(/)

Java Type System 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

Java Type System is a topic often brought up on technical interviews for Java developers. This article reviews some important questions that are asked most often and may be tricky to get right.

2. Questions

Q1. Describe the place of the *Object* class in the type hierarchy. Which types inherit from *Object*, and which don't? Do arrays inherit from *Object*? Can a lambda expression be assigned to an Object variable?

The *java.lang.Object* is at the top of the class hierarchy in Java. All classes inherit from it, either explicitly, implicitly (when the *extends* keyword is omitted from the class definition) or transitively via the chain of inheritance.

However, there are eight primitive types that do not inherit from *Object*, namely *boolean*, *byte*, *short*, *char*, *int*, *float*, *long* and *double*.

According to the Java Language Specification, arrays are objects too. They can be assigned to an *Object* reference, and all *Object* methods may be called on them.

Lambda expressions can't be assigned directly to an *Object* variable because *Object* is not a functional interface. But you can assign a lambda to a functional interface variable and then assign it to an *Object* variable(or simply assign it to an Object variable by casting it to a functional interface at the same time).

Q2. Explain the difference between primitive and reference types.

Reference types inherit from the top *java.lang.Object* class and are themselves inheritable (except *final* classes). Primitive types do not inherit and cannot be subclassed.

Primitively typed argument values are always passed via the stack, which means they are passed by value, not by reference. This has the following implication: changes made to a primitive argument value inside the method do not propagate to the actual argument value.

Primitive types are usually stored using the underlying hardware value types.

For instance, to store an *int* value, a 32-bit memory cell can be used. Reference types introduce the overhead of object header which is present in every instance of a reference type.

The size of an object header can be quite significant in relation to a simple numeric value size. This is why the primitive types were introduced in the first place — to save space on object overhead. The downside is that not everything in Java technically is an object — primitive values do not inherit from *Object* class.

Q3. Describe the different primitive types and the amount of memory they occupy.

Java has 8 primitive types:

- *boolean* — logical *true/ false* value. The size of boolean is not defined by the JVM specification and can vary in different implementations.
- *byte* — signed 8-bit value,
- *short* — signed 16-bit value,
- *char* — unsigned 16-bit value,
- *int* — signed 32-bit value,

- *long* — signed 64-bit value,
- *float* — 32-bit single precision floating point value corresponding to the IEEE 754 standard,
- *double* — 64-bit double precision floating point value corresponding to the IEEE 754 standard.

Q4. What is the difference between an abstract class and an interface? What are the use cases of one and the other?

An abstract class is a *class* with the *abstract* modifier in its definition. It can't be instantiated, but it can be subclassed. The interface is a type described with *interface* keyword. It also cannot be instantiated, but it can be implemented.

The main difference between an abstract class and an interface is that a class can implement multiple interfaces, but extend only one abstract class.

An *abstract* class is usually used as a base type in some class hierarchy, and it signifies the main intention of all classes that inherit from it.

An *abstract* class could also implement some basic methods needed in all subclasses. For instance, most map collections in JDK inherit from the *AbstractMap* class which implements many methods used by subclasses (such as the *equals* method).

An interface specifies some contract that the class agrees to. An implemented interface may signify not only the main intention of the class but also some additional contracts.

For instance, if a class implements the *Comparable* interface, this means that instances of this class may be compared, whatever the main purpose of this class is.

Q5. What are the restrictions on the members (fields and methods) of an interface type?

An interface can declare fields, but they are implicitly declared as *public*, *static* and *final*, even if you don't specify those modifiers. Consequently, you can't explicitly define an interface field as *private*. In essence, an interface may only have constant fields, not instance fields.

All methods of an interface are also implicitly *public*. They also can be either (implicitly) *abstract*, or *default*.

Q6. What is the difference between an inner class and a static nested class?

Simply put – a nested class is basically a class defined inside another class.

Nested classes fall into two categories with very different properties. An inner class is a class that can't be instantiated without instantiating the enclosing class first, i.e. any instance of an inner class is implicitly bound to some instance of the enclosing class.

Here's an example of an inner class – you can see that it can access the reference to the outer class instance in the form of *OuterClass1.this* construct:

```
public class OuterClass1 {

    public class InnerClass {

        public OuterClass1 getOuterInstance() {
            return OuterClass1.this;
        }

    }

}
```

To instantiate such inner class, you need to have an instance of an outer class:

```
OuterClass1 outerClass1 = new OuterClass1();
OuterClass1.InnerClass innerClass = outerClass1.new InnerClass();
```

Static nested class is quite different. Syntactically it is just a nested class with the *static* modifier in its definition.

In practice, it means that this class may be instantiated as any other class, without binding it to any instance of the enclosing class:

```
public class OuterClass2 {

    public static class StaticNestedClass {
    }

}
```

To instantiate such class, you don't need an instance of outer class:

```
OuterClass2.StaticNestedClass staticNestedClass = new OuterClass2.StaticNestedClass();
```

Q7. Does Java have multiple inheritance?

Java does not support the multiple inheritance for classes, which means that a class can only inherit from a single superclass.

But you can implement multiple interfaces with a single class, and some of the methods of those interfaces may be defined as *default* and have an implementation. This allows you to have a safer way of mixing different functionality in a single class.

Q8. What are the wrapper classes? What is autoboxing?

For each of the eight primitive types in Java, there is a wrapper class that can be used to wrap a primitive value and use it like an object. Those classes are, correspondingly, *Boolean*, *Byte*, *Short*, *Character*, *Integer*, *Float*, *Long*, and *Double*. These wrappers can be useful, for instance, when you need to put a primitive value into a generic collection, which only accepts reference objects.

```
1 List<Integer> list = new ArrayList<>();
2 list.add(new Integer(5));
```

To save the trouble of manually converting primitives back and forth, an automatic conversion known as autoboxing/auto unboxing is provided by the Java compiler.

```
1 List<Integer> list = new ArrayList<>();
2 list.add(5);
3 int value = list.get(0);
```

Q9. Describe the difference between equals() and ==

The == operator allows you to compare two objects for "sameness" (i.e. that both variables refer to the same object in memory). It is important to remember that the *new* keyword always creates a new object which will not pass the == equality with any other object, even if they seem to have the same value:

```
String string1 = new String("Hello");
String string2 = new String("Hello");

assertFalse(string1 == string2);
```

Also, the == operator allows to compare primitive values:

```
int i1 = 5;
int i2 = 5;

assertTrue(i1 == i2);
```

The *equals()* method is defined in the *java.lang.Object* class and is, therefore, available for any reference type. By default, it simply checks that the object is the same via the == operator. But it is usually overridden in subclasses to provide the specific semantics of comparison for a class.

For instance, for *String* class this method checks if the strings contain the same characters:

```
String string1 = new String("Hello");
String string2 = new String("Hello");

assertTrue(string1.equals(string2));
```

Q10. Suppose you have a variable that references an instance of a *Class* type. How do you check that an object is an instance of this class?

You cannot use *instanceof* keyword in this case because it only works if you provide the actual class name as a literal.

Thankfully, the *Class* class has a method *isInstance* that allows checking if an object is an instance of this class:

```
1 Class<?> integerClass = new Integer(5).getClass();
2 assertTrue(integerClass.isInstance(new Integer(4)));
```

Q11. What is an anonymous class? Describe its use case.

Anonymous class is a one-shot class that is defined in the same place where its instance is needed. This class is defined and instantiated in the same place, thus it does not need a name.

Before Java 8, you would often use an anonymous class to define the implementation of a single method interface, like *Runnable*. In Java 8, lambdas are used instead of single abstract method interfaces. But anonymous classes still have use cases, for example, when you need an instance of an interface with multiple methods or an instance of a class with some added features.

Here's how you could create and populate a map:

```
Map<String, Integer> ages = new HashMap<String, Integer>(){{
    put("David", 30);
    put("John", 25);
    put("Mary", 29);
    put("Sophie", 22);
}};
```

Next »

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

« Previous

Java Collections Interview Questions (https://www.baeldung.com/java-collections-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)