Java Generics Tutorial

Java Generics – Complete Tutorial
Java Generics – PECS

Popular Tutorials

Java Tutorial
Java 10 Tutorial
Java 9 Tutorial
Java 8 Tutorial
Java 7 Tutorial
Concurrency Tutorial
IO Tutorial
Interview Questions
Puzzles
JDBC Tutorial
Algorithms
Design Patterns
Best Practices
Regular Expressions
JAX-RS 2.0 Tutorial
Jersey Tutorial
Hibernate Tutorial
JUnit Tutorial
TestNG Tutorial
Log4j Tutorial

Complete Java Generics Tutorial

June 26, 2014 by Lokesh Gupta

Generics in java were introduced as one of **features in JDK 5**. Personally, I find the angular brackets "<>" used in generics very fascinating and it always force me to have another thought where I use it OR see it written in somebody else's code. To be very frank, I have been using generics since long time now but still I feel not fully confident to use it blindly. In this tutorial, I will be covering everything I find useful with **java generics**, and things related to them. If you think that I can use more precise words at any part of tutorial, or an example can be added or simply you do not agree with me; drop me a comment. I will be glad to know your point of view.

Table of content

Search Tutorials

Type and Press ENTER...

Maven Tutorial
Spring AOP Tutorial
Spring Boot Tutorial
Spring Cloud Tutorial
Spring MVC Tutorial
Spring Security Tutorial

Recommended

10 Life Lessons
How to Start New Blog
Secure Hash Algorithms
Regular Expressions
How Web Servers work?
How Java I/O Works Internally?
Best Way to Learn Java
Java Best Practices
Java Interview Questions
Microservices Tutorial
REST API Tutorial

"**Java Generics**" is a technical term denoting a set of language features related to the definition and use of generic types and methods . In java, Generic types or methods differ from regular types and methods in that they have type parameters.

> "Java Generics are a language feature that allows for definition and use of generic types and methods."

Generic types are instantiated to form parameterized types by providing actual type arguments that replace the formal type parameters. A class like `LinkedList<E>` is a generic type, that has a type parameter E . Instantiations, such as `LinkedList<Integer>` or a `LinkedList<String>`, are called parameterized types, and String and Integer are the respective actual type arguments.

1) Why Generics?

If you closely look at java collection framework classes then you will observe that most classes take parameter/argument of type `Object` and return values from methods as `Object`. Now, in this form, they can take any java type as argument and return the same. They are essentially heterogeneous i.e. not of a particular similar type.

Programmers like us often wanted to specify that a collection contains elements only of a certain type e.g. `Integer` or `String` or `Employee`. In original collection framework, having homogeneous collections was not possible without adding extra checks before adding some checks in code. Generics were introduced to remove this limitation to be very specific. They add this type checking of parameters in your code at compile time, automatically. This saves us writing a lot of un-necessary code which actually does not add any value in run-time, if written correctly.

> “In layman,s term, generics force type safety in java language.”

Without this type safety, your code could have infected by various bugs which get revealed only in runtime. Using generics, makes them highlighted in compile time itself and make you code robust even before you get the bytecode of your java sourcecode files.

> “Generics add stability to your code by making more of your bugs detectable at compile time.”

So now we have a fair idea why generics are present in java in the first place. Next step is get some knowledge about how they work in java. What actually happen when you use generics in your sourcecode.

2) How Generics works in Java

In the heart of generics is “**type safety**“. What exactly is type safety? It’s just a guarantee by compiler that if correct Types are used in correct places then there should not be any `ClassCastException` in runtime. A usecase can be list of `Integer` i.e. `List<Integer>`. If you declare a list in java like `List<Integer>`, then java guarantees that it will detect and report you any attempt to insert any non-integer type into above list.

Another important term in java generics is “**type erasure**“. It essentially means that all the extra information added using generics into sourcecode will be removed from bytecode generated from it. Inside bytecode, it will be old java syntax which you will get if you don’t use generics at all. This necessarily helps in generating and executing code written prior java 5 when generics were not added in language.

Let’s understand with an example.

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

list.add(1000);     //works fine

list.add("lokesh"); //compile time error;
```

When you write above code and compile it, you will get below error: "*The method add(Integer) in the type `List<Integer>` is not applicable for the arguments (String)*". Compiler warned you. This exactly is generics sole purpose i.e. Type Safety.

Second part is getting byte code after removing second line from above example. If you compare the bytecode of above example with/without generics, then there will not be any difference. Clearly compiler removed all generics information. So, above code is very much similar to below code without generics.

```
List list = new ArrayList();

list.add(1000);
```

> "Precisely, Generics in Java is nothing but a syntactic sugar to your code for Type Safety and all such type information is erased by Type Erasure feature by compiler."

3) Types of Generics?

Now we have some understanding about what generics are all about. Now start exploring other important concepts revolving around generics. I will start from identifying the various ways, generics can be applied into sourcecode.

Generic Type Class or Interface

A class is generic if it declares one or more type variables. These type variables are known as the type parameters of the class. Let's understand with an example.

`DemoClass` is simple java class, which have one property t (can be more than one also); and type of property is Object.

```
class DemoClass {
   private Object t;

   public void set(Object t) { this.t = t; }

   public Object get() { return t; }
}
```

Here we want that once initialized the class with a certain type, class should be used with that particular type only. e.g. If we want one instance of class to hold value t of type '`String`', then programmer should set and get the only `String` type. Since we have declared property type to `Object`, there is no way to enforce this restriction. A programmer can set any object; and can expect any return value type from get method since all java types are subtypes of `Object` class.

To enforce this type restriction, we can use generics as below:

```
class DemoClass<T> {
   //T stands for "Type"
   private T t;

   public void set(T t) { this.t = t; }
```

```
    public T get() { return t; }
}
```

Now we can be assured that class will not be misused with wrong types. A sample usage of `DemoClass` will look like this:

```
DemoClass<String> instance = new DemoClass<String>();
instance.set("lokesh");   //Correct usage
instance.set(1);          //This will raise compile time error
```

Above analogy is true for interface as well. Let's quickly look at an example to understand, how generics type information can be used in interfaces in java.

```
//Generic interface definition
interface DemoInterface<T1, T2>
{
    T2 doSomeOperation(T1 t);
    T1 doReverseOperation(T2 t);
}

//A class implementing generic interface
class DemoClass implements DemoInterface<String, Integer>
{
    public Integer doSomeOperation(String t)
    {
        //some code
    }
    public String doReverseOperation(Integer t)
    {
        //some code
    }
}
```

I hope, I was enough clear to put some light on generic classes and interfaces. Now it's time to look at generic methods and constructors.

Generic Type Method or Constructor

Generic methods are much similar to generic classes. They are different only in one aspect that scope of type information is inside method (or constructor) only. Generic methods are methods that introduce their own type parameters.

Let's understand this with an example. Below is code sample of a generic method which can be used to find all occurrences of a type parameters in a list of variables of that type only.

```
public static <T> int countAllOccurrences(T[] list, T item) {
  int count = 0;
  if (item == null) {
    for ( T listItem : list )
      if (listItem == null)
        count++;
  }
  else {
    for ( T listItem : list )
      if (item.equals(listItem))
        count++;
  }
  return count;
}
```

If you pass a list of `String` and another string to search in this method, it will work fine. But if you will try to find an `Number` into list of `String`, it will give compile time error.

Same as above can be example of generic constructor. Let's take a separate example for generic constructor as well.

```
class Dimension<T>
{
  private T length;
  private T width;
  private T height;

  //Generic constructor
  public Dimension(T length, T width, T height)
  {
    super();
    this.length = length;
    this.width = width;
    this.height = height;
  }
}
```

In this example, `Dimension` class’s constructor has the type information also. So you can have an instance of dimension with all attributes of a single type only.

4) Generic Type Arrays

Array in any language have same meaning i.e. an array is a collection of similar type of elements. In java, pushing any incompatible type in an array on runtime will throw `ArrayStoreException`. It means array preserve their type information in runtime, and generics use type erasure or remove any type information in runtime. Due to above conflict, instantiating a generic array in java is not permitted.

```
public class GenericArray<T> {
    // this one is fine
    public T[] notYetInstantiatedArray;
    // causes compiler error; Cannot create a generic array of T
    public T[] array = new T[5];
}
```

In the same line as above generic type classes and methods, we can have generic arrays in java. As we know that an array is a collection of similar type of elements and pushing any incompatible type will throw `ArrayStoreException` in runtime; which is not the case with `Collection` classes.

```
Object[] array = new String[10];
array[0] = "lokesh";
array[1] = 10;      //This will throw ArrayStoreException
```

Above mistake is not very hard to make. It can happen anytime. So it's better to provide the type information to array also so that error is caught at compile time itself.

Another reason why arrays does not support generics is that arrays are co-variant, which means that an array of supertype references is a supertype of an array of subtype references. That is, `Object[]` is a supertype of `String[]` and a string array can be accessed through a reference variable of type `Object[]`.

```
Object[] objArr = new String[10];  // fine
objArr[0] = new String();
```

5) Generics with Wildcards

In generic code, the question mark (?), called the wildcard, represents an unknown type. **A wildcard parameterized type is an instantiation of a generic type where at least one type argument is a wildcard.** Examples of wildcard parameterized types are `Collection<?<`, `List<? extends Number<`, `Comparator<? super String>` and `Pair<String,?>`. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

Having wild cards at difference places have different meanings as well. e.g.

- **Collection** denotes all instantiations of the Collection interface regardless of the type argument.
- **List** denotes all list types where the element type is a subtype of Number.
- **`Comparator<? super String<`** denotes all instantiations of the Comparator interface for type argument types that are supertypes of String.

A wildcard parameterized type is not a concrete type that could appear in a new expression. It just hints the rule enforced by java generics that which types are valid in any particular scenario where wild cards have been used.

For example, below are valid declarations involving wild cards:

```
Collection<?> coll = new ArrayList<String>();
//OR
List<? extends Number> list = new ArrayList<Long>();
//OR
Pair<String,?> pair = new Pair<String,Integer>();
```

And below are not valid uses of wildcards, and they will give compile time error.

```
List<? extends Number> list = new ArrayList<String>();  //String is not subclass of Number; so
error
//OR
```

```
Comparator<? super String> cmp = new RuleBasedCollator(new Integer(100)); //Integer is not
superclass of String
```

Wildcards in generics can be unbounded as well as bounded. Let's identify the difference in various terms.

Unbounded wildcard parameterized type

A generic type where all type arguments are the unbounded wildcard `"?"` without any restriction on type variables. e.g.

```
ArrayList<?>  list = new ArrayList<Long>();
//or
ArrayList<?>  list = new ArrayList<String>();
//or
ArrayList<?>  list = new ArrayList<Employee>();
```

Bounded wildcard parameterized type

Bounded wildcards put some restrictions over possible types, you can use to instantiate a parametrized type. This restriction is enforced using keywords "super" and "extends". To differentiate more clearly, let's divide them into upper bounded wildcards and lower bounded wildcards.

Upper bounded wildcards

For example, say you want to write a method that works on List<String>, List<Integer>, and List<double> you can achieve this by using an upper bounded wildcard e.g. you would specify List<? extends Number>. Here Integer, Double are subtypes of Number class. In layman’s terms, if you want generic expression to accept all subclasses of a particular type, you will use upper bound wildcard using “**extends**” keyword.

```
public class GenericsExample<T>
{
  public static void main(String[] args)
  {
    //List of Integers
    List<Integer> ints = Arrays.asList(1,2,3,4,5);
    System.out.println(sum(ints));

    //List of Doubles
    List<Double> doubles = Arrays.asList(1.5d,2d,3d);
    System.out.println(sum(doubles));

    List<String> strings = Arrays.asList("1","2");
    //This will give compilation error as :: The method sum(List<? extends Number>) in the
    //type GenericsExample<T> is not applicable for the arguments (List<String>)
    System.out.println(sum(strings));

  }

  //Method will accept
  private static Number sum (List<? extends Number> numbers){
    double s = 0.0;
    for (Number n : numbers)
      s += n.doubleValue();
    return s;
  }
}
```

Lower bounded wildcards

If you want a generic expression to accept all types which are "super" type of a particular type OR parent class of a particular class then you will use lower bound wildcard for this purpose, using 'super' keyword.

In below given example, I have created three classes i.e. `SuperClass`, `ChildClass` and `GrandChildClass`. There relationship is shown in code below. Now, we have to create a method which somehow get a `GrandChildClass` information (e.g. from DB) and create an instance of it. And we want to store this new `GrandChildClass` in an already existing list of `GrandChildClasses`.

Here problem is that `GrandChildClass` is subtype of `ChildClass` and `SuperClass` as well. So any generic list of SuperClasses and ChildClasses is capable of holding GrandChildClasses as well. Here we must take help of lower bound wildcard using '**super**' keyword.

```
package test.core;

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

public class GenericsExample<T>
{
  public static void main(String[] args)
  {
    //List of grand children
    List<GrandChildClass> grandChildren = new ArrayList<GrandChildClass>();
    grandChildren.add(new GrandChildClass());
    addGrandChildren(grandChildren);

    //List of grand childs
    List<ChildClass> childs = new ArrayList<ChildClass>();
    childs.add(new GrandChildClass());
    addGrandChildren(childs);
```

```
        //List of grand supers
        List<SuperClass> supers = new ArrayList<SuperClass>();
        supers.add(new GrandChildClass());
        addGrandChildren(supers);
    }

    public static void addGrandChildren(List<? super GrandChildClass> grandChildren)
    {
        grandChildren.add(new GrandChildClass());
        System.out.println(grandChildren);
    }
}

class SuperClass{

}
class ChildClass extends SuperClass{

}
class GrandChildClass extends ChildClass{

}
```

6) What is not allowed to do with Generics?

So far we have learned about a number of things which you can do with generics in java to avoid many `ClassCastException` instances in your application. We also saw the usage of wildcards as well. Now it's time to identify some tasks which are not allowed to do in java generics.

a) You can't have static field of type

You can not define a static generic parameterized member in your class. Any attempt to do so will generate compile time error: Cannot make a static reference to the non-static type T.

```
public class GenericsExample<T>
{
  private static T member; //This is not allowed
}
```

b) You can not create an instance of T

Any attempt to create an instance of T will fail with error: Cannot instantiate the type T.

```
public class GenericsExample<T>
{
  public GenericsExample(){
    new T();
  }
}
```

c) Generics are not compatible with primitives in declarations

Yes, it’s true. You can’t declare generic expression like List or Map<String, double>. Definitely you can use the wrapper classes in place of primitives and then use primitives when passing the actual values. These value primitives are accepted by using auto-boxing to convert primitives to respective wrapper classes.

```
final List<int> ids = new ArrayList<>();    //Not allowed

final List<Integer> ids = new ArrayList<>(); //Allowed
```

d) You can’t create Generic exception class

Sometimes, programmer might be in need of passing an instance of generic type along with exception being thrown. This is not possible to do in Java.

```
// causes compiler error
public class GenericException<T> extends Exception {}
```

When you try to create such an exception, you will end up with message like this: The generic class `GenericException` may not subclass `java.lang.Throwable`.

That’s all for now closing the discussion on **java generics** this time. I will come up with more interesting facts and features related to generics in coming posts.

Drop me a comment if something is unclear /OR you have any other question.

Happy Learning !!

About Lokesh Gupta

Founded HowToDoInJava.com in late 2012. I love computers, programming and solving problems everyday. A family guy with fun loving nature. You can find me on Facebook, Twitter and Google Plus.

Feedback, Discussion and Comments

Abhijit

April 10, 2018

Hi Lokesh

Great Explanation on Generics concepts. However I have question. Why Generics Type Arrays do not throw ArrayStoreException when we change your code to the below. If we change the type from String to Object then arrays accept heterogeneous types..

```
Object[] array = new Object[10];
array[0] = "lokesh";
array[1] = 10;
```

Reply

Lokesh

February 9, 2018

“For example, say you want to write a method that works on List<String>, List<Integer>, and List<double> you can achieve this by using an upper bounded wildcard e.g. you would specify List<? extends Number>.”

Correction Indeed, because List<String> isn’t the same thing as (or subtype of) List<? extends Number>.
In fact, The common parent in List<?>.

ref: https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html

Reply

Lokesh

February 9, 2018

In section 5, This is confusing:

Collection<?<, List<? extends Number<, Comparator and Pair

Problem with html? Maybe you mean:

Collection, List, Comparator and Pair

Reply

JavaDev

January 29, 2018

I have several questions .
1. What is the difference when writing and T as return type
2. Wildecard vs T .
3. When use S,U etc. What is exactly mean secondary type
4. Is it possible to explain more detailed what is not allowed by generics . I mean why it is not allowed
Thanks

Reply

Lakshmi N Galla

January 10, 2018

public static int countAllOccurrences(T[] list, T item) {
………..
return count;
}

countAllOccurrences(new String[]{"1″,"3″,"3″,"6″}, new Integer(2));

This won't throw any compile error. It just returns count as '0'

Reply

sonam

May 8, 2017

I’m seeing <T> instead of

Reply

Sam

October 26, 2016

THanks for the info but you should really change the

Reply

Akash Anand

June 29, 2016

Thanks for the important information about generic.

Reply

sundar

February 23, 2016

why generic exceptipn are not allowed?can u elaborate deeply with examples

Reply

Pati Ram Yadav

February 4, 2016

Comparator cmp = new RuleBasedCollator(new Integer(100)); //String is not superclass of Integer

This is wrong explanation because means something which is super of String (including String itself). So here it should be like: Integer is not superclass of String :)

Reply

Lokesh Gupta

February 4, 2016

You are right. Fixed it.

Reply

Garima

October 22, 2015

I think there is one type mistake in point no 4 : Genereic Arrays which is “generics use type erasure or remove any type information in runtime” where as Erasure remove Type information at compile time. Am I correct?
then this reason is not applicable when u said “Due to above conflict, instantiating a generic array in java is not permitted.” Please explain

Reply

varun kumar srivastawa

April 20, 2017

Type Erasure

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:

Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
Insert type casts if necessary to preserve type safety.
Generate bridge methods to preserve polymorphism in extended generic types.
Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

Reply

Nitish Goyal

April 9, 2015

Thanks for the article..!!

Reply

Vinay Trivedi

March 10, 2015

Thanks for the post.
Can you please explain in detail why we can not create generic exception class.

Reply

john doe

March 23, 2015

that was because Runtime exception type has no generics information . For more detail , look into JLS 8.1.2

Reply

Nirmal

December 1, 2014

Awesome explanation!!

Reply

Vijay

August 20, 2014

Thanks Logesh! The tutorial is good.

Reply

Veer

June 30, 2014

Hi Lokesh,
Good effort. One thing additional which you might want to mention while talking about generics is Joshua Bloch's PECS (Producer Extends Consumer Super).

This reminds when to use ? extends and when to use ? super.

Reply

Lokesh

June 30, 2014

Yes Veer.. I am planning to write a separate pot for PECS only, it's just so much detailed and interesting topic.

Reply

Santhosh

June 30, 2014

Thank you so much... useful for us

Reply

prabhakar

June 27, 2014

Hi Lokesh.any posts on GIT Hub

Reply

Lokesh

June 27, 2014

I am planning to start using GIT hub. Just give me some time.

Reply

Ruslan Pavlutskiy

July 21, 2014

Yep, it will be good.

Reply

Bhaskar

June 27, 2014

Thanks !! Lokesh For your wonderful effort

Reply

HIMANSU NAYAK

June 26, 2014

Hi Lokesh,

What is not allowed to do with Generics?

ArrayList list = new ArrayList() // allowed
ArrayList list = new ArrayList(); // sub-type not allowed

objects creation using wildcard parameterized type is not allowed
ArrayList list = new ArrayList(); // as told by you a generic class instance creation not allowed

Reply

Lokesh

June 26, 2014

Please wrap the code inside: [java] … [/java]

Reply

Mridul Vimal Singh

June 26, 2014

Thanks Sir, You include all these things about generic

Reply

Sudheer

June 26, 2014

Thanks for the post on Generics. It would be very helpful if you can write a post on WebServices

Reply

Lokesh

June 26, 2014

I have written lots of post on RESTful WS. If you need in SOAP, then sorry to disappoint you, i have not worked on SOAP till date, so it will take long time.

Reply

Ask Questions & Share Feedback

Your email address will not be published. Required fields are marked *

Comment

*Want to Post Code Snippets or XML content? Please use [java] ... [/java] tags otherwise code may not appear partially or even fully. e.g.

```
[java]
public static void main (String[] args) {
...
}
[/java]
```

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Help me fight spammers. Solve this simple math. *

+ 4 = 7

POST COMMENT