Generics – Java Tutorials

- Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods.
- (GFAQ)Generics are basically invented to create generic collections type. Because all collections accepts objects of different type hence while retriving them one needs to casts all of them. Hence with Generics one can have homogenous collections.
- (GFAQ)With Generics one can create collections which are homogeneous in nature like LinkedList<String> and LinkedList<Integer> which accepts only String and Integer elements respectively.
- (GFAQ)Generic is useful because type mismatch could be found out during compile time as error rather than during runtime as exceptions.

```
LinkedList<String> ls = new LinkedList<String>();

list.add(“abc”);//fine

list.add(new Date());//error
```

Without Generics above would be

```
LinkedList ls = new LinkedList();

list.add(“abc”);//fine

list.add(new Date());//fine
```

- Generics notations:

T – used to denote type
E – used to denote element
K – keys
V - values
N – for numbers

- Advantages of using Generics
  - Stronger type checks at compile time.

    Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

  - Elimination of casts.
    The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
```

```
String s = (String) list.get(0);
```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

- Enables programmers to implement generic algorithms.

  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

- Generic Types

  Generic type is generic class or interface which is parameterized over types.

  Example:

  Lets consider a non-generic class Box. It has two methods set() to put something inside the box and get() to retrieve it back.

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

We can see that Box accepts and returns an 'object'. One is free to pass in whatever he wants. One part of code may place an 'Integer' and expect 'Integer' back, while other part of code may pass a 'String' resulting in a runtime error.

Generic Class is defined as class name<T1, T2,....Tn> {/* . . . */} here T1, T2,...Tn are type parameters which could be used anywhere inside the class.

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

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

A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable. All types are allowed except enum types, anonymous inner classes and exception classes.

Example:

```
package javaTutorial;

class SimpleGeneric1<T>
{
	T t;
	void setType(T t1)
	{
		this.t = t1;
		System.out.println(t);
	}
	T getType()
	{
		return t;
	}
}

public class SimpleGeneric {
	public static void main(String[] args)
	{
		SimpleGeneric1<String> in = new SimpleGeneric1<String>();
		in.setType("jayant");
		System.out.println(in.getType());
	}
  }
```

```
Output:
jayant
jayant
```

- <u>Type parameter naming convention</u>

  By convention, type parameter names are single, uppercase letters. Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

  The most commonly used type parameter names are.

  E - Element (used extensively by the Java Collections Framework)

  K - Key

  N - Number

  T - Type

  V - Value

  S,U,V etc. - 2nd, 3rd, 4th types

- <u>Invoking and Instantiating a Generic Type</u>

  To reference the generic box class, one must perform ‘generic type invocation’ which is done by replacing type ‘T’ with some concrete value.

  ```
  Ex. Box<Integer> integerBox;
  ```

  It is something like method invocation, instead of passing an argument to a method here we pass ‘type argument’ – Integer this case.

Note: There is difference between ‘type parameter’ and ‘type argument’. In above example ‘T’ is type parameter and ‘Integer’ is type arguement.

To instantiate Box class we should do

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

From Java 7, ‘the diamond’ has been introduced, so now instance of Box can be created as

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

- Multiple type parameter

A generic class can have multiple type parameters too.

Example:

```
package javaTutorial;

class MultipleTypeParam1<K, V>
{
   K key;
   V value;
   MultipleTypeParam1(K key1, V value1)
   {
         this.key = key1;
         this.value = value1;
   }
   K getK()
   {
         return key;
   }
   V getV()
   {
         return value;
   }
}
public class MultipleTypeParam
{
   public static void main(String[] args)
   {
         MultipleTypeParam1<Integer, String> mtp1 = new
MultipleTypeParam1<Integer, String>(5, "odd");
         MultipleTypeParam1<String, String> mtp2 = new
MultipleTypeParam1<String, String>("Jayant", "Joshi");
         System.out.println(mtp1.getK());
         System.out.println(mtp1.getV());
         System.out.println(mtp2.getK());
         System.out.println(mtp2.getV());
   }
}
Output:
5
odd
Jayant
Joshi
```

- Parameterized types

- Raw Types
  - A *raw type* is the name of a generic class or interface without any type arguments. For example, given the genericBox class:

    ```
    public class Box<T> {
        public void set(T t) { /* ... */ }
        // ...
    }
    ```

    To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

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

    If the actual type argument is omitted, you create a raw type of Box<T>:

    ```
    Box rawBox = new Box();
    ```

    Therefore, Box is the raw type of the generic type Box<T>. However, a non-generic class or interface type is *not* a raw type.

  - (GFAQ)Raw types have been permitted to facilitate interfacing with non-generic (legacy) code.
  - For backward compatibility, assigning a parameterized type to its raw type is allowed:

    ```
    Box<String> stringBox = new Box<>();
    Box rawBox = stringBox;               // OK
    ```

    But if you assign a raw type to a parameterized type, you get a warning:

    ```
    Box rawBox = new Box();           // rawBox is a raw type of Box<T>
    Box<Integer> intBox = rawBox;     // warning: unchecked conversion
    ```

  - You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

    ```
    Box<String> stringBox = new Box<>();
    Box rawBox = stringBox;
    rawBox.set(8);  // warning: unchecked invocation to set(T)
    ```

    Example:

    ```
    package javaTutorial;

    class RawTypesClass1<R>
    {
          R r;
    ```

```
        void SetRaw(R r1)
        {
                r = r1;
        }
        void PrintRaw()
        {
                System.out.println(r);
        }
}

public class RawTypesClass {

        public static void main(String[] args)
        {
                RawTypesClass1<Integer> rt1 = new RawTypesClass1<Integer>();
                rt1.SetRaw(5);
                rt1.PrintRaw();

//Warning:RawTypesClass1 is a raw type. References to generic type
RawTypesClass1<R> should be parameterized


        /*warning on this line*/RawTypesClass1 rt2 = new RawTypesClass1();
        /*warning*/rt2.SetRaw(10);
                rt2.PrintRaw();
                rt2 = rt1;// assigning parameterized type to its raw type is
allowed
                rt2.PrintRaw();
                /*warning*/rt1 = rt2;//reverse generate a warning
                rt1.PrintRaw();
        }
}
Output:
5
10
5
5
```

- Generic methods

    - Generic methods are the one which introduces their own generic type but the scope of type parameter is limited to scope.
    - Syntax:
      The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.
    - Example:

```
package javaTutorial;

class GenericMethClass1<T>
{
   T t;
   GenericMethClass1(T t1)
   {
```

```
		t = t1;
	}
	<I> void meth1(I i1)
	{
		if(i1 == t)
		{
			System.out.println("success");
		}
		if(i1 != t)
		{
			System.out.println("failure");
		}
	}
}

public class GenericMethClass
{
	public static void main(String[] args)
	{
		GenericMethClass1<Integer> gm = new
GenericMethClass1<Integer>(10);
		gm.<Integer>meth1(10);
		//gm.<String>meth1(10);//The parameterized method
<String>meth1(String) of type GenericMethClass1<Integer> is not
applicable for the arguments (Integer)
		//gm.<Integer>meth1("Jayant");//The parameterized method
<Integer>meth1(Integer) of type GenericMethClass1<Integer> is not
applicable for the arguments (String)
		GenericMethClass1<String> gm1 = new
GenericMethClass1<String>("String");
		gm1.<String>meth1("Jayant");
	}
}
Output:
success
failure
```

Example 2:

```
package generics;

class GenericMeth1<T>
{
	T t;
	GenericMeth1(T t1)
	{
		t = t1;
	}
	<M extends String> void genMethod(M m)
	{
		if (m == t)
		{
			System.out.println("success");
		}
		else
```

```
            if (m != t)
            {
                  System.out.println("failure");
            }

      }
}

public class GenericMeth
{
      public static void main(String[] args)
      {
            GenericMeth1<String> gm11 = new GenericMeth1<>("jayant");
            GenericMeth1<Integer> gm12 = new GenericMeth1<>(10);

            gm11.<String>genMethod("jayant");
            gm12.<String>genMethod("joshi");

      }
}
Output:
success
failure
```

- <u>Bounded Type Parameters</u>

    - Sometimes there could be times when one needs to restrict type argument in parameterized types. In the previous program we have seen that method 'meth1' has been used for both <Integer> and <String> type arguments. But if one wanted to restrict them to particular type argument then java provide 'bounded type parameters'.
    - To declare bound type parameter, list the type parameter's name, followed by extends keyword, followed by upper bound. Example is shown below.

```
package javaTutorial;

class BoundedTypeParameters1<I extends Number>
{
      I i1;
      void set(I i2)
      {
            i1 = i2;
      }
      void printvalue()
      {
            System.out.println("value is "+i1);
      }
}

public class BoundedTypeParameters
{
      public static void main(String[] args)
      {
            BoundedTypeParameters1<Integer> bt = new
            BoundedTypeParameters1<Integer>();
            bt.set(5);
            bt.printvalue();
            //BoundedTypeParameters1<String> bt1 = new
   BoundedTypeParameters1<String>();
            //Error:Bound mismatch: The type String is not a valid
   substitute for the bounded parameter <I extends Number> of the type
   BoundedTypeParameters1<I>
      }
   }
   Output:
   value is 5
```

- We can have classes and Interfaces too as bounded types.
  - When we have a class statement as Class A <T extends SimpleClass> it means that T can only be SimpleClass or its subclasses.
  - Example:

```
package javaTutorial;

class AppleClass<C>
{
      C color1;
      AppleClass(C color)
      {
            color1 = color;
      }
      void setColor(C color)
      {
            color1 = color1;
      }
      C getColor()
      {
            return color1;
      }
}

class RedAppleContainer<A extends AppleClass>
{
      A apple;
      void checkApple(A apple1)
      {
            if(apple1.color1 == "red")
            {
                  apple = apple1;
                  System.out.println("Its a red apple");
            }
            else
            {
                  System.out.println("Its not a red apple its
                  "+apple1.color1);
            }
      }
}

class SmallApple<S> extends AppleClass<S>
{
      S color2;
```

```
        SmallApple(S color)
        {
              super(color);
              color2 = color;
        }
        S getColor1()
        {
              return color2;
        }
}

public class AppleContainerClass {

        public static void main(String[] args)
        {
              AppleClass<String> apple1 = new AppleClass<String>("red");
              System.out.println("color of apple is "+apple1.getColor());

              AppleClass<String> apple2 = new AppleClass<String>("green");
              System.out.println("color of apple is "+apple2.getColor());

              SmallApple<String> sa = new SmallApple<String>("red");
              System.out.println("color of small apple is 
"+sa.getColor1());

              SmallApple<String> sa2 = new SmallApple<String>("green");
              System.out.println("color of small apple is 
"+sa2.getColor1());


              RedAppleContainer<AppleClass<String>> rac = new 
              RedAppleContainer<AppleClass<String>>();
              rac.checkApple(apple1);
              rac.checkApple(apple2);
              rac.checkApple(sa);
              rac.checkApple(sa2);

              RedAppleContainer<SmallApple<String>> sac = new 
              RedAppleContainer<SmallApple<String>>();
              sac.checkApple(sa);
        }
}
```

```
Output:
color of apple is red
color of apple is green
color of small apple is red
color of small apple is green
Its a red apple
Its not a red apple its green
Its a red apple
Its not a red apple its green
Its a red apple
```

- Generic Method:
  The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

```
package javaTutorial;

class Pair<K, V>
{
   K key;
   V value;
   public Pair(K k1, V v1)
   {
         this.key = k1;
         this.value = v1;
   }
   public void setKey(K key1)
   {
         this.key = key1;
   }
   public void setValue(V value1)
   {
         this.value = value1;
   }
   public K getKey()
   {
         return key;
   }
   public V getValue()
   {
         return value;
   }
}

class Util
{
   //generic method
   public static <K, V> boolean compare(Pair<K,V> p1, Pair<K, V> p2)
   {
          return p1.getKey().equals(p2.getKey()) &&
         p1.getValue().equals(p2.getValue());
   }
}

public class GenericMethodClass
{
   public static void main(String[] args)
   {
         Pair<Integer, String> p1 = new Pair<Integer, String>(1, "apple");
         Pair<Integer, String> p2 = new Pair<Integer, String>(2, "pear");
         Pair<Integer, String> p3 = new Pair<Integer, String>(1, "apple");
         boolean same1 = Util.compare(p1, p2);
         boolean same2 = Util.compare(p1, p3);
         System.out.println(same1);
         System.out.println(same2);
   }
}
Output
false
true
```

Example 2

```
package javaTutorial;

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); //its printf
         }
         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
    }
   }
   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
```

- Like raw types we can call generic methods also as shown below

```
package generics;

class GenMethCheck1
{
      <T> void method()
      {
```

```
                System.out.println("Inside method");
        }
}

public class GenMethCheck
{
        public static void main(String[] args)
        {
                GenMethCheck1 gmc = new GenMethCheck1();
                gmc.method();
        }
}
Output:
Inside method
```

- Generic Interface:
    - Example

```
package javaTutorial;

        interface StringInterface<S extends String>
        {
                void printString(S s);
        }

        class GenStringClass<S extends String> implements
        StringInterface<String>
        {
                public void printString(String str)
                {
                        System.out.println(str);
                }
        }

        public class GenStringInterfaceClass
        {
                public static void main(String[] args)
                {
                        GenStringClass<String> gsc1 = new GenStringClass<String>();
                        gsc1.printString("java");
                        gsc1.printString("generics");
                        //GenStringClass<Integer> gsc2 = new
        GenStringClass<Integer>();//bound mismatch Integer
                }
        }
        Output:
        java
        generics
```

- Comparable Interface:

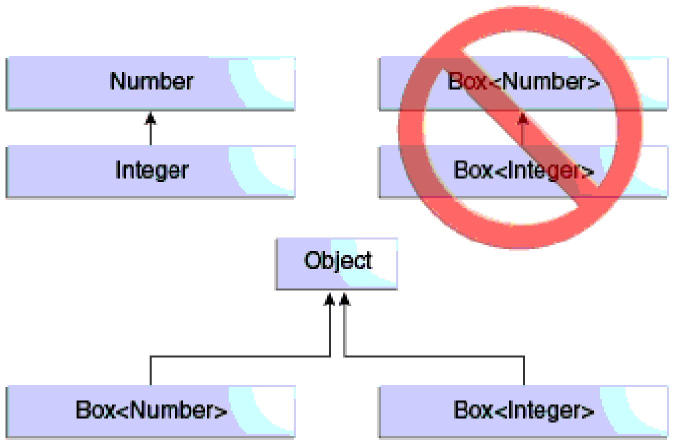
```
    package javaTutorial;

    class CompareToClass<T extends String> implements Comparable<T>
    {
```

```
    T t;
    CompareToClass(T t1)
    {
          t = t1;
    }
    public int compareTo(T t2)
    {
          int l = t.length() - t2.length();
          return l;
    }
}

public class GenCompareToInterfaceClass
{
    public static void main(String[] args)
    {
          CompareToClass<String> ctc = new
CompareToClass<String>("Generic");
          System.out.println(ctc.compareTo("java"));
    }
}
Output:
3
```

- Multiple Bounds:
- Suppose we have a class
  `public void boxTest(Box<Number> n) { /* ... */ }`
  One can wonder what is the argument boxTest() will accept ? It may seem that Box<Number> could accept Box<Integer> or Box<Double> but its not correct. Given two concrete types A and B (for example, Number and Integer), MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related. The common parent of MyClass<A> and MyClass<B> is Object.


Number
Integer
Box<Number>
Box<Integer>
Object
Box<Number>
Box<Integer>