Register | Login | Logout | Contact Us

Java-Success.com

Industrial strength Java/JEE Career Companion for those who want to go places

search here … Go

Home | Java FAQs | 600+ Java Q&As | Career | Tutorials | Member | Why?

Can u Debug? | Java 8 ready? | Top X | Productivity Tools | Judging Experience?

Home › Interview › Core Java Interview Q&A › Generics › ♦ 7 rules to remember on Java Generics for better understanding

♦ 7 rules to remember on Java Generics for better understanding

Posted on July 22, 2015 by Arulkumaran Kumaraswamipillai

Here are **7 rules** to remember regarding Java Generics to understand Generics and handle interview and coding questions on Core Java.

Object <– Fruit <– (Orange, Mango, etc siblings)

Rule 1: 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

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 (
⊞ Java Overview (4)
⊞ Data types (6)
⊞ constructors-methc
⊞ Reserved Key Wor
⊞ Classes (3)
⊞ Objects (8)
⊞ OOP (10)
⊞ GC (2)
⊟ Generics (5)

written prior to JDK 5.0. In other words, Java does not support runtime generics.

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

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.

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

The following code will issue a warning.

```
public void unCheckedCastWarning(Object o) {
    List<Integer> l = (List<Integer>)o;    //unche
}
```

The JVM can actually check to see if it’s a List, but it can’t check whether it is a list of Strings or Integers because the type information in angle brackets have been erased. Never ignore warnings like this from the compiler. You also cannot handle different
versions of the same exception as shown below because type erasure does not allow it.

```
//illegal
try {
   //....
}
catch(MyException<Integer> ex1){
   //....
}
catch(MyException<String> ex2){
   //....
}
```

Rule 2: Unlike an Object class is a super type for all objects like String, Integer, Fruit, etc, List<Object> is not a super type

♥ Java Generics
♥ Overloaded m
♦ 12 Java Gener
♦ 7 rules to reme
3 scenarios to ge
⊞ FP (8)
⊞ IO (7)
⊞ Multithreading (12)
⊞ Algorithms (5)
⊞ Annotations (2)
⊞ Collection and Data
⊞ Differences Betwee
⊞ Event Driven Progr
⊞ Exceptions (2)
⊞ Java 7 (2)
⊞ Java 8 (24)
⊞ JVM (6)
⊞ Reactive Programn
⊞ Swing & AWT (2)
⊞ JEE Interview Q&A (3
⊞ Pressed for time? Jav
⊞ SQL, XML, UML, JSC
⊞ Hadoop & BigData Int
⊞ Java Architecture Inte
⊞ Scala Interview Q&As
⊞ Spring, Hibernate, & I
⊞ Testing & Profiling/Sa
⊞ Other Interview Q&A f
⊞ ▶ Free Java Interview

As a Java Architect

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

for List<String>, List<Integer>, List<Fruit>, etc. So it is illegal to do the following:

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

Though Integer is a subtype of Object, List<Integer> 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. So how would you go about adding a method that accepts a collection of any type?

“processFruits” would not work!!!!!

```
import java.util.ArrayList;
import java.util.List;

public class Generics3 {
    public static void main(String[ ] args) {
      List<Fruit> fruitBasket = new ArrayList<Fr
      fruitBasket.add(new Orange( ));
      fruitBasket.add(new Mango( ));
      processFruits(fruitBasket); //compile-time
    }

    //Won't work with List<Object> because a Lis
    //type for all fruits (i.e. List<Fruit>).
    public static void processFruits(List<Object
       for (Object object : fruitBasket) {
          ((Fruit)object).peel( );
       }
    }
}
```

The processFruits(…) method can be fixed with the wild card character “?” as discussed next.

Rule 3: Collection<?> is the super type for all generic collection as Object[ ] is the super type for all arrays.

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

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 I
⊞ Low Latency (7)
⊞ Memory Managemen
⊞ Performance (13)
⊞ QoS (8)
⊞ Scalability (4)
⊞ SDLC (6)
⊞ Security (13)
⊞ Transaction Managem

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)

The processFruits(…) method can be fixed as follows to accept any kind of fruit.

```
public static void processFruits(List<? extends F
    for (Fruit fruit : fruitBasket) {
       fruit.peel( );
    }
}
```

The following code snippet will work with a collection of any type. But very rarely you may have a requirement to do this. It is a bad practice to mix unrelated object types into the same collection.

```
public static void processFruits(List<?> fruitBas
   for (Object object : fruitBasket) {
      ((Fruit)object).peel( );
   }
}
```

Q. Why not implement the method as follows?

```
public static void processFruits(List<Fruit> frui
    for (Fruit fruit : fruitBasket) {
       fruit.peel( );
    }
}
```

If you use List<Fruit> instead of List<? extends Fruit>, you will not be able to use this method for a List<Mango> or List<Orange>. As discussed earlier in **Rule 2**, a Mango or Orange might be a sub type of Fruit, but a List<Mango> or List<Orange> is not a sub type of List<Fruit>. This means the following declaration is illegal.

```
List<Fruit> fruitBasket = new ArrayList<Orange>(
```

Rule 4: The Collection<?> can only be used as a reference type, and you cannot instantiate it. The following statements are illegal.

```
List<?> fruitBasket = new ArrayList<?>( ); //ille
```

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 & Resur

```
List<?> fruitBasket = new ArrayList<? extends Fru
List<?> fruitBasket = new ArrayList<? extends Fru
```

```
List<?> fruitBasket = new ArrayList<Fruit>( ); //
List<? Extends Fruit> fruitBasket = new ArrayList
```

Rule 5: Hence, Collection<?> is almost a read-only collection allowing only remove( ) and clear( ) operations.

Even if you declare it as above, Collection<?> has a restriction (i.e. Rule 5) to add arbitrary objects as shown below.

```
List<? extends Fruit> fruitBasket = new ArrayList
fruitBasket.add(new Orange( )); //compile-time er
```

Q. Is it possible to generify your own Java class?

```
public class MyGenericClass<T> {
  T objType;

  public MyGenericClass(T type) {
     this.objType = type;
  }

  public T getObjType( ) {
     return objType;
  }

  public void setObjType(T objType) {
     this.objType = objType;
  }

  public static void main(String[ ] args) {
     MyGenericClass<Integer> val1 = new MyGeneri
     MyGenericClass<Long> val2 = new MyGenericCl
     long result = val1.getObjType( ).longValue(
     val2.getObjType( ).longValue( );
     System.out.println(result);
 }
}
```

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

```
public class MyGenericClass<T>
{
   T objType;

   public MyGenericClass(T type){
     this.objType = type;
   }

   public T getObjType( ) {
      return this.objType;
  }

  public void setObjType(T objType) {
    this.objType = objType;
  }

  public static void main(String[ ] args) {
     MyGenericClass val1 =
       new MyGenericClass(Integer.valueOf(37));
     MyGenericClass val2 =
       new MyGenericClass(Long.valueOf(250L)); /
     long result = ((Integer)val1.getObjType( ))
            ((Long)val2.getObjType( )).longValue
     System.out.println(result);
  }
}
```

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.

Rule 6: 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>.

```
 //T is inferred as an Integer
MyGenericClass<Integer> val1 = new MyGenericClass
//T is inferred as a Long
MyGenericClass<Long> val2 = new MyGenericClass<Lo
```

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

```
 import java.util.ArrayList;
import java.util.List;
public class MyBasket {

 /**
  * The 'src' is the inferred type T or its sub
  * inferred type T or its super type.
  */
  public static <T> void copy(List<? extends T>
     List<? super T> dest) {
     for (T obj : src) {
        dest.add(obj);
    }
 }

 public static void main(String[] args) {
     List<Orange> orangeBasket = new ArrayList<O
     List<Mango> mangoBasket = new ArrayList<Man
     orangeBasket.add(new Orange());
     mangoBasket.add(new Mango());

     List<Fruit> fruitBasket = new ArrayList<Fru

     List<Orange> orangeBasket2 = new ArrayList<
     orangeBasket2.add(new Orange());

     List<Mango> mangoBasket2 = new ArrayList<Ma
     mangoBasket2.add(new Mango());

     List<Fruit> fruitBasket2 = new ArrayList<Fr
     fruitBasket2.add(new Mango());

     MyBasket.copy(orangeBasket2, orangeBasket);
     MyBasket.copy(mangoBasket2, mangoBasket); /
     MyBasket.<Orange> copy(orangeBasket, fruitB
     MyBasket.<Mango> copy(mangoBasket, fruitBas
     MyBasket.copy(fruitBasket2, fruitBasket); /

     MyBasket.copy(fruitBasket, orangeBasket); /
                                               /
                                               /
                                               /

     MyBasket.<Orange> copy(fruitBasket, orangeB







    for (Fruit fruit : fruitBasket) {
        fruit.peel();
    }
  }
}
```

If you comment the 2 lines If you comment the 2 lines that result in compile-time error, you will get the following output,

```
peeling Orange
peeling Orange
peeling Mango
peeling Mango
peeling Mango
```

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.

Rule 7: Static members are not allowed to have reference to their type parameters due to Rule 1. Since the static members are not allowed to have reference to their type parameters, generic enum is also not allowed as enum values are static data members.

```
enum State <T> { //illegal
 //....
}
```

Q. Is it possible to generify methods in Java?
A. Yes.

```
import java.util.ArrayList;
import java.util.List;

public class MyGenericMethod {

   //Generified method
   public static <T> void addValue(T value, List
     list.add(value);
   }

   public static void main(String[ ] args) {
     List<Integer> listIntegers = new ArrayList<
     Integer value1 = new Integer(37);

     addValue(value1, listIntegers); //T is infe
     System.out.println("listIntegers=" + listIn
     List<String> listString = new ArrayList<Str
     String value2 = "Test";
     addValue(value2, listString); //T is inferr
     System.out.println("listString=" + listStri
  }
}
```

Note: If you had used the wildcard List instead of List on line A, it would not have been possible to add elements due to Rule 5. 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 as discussed in Rule 6. The generated class file looks pretty much the same as the source file without the and angle brackets as shown below once decompiled.

```
import java.util.ArrayList;
import java.util.List;

public class MyGenericMethod {

 public static <T> void addValue(T value, List<
    list.add(value);
 }

 public static void main(String[ ] args) {

  List listIntegers = new ArrayList( );
  Integer value1 = new Integer(37);
  addValue(value1, listIntegers);
  System.out.println("listIntegers=" + listInt

  List listString = new ArrayList( );
  String value2 = "Test";
  addValue(value2, listString);
  System.out.println("listString=" + listStrin
 }
}
```

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

```
public class Generics4<T> {

  public <T> void doSomething(T data) {
   System.out.println(data);
 }

 public static void main(String[ ] args) {
   Generics4<String> g4 = new Generics4<String
   g4.doSomething(123);
 }
}
```

What does “public <T> void doSomething(T data)” really mean?

A. es, 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,

```
public <T> void doSomething(T data)
```

really means:

```java
public void doSomething(Object data)
```

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.

‹ ♦ 9 Groovy meta programming interview questions & answers

Sum grades Groovy coding questions & answers ›

Posted in Generics, member-paid

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.