
public class CashPaymentServiceImpl implements PaymentService{
    public void processPayment(BigDecimal amount, Account account) {
         //.............
    }
}
```

Named annotations can be looked up via the JEE bean container.

```
public class PaymentMain {
    //...

    public static void main(String[] args) throws Exception {
        PaymentService svc = (PaymentService) beanContainer
                .getBeanByName("cash");

        svc.processPayment(new BigDecimal("9.00"), acc);
    }

}
```

@Inject to inject via fields and constructors.

```
public class PaymentMain {

    @Inject
    private PaymentService paymentService;
    //...
}
```

```
public class PaymentMain {

    private PaymentService paymentService;

    @Inject
    public PaymentMain(PaymentService paymentService) {
         this. paymentService = paymentService;
    }

    //....
}
```

@Produces for more complex object construction via **factory methods**.

```
import javax.enterprise.inject.Produces;

public class PaymentFactory {

    @Produces
    public PaymentService createPaymentService() {
        return new CashPaymentServiceImpl();
    }
    //.....
}

```

@Qualifier is required when an interface has multiple implementations to choose the one you want to inject. All objects and producers in CDI have qualifiers. If you do not assign a qualifier, by default has the qualifier **@Default** and **@Any**. So, if you don't specify a qualifier, you will be assigned one.

Meta meta annotations to define a qualifier

```
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface BPay {

}
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface CreditCard {

}
```

Use the qualifier annotations

```
@BPay
public class BPayPaymentServiceImpl implements PaymentService{
    public void processPayment(BigDecimal amount, Account account) {
        //.............
    }
}
```

```
@CreditCard
public class CreditCardPaymentServiceImpl implements PaymentService{
    public void processPayment(BigDecimal amount, Account account) {
        //.............
    }
}
```

BPay is injected

```
public class PaymentMain {

    private PaymentService paymentService;

    @Inject
    public PaymentMain(@BPay PaymentService paymentService) {
        this. paymentService = paymentService;
```

```
    }

    //....
}
```

Multiple types can be injected into the same bean

```
public class PaymentMain {
    //...

    @Inject @BPay
    private PaymentService bpayService;

    @Inject @CreditCard
    private PaymentService ccService;

    private PaymentService paymentService;

    //....

    @PostConstruct
    protected void init() {
      if(account.isBPay()) {
         this.paymentService = this.bpayService;
      }
      else {
         this.paymentService = this.ccService;

      }
    }
}
```

Note: @PostConstruct annotation is used to decide which service to use. Alternatively, you can use a factory method with **@Produces** annotation.

```
@Produces
public PaymentService createPayment (@BPay PaymentService bpayService,
                        @CreditCard PaymentService ccService) {

   if (account.isBPay()) {
       return bpayService;
   } else {
       return ccService;
   }
}
```

Q8. How would you unit test CDI with JUnit?
A8. **@RunWith** annotation and pass "CdiRunner.class".

```
@RunWith(CdiRunner.class)
class PaymentServiceTest {
   @Inject
   PaymentService paymentService;
   //.....
}
```

There are other annotations like @AdditionalClasses, @AdditionalPackages, @AdditionalClasspath, @Produces, @EnabledAlternatives, @ProducesAlternative, etc and scoping annotations like @InRequestScope, @InSessionScope, and @InConversationScope.

Q9. In Servlet 3.0, why is configuring your servlet via deployment descriptor file web.xml optional?
A9. Servlets 3.0 have come up with a set of new Annotations for the declarations of Servlet Mappings, Init-Params, Listeners, and Filters to make the code more readable by making the use of Deployment Descriptor (web.xml) absolutely optional.

```
@WebServlet(
  asyncSupported = false,
  name = "AccountingServlet",
  urlPatterns = { "/acount" },
  initParams = {
    @WebInitParam(name = "param1", value = "value1"),
    @WebInitParam(name = "param2", value = "value2")
  }
)
public class AccountingServlet extends HttpServlet {
    //...
}

```

Servlet filters, listeners, etc can be configured with annotations.

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

Have you completed this unit? Then mark this unit as completed.

Mark as Completed

« Previous Unit | Next Unit »

‹ Lambda expressions to work with Java 8 Collections

4 Java annotation types & processing interview Q&As ›

Core Java Interview Questions-Part4

Notebook:	Core Java		
Created:	23-06-2018 16:59	**Updated:**	23-06-2018 17:25
URL:	https://www.java-success.com/module-4/3-java-class-loading-interview-qas-ascertain-depth-java-knowledge/		

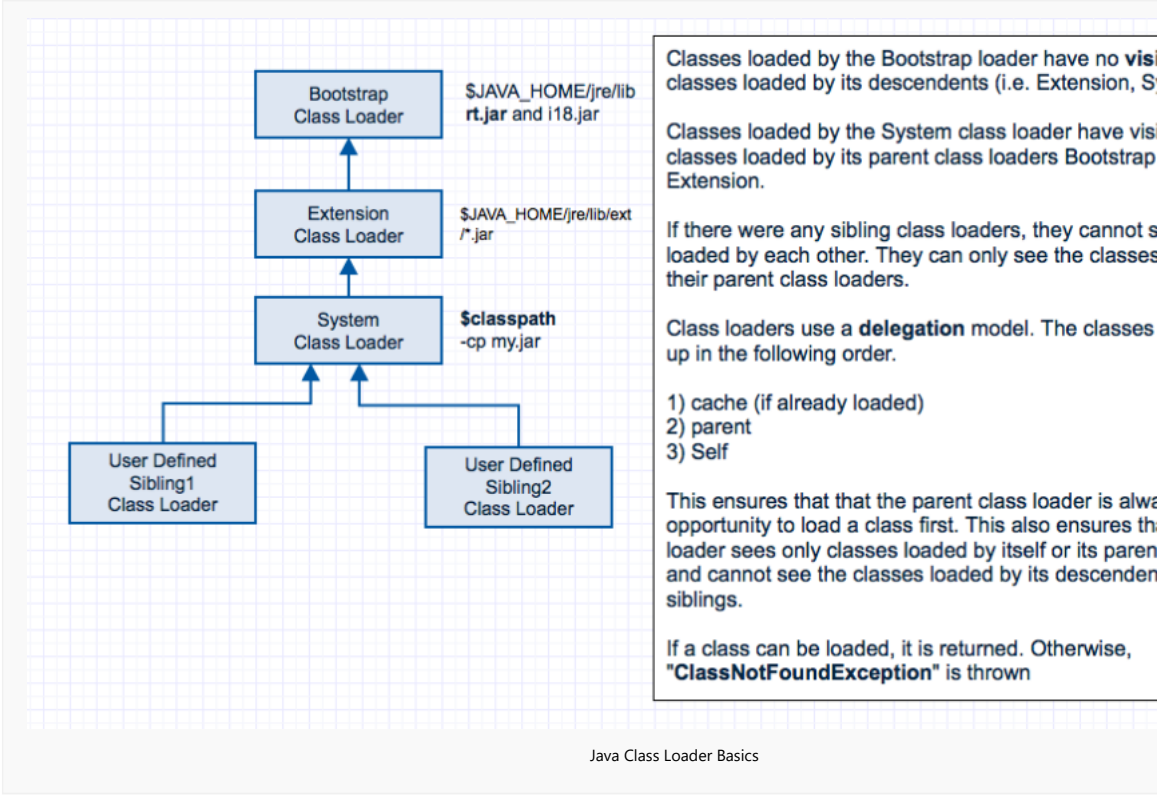
3 Java class loading interview Q&As to ascertain your depth of Java knowledge | Java-Success.com

SourceURL: https://www.java-success.com/module-4/3-java-class-loading-interview-qas-ascertain-depth-java-knowledge/

3 Java class loading interview Q&As to ascertain your de knowledge

Posted on July 1, 2017 by  Arulkumaran Kumaraswamipillai

Q1. What do you know about Java class loading? Explain Java class loaders?
A1. Class loaders are hierarchical. Classes are introduced into the JVM as they are referenced by name in a class that is already running in the JVM. So, how is th loaded with the help of static main( ) method declared in your class. All the subsequently loaded classes are loaded by the classes, which are already loaded an JVMs include at least one class loader that is embedded within the JVM called the primordial (or bootstrap) class loader. The JVM has hooks in it to allow user c class loader. Let us look at the class loaders created by the JVM.



Java Class Loader Basics

Class loaders are hierarchical and use a delegation model when loading a class. Class loaders request their parent to load the class first before attempting to lo child class loaders in the hierarchy will never reload the class again. Hence uniqueness is maintained. Classes loaded by a child class loader have visibility into c reverse is not true as explained in the above diagram.

Q2. Explain static vs. dynamic class loading?
A2. Classes are statically loaded with Java's "new" operator.

```
1
2 class MyClass {
3     public static void main(String args[]) {
4         Car c = new Car( );
5     }
6 }
```

Dynamic loading is a technique for programmatically invoking the functions of a class loader at run time. Let us look at how to load classes dynamically.

```

//static method which returns a Class
Class.forName (String className);

```

The above static method returns the class object associated with the class name. The string className can be supplied dynamically at run time. Unlike the stat load the class Car or the class Jeep at runtime based on a properties file and/or other runtime conditions. Once the class is dynamically loaded the following m like creating a class object with no arguments.

```

// A non-static method, which creates an instance of a
// class (i.e. creates an object).
class.newInstance ( );

```

Static class loading throws "**NoClassDefFoundError**" if the class is not found and the dynamic class loading throws "**ClassNotFoundException**" if the class is r

Q. What is the difference between the following approaches?

```
Class.forName("com.SomeClass");
```

and

```
classLoader.loadClass("com.SomeClass");
```

A.

Class.forName("com.SomeClass")

— Uses the caller's classloader and initializes the class (runs static intitializers, etc.)

classLoader.loadClass("com.SomeClass")

— Uses the "supplied class loader", and initializes the class **lazily** (i.e. on first use). So, if you use this way to load a JDBC driver, it won't get registered, and you

The "java.lang.API" has a method signature that takes a boolean flag indicating whether to initialize the class on loading or not, and a class loader reference.

```

forName(String name, boolean initialize, ClassLoader loader)

```

So, invoking

```
Class.forName("com.SomeClass")
```

is same as invoking

```
forName("com.SomeClass", true, currentClassLoader)
```

Q. What are the different ways to create a "ClassLoader" object?
A.

```

ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
ClassLoader classLoader = MyClass.class.getClassLoader();      // Assuming in class "MyClass"
ClassLoader classLoader = getClass().getClassLoader();         // works in any class

```

Q. How to load property file from classpath?
A. **getResourceAsStream()** is the method of java.lang.Class. This method finds the resource by implicitly delegating to this object's class loader.

```
final Properties properties = new Properties();
try (final InputStream stream = this.getClass().getResourceAsStream("myapp.properties")) {
    properties.load(stream);
    /* or properties.loadFromXML(...) */
}
```

Note: "Try with AutoCloseable resources" syntax introduced with Java 7 is used above.

Q. What is the benefit of loading a property file from classpath?
A. It is portable as your file is relative to the classpath. You can deploy the "jar" file containing your "property" file to **any location** where the JVM is.

Loading it from outside the classpath is NOT portable

```
final Properties properties = new Properties();
final String dir = System.getProperty("user.dir");

try (final InputStream stream =  new FileInputStream(dir + "/myapp/myapp.properties")) {
    properties.load(stream);
}
```

As the above code is NOT portable, you must document very clearly in the installation or deployment document as to where the property file is loaded from be it might not already have the path "dir" and "myapp" configured.

So, loading it via the classpath is recommended as it is a portable solution.

Q3. What tips would you give to someone who is experiencing a class loading or "Class Not Found" exception?
A3. "ClassNotFoundException" could be quite tricky to troubleshoot. When you get a ClassNotFoundException, it means the JVM has traversed the entire class reference.

1) Stand alone Java applications use -cp or -classpath to define all the folders and jar files to look for. In windows separated by ";" and in Unix separated by ":".

```
java -classpath "C:/myproject/classes;C:/myproject/lib/my-utility.jar;C:/myproject/lib/my-dep.jar" MyApp
</code>>/pre>

2) Determine the jar file that should contain the class file within the classpath -- war/ear archives and application server lib d

<pre class="prettyprint"><code class="language-java">
$ find . -name "*.jar" -print -exec jar -tf '{}' \; | grep -E  "jar$|String\.class"
```

You can also search for the class at www.jarfinder.com

3) Check the version of the jar in the manifest file MANIFEST.MF, access rights (e.g. read-only) of the jar file, presence of multiple versions of the same jar file ar ...". If the class is dynamically loaded with Class.forName("com.myapp.Util"), check if you have spelled the class name correctly.

4) Check if the application is running under the right JDK? Check the JAVA_HOME environment property

```
$  echo $JAVA_HOME
```

5) **-verbose:class** option in your JVM. With the -verbose option all the classes that are loaded are listed, along with the JAR file or directory from which they we information, such as when superclasses are being loaded, and when static initializers are being run.

6) Creating a **Java dump and analyzing the Java dump** for class loading issues. The Java dumps are created under following circumstances.

— When a fatal native JVM error.
— When the JVM runs out of heaps memory space.
— When a signal is sent to the JVM (e.g. Control-Break is pressed on Windows, Control-\ on Linux, or kill -3 on Unix)

There are tools like **jstack**, **jmap**, **hprof**, and Eclipse Memory Analyzer (MAT) to analyze the Java dumps.

7) Some of the libraries provide **API to list the version number**. For example, The Eclipse link MOXy library provides a method as shown below.

```
1
2    org.eclipse.persistence.Version.getVersion();
3
```

8) The org.jboss.test.util.Debug class has a method *displayClassInfo(Class clazz, StringBuffer results)* to display the loaded class details. This is done programmat

```
1
2       URL loc = MyClass.class.getProtectionDomain().getCodeSource().getLocation();
3
```

9) The http://www.findjar.com is an online search engine that can list possible jar files in which a particular class file like java.sql.Connection can be found.

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. A job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java

10 Java serialization, cloning, and casting interview Q&As | Java-Success.com

SourceURL: https://www.java-success.com/module-5/10-java-serialization-cloning-casting-interview-qas/

10 Java serialization, cloning, and casting interview Q&As

Posted on July 1, 2017 by Arulkumaran Kumaraswamipillai

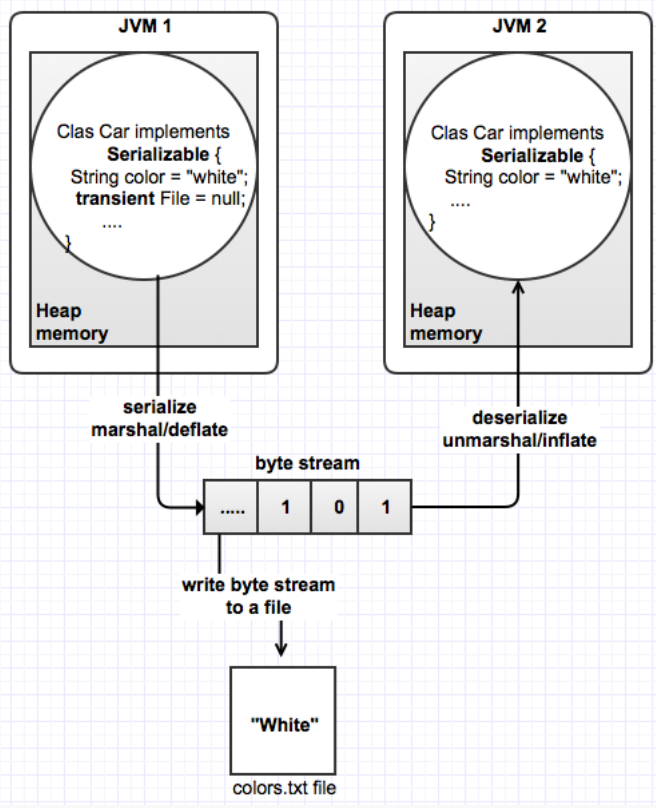
Q1. Which Java interface must be implemented by a class whose instances are transported via a Web service?

a. Accessible
b. BeanInfo
c. Remote
d. Serializable

A1. Answer is "d".

Q2. What is serialization?
A2. 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. This happens in between two different processes (i.e. JVM or heap memory). So, you can't serialize non memory resources like file handles, sockets, threads, etc.

can't serialize non-memory resources like file handles, sockets, threads, etc.



Java Serialization

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.

To persist objects, you need to keep 5 rules in mind:

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. In the example shown below, the Pet class implements the Serializable interface, and also the containing field types String and Integer are also serializable.

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.

Q3.How would you exclude a field of a class from serialization?
A3.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.

Q4. What happens to static fields during serialization?
A4. erialization persists only the state of a single object. Static fields are not part of the state of an object – they're effectively the state of the class shared by many other instances.

Q5. What are the common uses of serialization? Can you give me an instance where you used serialization?
A5.

1. allows you to **persist objects with state to a text file** on a disk, and re-assemble them by reading this back in. 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, Web service, etc.

3. allows you to **deeply clone** any arbitrary object graph.

Q6. What is a serial version id?
A6. 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, and serial version id has now changed. This exception is thrown when the serial version id of the class is not equal to the serial version id 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.

Best practice: 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.

Q7. Are there any disadvantages in using serialization?
A7. Yes. Serialization can adversely affect performance since it:

— Depends on reflection.
— Has an incredibly verbose data format.
— is very easy to send surplus data.

So don't use serialization if you do not have to.

Q8. What is the difference between Serializable and Externalizable interfaces? How can you customize the serialization process?
A8. An object must implement either Serializable or Externalizable interface before it can be written to a byte stream. When you use Serializable interface, your class is serialized automatically by default. But you can override writeObject(..) and readObject(...) methods to control or customize your object serialization process. For example, you can add the following methods to your Pet class.

```
1  private void writeObject(ObjectOutputStream out)  throws IOException {
2           //any write customization goes in this method
3          System.out.println("Started writing object ......................");
4          out.writeObject(this);
5      }
6
7  private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException    {
8        //any read customization goes in this method
9        System.out.println("Started reading object ......................");
10         in.readObject( );
11     }
12
```

Note: Both the above methods must be declared private.

No changes are required for FlattenPet and InflatePet classes.

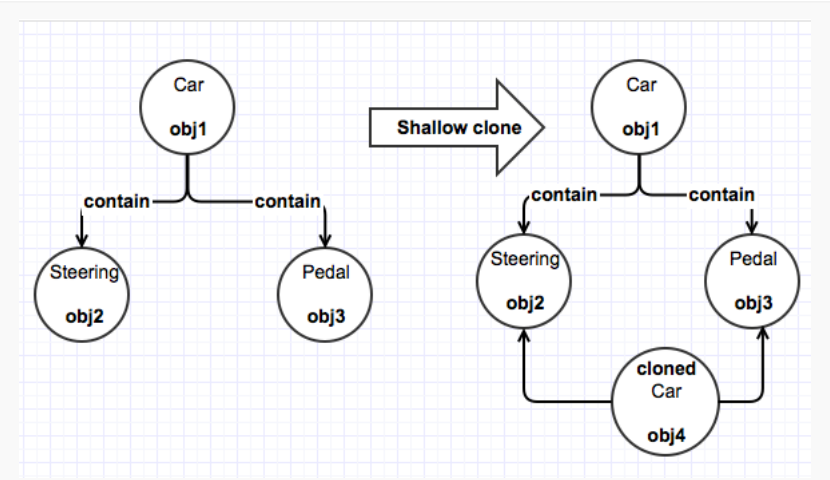
When you use **Externalizable** interface instead of the Serializable interface, you have a complete control over your class's serialization process. This interface contains two methods namely readExternal(...) and writeExternal(...) to achieve this total customization. You can change the Pet class to implement the Externalizable interface and then provide implementation for following 2 methods.

```
1
2  public void writeExternal(ObjectOutput out) throws IOException;
3  public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
4
```

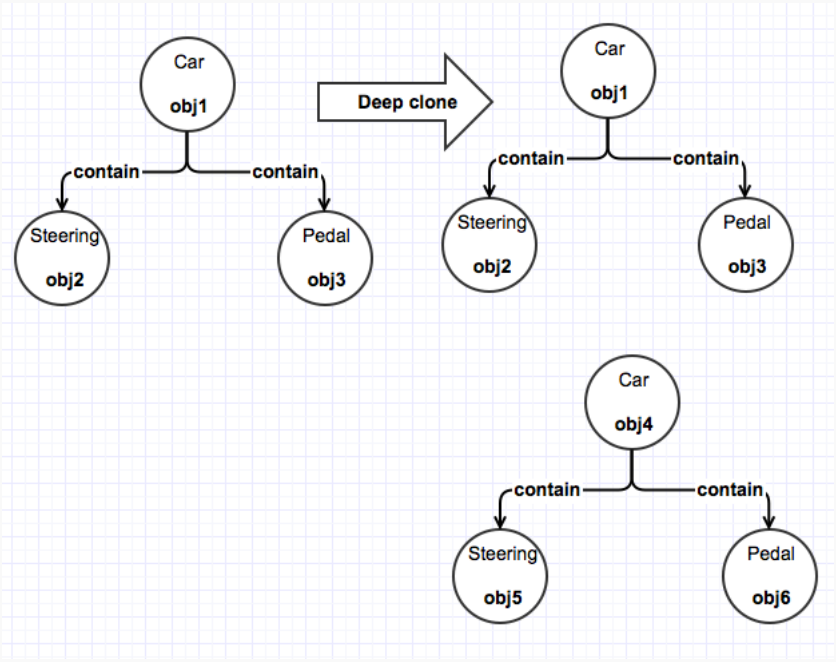
An example situation for this full control will be to read and write PDF files with a Java application. If you know how to write and read PDF (the sequence of bytes required), you could provide the PDF specific protocol in the *writeExternal(...)* and *readExternal(...)* methods.

Just as before, there is no difference in how a class that implements Externalizable is used. Just call writeObject( ) or readObject and, those externalizable methods will be called automatically.

Q9. What is the main difference between shallow cloning and deep cloning of objects?
A9. **Shallow copy:** If a shallow copy is performed, the contained objects are not cloned. Java supports shallow cloning of objects by default when a class implements the java.lang.Cloneable interface. For example, invoking clone( ) method on a collection like HashMap, List, etc returns a shallow copy of the HashMap, List, instances. This means if you clone a HashMap, the map instance is cloned but the keys and values themselves are not cloned, but are shared by pointing to the original objects.



Java shallow cloning



Java deep cloning or copying

Deep copy: If a deep copy is performed, then not only the original object has been copied, but the objects contained within it have been copied as well. Serialization can be used to achieve deep cloning. For example, you can serialize a HashMap to a ByteArrayOutputStream and then deserialize it. This creates a deep copy, but does require that all keys and values in the HashMap to be Serializable. The main advantage of this approach is that it will deep copy any arbitrary object graph. Deep cloning through serialization is faster to develop and easier to maintain, but carries a performance overhead. Alternatively, you can provide a static factory method to deep copy as shown below:

```
1
2  public static List deepCopy(List listCars) {
3       List copiedList = new ArrayList(10);
4       for (Object object : listCars) {
5          Car original = (Car)object;
6          Car carCopied = new Car( );  //instantiate a new Car object
7          carCopied.setColor((original.getColor( )));
8          copiedList.add(carCopied);
9       }
```

```
    return copiedList;
}

```

Q10. What is type casting? Explain up casting vs. down casting? When do you get *ClassCastException*?
A10. Type casting means treating a variable of one type as though it is another type.

byte (1 byte) → **short** (2 bytes) → **char** (2 bytes) → **int** (4 bytes) → **long** (8 bytes) → **float** (4 bytes) → **double** (8 bytes)

Note: Want anything larger than long or double? Then look at *BigInteger* and *BigDecimal*.

When up casting **primitives** from left to right, automatic conversion occurs. But if you go from right to left, down casting or explicit casting is required.

When it comes to object references, you can **always cast from a subclass to a super class** because a subclass object is also a super class object. You can cast an object implicitly to a super class type (i.e. up casting). If this were not the case, polymorphism wouldn't be possible.

You can cast down the hierarchy as well, but you must explicitly write the cast and the object must be a legitimate instance of the class you are casting to. The ClassCastException is thrown to indicate that code has attempted to cast an object to a subclass of which it is not an instance. If you are using JSE 5.0 or later version, then "**generics**" will minimize the need for casting, and otherwise you can deal with the problem of incorrect down casting in two ways:

1. Using the exception handling mechanism to catch ClassCastException:

```

Object o = null;
try{
 o = new Integer(1);
 System.out.println((String) o);
}
catch(ClassCastException cce) {
        logger.log("Invalid casting, String is expected…Not an Integer");
        System.out.println(((Integer) o).toString( ));
}

```

2. Using the **instanceof** statement to guard against incorrect casting:

```

if(o instanceof String) {
   String s2 = (String) o;
}
else if (o instanceof Integer) {
     Integer i2 = (Integer) o;
}


```

The "instanceof" and "typecast" constructs can make your code unmaintainable due to large "if" and "else if" statements, and also can adversely affect performance if used in frequently accessed methods or loops. Look at using generics or visitor design pattern to avoid or minimize these casting constructs where applicable.

Note: You can also get a ClassCastException when two different class loaders load the same class because they are treated as two different classes.

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

5 Java Object class methods interview Q&As | Java-Success.com

SourceURL: https://www.java-success.com/module-5/5-java-object-class-methods-interview-qas/

5 Java Object class methods interview Q&As

Posted on July 1, 2017 by  Arulkumaran Kumaraswamipillai

Q1. What are the non-final methods in Java Object class, which are meant primarily for extension?
A1. The non-final methods are

equals( ), hashCode( ), toString( ), clone( ), and finalize( ).

These methods are meant to be **overridden**. The equals( ) and hashCode( ) methods prove to be very important, when objects implementing these two methods are added to collections. If implemented incorrectly or not implemented at all, then your objects stored in a Map may behave strangely and also it is hard to debug.

The other methods like wait( ), notify( ), notifyAll( ), getClass( ), etc are final methods and therefore cannot be overridden. The methods clone( ) and finalize( ) have protected access.

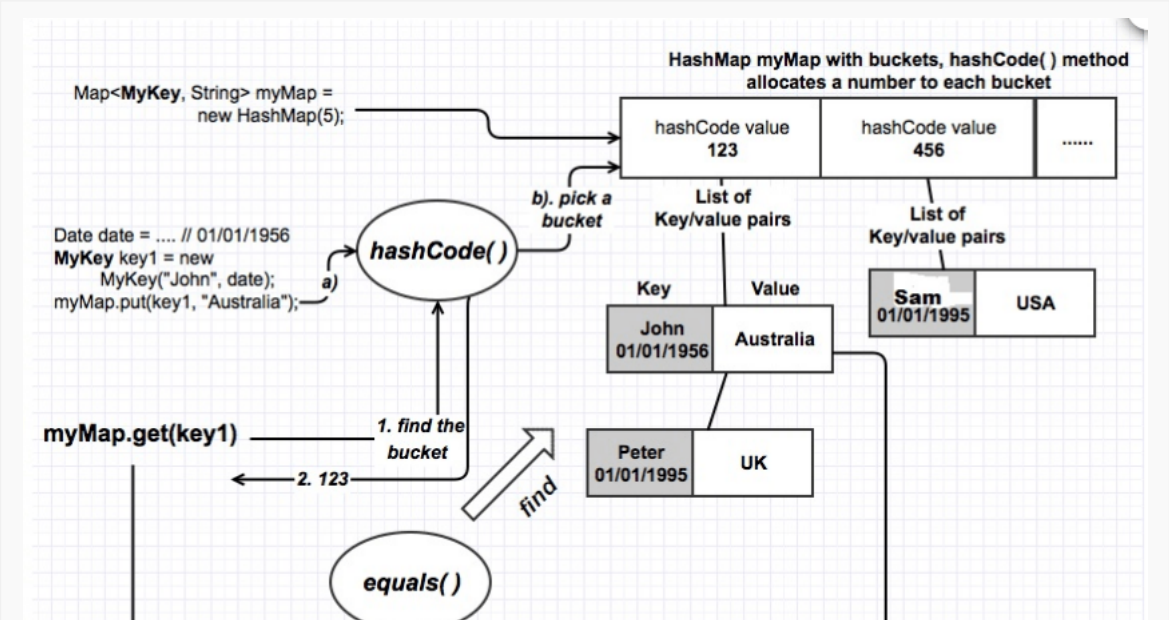
Q2. Are these methods easy to implement correctly?
A2. No. It is not easy to implement these methods correctly because,

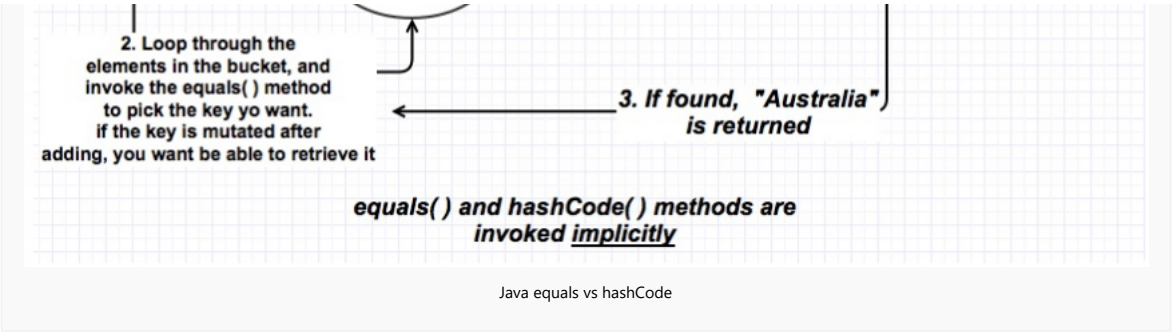
1) You must pay attention to whether your implementation of these methods will continue to work correctly if sub classed. If your class is not meant for extension, then declare your class as final.
2) These methods have to adhere to contracts, and when implementing or overriding a method, the contracts must be satisfied.

Equals() Vs hashCode()

Q3. What are the implications of implementing them incorrectly?
A3. In short, excess debug and rework time. Incorrect implementations of equals( ) and hashCode( ) methods can result in data being lost in HashMaps and Sets. You can also get intermittent data related problems that are harder to consistently reproduce over time.

Here is an example of equals() and hashCode() methods being invoked implicitly in adding and retrieving objects from a *Map*:





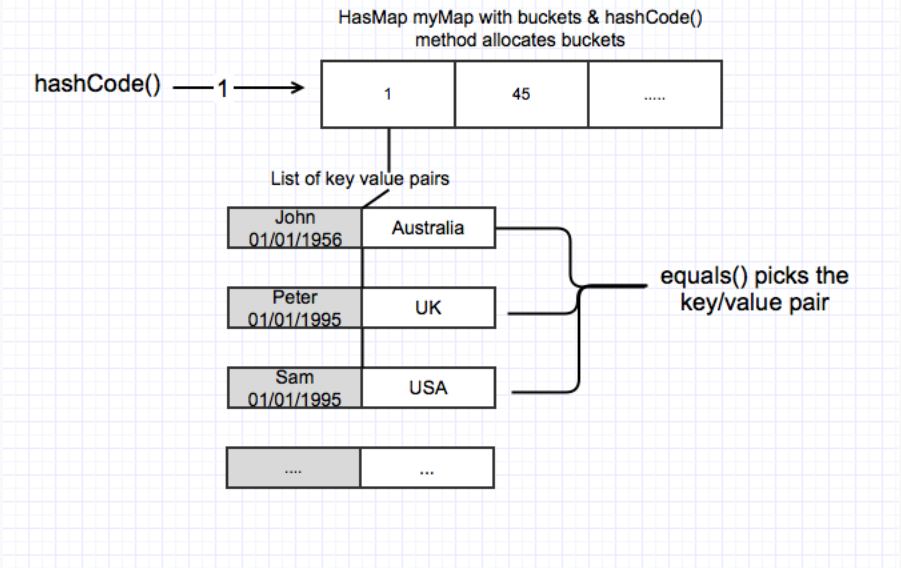
Java equals vs hashCode

The above example uses a custom key class **MyKey** that takes "*name*" and "*date*" as attributes. So, this key class needs to implement **equals()** and **hashCode()** methods using these 2 attributes.

a & b) When you put an object into a map with a key and a value, **hashCode()** method is implicitly invoked, and hash code value say 123 is returned. Two different keys can return the same hash value. A good hashing algorithm spreads out the numbers. In the above example, let's assume that ("John",01/01/1956) key and ("Peter", 01/01/1995) key return the same hash value of **123**.

Q. So, when we retrieve the object with a specific key, how does it know which one of two objects to return?
A. This is where the **equals()** method is implicitly invoked to get the object value with the right key. So, it needs to loop through each key/value pair in the bucket "123" to find the right key by comparing "name" & "date" via equals() method.

Q. What if you change the **hashCode()** method of the "MyKey" class to always return a constant value of? Will you be able to add objects to the map?
A. Yes, you will be able to add objects, but all the key/value pairs will be added to the single bucket, and the equals() method needs to loop through all the objects in the same bucket.



hashCode() returns 1

1 & 2). The hashCode() method picks the right bucket **123**. Then loop through all the values and invoke the equals() method to pick the right key, which is ("John",01/01/1956).

Q4. If a class overrides the equals( ) method, what other method it should override? What are some of the key considerations while implementing these methods?
A4. It should override the **hashCode( ) method**. The contract between the hashCode( ) and equals( ) methods is clearly defined on the Java API for the Object class under respective methods. Here are some key points to keep in mind while implementing these methods,

1) The equals( ) and hashCode( ) methods should be implemented together. You should not have one without the other.
2) If two objects return the same hashCode( ) integer value does not mean that those two objects are equal.
3) If two objects are equal, then they must return the same hashCode( ) integer value.
4) The implementation of the equals( ) method must be consistent with the hashCode( ) method to meet the previous bullet points.
5) Use @override annotation to ensure that the methods are correctly overridden.

6) Favor instanceof instead of getClass(..) in the equals( ) method which takes care of super types and null comparison as recommended by Joshua Bloch, but ensure that the equals implementation is final to preserve the symmetry contract of the method: x.equals(y) == y.equals(x). If final seems restrictive, carefully examine to see if overriding implementations can fully maintain the contract established by the Object class.
7) Check for self-comparison and null values where required.

```
public final class Pet {
    int id;
    String name;

    /**
     * use @override annotation to prevent the danger of misspelling
     * method name or incorrect method signature.
     */

    @Override
    public boolean equals(Object that){
        //check for self-comparison
        if ( this == that ) return true;

       /**
        * use instanceof instead of getClass here for two reasons
        * 1.  it can match any super type, and not just one class;
        * 2.  explicit check for "that == null" is not required as
        *     "null instanceof Pet" always  returns false.
        **/
        if ( ! (that instanceof Pet) )
             return false;

       Pet pet = (Pet)that;
       return  this.id == pet.id  && this != null && this.name.equals(pet.name);

    }

    /**
     * fields id & name are used in both equals( ) and
     * hashCode( ) methods.
     */

    @Override
    public int hashCode( ) {
        int hash = 9;
        hash = (31 * hash) + id;
        hash = (31 * hash) + (null == name ? 0 : name.hashCode( ));
        return hash;
    }
}

```

Use Apache's HashCodeBuilder & EqualsBuilder classes to simplify your implementation, especially when you have a large number of member variables. Commonclipse is an eclipse plugin for jakarta commons-lang users. It is very handy for automatic generation of toString( ), hashCode( ), equals(..), and compareTo( ) methods.

```
public final class Pet2 {
    int id;
    String name;

    @Override
    public boolean equals(Object that) {
        if (this == that)
            return true;

        if (!(that instanceof Pet2))
            return false;

        Pet2 pet = (Pet2) that;
        return new EqualsBuilder( ).append(this.id, pet.id).append(
                this.name, pet.name).isEquals( );
    }

    /**
     * both fields id & name are used in equals( ),
     * so both fields must be used in hashCode( ) as well.
     */

    @Override
    public int hashCode( ) {
        //pick 2 hard-coded, odd &  >0 int values as arguments
        return new HashCodeBuilder(1, 31).append(this.id).append(
                this.name).toHashCode( );
    }
}

```

toString() method

Q5. Why or when should you override a toString( ) method?
A5. You can use System.out.println( ) or logger.info(...) to print any object. The toString( ) method of an object gets invoked automatically, when an object

reference is passed in the System.out.println(refPet) or logger.info(refPet) method. However for good results, your class should have a toString( ) method that overrides Object class's default implementation by formatting the object's data in a sensible way and returning a String. Otherwise all that's printed is the class name followed by an "@" sign and then unsigned hexadecimal representation of the hashCode. For example, If the Pet class doesn't override the toString( ) method as shown below, by default Pet@162b91 will be printed via toString( ) default implementation in the Object class.

Q6. Can you override clone( ) and finalize( ) methods in the Object class? How do you disable a clone( ) method?
A6. es, but you need to do it very judiciously. Implementing a properly functioning clone( ) method is complex and it is rarely necessary. You are better off providing some alternative means of object copying through a copy constructor or a static factory method.

Unlike C++ destructors, the finalize( ) method in Java is unpredictable, often dangerous and generally unnecessary. Use finally{} blocks to close any resources or free memory. The finalize( ) method should only be used in rare instances as a safety net or to terminate noncritical native resources. If you do happen to call the finalize( ) method in some rare instances, then remember to follow the following guidelines:

1) You should call the finalize method of the super class in case it has to clean up.

```
protected void finalize( ) throws Throwable {
     try{
          //finalize subclass state
     }
     catch(Throwable t){
          //log the exception
     }
     finally {
         super.finalize( );
     }
}

```

2) You should not depend on the finalize method being called. There is no guarantee that when (or if) objects will be garbage collected and thus no guarantee that the finalize method will be called before the running program terminates.

3) Finally, the code in finalize method might fail and throw exceptions. Catch these exceptions so that the finalize method can continue.

How do you disable a clone( ) method?

```
public final Object clone( ) throws CloneNotSupportedException {
     throw new CloneNotSupportedException( );
}
```

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

‹ 10 Java serialization, cloning, and casting interview Q&As

8 Java Garbage Collection interview Q&As to ascertain your depth of Java knowledge ›

3 Object wait( ) & notify( ) interview Q&As | Java-Success.com

SourceURL: https://www.java-success.com/module-5/3-object-wait-notify-interview-qas/

3 Object wait( ) & notify( ) interview Q&As

5 Object wait() & notify() Interview Q&As

Posted on July 1, 2017 by  Arulkumaran Kumaraswamipillai

Q1. How do Java threads communicate with each other?
A1. In inter process communication, two or more processes communicate with each other using

— Pipes (e.g. Unix processes ps -ef | grep Java).
— Sockets(Java processes using java.io.Reader and Java.io.Writer, RMI, etc).
— Serialized data is passed between processes.

In inter thread communication, you can use both pipes and sockets, and in addition, use shared memory if happens within the same process. In Java, every object extends the java.lang.Object class, which has 3 final methods for inter-thread communication. Those methods are wait( ), notify ( ), and notifyAll( ), and these methods are used to provide an efficient way for threads to communicate with each other.

Here is a very simple example, demonstrating wait/notify methods from the object class communicate with each other via an object.

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 nothing is produced to consume
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().getName() + " after consuming " + count);
16   }
17
18   public synchronized void produce() {
19      count++;             //produce
20      System.out.println(Thread.currentThread().getName() + " after producing " + count);
21      notifyAll();         // notify waiting threads to resume
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 Runnable {
2
3    boolean isConsumer;
4    ConsumerProducer cp;
5
6    public ConsumerProducerTest(boolean isConsumer, ConsumerProducer cp) {
7        this.isConsumer = isConsumer;
8        this.cp = cp;
9    }
10
11   public static void main(String[] args) {
12       ConsumerProducer cp = new ConsumerProducer(); //shared by both threads to communicate
13
14       Thread producer = new Thread(new ConsumerProducerTest(false, cp));
15       Thread consumer = new Thread(new ConsumerProducerTest(true, cp));
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
```

Q2. What must you know about using wait () and notifyAll( ) properly?
A2. Here are 5 things you must know to use wait( ) / notify( ) for inter thread communication

1) Use the same object for calling wait() and notify() method as every object in Java has its own lock. so calling wait() on objA and notify() on obj B will not make any sense, and will not give you inter-thread communication.
2) In order to wait or notify, you need to "own" the object's lock first. So, the method or block must be synchronized.
3) You need to wait( ) or notify() on the same object you have acquired the lock for.
4) use notifyAll() instead of notify() if you expect more than one thread is waiting for lock.
5) Always call wait() method in a loop because if multiple threads are waiting for lock and one of them got lock and reset the condition and the other thread needs to check the condition after they got woken up to see whether they need to wait again or can start processing.

Q3. Why wait/notify methods are in the java.lang.Object class and not in the java.lang.Thread class?
A3.

— In Java, an object itself shared between threads, and each object intrinsically has a lock, which allows thread to share a object, and communicate with each other.
— If wait( ) and notify( ) were on the Thread instead then each thread would have to know the status of every other thread.
— Since wait/notify are in the java.lang.Object class, the threads don't need to have specific knowledge of each other and they can run asynchronously.

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/EE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

7 Object equals Vs == and pass by reference Vs value interview Q&As | Java-Success.com

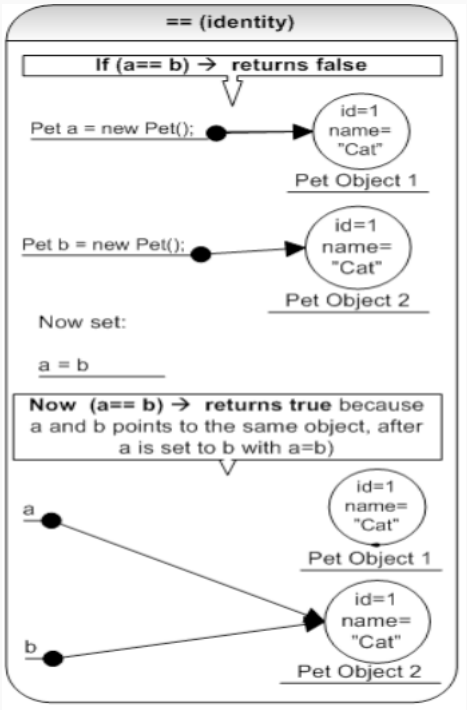
SourceURL: https://www.java-success.com/module-5/7-object-equals-vs-pass-reference-vs-value/

7 Object equals Vs == and pass by reference Vs value interview Q&As

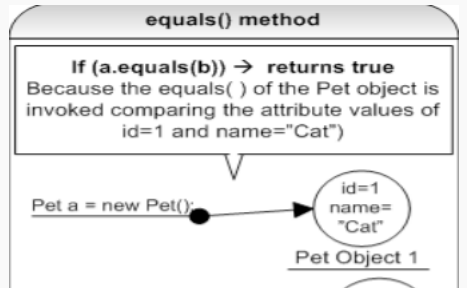
Posted on July 1, 2017 by Arulkumaran Kumaraswamipillai

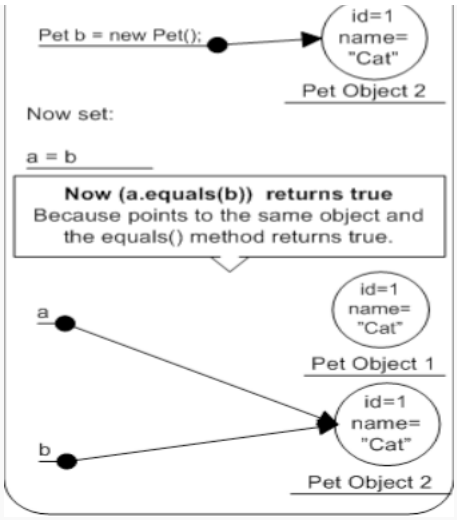
Q1. What is the difference between "==" and equals(..) method when comparing 2 objects?
A1.It is important to understand the difference between identity (i.e. ==) comparison, which is a shallow comparison that compares only the object references, and the equals( ) comparison, which is a **deeper comparison** that compares the object attributes. The diagram below explains the difference between the two. There are some exceptional conditions when using primitives, String objects, and enums.

If the equals(..) method is not overridden, then the Object class's default implementation is invoked, which only compares the object references. Invoking the equals(..) method of the Object class is equivalent to making a shallow comparison with "==". This is why it is imperative to override the **equals( )** method, and the **hashCode( )** methods in your custom classes like Pet. The Java API objects like String, and the wrapper classes like Integer, Double, Float, etc override the equals(..), hashCode( ), and the toString(..) methods.



Java identity ==





Deeper equals( ) compares values by invoking equals( ) method

If you were to implement the equals(...) and hashCode(...) methods:

```
public final class Pet {
    int id;
    String name;

    @Override
    public boolean equals(Object that){
        if ( this == that ) return true;

        if ( ! (that instanceof Pet) ){
             return false;
  }

       Pet pet = (Pet)that;
       return  this.id == pet.id  && this != null && this.name.equals(pet.name);
    }

    @Override
    public int hashCode( ) {
        int hash = 9;
        hash = (31 * hash) + id;
        hash = (31 * hash) + (null == name ? 0 : name.hashCode( ));
        return hash;
    }
}

```

You can learn more at "**Sorting objects in Java interview Q&As**"

Q2. What happens when you run the following code?

```
Boolean b1  = new Boolean(false);
Boolean b2  = Boolean.FALSE;

 if(b1 == b2) {
      System.out.println("Equal");
 }
 else{
      System.out.println("Not Equal");
 }

```

A2. Prints "Not Equal".

The == is a shallow comparison that only compares the references. The references are not equal. If you want to print "Equal", perform a deeper comparison as shown below, which compares the values.

```

If (b1.equals(b2)){
    System.out.println("Equal");//gets printed
}
else {
    System.out.println("Not Equal");
```

```
}
```

Or, you need to take advantage of the **flyweight design pattern** that reuses objects.

```
Boolean b1 = Boolean.valueOf("false"); // create a false object if not already present
Boolean b2 = Boolean.FALSE; //points to the same object as above
```

or

```
Boolean b1 = Boolean.valueOf("false"); // create a false object if not already present
Boolean b2 = Boolean.valueOf("false"); //points to the same object as above
```

Q3. Can you discuss the output of the following code?

```
public class PrimitiveAndObjectEquals {

    public static void main(String[ ] args) {
        int a = 5;
        int b = 5;

        Integer c = new Integer(5);
        Integer d = new Integer(5);

        if (a == b) {  //Line 1
            System.out.println("primitives a and b are ==");
        }

        if (c == d) {  //Line 2
            System.out.println("Objects c and d are ==");
        }

        if (c.equals(d)) {  //Line 3
            System.out.println("Objects c and d are equals( )");
        }

        if (a == d) { //Line 4
           System.out
               .println("Primitive a and Object d are == due to auto unboxing");
        }
    }
}
```

A3. **Output is:**

```
primitives a and b are ==
Objects c and d are equals( )
Primitive a and Object d are == due to auto unboxing
```

1) Line 1 is printed as both a and b are primitive data types, and primitives are compared with == as they don't have an equals() method.

2) Line 2 will not get printed as they are comparing the object references (shallow comparison). Line 3 will get printed as they are comparing the actual values (deep er comparison).

3) Line 4 is printed because the object reference "d" is auto-unboxed to a primitive int value and then compared with the primitive reference "a". This also illustrates a hidden chance of a **NullPointerException** being thrown if the reference "d" were to be null.

Q4. Can you discuss the output of the following code?

```
public class EnumEquals {

  public enum Action {START, STOP, CONTINUE}

  private static Action action = Action.STOP;

  public static void main(String[ ] args) {

      if(Action.STOP == action){
          System.out.println("Enumurations can be compared to ==.");
      }

      if(Action.STOP.equals(action)){
```

```
        if(Action.STOP.equals(action)){
            System.out.println("Enumurations can be compared to equals( ) also.");
        }
    }
}
```

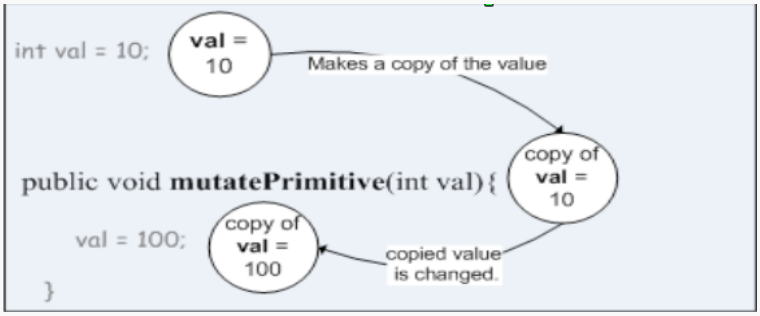
A4. Output is:

```
Enumurations can be compared to ==.
Enumurations can be compared to equals( ) also.
```

The best practice is to use the referential == for enums.

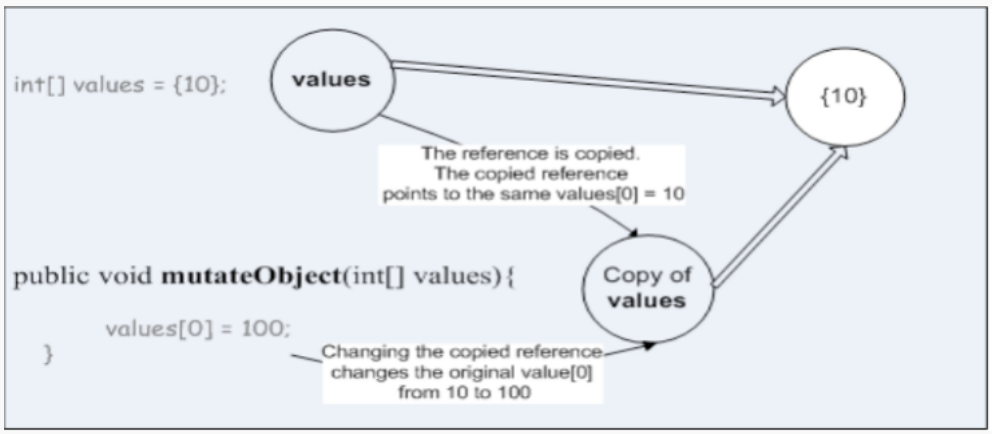
Q5. Explain the statement Java is always pass by value?
A5. Other languages use pass-by-reference or pass-by-pointer. But in Java, no matter what type of argument (i.e. a primitive variable or an object reference) you pass, the corresponding parameter will get a copy of that data, which is exactly how pass-by-value (i.e. copy-by-value) works. Even though the definition is quite straight forward, the way the primitives and object references behave when passed by value, will be different.

For example, If the passed in argument was a primitive value like int, char, etc, the passed in primitive value is copied to the method parameter. Modifying the copied parameter will not modify the original primitive value.



pass-by-value primitive variables like int, long, etc

On the contrary, if the passed in argument was an object reference, the passed in reference is copied to the method parameter. The copied reference will still be pointing to the same object. So if you modify the object value through the copied reference, the original object will be modified.



pass by value for objects like int[], Pet, Car, etc

Q6. The value of Point p before the following method calls is (10,20). What will be the value of Point p after executing the following method calls?

Scenario 1:

```
static void mutatePoint(Point p) {
     p.x =  50;
     p.y=100;
```

```
}
```

Scenario 2:

```
static void mutatePoint(Point p) {
     p = new Point(50,100);
}
```

A6.

Scenario 1:

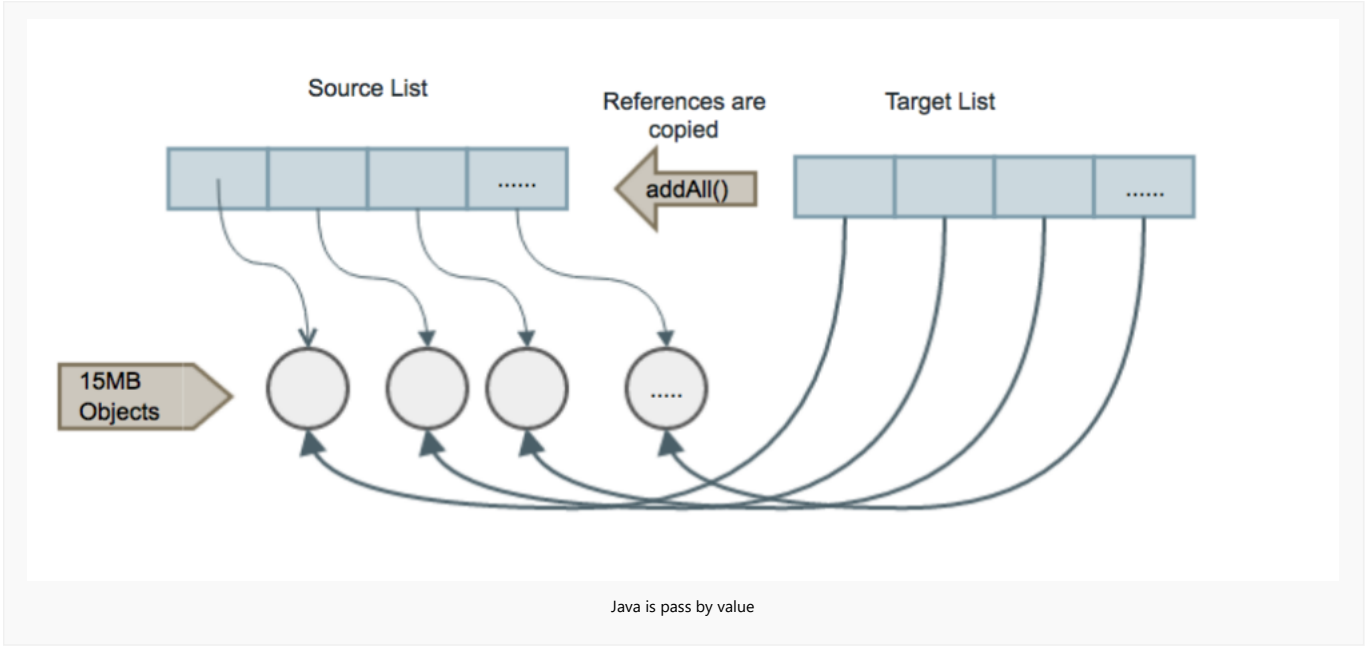
Point p = (50,100), as the copied reference will still be pointing and modifying the original Point (10,2 0) object through the mutatePoint( ) method.

Scenario 2:

Point p = (10,20), as the copied reference will be creating and pointing to the newly created Point (50, 100) object.

Q7. If there is a source array list with 15MB of data, and then you create a new target empty array list and copy the source to target with target.addAll(source). How much memory will be consumed after invoking the addAll(...) method?

A7. The memory will still be 15MB because of "**pass-by-value**" where new objects are not created when addAll(...) is invoked. Only the references are copied, but the copied references will still be pointing to the source list objects. For example,



Java is pass by value

```
package com.test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class PassByReference {

	public static void main(String[] args) {

		List<String> source = Arrays.asList("One", "Two", "Three", "Four"); //say 15MB

		List<String> target = new ArrayList<String>();

		target.addAll(source);

		//memory will still be 15MB, why?

		System.out.println(source.get(0) == target.get(0)); //outputs true.

		//This is because source and target lists reference the same objects.
		//No new objects are created by addAll(...)
```

```
            //Only the references are copied
        }
}
```

print

Arulkumaran Kumaraswamipillai

Mechanical Engineer to freelance Java developer within 3 years. Freelancing since 2003 for the major banks, telecoms, retail & government organizations. Attended 150+ Java job interviews, and most often got 3-6 job offers to choose from. Published Java/JEE books via Amazon.com in 2005, and sold 35K+ copies. Books are outdated and replaced with this online Java training. Join my LinkedIn group.

Have you completed this unit? Then mark this unit as completed.

Mark as Completed

« Previous Unit | Next Unit »

‹ 3 Object wait( ) & notify( ) interview Q&As

10 Java immutable objects interview Q&As ›

10 Java immutable objects interview Q&As | Java-Success.com

SourceURL: https://www.java-success.com/module-5/10-java-immutable-objects-interview-qas/

10 Java immutable objects interview Q&As

Posted on July 1, 2017 by  Arulkumaran Kumaraswamipillai

Best Practice: "Classes should be immutable unless there's a very good reason to make them mutable….If a class cannot be made immutable, limit its mutability as much as possible." — by Joshua Bloch

Q1. What is an immutable object?
Q2. Immutable objects are objects whose state (the object's data) cannot change after construction. Examples of immutable objects from the JDK include String and wrapper classes like Integer, Double, Character, etc.

Q2.How do you create an immutable type?
A2.

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

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

2. Make the fields private and final.

3. Don't provide any methods that can change the state of the immutable object in any way from outside the object – 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, and the immutable class should have exclusive access to fields that contain references to other mutable objects like arrays, collections and mutable classes like Date by:

a) Declaring the mutable references as private.
b) Not returning or exposing the mutable references to the caller. This can be done by defensively copying the objects by deeply cloning them.

Example: satisfying the above conditions

```
public class User {
    private final String firstName; //final, and String class itself is immutable
    private final String lastName;  //final, and String class itself is immutable

    // constructor is private
    private User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    //factory method
    public static User getInstance(String firstName, String lastName) {
        return new User(firstName, lastName);
    }

    //only getters, no setters

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

}
```

Q3. Is the following class immutable?

```
import java.util.Arrays;

public final class MyImmutable {

    private final Integer[ ] myArray;

    public MyImmutable(Integer[ ] anArray) {
          this.myArray = anArray;
    }

    public Integer[ ] getMyArray( ) {
        return myArray;
    }

    //....equals(), hashCode(), etc

    @Override
    public String toString() {
        return Arrays.toString(myArray);
    }
}
```

A3. No. The above class is **not immutable** as it fails **#4** condition where the "myArray" reference can escape, and mutated from outside as demonstrated below.

```
public class MyImmutableTest {

    public static void main(String[] args) {
        Integer[] array1 = {1,2,3};
        MyImmutable mi = new MyImmutable(array1);
        System.out.println("before modifying: " + mi);

        mi.getMyArray()[2] = 4; //change 3 to 4
        System.out.println("after modifying: " + mi);
    }
```

```
}
```

Output:

```
before modifying: [1, 2, 3]
after modifying: [1, 2, 4]
```

FIX: “myArray” reference is deeply copied, and can’t escape

```
import java.util.Arrays;

public final class MyImmutable {

    private final Integer[] myArray;

    public MyImmutable(Integer[] anArray) {
        this.myArray = anArray.clone(); //cloned array assigned
    }

    public Integer[] getMyArray() {
        return myArray.clone(); // cloned array is returned
    }

    // ....equals(), hashCode(), etc

    @Override
    public String toString() {
        return Arrays.toString(myArray);
    }
}
```

If you run the “MyImmutableTest” again,

```
before modifying: [1, 2, 3]
after modifying: [1, 2, 3]
```

Q4. What are the advantages of immutable objects?
A4.

1) 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.

2) 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.

3) 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.

4) Eliminates the need for class invariant check once constructed.

5) Allow hashCode( ) method to use lazy initialization, by caching its return value.

6) Cloning is not required.

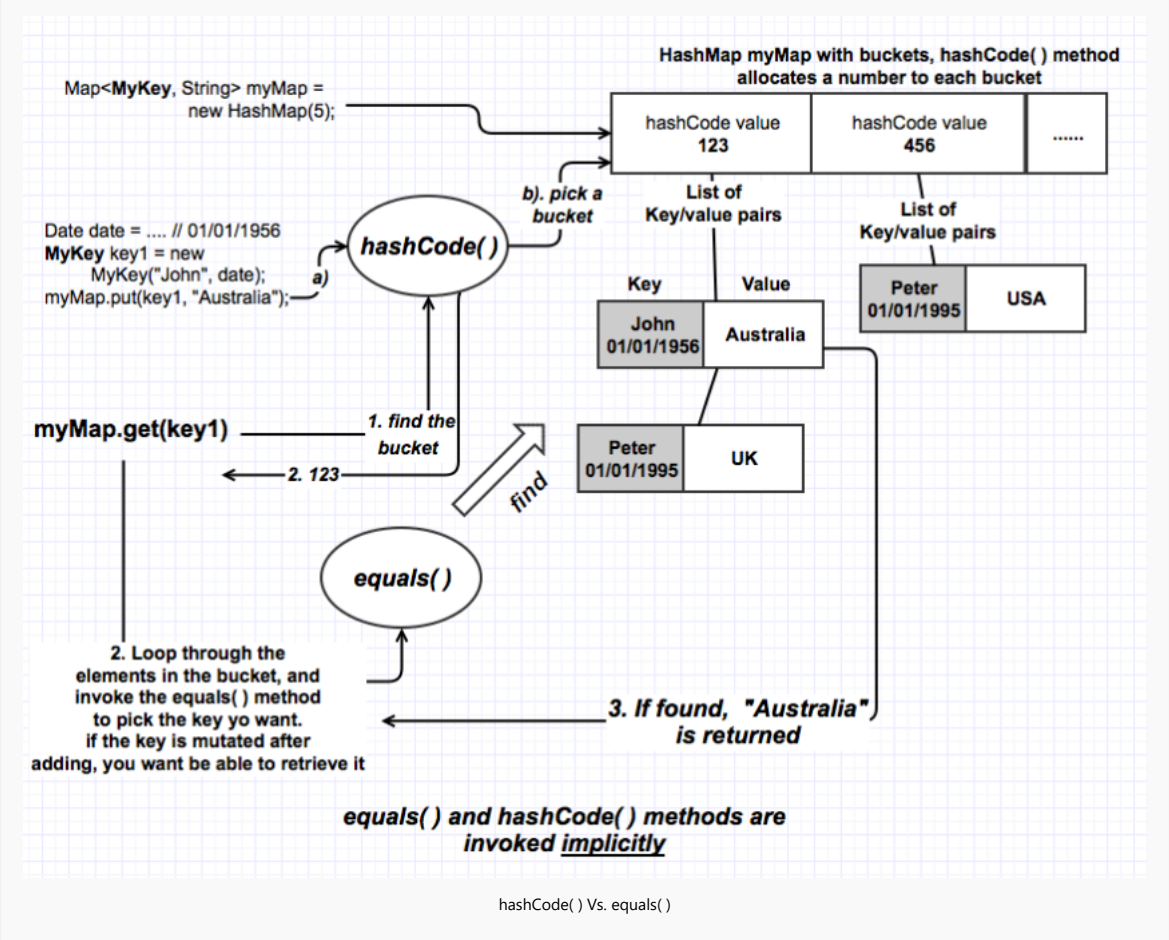
7) Simpler to construct, use, and test due to its deterministic state.

Q5. Why is it a best practice to implement the user defined key class as an immutable object?
A5. Immutable objects generally make the best map keys as the keys cannot be modified once they have been added to the Map. In general *String*, *Integer*, or *Long* are used as keys, which are immutable objects. If you define your own key class, make sure that they are immutable. Otherwise, if the keys are accidentally modified after adding to a Map, you will never be able to retrieve the stored value as the key values have been changed. This is a common pitfall many Java developers, especially beginners fall for.

Example:

Immutable key class

```
import java.util.Date;

public final class MyKey {

    private final String name;
    private final Date myDate;

    public MyDiary(Date aDate) {
        this.myDate = new Date(aDate.getTime());//defensive copying by not exposing the "myDate" reference
    }

    public Date getDate() {
        return new Date(myDate.getTime()); //defensive copying by not exposing the "myDate"
    }

    public String getName() {
        return name; //String is immutable
    }

    //...equals( ), hashCode(), etc
}
```



hashCode( ) Vs. equals( )

As shown, when Maps are used in Java, the **equals( )** *and* ***hashCode( )*** methods are implicitly invoked. If these methods are incorrectly implemented or the keys are modified once added to the map, then unpredictable behavior will be experienced, and these behaviors are harder to debug. The hashCode( ) and equals( ) methods are implicitly invoked to determine where the key is stored, and to retrieve the value for a particular key respectively. More than one key/value pairs can be stored in the same bucket.

The *hashCode( )* method does not give a unique value each time. Its duty is to spread out the numbers so that your key value pairs get spread out in multiple buckets. So, always remember this.

The *hashCode( )* method is used to store the key/value in a bucket, and both the hashCode() and equals() methods are called to retrieve the stored key/value. If they are implemented inconsistently or the key is mutated, then the stored object cannot be retrieved as the returned values of these methods will vary in between different invocations. To be more specific, the hashCode( ) method is called to determine the key index (aka the bucket) of the array, and the equals( ) method is called to retrieve the exact key from the list of keys belonging to that particular key index (or bucket) as the same bucket will be holding multiple keys linked to multiple values. Remember the contract between these two methods?

"If 2 objects are equal, they must return the same hashCode( ) value, but the reverse is not true. Which means, if 2 objects return the same hashCode( ) value does not mean that those 2 objects are equal( )".

Q6. How would you defensively copy a Date field in your immutable class?
A6.

```
public final class MyDiary {

    private final Date myDate;

    public MyDiary(Date aDate) {
        this.myDate = new Date(aDate.getTime( )); //defensive copying by not exposing the "myDate" reference
    }

    public Date getDate( ) {
        return new Date(myDate.getTime( )); //defensive copying by not exposing the "myDate"
    }
}
```

Q7. How will you prevent the caller from adding or removing elements from a collection of pets?
A7.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class PetCage {
    private final List<Pet> pets;

    public PetCage(List<Pet> pets) {
          this.pets = Collections.unmodifiableList(new ArrayList<Pet>(pets));
    }

    public List<Pet> getPets( ) {
          return pets;
    }
}
```

It ensures that you cannot add or remove pets. However, there is no guarantee that the pets are also immutable. To make this instance fully immutable, the Pet instance itself must be immutable or use the decorator pattern as a wrapper around each of the pets to make them also immutable. For example, The *Integer* **wrapper class provides immutability to mutable primitive int value** . You could also defensively deep copy the list of pets in the constructor and getPets( ) method.

Q8. How about data that needs to be mutable, but less frequently? Is there any way to obtain the benefits of immutability with the added benefit of thread-safety for data that changes less frequently?

A8. The **Copy-On-Write** collections like *CopyOnWriteArrayList* and *CopyOnWriteArraySet* classes introduced from JSE 5.0 util.concurrent package are good examples of how to harness the power of immutability whist permitting occasional modifications for infrequently changing data. CopyOnWriteArrayList behaves much like the ArrayList class, except that when the list is modified, instead of mutating the underlying array, a new array is created and the old array is discarded. CopyOnWriteArrayList is designed for cases where:

— reads hugely outnumber writes.
— the array is small (or writes are very infrequent).
— the caller genuinely needs the functionality of a list rather than an array.

When you obtain an iterator, which holds a reference to the underlying array, the array referenced by the iterator is effectively immutable and therefore can be traversed without synchronization or risk of concurrent modification. This eliminates the need to either clone the list before traversal or synchronize on the list during traversal. If reads are much more frequent than insertions or removals, which is the case very often, the Copy-on-Write collections and *ConcurrentHashMaps*offer better performance and development convenience. The development convenience is provided not needing to deal with synchronization, deep cloning, or "ConcurrentModificationException". The ConcurrentModificationException is generally thrown by an ArrayList, HashSet, or a HashMap implementation when you try to remove an object from a collection while iterating over it.

Q9. Can builder design pattern be used to create immutable objects?
A9. Yes.

Q10. Can you give a builder design pattern example to create immutable objects?
A10. **Example**: Builder pattern and immutability in Java.