Restriction of Genric class

* Type parameters cannot be instantiated
* Restriction on using static members
* Generic array Restrictions
* primitive data types are not used with generic types
* Generic Exception Restriction

**1. Type parameters cannot be instantiated**

It is not possible to create an instance of a type parameter. The compiler doesn't know what type of object to create, their T is simply a placeholder. Below is the example code that shows the invalid creation of an instance of T which leads to a compilation error.

### ****2.Restriction on using static members****

It is illegal for the static members(variables, methods) to use a type parameter declared by the enclosing class. We will use the above example in this case to make the variable and method static, this also leads to compile time error.

// Java Program to demonstrate Generic class creation with

// type parameter T.

class GenType<T> {

// illegal to make a variable as static.

// static T data; //compile-time Error:Cannot make a

// static reference to the non-static type T

T data;

GenType() {}

GenType(T data)

{

// parameterized constructor

this.data = data;

}

T getData() { return data; }

public static void main(String[] args)

{

GenType<String> gt

= new GenType<>("Geek For Geeks!!");

System.out.println(gt.getData());

}

}

### ****3. Generic array Restrictions****

There are two important generic restrictions that applied to arrays.

1. We cannot instantiate an array whose element type is a type parameter. There is no way for the compiler to know what type of array to actually create. However, we can pass a reference to a type-compatible array as a parameter and assign it to the object created. This is acceptable because the array passed as a parameter has a known type, which will be of the same type as T at the time of object creation.
2. We cannot create an array of type-specific generic references. The reason could be the same as in the above case the compiler doesn't know what kind of array to create. This can be resolved by using a wildcard, which is better than using a raw type because at least some type checking will be done.

// Java Program to implement

// Generic array Restrictions

import java.util.Arrays;

public class GenArray<T extends Number> {

T obj;

T arr[];

GenArray(T o, T[] vals)

{

this.obj = o;

System.out.println("value: " + obj);

// Invalid

// arr = new T[10];

// compile-time Error:Cannot

// create a generic array of T

// But, this is allowed because we are assigning the

// reference to the existing array.

arr = vals;

}

T[] getArray() { return arr; }

public static void main(String[] args)

{

Integer[] Array = { 1, 2, 3, 4, 5 };

GenArray<Integer> obj1

= new GenArray<Integer>(10, Array);

System.out.println(

Arrays.toString(obj1.getArray()));

// illegal to create an array of type-specific

// generic references.

}

}

### ****4. Primitive data types are not used with the generic types****

We will get the compilation error if we use the primitive data types at the time of object creation. The following code demonstrates the situation:

// Java Program to implement

// Primitive data types are not

// used with the generic types

import java.io.\*;

// Driver Class

class Box<T> {

private T data;

Box(T data) { this.data = data; }

T getData() { return data; }

public static void main(String[] args)

{

Box<Integer> b1 = new Box<Integer>(10);

// use of wrapper classes

Box<String> b2 = new Box<String>("Geek For Geeks");

System.out.println("value: " + b1.getData());

System.out.println("value: " + b2.getData());

}

}

### ****5. Generic Exception Restriction****

We cannot create generic exception classes and cannot extend **throwable**(which is superior to all exception classes in the exception class hierarchy)**.**We will use the above example to understand this, although we get an error if we execute this code, give it a try.

// Java Program to implement

// Generic Exception Restriction

import java.io.\*;

// generic class cannot extend throwable

class Box<T> extends Throwable {

private T data;

Box(T data) { this.data = data; }

T getData() { return data; }

// main function

public static void main(String[] args)

{

Box<Integer> b1 = new Box<Integer>(10);

System.out.println("value: " + b1.getData());

}

}

Output:

Error: Could not find or load main class Box

Caused by: java.lang.ClassNotFoundException: Box