# Java Technologies Week 4

Prepared for: DDU CE Semester 4

Prepared by: Prof. Niyati J. Buch

# Topics

- Generics
  - Need of Generics
  - Generic class with more than one type parameter
  - Bounded Types
  - Use of wildcard
  - Bounded wildcard
  - Generic method / constructor in non-generic class
  - Generic interface
  - Raw type
  - Generics and inheritance
  - Generics and polymorphism
  - Restrictions in generics

# Pre-generics → Generics

- In pre-generics code, generalized classes, interfaces, and methods used Object references to operate on various types of objects.
- The problem was that they could not do so with type safety.
- Generics added the type safety.
- They also streamlined the process, because it is no longer necessary to explicitly employ casts to translate between Object and the type of data that is actually being operated upon.
- With generics, all casts are automatic and implicit.
- Thus, generics expanded your ability to reuse code and let you do so safely and easily.
- A class, interface, or method that operates on a parameterized type is called generic, as in generic class or generic method.

# Program for all types without Generic. Using Object class.

```
class NonGen {
    Object ob;

    NonGen(Object o) {
        ob = o;
    }

    Object getob() {
        return ob; // Return type Object.
    }

    void showType() {
        System.out.println("Type of ob is " + ob.getClass().getName());
    }
}
```

```
public class NonGenDemo {

  public static void main(String args[]) {
    NonGen iOb;
    iOb = new NonGen(88); // autoboxing
    iOb.showType();

    int v = (Integer) iOb.getob();
    System.out.println("value: " + v);

    NonGen strOb = new NonGen("Non-Generics Test");
    strOb.showType();

    String str = (String) strOb.getob();
    System.out.println("value: " + str);

    iOb = strOb; // This compiles, but is conceptually wrong!
    v = (Integer) iOb.getob(); // run-time exception!
  }
}
```

```
OUTPUT
Type of ob is java.lang.Integer
value: 88
Type of ob is java.lang.String
value: Non-Generics Test
Exception in thread "main"
java.lang.ClassCastException:
class java.lang.String cannot
be cast to class
java.lang.Integer
```

```
class Gen<T> {

    T ob;

    Gen(T o) {
        ob = o;
    }

    T getob() {
        return ob;
    }

    void showType() {
        System.out.println("Type of ob is " + ob.getClass().getName());
    }
}
```

```
public class GenDemo {

    public static void main(String args[]) {
        Gen<Integer> iOb;
        iOb = new Gen<Integer>(88);
        iOb.showType();

        int v = iOb.getob();
        System.out.println("value: " + v);
        System.out.println();

        Gen<String> strOb = new Gen<String>("Generics Test");
        strOb.showType();
        String str = strOb.getob();
        System.out.println("value: " + str);

        //iOb = strOb;
        //error: incompatible types:
        //Gen<String> cannot be converted to Gen<Integer>
    }
}
```

```
OUTPUT
Type of ob is java.lang.Integer
value: 88

Type of ob is java.lang.String
value: Generics Test
```

# Points to note

1. Generics Work Only with Reference Types
   - Gen<int> intOb = new Gen<int>(53); // Error, can't use primitive type
   - So, use the wrapper classes instead.
2. Generic Types Differ Based on Their Type Arguments
   - iOb = strOb; // Wrong!
   - Even though both iOb and strOb are of type Gen<T>, they are references to different types because their type parameters differ.
   - This is part of the way that generics add **type safety and prevent errors.**

# A Generic Class with Two Type Parameters

- You can declare more than one type parameter in a generic type.
- To specify two or more type parameters, simply use a comma-separated list.

```
class TwoGen<T, V> {

    T ob1;
    V ob2;

   public  TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }

    public void showTypes() {
        System.out.println("Type of T is " + ob1.getClass().getName());
        System.out.println("Type of V is " + ob2.getClass().getName());
    }

    public T getOb1() {
        return ob1;
    }

    public V getOb2() {
        return ob2;
    }

}
```

```
public class TwoGenDemo {

   public static void main(String args[]) {
      TwoGen<Integer, String> tgObj = new TwoGen<Integer, String>(88, "Generics");
      tgObj.showTypes();

      int v = tgObj.getOb1();
      System.out.println("value: " + v);

      String str = tgObj.getOb2();
      System.out.println("value: " + str);

   }
}
```

```
OUTPUT
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

# Bounded Types

- Assume that you want to create a generic class that contains a method that returns the average of an array of numbers.
- Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, floats, and doubles.
- Thus, you want to specify the type of the numbers generically, using a type parameter.

```
class Stats<T extends Number> {

   T[] nums; // array of Number or subclass

   Stats(T[] o) {
      nums = o;
   }

   double average() {
      double sum = 0.0;
      for (int i = 0; i < nums.length; i++) {
         sum += nums[i].doubleValue();
      }
      return sum / nums.length;
   }

}
```

```
public class BoundTypesDemo {

    public static void main(String args[]) {
        Integer inums[] = {1, 2, 3, 4, 5};
        Stats<Integer> iob = new Stats<Integer>(inums);
        double v = iob.average();
        System.out.println("iob average is " + v);

        Double dnums[] = {1.1, 2.2, 3.3, 4.4, 5.5};
        Stats<Double> dob = new Stats<Double>(dnums);
        double w = dob.average();
        System.out.println("dob average is " + w);

        // This won't compile because String is not a subclass of Number.
        // String strs[] = { "1", "2", "3", "4", "5" };
        // Stats<String> strob = new Stats<String>(strs);
        // double x = strob.average();
        // System.out.println("strob average is " + v);
    }
}
```

```
OUTPUT
iob average is 3.0
dob average is 3.3
```

```
class Person {

    String name;

    @Override
    public String toString() {
        return "Name = " + name;
    }
}
```

```
class Student extends Person {

    int rollno;

    @Override
    public String toString() {
        return super.toString() + " Rollno = " + rollno;
    }
}
```

```
class Employee extends Person {

    int id;

    @Override
    public String toString() {
        return super.toString() + " Id = " + id;
    }
}
```

```
class GenPeople<T extends Person> {

    T[] people;

    GenPeople(T[] o) {
        people = o;
    }

    void show() {
        for (T p : people) {
            System.out.println(p);
        }
    }
}
```

```
public class BoundTypesPerson {
    public static void main(String args[]) {
        Student s[] = new Student[3];
        for (int i = 0; i < s.length; i++) {
            s[i] = new Student();
            s[i].rollno = i + 1;
        }
        s[0].name = "abc";
        s[1].name = "pqr";
        s[2].name = "xyz";
        GenPeople<Student> ob = new GenPeople<Student>(s);
        ob.show();
        Employee e[] = new Employee[3];
        for (int i = 0; i < e.length; i++) {
            e[i] = new Employee();
            e[i].id = i + 1;
        }
        e[0].name = "ABCD";
        e[1].name = "PQRS";
        e[2].name = "WXYZ";
        GenPeople<Employee> ob1 = new GenPeople<Employee>(e);
        ob1.show();
    }
}
```

```
OUTPUT
Name = abc Rollno = 1
Name = pqr Rollno = 2
Name = xyz Rollno = 3
Name = ABCD Id = 1
Name = PQRS Id = 2
Name = WXYZ Id = 3
```

# Need of wildcard

- Assume that you want to add a method called sameAvg( ) that determines if two Stats objects contain arrays that yield the same average, no matter what type of numeric data each object holds.

```
Integer inums[] = { 1, 2, 3, 4, 5 };
Double dnums[] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);
if(iob.sameAvg(dob))
        System.out.println("Averages are the same.");
else
        System.out.println("Averages differ.");
```

# Is this the sameAvg() to compare averages?

```
boolean sameAvg(Stats<T> ob) {
        if(average() == ob.average())
                return true;
        return false;
}
```

# Issues with this sameAvg() method

```
boolean sameAvg(Stats<T> ob) {
        if(average() == ob.average())
                return true;
        return false;
}
```

- The trouble with this attempt is that it will work only with other Stats objects whose type is the same as the invoking object.
- For example, if the invoking object is of type Stats<Integer>, then the parameter ob must also be of type Stats<Integer>.
- It can’t be used to compare the average of an object of type Stats<Double>with the average of an object of type Stats<Short>, for example.
- Therefore, this approach won’t work except in a very narrow context and does not yield a general (that is, generic) solution.

# Solution: Use the wildcard argument

- The wildcard argument is specified by the ?, and it represents an unknown type.

```
class Stats<T extends Number> {

    T[] nums;

    Stats(T[] o) {
        nums = o;
    }

    double average() {
        double sum = 0.0;
        for (int i = 0; i < nums.length; i++) {
            sum += nums[i].doubleValue();
        }
        return sum / nums.length;
    }

    boolean sameAvg(Stats<?> ob) {
        if (this.average() == ob.average()) {
            return true;
        }
        return false;
    }
}
```

Here, Stats<?> matches any Stats object, allowing any two Stats objects to have their averages compared.

```
public class WildCardDemo {
   public static void main(String args[]) {
      Integer inums[] = {1, 2, 3, 4, 5};
      Stats<Integer> iob = new Stats<Integer>(inums);
      double v = iob.average();
      System.out.println("iob average is " + v);

      Double dnums[] = {1.1, 2.2, 3.3, 4.4, 5.5};
      Stats<Double> dob = new Stats<Double>(dnums);
      double w = dob.average();
      System.out.println("dob average is " + w);

      Float fnums[] = {1.0F, 2.0F, 3.0F, 4.0F, 5.0F};
      Stats<Float> fob = new Stats<Float>(fnums);
      double x = fob.average();
      System.out.println("fob average is " + x);

      System.out.print("Averages of iob and dob ");
      if (iob.sameAvg(dob)) {
         System.out.println("are the same.");
      } else {
         System.out.println("differ.");
      }
   }
}
```

```
System.out.print("Averages of iob and fob ");
      if (iob.sameAvg(fob)) {
         System.out.println("are the same.");
      } else {
         System.out.println("differ.");
      }
```

```
OUTPUT
iob average is 3.0
dob average is 3.3
fob average is 3.0
Averages of iob and dob differ.
Averages of iob and fob are the same.
```

# Bounded Wildcard

```
// Two-dimensional coordinates.
class TwoD {
    int x, y;
    TwoD(int a, int b) {
        x = a;
        y = b;
    }
}
```

```
// Three-dimensional coordinates.
class ThreeD extends TwoD {
    int z;
    ThreeD(int a, int b, int c) {
        super(a, b);
        z = c;
    }
}
```

```
// Four-dimensional coordinates.
class FourD extends ThreeD {
    int t;
    FourD(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }
}
```

```
// This class holds an array of coordinate
//objects.
class Coords<T extends TwoD> {
    T[] coords;
    Coords(T[] o) {
        coords = o;
    }
}
```

```
public class BoundedWildcard {
    static void showXY(Coords<?> c) {
        System.out.println("X Y Coordinates:");
        for (int i = 0; i < c.coords.length; i++) {
            System.out.println(c.coords[i].x + " " + c.coords[i].y);
        }
        System.out.println();
    }
    //static void showXYZ(Coords<?> c) {
    static void showXYZ(Coords<? extends ThreeD> c) {
        System.out.println("X Y Z Coordinates:");
        for (int i = 0; i < c.coords.length; i++) {
            System.out.println(c.coords[i].x + " " + c.coords[i].y+" " + c.coords[i].z);
        }
        System.out.println();
    }
   static void showAll(Coords<? extends FourD> c) {
        System.out.println("X Y Z T Coordinates:");
        for (int i = 0; i < c.coords.length; i++) {
            System.out.println(c.coords[i].x+" " + c.coords[i].y+" " + c.coords[i].z + " " + c.coords[i].t);
        }
        System.out.println();
    }
```

```
  public static void main(String args[]) {
    TwoD td[] = {
      new TwoD(0, 0),          new TwoD(7, 9),
      new TwoD(18, 4),         new TwoD(-1, -23)
    };
    Coords<TwoD> tdlocs = new Coords<TwoD>(td);
    System.out.println("Contents of tdlocs.");
    showXY(tdlocs); // OK, is a TwoD
    // showXYZ(tdlocs); // Error, not a ThreeD
    // showAll(tdlocs); // Error, not a FourD

    // Now, create some FourD objects.
    FourD fd[] = {
      new FourD(1, 2, 3, 4),  new FourD(6, 8, 14, 8),
      new FourD(22, 9, 4, 9), new FourD(3, -2, -23, 17)
    };
    Coords<FourD> fdlocs = new Coords<FourD>(fd);
    System.out.println("Contents of fdlocs.");
    // These are all OK.
    showXY(fdlocs);
    showXYZ(fdlocs);
    showAll(fdlocs);
  }
}
```

```
OUTPUT
Contents of tdlocs.
X Y Coordinates:
0 0
7 9
18 4
-1 -23

Contents of fdlocs.
X Y Coordinates:
1 2
6 8
22 9
3 -2

X Y Z Coordinates:
1 2 3
6 8 14
22 9 4
3 -2 -23

X Y Z T Coordinates:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17
```

# Generic method enclosed within a non-generic class

```
public class GenMethDemo {

   static <T> boolean isIn(T x, T[] y) {
      for (T y1 : y) {
         if (x.equals(y1)) {
            return true;
         }
      }
      return false;
   }
```

```
public static void main(String args[]) {
      Integer nums[] = {1, 2, 3, 4, 5};
      if (isIn(2, nums)) {
         System.out.println("2 is in nums");
      }
      if (!isIn(7, nums)) {
         System.out.println("7 is not in nums");
      }

      String strs[] = {"one", "two", "three", "four", "five"};
      if (isIn("two", strs)) {
         System.out.println("two is in strs");
      }
      if (!isIn("seven", strs)) {
         System.out.println("seven is not in strs");
      }
      // Oops! Won't compile! Types must be compatible.
      // if(isIn("two", nums))
      //      System.out.println("two is in strs");
   }
}
```

```
OUTPUT
2 is in nums
7 is not in nums

two is in strs
seven is not in strs
```

# Generic constructors for non-generic class

```
class GenCons {

    private double val;

    <T extends Number> GenCons(T arg) {
        val = arg.doubleValue();
    }

    void showval() {
        System.out.println("val: " + val);
    }
}
```

```
public class GenConsDemo {

    public static void main(String args[]) {
        GenCons test = new GenCons(100);
        GenCons test2 = new GenCons(123.5F);
        test.showval();
        test2.showval();
    }
}
```

```
OUTPUT
val: 100.0
val: 123.5
```

# Generic Interface

```
interface MinMax<T extends Comparable<T>> {
   T min();
   T max();
}
```

- Comparable is an interface defined by java.lang that specifies how objects are compared.
- Its type parameter specifies the type of the objects being  compared.

```
class MinMaxImpl<T extends Comparable<T>> implements MinMax<T> {

    T[] vals;

    MinMaxImpl(T[] o) {
        vals = o;
    }

    @Override
    public T min() {
        T v = vals[0];
        for (int i = 1; i < vals.length; i++) {
            if (vals[i].compareTo(v) < 0) {
                v = vals[i];
            }
        }
        return v;
    }

    @Override
        public T max() {
            T v = vals[0];
            for (int i = 1; i < vals.length; i++) {
                if (vals[i].compareTo(v) > 0) {
                    v = vals[i];
                }
            }
            return v;
        }

}
```

```
class Student implements Comparable<Student> {
    int rollno;
    String name;
    double cpi;
    Student(int rollno, String name, double cpi) {
        this.rollno = rollno;
        this.name = name;
        this.cpi = cpi;
    }
    @Override
    public String toString() {
        return "Roll no = " + rollno + " Name = " + name + " CPI = " + cpi;
    }
    @Override
    public int compareTo(Student obj) {
        if (this.cpi > obj.cpi) {
            return 1;
        } else if (this.cpi < obj.cpi) {
            return -1;
        } else {
            return this.name.compareTo(obj.name);
        }
    }
}
```

```
public class MinMaxDemo {

    public static void main(String args[]) {
        Integer inums[] = {3, 6, 2, 8, 6};
        Character chs[] = {'b', 'r', 'p', 'w'};
        MinMaxImpl<Integer> iob = new MinMaxImpl<>(inums);
        MinMaxImpl<Character> cob = new MinMaxImpl<>(chs);
        System.out.println("Max value in inums: " + iob.max());
        System.out.println("Min value in inums: " + iob.min());
        System.out.println("Max value in chs: " + cob.max());
        System.out.println("Min value in chs: " + cob.min());
    }
}
```

```
OUTPUT
Max value in inums: 8
Min value in inums: 2
Max value in chs: w
Min value in chs: b
```

```
public class MinMaxDemo {

    public static void main(String args[]) {
        Student s[]={new Student(1,"abc",9.0), new Student(3,"xyz",7.0),
                                   new Student(2,"pqr",9.8)};

        MinMaxImpl<Student> ob = new MinMaxImpl<>(s);
        for (Student ss : s) {
            System.out.println(ss);
        }
        System.out.println("Max value in ob: " + ob.max());
        System.out.println("Min value in ob: " + ob.min());
    }
}
```

```
OUTPUT
Roll no = 1 Name = abc CPI = 9.0
Roll no = 3 Name = xyz CPI = 7.0
Roll no = 2 Name = pqr CPI = 9.8
Max value in ob: Roll no = 2 Name = pqr CPI = 9.8
Min value in ob: Roll no = 3 Name = xyz CPI = 7.0
```

# Raw Type

- To handle the transition to generics, Java allows a generic class to be used without any type arguments.
- This creates a raw type for the class.
- This raw type is compatible with legacy code, which has no knowledge of generics.
- The main drawback to using the raw type is that the type safety of generics is lost.

```
class Gen<T> {

    T ob;

    public Gen(T ob) {
        this.ob = ob;
    }

    public T getOb() {
        return ob;
    }

    public void showType() {
        System.out.println("Type of ob is " + this.ob.getClass().getName());
    }
}
```

```
public class RawTypeDemo {

    public static void main(String args[]) {
        Gen<Integer> iOb = new Gen<Integer>(88);
        Gen<String> strOb = new Gen<String>("Generics Test");
        Gen raw = new Gen(98.6);

        // Cast here is necessary because type is unknown.
        double d = (Double) raw.getOb();
        System.out.println("value: " + d);

        //int i = (Integer) raw.getOb();//run-time error
        strOb = raw; // OK, but potentially wrong
        //String str = strOb.getOb(); // run-time error
        raw = iOb; // OK, but potentially wrong
        //d = (Double) raw.getOb(); // run-time error
    }
}
```

```
OUTPUT
value: 98.6
```

# Generics and Inheritance

- Deriving a generic subclass from a generic superclass
- Deriving a generic subclass from a non-generic superclass

# Deriving a generic subclass from a generic superclass

```
class GenA<T> {

    T ob;

    public GenA(T o) {
        ob = o;
    }

    public T getOb() {
        return ob;
    }
}
```

```
class GenB<T> extends GenA<T> {

    public GenB(T o) {
        super(o);
    }
}
```

```
class GenC<T, V> extends GenA<T> {

    V ob2;

    public GenC(T o, V o2) {
        super(o);
        ob2 = o2;
    }

    public V getOb2() {
        return ob2;
    }
}
```

```
public class GenInheritance1 {

    public static void main(String args[]) {
        GenA<String> a = new GenA<>("Hello");
        System.out.println(a.getOb());

        GenB<Integer> b = new GenB<>(99);
        System.out.println(b.getOb());

        GenC<String, Integer> c = new GenC<>("Value is: ", 99);
        System.out.print(c.getOb());
        System.out.println(c.getOb2());
    }
}
```

```
OUTPUT
Hello
99
Value is: 99
```

# Deriving a generic subclass from a non-generic superclass

```
class NonGen {

    private int num;

    public NonGen(int num) {
        this.num = num;
    }

    public int getNum() {
        return num;
    }
}
```

```
class Gen<T> extends NonGen {

    private T ob;

    public Gen(T ob, int num) {
        super(num);
        this.ob = ob;
    }

    public T getOb() {
        return ob;
    }
}
```

```
public class GenInheritance2 {
    public static void main(String args[]) {
        Gen<String> w = new Gen<>("Hello", 47);
        System.out.println(w.getOb() + " " + w.getNum());
    }
}
```

```
OUTPUT
Hello 47
```

# Use the instanceof operator with a generic class hierarchy.

```
class GenA<T> {

    T ob;

    public GenA(T o) {
        ob = o;
    }

    public T getOb() {
        return ob;
    }
}
```

```
class GenB<T> extends GenA<T> {

    public GenB(T o) {
        super(o);
    }
}
```

```
public class Usinginstanceof {
    public static void main(String args[]) {
        GenA<Integer> iOb = new GenA<>(88);
        GenA<Integer> iOb2 = new GenB<>(99);
        GenB<String> strOb2 = new GenB<>("Generics Test");

        System.out.println("iOb is instance of GenA:- " + (iOb instanceof GenA<?>));
        System.out.println("iOb is instance of GenB:- " + (iOb instanceof GenB<?>));

        System.out.println("iOb2 is instance of GenA:- " + (iOb2 instanceof GenA<?>));
        System.out.println("iOb2 is instance of GenB:- " + (iOb2 instanceof GenB<?>));

        System.out.println("strOb2 is instance of GenA:- " + (strOb2 instanceof GenA<?>));
        System.out.println("strOb2 is instance of GenB:- " + (strOb2 instanceof GenB<?>));
    }
}
```

```
OUTPUT
iOb is instance of GenA:- true
iOb is instance of GenB:- false
iOb2 is instance of GenA:- true
iOb2 is instance of GenB:- true
strOb2 is instance of GenA:- true
strOb2 is instance of GenB:- true
```

# Question

- System.out.println("iOb is instance of GenA:- " + (iOb instanceof GenA<Integer>));
- Gives the error: illegal generic type for instanceof
- Why?

# Answer

- System.out.println("iOb is instance of GenA:- " + (iOb instanceof GenA<Integer>));
- Gives the error: illegal generic type for instanceof
- **The generic type info does not exist at run-time.**

# Type Casting in Generics

```
class Person {
}

class Student extends Person {
}

class Employee extends Person{
}
```

```
class GenP<T> {
    T ob;
    public GenP(T o) {
        ob = o;
    }
}
```

```
public class TypeCastingGenericDemo {

    public static void main(String args[]) {

        GenP<Person> p = new GenP<>(new Person());
        GenP<Person> p1 = p;

        GenP<Person> p2 = new GenP<>(new Student());
        GenP<Person> p3 = p2;

        //GenP<Student> p4 = p2;
//incompatible types GenP<Person> cannot be converted to GenP<Student>
        //GenP<Student> p4 =(GenP<Student>)p2;
//incompatible types GenP<Person> cannot be converted to GenP<Student>
        //GenP<Employee> p5 = p2;
//incompatible types GenP<Person> cannot be converted to GenP<Employee>

        //GenP<Student> s = new GenP<>(new Person()); //incompatible types
    }
}
```

# Overriding and Generics

```
class A<T> {

    T ob;

    public A(T ob) {
        this.ob = ob;
    }

    @Override
    public String toString() {
        return ob.toString();
    }
}
```

```
class B<V, T> extends A<T> {

    V ob1;

    public B(V ob1, T ob) {
        super(ob);
        this.ob1 = ob1;
    }

    @Override
    public String toString() {
        return super.toString() + " " + ob1.toString();
    }
}
```

```
public class OverriddingInGenerics {

    public static void main(String args[]) {
        A<String> a1 = new A<>("hello");
        System.out.println(a1);
        B<String, String> b1 = new B<>("aaa", "bbb");
        System.out.println(b1);
        B<String, Integer> b2 = new B<>("aaa", 50);
        System.out.println(b2);

    }
}
```

```
OUTPUT
hello
bbb aaa
50 aaa
```

# Overloading in Generics

```
class X<T, V> {
    T ob1;
    V ob2;

    void set(T ob1) {
        this.ob1 = ob1;
    }

    //void set(V ob2) {
    //    this.ob2 = ob2;
    //}

}
```

```
 /*
    error: name clash: set(V) and set(T) have the
same erasure
    void set(V ob2) {
  where V,T are type-variables:
    V extends Object declared in class X
    T extends Object declared in class X
1 error
    */
```

# Overloading in Generics

```
class X<T, V> {
    T ob1;
    V ob2;
    int n;

    void set(int n) {
        this.n = n;
    }

    void set(T ob1) {
        this.ob1 = ob1;
    }

}
```

# Generic Restrictions

1. Type parameters cannot be instantiated.
2. No static member can use a type parameter declared by the enclosing class.
3. Generic Array Restrictions
   a) You cannot instantiate an array whose element type is a type parameter.
   b) You cannot create an array of type-specific generic references.
4. A generic class cannot extend Throwable. This means that you cannot create generic exception classes.