Java Generics Tutorial

[Generics](http://docs.oracle.com/javase/tutorial/extra/generics/" \o "generics) in java were introduced as one of [**features in JDK 5**](http://howtodoinjava.com/category/new-jdk-features/java-5-features/). 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**

[1) Why Generics?](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#why_generics)

[2) How Generics works in Java](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#how_generics_work)

[3) Types of Generics?](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#generic_types)

[i) Generic Type Class or Interface](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#member_types)

[ii) Generic Type Method or Constructor](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#method_types)

[4) Generic Type Arrays](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#generic_arrays)

[5) Generics with Wildcards](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#wildcards)

[i) Unbounded Wildcards](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#unbounded_wildcards)

[ii) Bounded Wildcards](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#bounded_wildcards)

[a) Upper Bounded Wildcards](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#lower_bounded)

[b) Lower Bounded Wildcards](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#upper_bounded)

[6) What is not allowed to do with Generics?](http://howtodoinjava.com/2014/06/26/complete-java-generics-tutorial/#things_not_allowed)

“**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**](http://howtodoinjava.com/2013/07/09/useful-java-collection-interview-questions/) 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**](http://en.wikipedia.org/wiki/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**](http://en.wikipedia.org/wiki/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.

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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.

|  |  |
| --- | --- |
| 1  2  3 | 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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:

|  |  |
| --- | --- |
| 1  2  3 | 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | //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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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 throwArrayStoreException. 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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.

|  |  |
| --- | --- |
| 1  2  3 | 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[].

|  |  |
| --- | --- |
| 1  2 | 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<? extends Number>** 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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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.

|  |  |
| --- | --- |
| 1  2  3 | 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)); //String is not superclass of Integer |

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.

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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 devide 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, List, and List; 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | 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 andGrandChildClass. 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | 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 parametrized 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.

|  |  |
| --- | --- |
| 1  2  3  4 | 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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.

|  |  |
| --- | --- |
| 1  2  3 | 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.

|  |  |
| --- | --- |
| 1  2 | // 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.