Java Generics - Quick Guide

Previous Page Next Page

Java Generics - Overview

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Java Generics - Environment Setup

Local Environment Setup

JUnit is a framework for Java, so the very first requirement is to have JDK installed in your machine.

System Requirement

JDK	1.5 or above.
Memory	No minimum requirement.
Disk Space	No minimum requirement.
Operating System	No minimum requirement.

Step 1: Verify Java Installation in Your Machine

First of all, open the console and execute a java command based on the operating system you are working on.

OS	Task	Command
Windows	Open Command Console	c:\> java -version
Linux	Open Command Terminal	$ java -version
Mac	Open Terminal	machine:< joseph$ java -version

Let's verify the output for all the operating systems –

OS	Output
Windows	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Linux	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Mac	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM)64-Bit Server VM (build 17.0-b17, mixed mode, sharing)

If you do not have Java installed on your system, then download the Java Software Development Kit (SDK) from the following link https://www.oracle.com . We are assuming Java 1.6.0_21 as the installed version for this tutorial.

Step 2: Set JAVA Environment

Set the **JAVA_HOME** environment variable to point to the base directory location where Java is installed on your machine. For example.

OS	Output

Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdk1.6.0_21
Linux	export JAVA_HOME = /usr/local/java-current
Mac	export JAVA_HOME = /Library/Java/Home

Append Java compiler location to the System Path.

OS	Output
Windows	Append the string **C:\Program Files\Java\jdk1.6.0_21\bin** at the end of the system variable, **Path**.
Linux	export PATH = $PATH:$JAVA_HOME/bin/
Mac	not required

Verify Java installation using the command **java -version** as explained above.

Java Generics - Classes

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

The type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Syntax

```
public class Box<T> {
   private T t;
}
```

Where

Box – Box is a generic class.

T – The generic type parameter passed to generic class. It can take any Object.

t – Instance of generic type T.

Description

The T is a type parameter passed to the generic class Box and should be passed when a Box object is created.

Example

Create the following java program using any editor of your choice.

GenericsTester.java

Live Demo

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      Box<String> stringBox = new Box<String>();

      integerBox.add(new Integer(10));
      stringBox.add(new String("Hello World"));

      System.out.printf("Integer Value :%d\n", integerBox.get());
      System.out.printf("String Value :%s\n", stringBox.get());
   }
}

class Box<T> {
   private T t;

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

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

This will produce the following result.

Output

```
Integer Value :10
String Value :Hello World
```

Type Parameter Naming Conventions

By convention, type parameter names are named as single, uppercase letters so that a type parameter can be distinguished easily with an ordinary class or interface name. Following is the list of commonly used type parameter names –

- **E** – Element, and is mainly used by Java Collections framework.
- **K** – Key, and is mainly used to represent parameter type of key of a map.
- **V** – Value, and is mainly used to represent parameter type of value of a map.
- **N** – Number, and is mainly used to represent numbers.

T – Type, and is mainly used to represent first generic type parameter.

S – Type, and is mainly used to represent second generic type parameter.

U – Type, and is mainly used to represent third generic type parameter.

V – Type, and is mainly used to represent fourth generic type parameter.

Following example will showcase above mentioned concept.

Example

Create the following java program using any editor of your choice.

GenericsTester.java

Live Demo

```
package com.tutorialspoint;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer, String> box = new Box<Integer, String>();
      box.add(Integer.valueOf(10),"Hello World");
      System.out.printf("Integer Value :%d\n", box.getFirst());
      System.out.printf("String Value :%s\n", box.getSecond());

      Pair<String, Integer> pair = new Pair<String, Integer>();
      pair.addKeyValue("1", Integer.valueOf(10));
      System.out.printf("(Pair)Integer Value :%d\n", pair.getValue("1"));

      CustomList<Box> list = new CustomList<Box>();
      list.addItem(box);
      System.out.printf("(CustomList)Integer Value :%d\n", list.getItem(0).getFirst());
   }
}

class Box<T, S> {
   private T t;
   private S s;

   public void add(T t, S s) {
      this.t = t;
      this.s = s;
   }

   public T getFirst() {
      return t;
   }

   public S getSecond() {
      return s;
   }
}
```

```
class Pair<K,V>{
   private Map<K,V> map = new HashMap<K,V>();

   public void addKeyValue(K key, V value) {
      map.put(key, value);
   }

   public V getValue(K key) {
      return map.get(key);
   }
}

class CustomList<E>{
   private List<E> list = new ArrayList<E>();

   public void addItem(E value) {
      list.add(value);
   }

   public E getItem(int index) {
      return list.get(index);
   }
}
```

This will produce the following result.

Output

```
Integer Value :10
String Value :Hello World
(Pair)Integer Value :10
(CustomList)Integer Value :10
```

Java Generics - Type Inference

Type inference represents the Java compiler's ability to look at a method invocation and its corresponding declaration to check and determine the type argument(s). The inference algorithm checks the types of the arguments and, if available, assigned type is returned. Inference algorithms tries to find a specific type which can fullfill all type parameters.

Compiler generates unchecked conversion warning in-case type inference is not used.

Syntax

```
Box<Integer> integerBox = new Box<>();
```

Where

Box – Box is a generic class.

<> – The diamond operator denotes type inference.

Description

Using diamond operator, compiler determines the type of the parameter. This operator is avalilable from Java SE 7 version onwards.

Example

Create the following java program using any editor of your choice.

GenericsTester.java

Live Demo

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      //type inference
      Box<Integer> integerBox = new Box<>();
      //unchecked conversion warning
      Box<String> stringBox = new Box<String>();

      integerBox.add(new Integer(10));
      stringBox.add(new String("Hello World"));

      System.out.printf("Integer Value :%d\n", integerBox.get());
      System.out.printf("String Value :%s\n", stringBox.get());
   }
}

class Box<T> {
   private T t;

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

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

This will produce the following result.

Output

```
Integer Value :10
String Value :Hello World
```

Java Generics - Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the

compiler handles each method call appropriately. Following are the rules to define Generic Methods –

All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).

Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.

A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example

Following example illustrates how we can print an array of different type using a single Generic method –

Live Demo

```
public class GenericMethodTest {
   // generic method printArray
   public static < E > void printArray( E[] inputArray ) {
      // Display array elements
      for(E element : inputArray) {
         System.out.printf("%s ", element);
      }
      System.out.println();
   }

   public static void main(String args[]) {
      // Create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println("Array integerArray contains:");
      printArray(intArray);   // pass an Integer array

      System.out.println("\nArray doubleArray contains:");
      printArray(doubleArray);   // pass a Double array

      System.out.println("\nArray characterArray contains:");
      printArray(charArray);   // pass a Character array
   }
}
```

This will produce the following result –

Output

```
Array integerArray contains:
1 2 3 4 5

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

Java Generics - Multiple Type Parameters

A Generic class can have muliple type parameters. Following example will showcase above mentioned concept.

Example

Create the following java program using any editor of your choice.

GenericsTester.java

Live Demo

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer, String> box = new Box<Integer, String>();
      box.add(Integer.valueOf(10),"Hello World");
      System.out.printf("Integer Value :%d\n", box.getFirst());
      System.out.printf("String Value :%s\n", box.getSecond());

      Box<String, String> box1 = new Box<String, String>();
      box1.add("Message","Hello World");
      System.out.printf("String Value :%s\n", box1.getFirst());
      System.out.printf("String Value :%s\n", box1.getSecond());
   }
}

class Box<T, S> {
   private T t;
   private S s;

   public void add(T t, S s) {
      this.t = t;
      this.s = s;
   }

   public T getFirst() {
      return t;
   }

   public S getSecond() {
      return s;
```

```
   }
}
```

This will produce the following result.

Output

```
Integer Value :10
String Value :Hello World
String Value :Message
String Value :Hello World
```

Java Generics - Parameterized Types

A Generic class can have parameterized types where a type parameter can be substituted with a parameterized type. Following example will showcase above mentioned concept.

Example

Create the following java program using any editor of your choice.

GenericsTester.java

Live Demo

```
package com.tutorialspoint;

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


public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer, List<String>> box
         = new Box<Integer, List<String>>();

      List<String> messages = new ArrayList<String>();

      messages.add("Hi");
      messages.add("Hello");
      messages.add("Bye");

      box.add(Integer.valueOf(10),messages);
      System.out.printf("Integer Value :%d\n", box.getFirst());
      System.out.printf("String Value :%s\n", box.getSecond());


   }
}

class Box<T, S> {
   private T t;
   private S s;

   public void add(T t, S s) {
      this.t = t;
```

```
      this.s = s;
   }

   public T getFirst() {
      return t;
   }

   public S getSecond() {
      return s;
   }
}
```

This will produce the following result.

Output

```
Integer Value :10
String Value :[Hi, Hello, Bye]
```

Java Generics - Raw Types

A raw type is an object of a generic class or interface if its type arguments are npt passed during its creation. Following example will showcase above mentioned concept.

Example

Create the following java program using any editor of your choice.

GenericsTester.java

Live Demo

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer> box = new Box<Integer>();

      box.set(Integer.valueOf(10));
      System.out.printf("Integer Value :%d\n", box.getData());


      Box rawBox = new Box();

      //No warning
      rawBox = box;
      System.out.printf("Integer Value :%d\n", rawBox.getData());

      //Warning for unchecked invocation to set(T)
      rawBox.set(Integer.valueOf(10));
      System.out.printf("Integer Value :%d\n", rawBox.getData());

      //Warning for unchecked conversion
      box = rawBox;
      System.out.printf("Integer Value :%d\n", box.getData());
   }
}
```

```
class Box<T> {
   private T t;

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

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

This will produce the following result.

Output

```
Integer Value :10

Integer Value :10

Integer Value :10

Integer Value :10
```

Java Generics - Bounded Type Parameters

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Example

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects –

Live Demo

```
public class MaximumTest {
   // determines the largest of three Comparable objects

   public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
      T max = x;   // assume x is initially the largest

      if(y.compareTo(max) > 0) {
         max = y;   // y is the largest so far
      }

      if(z.compareTo(max) > 0) {
         max = z;   // z is the largest now
      }
      return max;   // returns the largest object
```

```
   }

   public static void main(String args[]) {
      System.out.printf("Max of %d, %d and %d is %d\n\n",
         3, 4, 5, maximum( 3, 4, 5 ));

      System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
         6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

      System.out.printf("Max of %s, %s and %s is %s\n","pear",
         "apple", "orange", maximum("pear", "apple", "orange"));
   }
}
```

This will produce the following result –

Output

```
Max of 3, 4 and 5 is 5


Max of 6.6,8.8 and 7.7 is 8.8


Max of pear, apple and orange is pear
```

Java Generics - Multiple Bounds

A type parameter can have multiple bounds.

Syntax

```
public static <T extends Number & Comparable<T>> T maximum(T x, T y, T z)
```

Where

maximum – maximum is a generic method.

T – The generic type parameter passed to generic method. It can take any Object.

Description

The T is a type parameter passed to the generic class Box and should be subtype of Number class and must implments Comparable interface. In case a class is passed as bound, it should be passed first before interface otherwise compile time error will occur.

Example

Create the following java program using any editor of your choice.

Live Demo

```
package com.tutorialspoint;
```

```
public class GenericsTester {
   public static void main(String[] args) {
      System.out.printf("Max of %d, %d and %d is %d\n\n",
         3, 4, 5, maximum( 3, 4, 5 ));

      System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
         6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));
   }

   public static <T extends Number
      & Comparable<T>> T maximum(T x, T y, T z) {
      T max = x;
      if(y.compareTo(max) > 0) {
         max = y;
      }

      if(z.compareTo(max) > 0) {
         max = z;
      }
      return max;
   }

   // Compiler throws error in case of below declaration
   /* public static <T extends Comparable<T>
      & Number> T maximum1(T x, T y, T z) {
      T max = x;
      if(y.compareTo(max) > 0) {
         max = y;
      }

      if(z.compareTo(max) > 0) {
         max = z;
      }
      return max;
   }*/
}
```

This will produce the following result –

Output

```
Max of 3, 4 and 5 is 5


Max of 6.6,8.8 and 7.7 is 8.8
```

Java Generics - List

Java has provided generic support in List interface.

Syntax

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

Where

list – object of List interface.

T – The generic type parameter passed during list declaration.

Description

The T is a type parameter passed to the generic interface List and its implemenation class ArrayList.

Example

Create the following java program using any editor of your choice.

Live Demo

```
package com.tutorialspoint;

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

public class GenericsTester {
   public static void main(String[] args) {

      List<Integer> integerList = new ArrayList<Integer>();

      integerList.add(Integer.valueOf(10));
      integerList.add(Integer.valueOf(11));

      List<String> stringList = new ArrayList<String>();

      stringList.add("Hello World");
      stringList.add("Hi World");


      System.out.printf("Integer Value :%d\n", integerList.get(0));
      System.out.printf("String Value :%s\n", stringList.get(0));

      for(Integer data: integerList) {
         System.out.printf("Integer Value :%d\n", data);
      }

      Iterator<String> stringIterator = stringList.iterator();

      while(stringIterator.hasNext()) {
         System.out.printf("String Value :%s\n", stringIterator.next());
      }
   }
}
```

This will produce the following result –

Output

```
Integer Value :10
String Value :Hello World
```

```
Integer Value :10
Integer Value :11
String Value :Hello World
String Value :Hi World
```

Java Generics - Set

Java has provided generic support in Set interface.

Syntax

```
Set<T> set = new HashSet<T>();
```

Where

set – object of Set Interface.

T – The generic type parameter passed during set declaration.

Description

The T is a type parameter passed to the generic interface Set and its implemenation class HashSet.

Example

Create the following java program using any editor of your choice.

Live Demo

```
package com.tutorialspoint;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class GenericsTester {
   public static void main(String[] args) {

      Set<Integer> integerSet = new HashSet<Integer>();

      integerSet.add(Integer.valueOf(10));
      integerSet.add(Integer.valueOf(11));

      Set<String> stringSet = new HashSet<String>();

      stringSet.add("Hello World");
      stringSet.add("Hi World");


      for(Integer data: integerSet) {
         System.out.printf("Integer Value :%d\n", data);
      }

      Iterator<String> stringIterator = stringSet.iterator();
```

```
      while(stringIterator.hasNext()) {
         System.out.printf("String Value :%s\n", stringIterator.next());
      }
   }
}
```

This will produce the following result –

Output

```
Integer Value :10
Integer Value :11
String Value :Hello World
String Value :Hi World
```

Java Generics - Map

Java has provided generic support in Map interface.

Syntax

```
Set<T> set = new HashSet<T>();
```

Where

set – object of Set Interface.

T – The generic type parameter passed during set declaration.

Description

The T is a type parameter passed to the generic interface Set and its implemenation class HashSet.

Example

Create the following java program using any editor of your choice.

Live Demo

```
package com.tutorialspoint;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class GenericsTester {
   public static void main(String[] args) {

      Map<Integer,Integer> integerMap
         = new HashMap<Integer,Integer>();
```

```
      integerMap.put(1, 10);
      integerMap.put(2, 11);

      Map<String,String> stringMap = new HashMap<String,String>();

      stringMap.put("1", "Hello World");
      stringMap.put("2","Hi World");


      System.out.printf("Integer Value :%d\n", integerMap.get(1));
      System.out.printf("String Value :%s\n", stringMap.get("1"));

      // iterate keys.
      Iterator<Integer> integerIterator   = integerMap.keySet().iterator();

      while(integerIterator.hasNext()) {
         System.out.printf("Integer Value :%d\n", integerIterator.next());
      }

      // iterate values.
      Iterator<String> stringIterator   = stringMap.values().iterator();

      while(stringIterator.hasNext()) {
         System.out.printf("String Value :%s\n", stringIterator.next());
      }
   }
}
```

This will produce the following result –

Output

```
Integer Value :10
String Value :Hello World
Integer Value :1
Integer Value :2
String Value :Hello World
String Value :Hi World
```

Java Generics - Upper Bounded Wildcards

The question mark (?), represents the wildcard, stands for unknown type in generics. There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.

To declare a upper bounded Wildcard parameter, list the ?, followed by the extends keyword, followed by its upper bound.

Example

Following example illustrates how extends is used to specify an upper bound wildcard.

Live Demo

```
package com.tutorialspoint;

import java.util.Arrays;
import java.util.List;

public class GenericsTester {

   public static double sum(List<? extends Number> numberlist) {
      double sum = 0.0;
      for (Number n : numberlist) sum += n.doubleValue();
      return sum;
   }

   public static void main(String args[]) {
      List<Integer> integerList = Arrays.asList(1, 2, 3);
      System.out.println("sum = " + sum(integerList));

      List<Double> doubleList = Arrays.asList(1.2, 2.3, 3.5);
      System.out.println("sum = " + sum(doubleList));
   }
}
```

This will produce the following result –

Output

```
sum = 6.0
sum = 7.0
```

Java Generics - Unbounded Wildcards

The question mark (?), represents the wildcard, stands for unknown type in generics. There may be times when any object can be used when a method can be implemented using functionality provided in the Object class or When the code is independent of the type parameter.

To declare a Unbounded Wildcard parameter, list the ? only.

Example

Following example illustrates how extends is used to specify an unbounded wildcard.

Live Demo

```
package com.tutorialspoint;

import java.util.Arrays;
import java.util.List;

public class GenericsTester {
   public static void printAll(List<?> list) {
      for (Object item : list)
         System.out.println(item + " ");
   }

   public static void main(String args[]) {
```

```
      List<Integer> integerList = Arrays.asList(1, 2, 3);
      printAll(integerList);
      List<Double> doubleList = Arrays.asList(1.2, 2.3, 3.5);
      printAll(doubleList);
   }
}
```

This will produce the following result –

Output

```
1
2
3
1.2
2.3
3.5
```

Java Generics - Lower Bounded Wildcards

The question mark (?), represents the wildcard, stands for unknown type in generics. There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Integer or its superclasses like Number.

To declare a lower bounded Wildcard parameter, list the ?, followed by the super keyword, followed by its lower bound.

Example

Following example illustrates how super is used to specify an lower bound wildcard.

Live Demo

```
package com.tutorialspoint;

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

public class GenericsTester {

   public static void addCat(List<? super Cat> catList) {
      catList.add(new RedCat());
      System.out.println("Cat Added");
   }

   public static void main(String[] args) {
      List<Animal> animalList= new ArrayList<Animal>();
      List<Cat> catList= new ArrayList<Cat>();
      List<RedCat> redCatList= new ArrayList<RedCat>();
      List<Dog> dogList= new ArrayList<Dog>();

      //add list of super class Animal of Cat class
      addCat(animalList);
```

```
      //add list of Cat class
      addCat(catList);

      //compile time error
      //can not add list of subclass RedCat of Cat class
      //addCat(redCatList);

      //compile time error
      //can not add list of subclass Dog of Superclass Animal of Cat class
      //addCat.addMethod(dogList);
   }
}
class Animal {}

class Cat extends Animal {}

class RedCat extends Cat {}

class Dog extends Animal {}
```

This will produce the following result –

```
Cat Added
Cat Added
```

Java Generics - Guidelines for Wildcard Use

Wildcards can be used in three ways –

Upper bound Wildcard – ? extends Type.

Lower bound Wildcard – ? super Type.

Unbounded Wildcard – ?

In order to decide which type of wildcard best suits the condition, let's first classify the type of parameters passed to a method as **in** and **out** parameter.

in variable – An in variable provides data to the code. For example, copy(src, dest). Here src acts as in variable being data to be copied.

out variable – An out variable holds data updated by the code. For example, copy(src, dest). Here dest acts as in variable having copied data.

Guidelines for Wildcards.

Upper bound wildcard – If a variable is of **in** category, use extends keyword with wildcard.

Lower bound wildcard – If a variable is of **out** category, use super keyword with wildcard.

Unbounded wildcard – If a variable can be accessed using Object class method then use an unbound wildcard.

No wildcard – If code is accessing variable in both **in** and **out** category then do not use wildcards.

Example

Following example illustrates the above mentioned concepts.

Live Demo

```
package com.tutorialspoint;

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

public class GenericsTester {

   //Upper bound wildcard
   //in category
   public static void deleteCat(List<? extends Cat> catList, Cat cat) {
      catList.remove(cat);
      System.out.println("Cat Removed");
   }

   //Lower bound wildcard
   //out category
   public static void addCat(List<? super RedCat> catList) {
      catList.add(new RedCat("Red Cat"));
      System.out.println("Cat Added");
   }

   //Unbounded wildcard
   //Using Object method toString()
   public static void printAll(List<?> list) {
      for (Object item : list)
         System.out.println(item + " ");
   }

   public static void main(String[] args) {

      List<Animal> animalList= new ArrayList<Animal>();
      List<RedCat> redCatList= new ArrayList<RedCat>();

      //add list of super class Animal of Cat class
      addCat(animalList);
      //add list of Cat class
      addCat(redCatList);
      addCat(redCatList);

      //print all animals
      printAll(animalList);
      printAll(redCatList);

      Cat cat = redCatList.get(0);
      //delete cat
      deleteCat(redCatList, cat);
      printAll(redCatList);
```

```
   }
}

class Animal {
   String name;
   Animal(String name) {
      this.name = name;
   }
   public String toString() {
      return name;
   }
}

class Cat extends Animal {
   Cat(String name) {
      super(name);
   }
}

class RedCat extends Cat {
   RedCat(String name) {
      super(name);
   }
}

class Dog extends Animal {
   Dog(String name) {
      super(name);
   }
}
```

This will produce the following result –

```
Cat Added
Cat Added
Cat Added
Red Cat
Red Cat
Red Cat
Cat Removed
Red Cat
```

Java Generics - Type Erasure

Generics are used for tighter type checks at compile time and to provide a generic programming. To implement generic behaviour, java compiler apply type erasure. Type erasure is a process in which compiler replaces a generic parameter with actual class or bridge method. In type erasure, compiler ensures that no extra classes are created and there is no runtime overhead.

Type Erasure rules

Replace type parameters in generic type with their bound if bounded type parameters are used.

Replace type parameters in generic type with Object if unbounded type parameters are used.

Insert type casts to preserve type safety.

Generate bridge methods to keep polymorphism in extended generic types.

Java Generics - Bound Types Erasure

Java Compiler replaces type parameters in generic type with their bound if bounded type parameters are used.

Example

Live Demo

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      Box<Double> doubleBox = new Box<Double>();

      integerBox.add(new Integer(10));
      doubleBox.add(new Double(10.0));

      System.out.printf("Integer Value :%d\n", integerBox.get());
      System.out.printf("Double Value :%s\n", doubleBox.get());
   }
}

class Box<T extends Number> {
   private T t;

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

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

In this case, java compiler will replace T with Number class and after type erasure,compiler will generate bytecode for the following code.

Live Demo

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      Box integerBox = new Box();
      Box doubleBox = new Box();

      integerBox.add(new Integer(10));
```

```
      doubleBox.add(new Double(10.0));

      System.out.printf("Integer Value :%d\n", integerBox.get());
      System.out.printf("Double Value :%s\n", doubleBox.get());
   }
}

class Box {
   private Number t;

   public void add(Number t) {
      this.t = t;
   }

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

In both case, result is same –

Output

```
Integer Value :10
Double Value :10.0
```

Java Generics - Unbounded Types Erasure

Java Compiler replaces type parameters in generic type with Object if unbounded type parameters are used.

Example

Live Demo

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      Box<String> stringBox = new Box<String>();

      integerBox.add(new Integer(10));
      stringBox.add(new String("Hello World"));

      System.out.printf("Integer Value :%d\n", integerBox.get());
      System.out.printf("String Value :%s\n", stringBox.get());
   }
}

class Box<T> {
   private T t;

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

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

In this case, java compiler will replace T with Object class and after type erasure,compiler will generate bytecode for the following code.

Live Demo

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      Box integerBox = new Box();
      Box stringBox = new Box();

      integerBox.add(new Integer(10));
      stringBox.add(new String("Hello World"));

      System.out.printf("Integer Value :%d\n", integerBox.get());
      System.out.printf("String Value :%s\n", stringBox.get());
   }
}

class Box {
   private Object t;

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

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

In both case, result is same –

Output

```
Integer Value :10
String Value :Hello World
```

Java Generics - Generic Methods Erasure

Java Compiler replaces type parameters in generic type with Object if unbounded type parameters are used, and with type if bound parameters are used as method parameters.

Example

Live Demo

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
```

```
      Box<String> stringBox = new Box<String>();

      integerBox.add(new Integer(10));
      stringBox.add(new String("Hello World"));

      printBox(integerBox);
      printBox1(stringBox);
   }

   private static <T extends Box> void printBox(T box) {
      System.out.println("Integer Value :" + box.get());
   }

   private static <T> void printBox1(T box) {
      System.out.println("String Value :" + ((Box)box).get());
   }
}

class Box<T> {
   private T t;

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

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

In this case, java compiler will replace T with Object class and after type erasure,compiler will generate bytecode for the following code.

Live Demo

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      Box integerBox = new Box();
      Box stringBox = new Box();

      integerBox.add(new Integer(10));
      stringBox.add(new String("Hello World"));

      printBox(integerBox);
      printBox1(stringBox);
   }

   //Bounded Types Erasure
   private static void printBox(Box box) {
      System.out.println("Integer Value :" + box.get());
   }

   //Unbounded Types Erasure
   private static void printBox1(Object box) {
      System.out.println("String Value :" + ((Box)box).get());
   }
}

class Box {
```

```
   private Object t;

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

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

In both case, result is same –

Output

```
Integer Value :10
String Value :Hello World
```

Java Generics - No Primitive Types

Using generics, primitive types can not be passed as type parameters. In the example given below, if we pass int primitive type to box class, then compiler will complain. To mitigate the same, we need to pass the Integer object instead of int primitive type.

Example

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();

      //compiler errror
      //ReferenceType
      //- Syntax error, insert "Dimensions" to complete
      ReferenceType
      //Box<int> stringBox = new Box<int>();

      integerBox.add(new Integer(10));
      printBox(integerBox);
   }

   private static void printBox(Box box) {
      System.out.println("Value: " + box.get());
   }
}

class Box<T> {
   private T t;

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

   public T get() {
```

```
      return t;
   }
}
```

This will produce the following result –

Output

```
Value: 10
```

Java Generics - No Instance

A type parameter cannot be used to instantiate its object inside a method.

```
public static <T> void add(Box<T> box) {
   //compiler error
   //Cannot instantiate the type T
   //T item = new T();
   //box.add(item);
}
```

To achieve such functionality, use reflection.

```
public static <T> void add(Box<T> box, Class<T> clazz)
   throws InstantiationException, IllegalAccessException{
   T item = clazz.newInstance();   // OK
   box.add(item);
   System.out.println("Item added.");
}
```

Example

Live Demo

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args)
      throws InstantiationException, IllegalAccessException {
      Box<String> stringBox = new Box<String>();
      add(stringBox, String.class);
   }

   public static <T> void add(Box<T> box) {
      //compiler error
      //Cannot instantiate the type T
      //T item = new T();
      //box.add(item);
   }

   public static <T> void add(Box<T> box, Class<T> clazz)
      throws InstantiationException, IllegalAccessException{
      T item = clazz.newInstance();   // OK
      box.add(item);
      System.out.println("Item added.");
   }
}
```

```
class Box<T> {
   private T t;

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

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

This will produce the following result –

```
Item added.
```

Java Generics - No Static field

Using generics, type parameters are not allowed to be static. As static variable is shared among object so compiler can not determine which type to used. Consider the following example if static type parameters were allowed.

Example

```
package com.tutorialspoint;

public class GenericsTester {
   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
          Box<String> stringBox = new Box<String>();

      integerBox.add(new Integer(10));
      printBox(integerBox);
   }

   private static void printBox(Box box) {
      System.out.println("Value: " + box.get());
   }
}

class Box<T> {
   //compiler error
   private static T t;

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

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

As stringBox and integerBox both have a stared static type variable, its type can not be determined. Hence static type parameters are not allowed.

Java Generics - No Cast

Casting to a parameterized type is not allowed unless it is parameterized by unbounded wildcards.

```
Box<Integer> integerBox = new Box<Integer>();
Box<Number> numberBox = new Box<Number>();
//Compiler Error: Cannot cast from Box<Number> to Box<Integer>
integerBox = (Box<Integer>)numberBox;
```

To achive the same, unbounded wildcards can be used.

```
private static void add(Box<?> box) {
   Box<Integer> integerBox = (Box<Integer>)box;
}
```

Java Generics - No instanceOf

Because compiler uses type erasure, the runtime does not keep track of type parameters, so at runtime difference between Box<Integer> and Box<String> cannot be verified using instanceOf operator.

```
Box<Integer> integerBox = new Box<Integer>();

//Compiler Error:
//Cannot perform instanceof check against
//parameterized type Box<Integer>.
//Use the form Box<?> instead since further
//generic type information will be erased at runtime
if(integerBox instanceof Box<Integer>) { }
```

Java Generics - No Array

Arrays of parameterized types are not allowed.

```
//Cannot create a generic array of Box<Integer>
Box<Integer>[] arrayOfLists = new Box<Integer>[2];
```

Because compiler uses type erasure, the type parameter is replaced with Object and user can add any type of object to the array. And at runtime, code will not able to throw ArrayStoreException.

```
// compiler error, but if it is allowed
Object[] stringBoxes = new Box<String>[];

// OK
stringBoxes[0] = new Box<String>();

// An ArrayStoreException should be thrown,
//but the runtime can't detect it.
stringBoxes[1] = new Box<Integer>();
```

Java Generics - No Exception

A generic class is not allowed to extend the Throwable class directly or indirectly.

```
//The generic class Box<T> may not subclass java.lang.Throwable
class Box<T> extends Exception {}

//The generic class Box<T> may not subclass java.lang.Throwable
class Box1<T> extends Throwable {}
```

A method is not allowed to catch an instance of a type parameter.

```
public static <T extends Exception, J>
   void execute(List<J> jobs) {
      try {
         for (J job : jobs) {}

         // compile-time error
         //Cannot use the type parameter T in a catch block
      } catch (T e) {
         // ...
   }
}
```

Type parameters are allowed in a throws clause.

```
class Box<T extends Exception>  {
   private int t;

   public void add(int t) throws T {
      this.t = t;
   }

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

Java Generics - No Overload

A class is not allowed to have two overloaded methods that can have the same signature after type erasure.

```
class Box  {
   //Compiler error
   //Erasure of method print(List<String>)
   //is the same as another method in type Box
   public void print(List<String> stringList) { }
   public void print(List<Integer> integerList) { }
}
```

Previous Page Next Page

Enter email for newsletter go