Generics

Generics means parameterized types. It was first introduced in Java 5. It enables the programmer to create classes, interfaces and methods in which type of data is specified as a parameter.
The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Why Generics ?

- The usage of Object class supports the reference to all other classes however type safety is not provided by Object class. Generics provides the type safety.
- Type safety ensures that an operation is being performed on the right type of data.
- Generics also support the code reusability- can create a class,method or interface that works with all types of data. No need to create different methods or interfaces or classes for different data types separately.
- Compile time type checking is done with generics so there won't be error at runtime.
- There is no necessity to typecast an object.

Restrictions and limitations on generics

- Primitive types cannot be used in generics.
  A generic class cannot extend Throwable tytpes.These declarations are incorrect
  class C<T> extends Throwable{…}
  class C<T> extends Exception{...}
- An instance of type parameter cannot be created as shown in the code below.
  public static<E> void append(ArrayList<E> b)
  {
  E e=new E();
  b.add(e);
  }
- The arrays of parametrized types may not be created. The following is syntactically wrong.
  new ArrayList<Integer>[2]
  However the following is correct:
  ArrayList<Integer>[] al=new ArrayList[2];
- Static fields of type parameters are not allowed. The code below is incorrrect.
  public class Test<T>
  {
  static T t1;
  }

Generic Class

A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

Example of Generic Class:

```
class Test<T> {
T a,b;
public void printData() {
System.out.println(a+ " "+b);
}
}
class Generics1{
public static void main(String[] args) {
Test<String> t=new Test< >();
t.a= "Hello";
t.b= "Ashesh";
t.printData();
Test<Integer> t1=new Test< >();
t1.a=1;
t1.b=2;
t1.printData();
}
}
```

Another example with multiple generic parameter.

```
class Test2<T1,T2,T3> { //Generic class can have multiple generic parameter types.
T1 a,b;
T2 c,d;
T3 e,f;
public void printData() {
System.out.println(a);
System.out.println(b);
System.out.println(c);
System.out.println(d);
System.out.println(e);
System.out.println(f);
}
}
class GenericsDemo{
public static void main(String[] args) {
Test2<Integer,String,Double> t1=new Test2< >();
t1.a=1;
t1.b=2;
t1.c="hello";
t1.d="world";
t1.e=2.3;
t1.f=3.4;
t1.printData();
}
}
```

Output

```
1
2
hello
world
2.3
3.4
```

Generic Method

Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.
Example of Generic Method:

```
package Generics;
class ProcessArray{
public static <T>void printArray(T[] array) {
for (T element : array) {
System.out.println(element);
}
}
public static <T> T getValue(T[] array, int index) {
return array[index];
}
}
public class Generics2 {
public static void main(String[] args) {
Integer[] i={1,2,3};
String[] s={“Ashesh”,“neupane”,“hi”};
ProcessArray.printArray(i);
ProcessArray.printArray(s);
}
}
```

Generic Constructor

A constructor can also be of generic type. Example of Generic Constructor:

```
package Generics;
class GenConst{
public <T> GenConst(T a, T b){
System.out.println(a);
System.out.println(b);
}
}
public class GenericConstructorDemo {
public static void main(String[] args) {
GenConst g=new GenConst(1,2);
}
}
```

Generic Interfaces

Generic interfaces are specified just like a generic class. It can be implemented for different types of data. It allows us to put constraints on type of data for which interface can be implemented.
General Syntax:

interface interface_name<type_parameter_list>

After the generic interface is implemetned the class implementing the interface should have following syntax:

class class_name implements interface_name<type_parameter_list>{}

```
package Generics;
interface Example<T> {
void add(T t);
}
class GenInterface implements Example<Integer>{
int t;
public void add(Integer t) {
this.t=t;
}
public int get(){
return t;
}
}
public class GenericInterface {
public static void main(String[] args) {
GenInterface ob=new GenInterface();
ob.add(10);
System.out.println(ob.get());
}
}
```

Bound types for generic types:

- ☑ limiting the certain datatypes for the generic types.
- ☑ by creating the bound types compiler accepts only certain datatypes for generics.
- ☑ bound can be created as:
  class class_name<T extends superclass>{…}
  this will only accept the superclass types as generic parameter.

Example:

```
class MyNumber<T extends Number>{
T a,b;
public void sum(){
System.out.println(a.getClass().getName());
if(a.getClass()==Integer.class)
System.out.println((Integer)a+(Integer)b);
}
}
public class Boundtype {
public static void main(String[] args) {
MyNumber<Integer> t=new MyNumber< >();
t.a=1;
t.b=2;
t.sum();
}
}
```

Wildcard and Generics

Wildcard in java generics is used to specify unknown types. It can accept any generic parameter type. How ever upper and lower bounds can be set on the wildcard to provide some restriction on data type.

```
package Generics;
import java.lang.reflect.Array;
import java.util.ArrayList;
//Wild card in java generics is used to specify unknown type.
class Employee{
public void work(){
System.out.println("working...");
}
}
class Programmer extends Employee{
String name;
public Programmer(String name){
this.name=name;
}
public void work(){
System.out.println(name+" Developing software....");
}
}
public class Wildcarddemo {
public static void main(String[] args) {
ArrayList <Employee> e=new ArrayList< >();
ArrayList <Programmer> p= new ArrayList< >();
p.add(new Programmer("Ashesh"));
p.add(new Programmer("Neupane"));
//e=p; This statement will give compile time error
ArrayList<?> elist=new ArrayList< >();
elist= p;
ArrayList<? extends Employee> elist2=new ArrayList< >();
//Setting the upper bound for ? type. upper bound accepts the Employee and it's child only
elist2=p;
ArrayList<Integer> ilist=new ArrayList< >();
elist=ilist;
//elist2=ilist; This will give compile time error.
ArrayList<? super Employee> elist3=new ArrayList< >();
//setting the lower bound for ? type.lower bound accepts the emloyee type and it's parent type only elist3=p;
for (Employee val:elist2) {
val.work();
}
}
}
```

Output

```
Ashesh Developing software....

Neupane Developing software.…
```

Generic Code and Virtual Machine

The virtual machine does not have objects of generic types- all objects belong to ordinary classes. Even the earlier version of the virtual machine can run the bytecode that have been compiled with the java code with generics.

The class files are rewritten in such a way that they can run on older virtual machines. Whenever the generic type is defined it's corresponding raw type is automatically provided. The name of the raw type is simply the name of generic type with parameters removed. The type variables are removed and replaced by their bouding type(variables with bounds) or Object(for variables without bounds).

```
//This is a generic type
class Test&lt;T&gt;{
private T a,b;
public void setValues(T a,T b) {
this.a=a;
this.b=b;
}
public void printData()
{
System.out.println(a+“ ”+b);
}
}
//This is a raw type
class Test{
private Object a,b;
public void setValues(Object a,Object b) {
this.a=a;
this.b=b;
}
public void printData() {
System.out.println(a+&amp;quot; &amp;quot;+b);
}
}
```

Inheritance Rules for Generics

A generic class can extend a non-generic class.

```
class NonGenericClass {
   //Non Generic Class
}
class GenericClass<T> extends NonGenericClass {
   //Generic class extending non-generic class
}
```

- Generic class can also extend another generic class. When generic class extends another generic class, sub class should have at least same type and same number of type parameters and at most can have any number and any type of parameters.

```
class GenericSuperClass<T> {
    //Generic super class with one type parameter
}
class GenericSubClass1<T> extends GenericSuperClass<T> {
    //sub class with same type parameter
}

class GenericSubClass2<T, V> extends GenericSuperClass<T> {

    //sub class with two type parameters
}
class GenericSubClass3<T1, T2> extends GenericSuperClass<T> {
    //Compile time error, sub class having different type of parameters
}
```

- When generic class extends another generic class, the type parameters are passed from sub class to super class same as in the case of constructor chaining where super class constructor is called by sub class constructor by passing required arguments. For example, in the below program 'T' in 'GenericSuperClass' will be replaced by String.

```
class GenericSuperClass<T> {
T t;
public GenericSuperClass(T t) {
this.t = t;
}
}
class GenericSubClass<T> extends GenericSuperClass<T> {
public GenericSubClass(T t) {
super(t);
}
}
public class GenericsInJava {
public static void main(String[] args) {
GenericSubClass<String> gen = new GenericSubClass<String>("I am string");
System.out.println(gen.t); //Output : I am string
}
}
```

- A generic class can extend only one generic class and one or more generic interfaces.

```
class GenericSuperClass<T1> {
  //Generic class with one type parameter
}
interface GenericInterface1<T1, T2> {
   //Generic interface with two type parameters
}
interface GenericInterface2<T2, T3> {
   //Generic interface with two type parameters
}
class GenericClass<T1,T2, T3> extends GenericSuperClass<T1> implements GenericInterface1<T1, T2>,
GenericInterface2<T2, T3> {
   //Class having parameters of both the interfaces and super class
}
```

- Non-generic class can't extend generic class except of those generic classes which have already pre defined types as their type parameters.

```
class GenericSuperClass<T> {
   //Generic class with one type parameter
}
class NonGenericClass extends GenericSuperClass<T> {
   //Compile time error, non-generic class can't extend generic class
}
class A {
   //Pre defined class
}
class GenericSuperClass1<A> {
   //Generic class with pre defined type 'A' as type parameter
}
class NonGenericClass1 extends GenericSuperClass1<A> {
   //No compile time error, It is legal
}
```

- Non-generic class can extend generic class by removing the type parameters. i.e as a raw type. But, it gives a warning.

```
class GenericClass<T> {
   T t;
   public GenericClass(T t) {
      this.t = t;
   }
}
class NonGenericClass extends GenericClass      //Warning
{
   public NonGenericClass(String s) {
      super(s);          //Warning
   }
}
public class GenericsInJava {
   public static void main(String[] args)  {
      NonGenericClass nonGen = new NonGenericClass("I am String");
      System.out.println(nonGen.t);    //Output : I am String
   }
}
```

- While extending a generic class having bounded type parameter, type parameter must be replaced by either upper bound or it's sub classes.

```
class GenericSuperClass<T extends Number> {
   //Generic super class with bounded type parameter
}
class GenericSubClass1 extends GenericSuperClass<Number> {
   //type parameter replaced by upper bound
}
class GenericSubClass2 extends GenericSuperClass<Integer> {
   //type parameter replaced by sub class of upper bound
}
class GenericSubClass3 extends GenericSuperClass<T extends Number> {
   //Compile time error
}
   • Generic methods of super class can be overriden in the sub class like normal methods.
class GenericClass {
   <T> void genericMethod(T t) {
      System.out.println(1);
   }
}
class NonGenericClass extends GenericClass {
   @Override
   <T> void genericMethod(T t)
   {
      System.out.println(2);
   }
}
public class GenericsInJava
{
   public static void main(String[] args)
   {
      new GenericClass().genericMethod("I am String");      //Output : 1
      new NonGenericClass().genericMethod("I am String");   //Output : 2
   }
}
```

Relection and Generics:

Reflection allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program like getting the methods,constructors, variables declared in the class and many more.

In the code below the reflection is used to get the parameter type of the generics using the java reflection.

```
import java.util.ArrayList;

import java.util.List;
import java.lang.reflect.*;
class MyClass {
   protected List<String> stringList = new ArrayList<>();
   public void setStringList(List<String> list){
      this.stringList = list;
   }
}
public class GenericsandReflection {
   public static void main(String[] args) {
      try {
         Method method = MyClass.class.getMethod("setStringList", List.class);
         Type[] genericParameterTypes = method.getGenericParameterTypes();
         for (Type genericParameterType : genericParameterTypes) {
            if (genericParameterType instanceof ParameterizedType) {
               ParameterizedType aType = (ParameterizedType) genericParameterType;
               Type[] parameterArgTypes = aType.getActualTypeArguments();
               for (Type parameterArgType : parameterArgTypes) {
                  Class parameterArgClass = (Class) parameterArgType;
                  System.out.println("parameterArgClass = " + parameterArgClass);
               }
            }
         }
      } catch (Exception ex) {
         System.out.println(ex);
      }
   }
}
```