# ♦ 12 Java Generics interview Q&A

Posted on August 13, 2014 by Arulkumaran Kumaraswamipillai — 2 Comments ↓

0 Like Share 0 G+1

Q1 What do you understand by the term type erasure with regards to generics?

A1. The term **type erasure** is used in Java generics. In the interest of backward compatibility, robustness of generics has been sacrificed through type erasure. Type erasure takes place at compile-time. So, after compiling List and List, both end up as List at runtime. They are both just lists.

```
1 List<String> myList = new ArrayList<String>( );
2 List<String> myList = new ArrayList<>(); // In Ja
3
```

9 tips to earn more | What can u do to go places? | **945+** members. LinkedIn Group. **Reviews**

## 600+ Full Stack Java/JEE Interview Q&As ♥Free ♦FAQs

open all | close all

- Ice Breaker Interview
- Core Java Interview C
  - Java Overview (4)
  - Data types (6)
  - constructors-metho
  - Reserved Key Wor
  - Classes (3)
  - Objects (8)
  - OOP (10)
  - GC (2)
  - Generics (5)

after compilation becomes backward compatible without any angular brackets.

```
1 List myList = new ArrayList( );
2
```

Java generics differ from C++ templates. Java generics (at least until JDK 8), generate only one compiled version of a generic class or method regardless of the number of types used. During compile-time, all the parametrized type information within the angle brackets are erased and the compiled class file will look similar to code written prior to JDK 5.0. In other words, J**ava does not support runtime generics**.

Q2. Why do you need generics?
A2. Generics was introduced in JDK 5.0, and allows you to abstract over types. Without generics, you could put heterogeneous objects into a collection. This can encourage developers to write programs that are harder to read and maintain. For example,

```
1 List list = new ArrayList( );
2 list.add(new Integer( ));
3 list.add("A String");
4 list.add(new Mango( ));
5
```

As demonstrated above, without generics you can add any type of object to a collection. This means you would not only have to use "instanceof" operator, but also have to explicitly **cast** any objects returned from this list. The code is also less readable. The following code with generics is not only more readable,

```
1 List<String> list1 = new ArrayList<String>( );
2 List<Integer> list2 = new ArrayList<Integer>( );
3
```

but also throws a compile time error if you try to add an Integer object to list1 or a String object to list2.




## As a Java Architect

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

Q3. What are the differences among?

– Raw or plain old collection type e.g. Collection
– Collection of unknown e.g. Collection<?>
– Collection of type object e.g. Collection<Object>

A3. **1) The plain old Collection**: is a heterogeneous mixture or a mixed bag that contains elements of all types, for example Integer, String, Fruit, Vegetable, etc.

**2) The Collection<object>**: is also a heterogeneous mixture like the plain old Collection, but not the same and can be more restrictive than a plain old Collection discussed above. It is incorrect to think of this as the super type for a collection of any object types.

Unlike an Object class is a super type for all objects like String, Integer, Fruit, etc, List<Object> is not a super type for List<String>, List<Integer>, List<Fruit>, etc. So it is illegal to do the following:

```
1 List<Object> list = new ArrayList<Integer>( );//i
2
```

Though Integer is a subtype of Object, List is not a subtype of List<Object> because List of Objects is a bigger set comprising of elements of various types like Strings, Integers, Fruits, etc. A List of Integer should only contain Integers, hence the above line is illegal. If the above line was legal, then you can end up adding objects of any type to the list, violating the purpose of generics.

**3) The Collection<?>:** is a homogenous collection that represents a family of generic instantiations of Collection like Collection<String>, Collection<Integer>, Collection<Fruit>, etc.

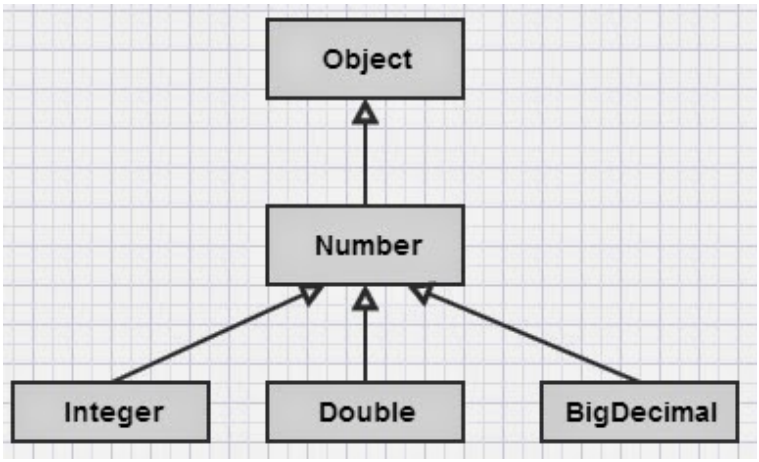
Collection<?> is the super type for all generic collection as Object[ ] is the super type for all arrays.


## Senior Java developers must have a good handle on

open all | close all
⊞ Best Practice (6)
⊞ Coding (26)
⊞ Concurrency (6)
⊞ Design Concepts (7)
⊞ Design Patterns (11)
⊞ Exception Handling (3
⊞ Java Debugging (21)
⊞ Judging Experience Ir
⊞ Low Latency (7)
⊞ Memory Managemen
⊞ Performance (13)
⊞ QoS (8)
⊞ Scalability (4)
⊞ SDLC (6)
⊞ Security (13)
⊞ Transaction Managen

## 80+ step by step Java Tutorials

open all | close all
⊞ Setting up Tutorial (6)
⊞ Tutorial - Diagnosis (2
⊞ Akka Tutorial (9)
⊞ Core Java Tutorials (2
⊞ Hadoop & Spark Tuto
⊞ JEE Tutorials (19)
⊞ Scala Tutorials (1)
⊞ Spring & HIbernate Tu
⊞ Tools Tutorials (19)
⊞ Other Tutorials (45)




Number class hierachy

```
1 List<?> list = new ArrayList<Integer>( );       //
2 List<? extends Number> list = new ArrayList<Integ
3
```

Q4. How will you go about deciding which of the following to use?

1) <Number>
2) <? extends Number>
3) <? super Number>

A4. Many developers struggle with the wild cards. Here is the guide:

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 put objects in a data structure.
3. If you need to do both read and add elements, **don't use any wildcard**.

Q5. What does the following code fragment print?

```
1
2 List<String> list1 = new ArrayList<String>( );
3 List<Integer> list2 = new ArrayList<Integer>( );
4 System.out.println(list1.getClass( ) == list2.get
```

**Preparing for Java written & coding tests**

open all | close all
♦ Complete the given
Can you write code? (
Converting from A to I
Designing your classe
Java Data Structures
Passing the unit tests
What is wrong with th
Writing Code Home A
Written Test Core Jav
Written Test JEE (1)

**How good are your...to go places?**

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

```
5
```

A5 It prints “true” because of type erasure(i.e. Rule 1), all instances of a generic class have the same runtime class, regardless of their actual type parameter. This also mean, there is no sense in checking generic information at runtime. The following code is illegal.

```
1 if(list1 instanceof List<String>) //illegal
2
```

Q6. Is the following line legal? If not, how will you fix it?

```
1
2 List<Object> list = new ArrayList<Integer>( );
3
```

A6. It is Illegal because Unlike an Object class is a super type for all objects like String, Integer, Fruit, etc, List&l;tObject> is not a super type for List<String>, List<Integer>, List<Fruit>, etc. List<?> is the super type.

```
1
2 List<?> list = new ArrayList<Integer>( );
3 List<? extends Number> list = new ArrayList<Integ
4
```

Note:<? extends Number> is read only and <?> is almost read only allowing only removce( ) and clear ( ) operations.

The Collection<?> can only be used as a reference type, and you cannot instantiate it.

```
1 List<?> fruitBasket = new ArrayList<?>( ); //ille
2 List<?> fruitBasket = new ArrayList<? extends Fru
3
```

Q7. How will you generify your own Java class?
A7

```
1
```

```
2  public class MyGenericClass<T> {
3
4      T objType;
5
6      public MyGenericClass(T type) {
7          this.objType = type;
8      }
9
10     public T getObjType( ) {
11         return objType;
12     }
13
14     public void setObjType(T objType) {
15         this.objType = objType;
16     }
17
18     public static void main(String[ ] args) {
19         MyGenericClass<Integer> val1 = new  MyGe
20         MyGenericClass<Long> val2 = new MyGeneri
21         long result = val1.getObjType( ).longVal
22
23         System.out.println(result);
24     }
25 }
26
```

If you decompile the converted class file, you will get,

```
1  public class MyGenericClass<T>
2  {
3      T objType;
4
5    public MyGenericClass(T type)
6    {
7        this.objType = type;
8    }
9
10   public T getObjType( ) {
11       return this.objType;
12   }
13
14   public void setObjType(T objType) {
15       this.objType = objType;
16   }
17
18   public static void main(String[ ] args) {
19       MyGenericClass val1 = new  MyGenericClass(
20       MyGenericClass val2 = new MyGenericClass(L
21       long result = ((Integer)val1.getObjType( )
22       System.out.println(result);
23    }
24 }
25
```

If you closely examine the above code, you would notice that the compiler has performed auto-boxing as generics does not support primitive types. The angle brackets have been removed for val1 & val2 declarations and appropriate

castings have been added to convert from type T to Integer and Long types.

Q8. What do you understand by the term type argument inference?

A8. The type inference happens when the compiler can deduce the type arguments of a generic type or method from a given context information. There are 2 situations in which the type argument inference is attempted during compile-time.

1. When an object of a generic type is created as demonstrated in the MyGenericClass<T>.

```
1
2 //T is inferred as an Integer
3  MyGenericClass<Integer> val1 = new  MyGenericCla
4  //T is inferred as  a Long
5  MyGenericClass<Long> val2 = new MyGenericClass<L
6
```

2. When a generic method is invoked. For example,

```
1
2  import java.util.ArrayList;
3  import java.util.List;
4
5  public class MyBasket {
6
7      /**
8       * The 'src' is the inferred type T or its s
9       * inferred type T or its super type.
10      */
11     public static <T> void copy(List<? extends T
12         for (T obj : src) {
13             dest.add(obj);
14         }
15     }
16
17     public static void main(String[] args) {
18         List<Orange> orangeBasket = new ArrayLis
19         List<Mango> mangoBasket = new ArrayList<
20         orangeBasket.add(new Orange());
21         mangoBasket.add(new Mango());
22         List<Fruit> fruitBasket = new ArrayList<
23
24         List<Orange> orangeBasket2 = new ArrayLi
25         orangeBasket2.add(new Orange());
26         List<Mango> mangoBasket2 = new ArrayList
27         mangoBasket2.add(new Mango());
28         List<Fruit> fruitBasket2 = new ArrayList
29         fruitBasket2.add(new Mango());
```

```
30
31          MyBasket.copy(orangeBasket2, orangeBaske
32          MyBasket.copy(mangoBasket2, mangoBasket)
33
34          MyBasket.<Orange> copy(orangeBasket, fru
35          MyBasket.<Mango> copy(mangoBasket, fruit
36
37          MyBasket.copy(fruitBasket2, fruitBasket)
38
39          for (Fruit fruit : fruitBasket) {
40              fruit.peel();
41          }
42      }
43  }
44
45
```

The copy(…) method ensures that fruits from a mixed fruit basket cannot be copied to a basket that only holds oranges or mangoes. But a mixed fruit basket allows fruits to be copied from any basket.

Q9. Is the following code snippet legal? If yes why and if not why not?

```
1
2  public MyGenericClass( ) {
3       this.objType = new T( );
4  }
5
```

A9. It is not legal as new T( ) will cause a compile-time error. This is partially because there's no guarantee that the target class for raw type "T" has a constructor that takes zero parameters and partially due to type erasure where the raw type "T" does not have any way of knowing the type of object you want to construct at runtime.

Q10. Is it possible to generify methods in Java?
A10. Yes.

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class MyGenericMethod {
5
6      //Generified method
7      public static <T> void addValue(T value, Lis
8           list.add(value);
9      }
```

```
10
11      public static void main(String[ ] args) {
12          List<Integer> listIntegers = new ArrayLi
13          Integer value1 = new Integer(37);
14          addValue(value1, listIntegers);
15          System.out.println("listIntegers=" + lis
16
17          List<String> listString = new ArrayList<
18          String value2 = "Test";
19          addValue(value2, listString);
20          System.out.println("listString=" + listS
21      }
22  }
23
```

**Note**: If you had used the wildcard List<?> instead of List<T> on line A, it would not have been possible to add elements. You will get a compile-time error. So how does the compiler know the type of “T”? It infers this from your use of the method. The generated class file looks pretty much the same as the source file without the <Integer> and <String> angle brackets

Q11. Does the following code snippet compile? What does it demonstrate?

```
1
2  public class Generics4<T> {
3
4      public <T> void doSomething(T data) {
5          System.out.println(data);
6      }
7
8      public static void main(String[ ] args) {
9          Generics4<String> g4 = new Generics4<Str
10         g4.doSomething(123);
11     }
12 }
13
```

A11. Yes, the above code snippet does compile. It demonstrates that the type parameter in the class name and the type parameter in the method are actually different parameters. The method signature,

```
1
2     public <T> void doSomething(T data)
3
```

really means,

```
1
2 public  void doSomething(Object data)
3
```

Q12. Can you identify any issues with the following code?

```
1
2  import java.util.ArrayList;
3  import java.util.Iterator;
4  import java.util.List;
5
6  public class GenericsWithIterators {
7
8      public static void main(String[ ] args) {
9          List<Integer> listIntegers = new ArrayLi
10         listIntegers.add(5);   //2
11         listIntegers.add(3);  //3
12
13         Iterator it = listIntegers.listIterator(
14
15         while(it.hasNext( )){  //5
16             Integer i =  it.next( ); //6
17             System.out.println(i); //7
18         }
19     }
20 }
21
22
```

A12.

Line 4 will cause compile-time error on line 6 as the iterator is not generic. To fix this, replace line 4 with:

```
1
2 Iterator<Integer> it = listIntegers.listIterator(
3
```

or add an explicit cast to line 6.

```
1
2 Integer i =  (Integer) it.next( );// fix 2
3
```

The fix 1 is preferred. When you get an iterator, keyset, or values from a collection, assign it to an appropriate parametrized type as shown in fix 1.

# Popular Posts

♦ 11 Spring boot interview questions & answers

**861 views**

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

**829 views**

18 Java scenarios based interview Questions and Answers

**448 views**

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

**407 views**

♦ 7 Java debugging interview questions & answers

**311 views**

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

**303 views**

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

**294 views**

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

**288 views**

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

**263 views**

8 Git Source control system interview questions & answers

**215 views**

Bio | Latest Posts

## Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in

2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

**About** Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

‹ ♦ Java classes and interfaces are the building blocks

♦ Java Garbage Collection interview Q&A to ascertain your depth of Java knowledge ›

**Posted in** Generics, member-paid

**Tags:** Core Java FAQs, Java/JEE FAQs, Novice FAQs

## 2 comments on "♦ 12 Java Generics interview Q&A"

gurijala.karthik says:

May 10, 2015 at 3:25 pm

Thanks, this is a nice summary.

Here i have one basic question. Before generics how we achieve type safety.
can you please explain.

Reply

Arulkumaran says:

May 11, 2015 at 10:06 am

You could handle it 2 ways.

1. Using the “instanceof” operator

Object –> Account –> ChequeAccount

```
1 ChequeAccount ca;
2  if(o instanceof ChequeAccount) {
3      ca = (ChequeAccount) o;
4  } else {
5      //what you need to do if not
6  }
7
```

2. Using the “ClasscastException”

```
1 try {
2       ca = (ChequeAccount) o;
3  }
4  catch(ClassCAstException cce) {
5     //what you need to do if not
6  }
7
```

Reply

# Leave a Reply

Logged in as geethika. Log out?

**Comment**

Post Comment

# Empowers you to open more doors, and fast-track

### Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

### Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

# Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category

# © Disclaimer

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

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