(/)

Java Class Structure and Initialization Interview Questions

Last modified: November 11, 2018

by baeldung (https://www.baeldung.com/author/baeldung/)

Java (https://www.baeldung.com/category/java/)

Interview (https://www.baeldung.com/tag/interview/)

This article is part of a series:

1. Introduction

Class structure and initialization are the basics that every Java programmer should be familiar with. This article provides answers to some of the interview questions on the topic that you may encounter.

Q1. Describe the meaning of the *final* keyword when applied to a class, method, field or a local variable.

The *final* keyword has multiple different meanings when applied to different language constructs:

- A *final* class is a class that cannot be subclassed
- A *final* method is a method that cannot be overridden in subclasses
- A *final* field is a field that has to be initialized in the constructor or initializer block and cannot be modified after that
- A *final* variable is a variable that may be assigned (and has to be assigned) only once and is never modified after that

Q2. What is a *default* method?

Prior to Java 8, interfaces could only have abstract methods, i.e. methods without a body. Starting with Java 8, interface methods can have a default implementation. If an implementing class does not override this method, then the default implementation is used. Such methods are suitably marked with a *default* keyword.

One of the prominent use cases of a *default* method is adding a method to an existing interface. If you don't mark such interface method as *default*, then all existing implementations of this interface will break. Adding a method with a *default* implementation ensures binary compatibility of legacy code with

the new version of this interface.

A good example of this is the *Iterator* interface which allows a class to be a target of the for-each loop. This interface first appeared in Java 5, but in Java 8 it received two additional methods, *forEach*, and *spliterator*. They are defined as default methods with implementations and thus do not break backward compatibility:

```
public interface Iterable<T> {

    Iterator<T> iterator();

    default void forEach(Consumer<? super T> action) { /* */ }

    default Spliterator<T> spliterator() { /* */ }
}
```

Q3. What are *static* class members?

Static fields and methods of a class are not bound to a specific instance of a class. Instead, they are bound to the class object itself. The call of a *static* method or addressing a *static* field is resolved at compile time because, contrary to instance methods and fields, we don't need to walk the reference and determine an actual object we're referring to.

Q4. May a class be declared abstract if it does not have any abstract members? What could be the purpose of such class?

Yes, a class can be declared *abstract* even if it does not contain any *abstract* members. As an abstract class, it cannot be instantiated, but it can serve as a root object of some hierarchy, providing methods that can be useful to its implementations.

Q5. What is Constructor Chaining?

Constructor chaining is a way of simplifying object construction by providing multiple constructors that call each other in sequence.

The most specific constructor may take all possible arguments and may be used for the most detailed object configuration. A less specific constructor may call the more specific constructor by providing some of its arguments with default values. At the top of the chain, a no-argument constructor could instantiate an object with default values.

Here's an example with a class that models a discount in percents that are available within a certain amount of days. The default values of 10% and 2 days are used if we don't specify them when using a no-arg constructor:

```
1  public class Discount {
2
3      private int percent;
4
5      private int days;
6
7      public Discount() {
8          this(10);
9      }
10
11     public Discount(int percent) {
12         this(percent, 2);
13     }
14
15     public Discount(int percent, int days) {
16         this.percent = percent;
17         this.days = days;
18     }
19
20 }
```

Q6. What is overriding and overloading of methods? How are they different?

Overriding of a method is done in a subclass when you define a method with the same signature as in superclass. This allows the runtime to pick a method depending on the actual object type that you call the method on. Methods *toString*, *equals*, and *hashCode* are overridden quite often in subclasses.

Overloading of a method happens in the same class. Overloading occurs when you create a method with the same name but with different types or number of arguments. This allows you to execute a certain code depending on the types of arguments you provide, while the name of the method remains the same.

Here's an example of overloading in the *java.io.Writer* abstract class. The following methods are both named *write*, but one of them receives an *int* while another receives a *char* array.

```
1  public abstract class Writer {
2
3      public void write(int c) throws IOException {
4          // ...
5      }
6
7      public void write(char cbuf[]) throws IOException {
8          // ...
9      }
10
11 }
```

Q7. Can you override a *static* method?

No, you can't. By definition, you can only override a method if its implementation is determined at runtime by the type of the actual instance (a process known as the dynamic method lookup). The *static* method implementation is determined at compile time using the type of the reference, so overriding would not make much sense anyway. Although you can add to subclass a *static* method with the exact same signature as in superclass, this is not technically overriding.

Q8. What is an immutable class, and how can you create one?

An instance of an immutable class cannot be changed after it's created. By changing we mean mutating the state by modifying the values of the fields of the instance. Immutable classes have many advantages: they are thread-safe, and it is much easier to reason about them when you have no mutable state to consider.

To make a class immutable, you should ensure the following:

- All fields should be declared *private* and *final*; this infers that they should be initialized in the constructor and not changed ever since;
- The class should have no setters or other methods that mutate the values of the fields;
- All fields of the class that were passed via constructor should either be also immutable, or their values should be copied before field initialization (or else we could change the state of this class by holding on to these values and modifying them);
- The methods of the class should not be overridable; either all methods should be *final*, or the constructor should be *private* and only invoked via *static* factory method.

Q9. How do you compare two *enum* values: with *equals()* or with *==*?

Actually, you can use both. The *enum* values are objects, so they can be compared with *equals()*, but they are also implemented as *static* constants under the hood, so you might as well compare them with `==`. This is mostly a matter of code style, but if you want to save character space (and possibly skip an unneeded method call), you should compare enums with `==`.

Q10. What is an initializer block? What is a *static* initializer block?

An initializer block is a curly-braced block of code in the class scope which is executed during the instance creation. You can use it to initialize fields with something more complex than in-place initialization one-liners.

Actually, the compiler just copies this block inside every constructor, so it is a nice way to extract common code from all constructors.

A static initializer block is a curly-braced block of code with the *static* modifier in front of it. It is executed once during the class loading and can be used for initializing static fields or for some side effects.

Q11. What is a marker interface? What are the notable examples of marker interfaces in Java?

A marker interface is an interface without any methods. It is usually implemented by a class or extended by another interface to signify a certain property. The most widely known marker interfaces in standard Java library are the following:

- *Serializable* is used to explicitly express that this class can be serialized;
- *Cloneable* allows cloning objects using the *clone* method (without *Cloneable* interface in place, this method throws a *CloneNotSupportedException*);
- *Remote* is used in RMI to specify an interface which methods could be called remotely.

Q12. What is a singleton and how can it be implemented in Java?

Singleton is a pattern of object-oriented programming. A singleton class may only have one instance, usually globally visible and accessible.

There are multiple ways of creating a singleton in Java. The following is the most simple example with a *static* field that is initialized in-place. The initialization is thread-safe because *static* fields are guaranteed to be initialized in a thread-safe manner. The constructor is *private*, so there is no way for outer code to create more than one instance of the class.

```
1  public class SingletonExample {
2
3      private static SingletonExample instance = new SingletonExample();
4
5      private SingletonExample() {}
6
7      public static SingletonExample getInstance() {
8          return instance;
9      }
10 }
```

But this approach could have a serious drawback — the instance would be instantiated when this class is first accessed. If initialization of this class is a heavy operation, and we would probably like to defer it until the instance is actually needed (possibly never), but at the same time keep it thread-safe. In this case, we should use a technique known as **double-checked locking**.

Q13. What is a var-arg? What are the restrictions on a var-arg? How can you use it inside the method body?

Var-arg is a variable-length argument for a method. A method may have only one var-arg, and it has to come last in the list of arguments. It is specified as a type name followed by an ellipsis and an argument name. Inside the method body, a var-arg is used as an array of specified type.

Here's an example from the standard library — the *Collections.addAll* method that receives a collection, a variable number of elements, and adds all elements to the collection:

```
public static <T> boolean addAll(
  Collection<? super T> c, T... elements) {
    boolean result = false;
    for (T element : elements)
        result |= c.add(element);
    return result;
}
```

Q14. Can you access an overridden method of a superclass? Can you access an overridden method of a super-superclass in a similar way?

To access an overridden method of a superclass, you can use the *super* keyword. But you don't have a similar way of accessing the overridden method of a super-superclass.

As an example from the standard library, *LinkedHashMap* class extends *HashMap* and mostly re-uses its functionality, adding a linked list over its values to preserve iteration order. *LinkedHashMap* re-uses the *clear* method of its superclass and then clears head and tail references of its linked list:

```
public void clear() {
    super.clear();
    head = tail = null;
}
```

Next »

Java 8 Interview Questions(+ Answers) (https://www.baeldung.com/java-8-interview-questions)

« Previous

Java Concurrency Interview Questions (+ Answers) (https://www.baeldung.com/java-concurrency-interview-questions)

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)

REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)

JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)

SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)

PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)

JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)

HTTP CLIENT (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)