(/)

Java Generics Interview Questions (+Answers)

Last modified: November 23, 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

In this article, we'll go through some example Java generics interview questions and answers.

Generics are a core concept in Java, first introduced in Java 5. Because of this, nearly all Java codebases will make use of them, almost guaranteeing that a developer will run into them at some point. This is why it's essential to understand them correctly, and is why they are more than likely to be asked about during an interview process.

2. Questions

Q1. What is a Generic Type Parameter?

Type is the name of a *class* or *interface*. As implied by the name, **a generic type parameter is when a *type* can be used as a parameter in a class, method or interface declaration.**

Let's start with a simple example, one without generics, to demonstrate this:

```
1 public interface Consumer {
2     public void consume(String parameter)
3 }
```

In this case, the method parameter type of the *consume()* method is *String*. It is not parameterized and not configurable.

Now let's replace our *String* type with a generic type that we will call *T*. It is named like this by convention:

```
public interface Consumer<T> {
    public void consume(T parameter)
}
```

When we implement our consumer, we can provide the *type* that we want it to consume as an argument. This is a generic type parameter:

```
public class IntegerConsumer implements Consumer<Integer> {
    public void consume(Integer parameter)
}
```

In this case, now we can consume integers. We can swap out this *type* for whatever we require.

Q2. What Are Some Advantages of Using Generic Types?

One advantage of using generics is avoiding costs and provide type safety. This which is particularly useful when working with collections. Let's demonstrate this:

Let's demonstrate this:

```
List list = new ArrayList();
list.add("foo");
Object o = list.get(1);
String foo = (String) foo;
```

In our example, the element type in our list is unknown to the compiler. This means that the only thing that can be guaranteed is that it is an object. So when we retrieve our element, an object is what we get back. As the authors of the code, we know it's a *String,* but we have to cast our object to one to fix the problem explicitly. This produces a lot of noise and boilerplate.

Next, if we start to think about the room for manual error, the casting problem gets worse. What if we accidentally had an integer in our list?

```
1  list.add(1)
2  Object o = list.get(1);
3  String foo = (String) foo;
```

In this case, we would get a *ClassCastException (https://docs.oracle.com/javase/8/docs/api/java/lang/ClassCastException.html)* at runtime, as an *Integer* cannot be cast to *String.*

Now, let's try repeating ourselves, this time using generics:

```
1  List<String> list = new ArrayList<>();
2  list.add("foo");
3  String o = list.get(1);    // No cast
4  Integer foo = list.get(1); // Compilation error
```

As we can see, **by using generics we have a compile type check which prevents *ClassCastExceptions* and removes the need for casting.**

The other advantage is to avoid code duplication. Without generics, we have to copy and paste the same code but for different types. With generics, we do not have to do this. We can even implement algorithms which apply to generic types.

Q3. What is Type Erasure?

It's important to realize that generic type information is only available to the compiler, not the JVM. In other words**, type erasure means that generic type information is not available to the JVM at runtime, only compile time**.

The reasoning behind major implementation choice is simple – preserving backward compatibility with older versions of Java. When a generic code is compiled into bytecode, it will be as if the generic type never existed. This means that compilation will:

1. Replace generic types with objects
2. Replace bounded types (More on these in a later question) with the first bound class
3. Insert the equivalent of casts when retrieving generic objects.

It's important to understand type erasure. Otherwise, a developer might get confused and think they'd be able to get the type at runtime:

```
public foo(Consumer<T> consumer) {
   Type type = consumer.getGenericTypeParameter()
}
```

The above example is a pseudo code equivalent of what things might look like without type erasure, but unfortunately, it is impossible. Once again, **the generic type information is not available at runtime.**

Q4. If a Generic Type is Omitted When Instantiating an Object, will the Code Still Compile?

As generics did not exist before Java 5, it is possible not to use them at all. For example, generics were retrofitted to most of the standard Java classes such as collections. If we look at our list from question one, then we will see that we already have an example of omitting the generic type:

```
List list = new ArrayList();
```

Despite being able to compile, it's still likely that there will be a warning from the compiler. This is because we are losing the extra compile time check that we get from using generics.

The point to remember is that **while backward compatibility and type erasure make it possible to omit generic types, it is bad practice.**

Q5. How Does a Generic Method Differ From a Generic Type?

A generic method is where a type parameter is introduced to a method, living within the scope of that method. Let's try this with an example:

```
public static <T> T returnType(T argument) {
    return argument;
}
```

We've used a static method but could have also used a non-static one if we wished. By leveraging type inference (covered in the next question), we can invoke this like any ordinary method, without having to specify any type arguments when we do so.

Q6. What is Type Inference?

Type inference is when the compiler can look at the type of a method argument to infer a generic type. For example, if we passed in *T* to a method which returns *T,* then the compiler can figure out the return type. Let's try this out by invoking our generic method from the previous question:

```
Integer inferredInteger = returnType(1);
String inferredString = returnType("String");
```

As we can see, there's no need for a cast, and no need to pass in any generic type argument. The argument type only infers the return type.

Q7. What is a Bounded Type Parameter?

So far all our questions have covered generic types arguments which are unbounded. This means that our generic type arguments could be any type that we want.

When we use bounded parameters, we are restricting the types that can be used as generic type arguments.

As an example, let's say we want to force our generic type always to be a subclass of animal:

```
1 public abstract class Cage<T extends Animal> {
2     abstract void addAnimal(T animal)
3 }
```

By using extends, we are forcing *T* to be a subclass of animal. We could then have a cage of cats:

```
1 Cage<Cat> catCage;
```

But we could not have a cage of objects, as an object is not a subclass of an animal:

```
1 Cage<Object> objectCage; // Compilation error
```

One advantage of this is that all the methods of animal are available to the compiler. We know our type extends it, so we could write a generic algorithm which operates on any animal. This means we don't have to reproduce our method for different animal subclasses:

```
1 public void firstAnimalJump() {
2     T animal = animals.get(0);
3     animal.jump();
4 }
```

Q8. Is it Possible to Declared a Multiple Bounded Type Parameter?

Declaring multiple bounds for our generic types is possible. In our previous example, we specified a single bound, but we could also specify more if we wish:

```
1 public abstract class Cage<T extends Animal & Comparable>
```

In our example, the animal is a class and comparable is an interface. Now, our type must respect both of these upper bounds. If our type were a subclass of animal but did not implement comparable, then the code would not compile. **It's also worth remembering that if one of the upper bounds is a class, it must be the first argument.**

Q9. What is a Wildcard type?

A wildcard type represents an unknown *type*. It's detonated with a question mark as follows:

```
1 public static consumeListOfWildcardType(List<?> list)
```

Here, we are specifying a list which could be of any *type*. We could pass a list of anything into this method.

Q10. What is an Upper Bounded Wildcard?

An upper bounded wildcard is when a wildcard type inherits from a concrete type. This is particularly useful when working with collections and inheritance.

Let's try demonstrating this with a farm class which will store animals, first without the wildcard type:

```
public class Farm {
  private List<Animal> animals;

  public void addAnimals(Collection<Animal> newAnimals) {
    animals.addAll(newAnimals);
  }
}
```

If we had multiple subclasses of animal, such as cat and dog, we might make the incorrect assumption that we can add them all to our farm:

```
farm.addAnimals(cats); // Compilation error
farm.addAnimals(dogs); // Compilation error
```

This is because the compiler expects a collection of the concrete type animal, not one it subclasses.

Now, let's introduce an upper bounded wildcard to our add animals method:

```
public void addAnimals(Collection<? extends Animal> newAnimals)
```

Now if we try again, our code will compile. This is because we are now telling the compiler to accept a collection of any subtype of animal.

Q11. What is an Unbounded Wildcard?

An unbounded wildcard is a wildcard with no upper or lower bound, that can represent any type.

It's also important to know that the wildcard type is not synonymous to object. This is because a wildcard can be any type whereas an object type is specifically an object (and cannot be a subclass of an object). Let's demonstrate this with an example:

```
List<?> wildcardList = new ArrayList<String>();
List<Object> objectList = new ArrayList<String>(); // Compilation error
```

Again, the reason the second line does not compile is that a list of objects is required, not a list of strings. The first line compiles because a list of any unknown type is acceptable.

Q12. What is a Lower Bounded Wildcard?

A lower bounded wildcard is when instead of providing an upper bound, we provide a lower bound by using the *super* keyword. In other words, **a lower bounded wildcard means we are forcing the type to be a superclass of our bounded type**. Let's try this with an example:

```
public static void addDogs(List<? super Animal> list) {
   list.add(new Dog("tom"))
}
```

By using *super,* we could call addDogs on a list of objects:

```
ArrayList<Object> objects = new ArrayList<>();
addDogs(objects);
```

This makes sense, as an object is a superclass of animal. If we did not use the lower bounded wildcard, the code would not compile, as a list of objects is not a list of animals.

If we think about it, we wouldn't be able to add a dog to a list of any subclass of animal, such as cats, or even dogs. Only a superclass of animal. For example, this would not compile:

```
ArrayList<Cat> objects = new ArrayList<>();
addDogs(objects);
```

Q13. When Would You Choose to Use a Lower Bounded Type vs. an Upper Bounded Type?

When dealing with collections, a common rule for selecting between upper or lower bounded wildcards is PECS. PECS stands for **producer extends, consumer super.**

This can be easily demonstrated through the use of some standard Java interfaces and classes.

Producer extends just means that if you are creating a producer of a generic type, then use the *extends* keyword. Let's try applying this principle to a collection, to see why it makes sense:

```
1 public static void makeLotsOfNoise(List<? extends Animal> animals) {
2     animals.forEach(Animal::makeNoise);
3 }
```

Here, we want to call *makeNoise()* on each animal in our collection. This means our collection is a producer, as all we are doing with it is getting it to return animals for us to perform our operation on. If we got rid of *extends*, we wouldn't be able to pass in lists of cats, dogs or any other subclasses of animals. By applying the producer extends principle, we have the most flexibility possible.

Consumer super means the opposite to *producer extends.* All it means is that if we are dealing with something which consumers elements, then we should use the *super* keyword. We can demonstrate this by repeating our previous example:

```
1 public static void addCats(List<? super Animal> animals) {
2     animals.add(new Cat());
3 }
```

We are only adding to our list of animals, so our list of animals is a consumer. This is why we use the *super* keyword. It means that we could pass in a list of any superclass of animal, but not a subclass. For example, if we tried passing in a list of dogs or cats then the code would not compile.

The final thing to consider is what to do if a collection is both a consumer and a producer. An example of this might be a collection where elements are both added and removed. In this case, an unbounded wildcard should be used.

Q14. Are There Any Situations Where Generic Type Information is Available at Runtime?

There is one situation where a generic type is available at runtime. This is when a generic type is part of the class signature like so:

```
1 public class CatCage implements Cage<Cat>
```

By using reflection, we get this type parameter:

```
1 (Class<T>) ((ParameterizedType) getClass()
2   .getGenericSuperclass()).getActualTypeArguments()[0];
```

This code is somewhat brittle. For example, it's dependant on the type parameter being defined on the immediate superclass. But, it demonstrates the JVM has does have this type information.

Next »
Java Flow Control Interview Questions (+ Answers)
(https://www.baeldung.com/java-flow-control-interview-questions)

« Previous

Memory Management in Java Interview Questions (+Answers) (https://www.baeldung.com/java-memory-management-interview-questions)

▲ newest ▲ oldest ▲ most voted

Guest

Ruslan Stelmachenko

Q13. “However, if we passed in a list of dogs, it would not compile because a cat is not a dog.” No. It would not compile because a cat is not a superclass of Animal. Even if we passed in a list of cats, it would still not compile and we can’t add a cat to cats. 🙂 The same is for Q12. “If we think about it, we wouldn’t be able to add a dog to a list of another subclass of animal, such as cats.”. Yes, but with this code, we also wouldn’t be able to add a… Read more »

✚ 2 ━

2 years ago

Guest

Grzegorz Piwowarek

Thanks, good catches. We will update the article

+ 1 −

1 year ago

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)