Advance Java Programming

Topic	**Why ?**
Generics	• Detecting the bugs at compile-time rather than at run-time • Make the code generic that supports any data type
Lambda Expressions	• An Alternate to anonymous classes • Treat functionality as method argument or code as data

Collections framework (Utility Classes)	• It is a Unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.
	• Advantages
	  • Reduces programming effort
	  • Increases performance
	  • Provide interoperability between unrelated APIs
	  • Reduces the effort required to learn APIs
	  • Reduces the effort required to design and implement APIs
	  • Foster software reuse

	Interfaces : List, Set, SortedSet, NavigableSet, Queue, Deque Classes: ArrayList,LinkedList,HashSet,LinkedHashSet,TreeSet,PriorityQueue,ArrayDeque,EnumSet

Utility Classes	Date : The class **Date** represents a specific instant in time, with millisecond precision. Calendar: The **Calendar** class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields. Gregorian Calendar : **GregorianCalendar** is a concrete subclass of **Calendar** and provides the standard calendar system used by most of the world. TimeZone : **TimeZone** represents a time zone offset, and also figures out daylight savings. SimpleTimeZone : **SimpleTimeZone** is a concrete subclass of **TimeZone** that represents a time zone for use with a Gregorian calendar. Locale : A **Locale** object represents a specific geographical, political, or cultural region., For example, displaying a number is a locale-sensitive operation— the number should be formatted according to the customs and conventions of the user's native country, region, or culture. Random : An instance of this class is used to generate a stream of pseudorandom numbers.
Regular Expression	*Regular expressions* are a way to describe a set of strings based on common characteristics shared by each string in the set

Unit - 1: Generics

- Why Generics?
- What are generics?
- A simple Generic Example
- The general form of a generic class
- A generic class with two type parameters.
- Bounded Types, Using wild card arguments,
- Creating generic method, Generic Interfaces,
- Some Generic Restrictions

Why Generics?

Non-generic code for creating an Dynamic sized array

```
public class MyArrayList {
   private int size;
   private Object[] elements;
   public MyArrayList() {
        elements = new Object[10];
        size = 0;
   }
   public void add(Object o) {
     if (size >= elements.length) {
       Object[] newElements = new Object[size + 10];
       for (int i = 0; i < size; ++i)
         newElements[i] = elements[i];
       elements = newElements;
     }
     elements[size] = o;
     ++size;
   }
   public Object get(int index) {
     if (index >= size)
       throw new IndexOutOfBoundsException("Index: " + index +
                                           ", Size: " + size);
     return elements[index];
   }
   public int size() { return size; }
}
```

```
public class MyArrayListDemo {
    public static void main(String[] args) {
      MyArrayList strLst = new MyArrayList();
      strLst.add("alpha");
      strLst.add("beta");

      for (int i=0;i< strLst.size();++i) {
           String str =(String)strLst.get(i);
           System.out.println(str);
      }
      strLst.add(1234);
      String str = (String)strLst.get(2);
      System.out.println(str);
    }
}
```

upcast from String to Object by compiler .

Need to downcast from Object to String explicitly.

Accidently added the Integer, results no compile-time error but, run-time error is generated

This `MyArrayList` is not *type-safe*. It suffers from the following drawbacks:

1. The upcasting to `java.lang.Object` is done implicitly by the compiler. But, the programmer has to explicitly downcast the `Object` retrieved back to their original class (e.g., `String`).
2. The compiler is not able to check whether the downcasting is valid at *compile-time*. Incorrect downcasting will show up only at *runtime*, as a `ClassCastException`.

Generics will resolve the above problems

Generic code for creating an Dynamic sized array

```
public class MyArrayList<T>{
    private int size;
    private T[] elements;
    public MyArrayList() {
        elements = (T[])new Object[10];
        size = 0;
    }
    public void add(T o) {
      if (size >= elements.length) {
        T[] newElements = (T[])new Object[size + 10];
        for (int i = 0; i < size; ++i)
          newElements[i] = elements[i];
        elements = newElements;
      }
      elements[size] = o;
      ++size;
    }
    public T get(int index) {
      if (index >= size)
        throw new IndexOutOfBoundsException("Index: " + index +
                                            ", Size: " + size);
      return elements[index];
    }
    public int size() { return size; }
}
```

Need to create Object array and cast it explicitly to T because, Generics does not create generic arrays.

```
public class MyArrayListDemo {
    public static void main(String[] args) {
     MyArrayList<String> strLst = new MyArrayList<String>();
        strLst.add("alpha");
        strLst.add("beta");

      for (int i=0;i< strLst.size();++i) {
            String str = strLst.get(i);
            System.out.println(str);
        }
        strLst.add(1234);
        String str = (String)strLst.get(2);
        System.out.println(str);
    }
}
```

upcast from String to Object by compiler .

Explicit downcast is not required.

Error will be caught at compile-time rather than at run-time.

JDK 5 introduces the so-called *generics* to resolve this problem. *Generics* allow us to *abstract over types*. The class designer can design a class with a *generic type*. The users can create specialized instance of the class by providing the *specific type* during instantiation. Generics allow us to *pass type information*, in the form of <type>, to the compiler, so that the compiler can perform all the necessary type-check during compilation to ensure type-safety at runtime.

Generics

- Generics (known as parameterized types) enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods which are similar to formal parameters in method declarations.
- The ***type parameters*** provide a way for you to re-use the same code with different inputs.

- The difference between ***formal parameters*** and ***type parameters*** is that the inputs to *formal parameters are values*, while the *inputs to type parameters are types*.

Code that uses generics has many benefits over non-generic code:

- **Stronger type checking at compile time**
  - A 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 run-time errors
- **Elimination of casts(type casting)**

Without generics	**With generics**
List list=new ArrayList(); list.add(“hello”); String s=(String)list.get(0);	List<String>= new ArrayList<String>(); List.add(“hello”); String s=list.get(0);

- **Enabling programmers to implement generic algorithms**.
  - Algorithms that work on collections of different types, can be customized and are type safe and easier to read

Generics (contd…)

- A *generic class* is defined with the following format:
  class name<T_1, T_2, ..., T_n> { /* ... */ }
- The type parameter section, delimited by angle brackets (<>), follows the class name.It specifies the *type parameters* (also called *type variables*) T_1, T_2, ..., and T_n.
- 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.

Formal Type Parameter Naming Convention

Use an uppercase single-character for formal type parameter. For example,

- <E> for an element of a collection;
- <T> for type;
- <K, V> for key and value.
- <N> for number
- S,U,V, etc. for 2nd, 3rd, 4th type parameters

Non-Generic Code(without generics)	Generic Code (with Generics)

public class GenericBox {
  private **Object** content;
  public GenericBox(**Object** content) { **// constructor**
  this.content = content;
  }
  public **Object** getContent() { **// getter**
  return content;
  }
  public void setContent(**Object** content) { **// setter**
  this.content = content;
  }
}
	public class GenericBox<**T**> {
	  private **T** content;
	  public GenericBox(**T** content) { **// constructor**
	  this.content = content;
	  }
	  public **T** getContent() { **// getter**
	  return content;
	  }
	  public void setContent(**T** content) { **// setter**
	  this.content = content;
	  }
	}

• Need to downcast explicitly by the programmer Box b1=new Box("non-generic"); String s=**(String)**b1.getContent(); Box b2= new Box(101); Integer i=**(Integer)**b2.getContent();	• No explicit downcast is required Box<String> b1=new Box<String>("generic"); String s= b1.getContent(); Box<Integer> b2=new Box<Integer>(101); String s= b2.getContent();
• b1.setContent(101) **// Accidently set Integer** **// retrieving and storing in String Object** String s=(String)b1.getContent() No compile-time error, but run-time error is occurred.	• b1.setContent(101) ***// Accidently set Integer*** ***// retrieving and storing in String Object*** String s=(String)b1.getContent() compiler catches the error at compile-time.

```
// A Generic Box with a content
public class GenericBox<E> {
        private E content; // private variable of generic type E
        public GenericBox(E content) { // constructor
                this.content = content;
        }
        public E getContent() { // getter
                return content;
        }
        public void setContent(E content) { // setter
                this.content = content;
        }
        public String toString() { // describe itself
          return "GenericBox[content=" + content + "(" + content.getClass() + ")]";
        }
}
```

```java
// A Generic Box with a content
public class GenericBox<T> {
        private T content; // private variable of generic type T
        public GenericBox(T content) { // constructor
                this.content = content;
        }
        public T getContent() { // getter
                return content;
        }
        public void setContent(T content) { // setter
                this.content = content;
        }
        public String toString() { // describe itself
        return "GenericBox[content=" + content + "(" + content.getClass() + ")]";  }
}
public class GenericBoxTest {
    public static void main(String[] args) {
        GenericBox<String> box1 = new GenericBox<String>("hello");
        String str = box1.getContent(); // no explicit downcasting needed
        System.out.println(box1);
        //GenericBox[content=hello(class java.lang.String)]
        GenericBox<Integer> box2 = new GenericBox<Integer>(123); // int auto-box to Integer
        int i = box2.getContent(); // Integer auto-unbox to int
        System.out.println(box2);
        //GenericBox[content=123(class java.lang.Integer)]
        GenericBox<Double> box3 = new GenericBox<>(55.66); // double auto-box to Double
        double d = box3.getContent(); // Double auto-unbox to double
        System.out.println(box3);
        //GenericBox[content=55.66(class java.lang.Double)]
    }
}
```

Type Parameter

Type Argument

It supports from JDK7 onwards

Generics

Generics, which supports *abstraction over types* (or *parameterized types*) on classes and methods.

The class or method designers can be *generic about types in the definition*, while the users are to provide the *specific types (actual type) during the object instantiation or method invocation*.

The primary usage of generics is to *abstract over types* for the Collection Framework.

Generic class – two type parameters

```
public class OrderedPair<K,V> {
        private K key;
        private V value;
        public OrderedPair(K key,V value) {
                this.key=key;
                this.value=value;
        }
        public K getKey() {
                return key;
        }
        public V getValue() {
                return value;
        }
}
public class OrderedPairDemo {
    public static void main(String... args ) {
        OrderedPair<String,Integer> p1=new OrderedPair<String,Integer>("Even",8);
        OrderedPair<String,String> p2=new OrderedPair<String,String>("Hello","world");
        OrderedPair<String,Box<Integer>> p3=new OrderedPair<>("primes",new Box<Integer>(10));
        System.out.println("key="+p1.getKey()+
                           " type="+p1.getKey().getClass().getName());
        System.out.println("value="+p1.getValue()+
                           " type="+p1.getValue().getClass().getName());

    }
}
```

Output

```
key=Even type=java.lang.String
Value=8 type=java.lang.Integer
```

```
public class OrderedPairDemo {
    public static void main(String... args ) {
       Pair<String,Integer> p1=new OrderedPair<String,Integer>("Even",8);
       Pair<String,String> p2=new OrderedPair<String,String>("Hello","world");
       Pair<String,String> p3=new OrderedPair<String,String>(“java",“program");
       Pair<String,Box<Integer>> p4=new OrderedPair<>("primes",new Box<Integer>(10));
       System.out.println(“pair p1 key:”+p1.getKey()+ ”values =“+p1.getValue());
    }
}
```

```
public class Stats<T> {
    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;
    }
}
public class BoundsDemo {
    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("double average is"+w);
    }
}
```

An array of type T

Constructor to initialize an array

To calculate average of elements in an array

Compiler reports an error doubleValue() undefined for T

Stats object iob of type Integer

Calculate average of inums[]

Stats object dob of type Integer

Calculate average of dnums[]

Bounded Types

It is used to limit the types that can be passed to a type parameter.

```
public class Stats<T> {
    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;
    }
}
```

- The doubleValue() method is available in Number class so , it is available to all the subclasses of it.
- There is no way the compiler knows Stats Objects are used for numeric types only. So, the compiler report an error it is undefined to T

So, to overcome the above problem java provides bounded types.

Upperbound : to accept superclass and its subclasses

<T ***extends*** *superclass*>

T can be superclass or subclasses of the superclass.

Example : <T extends Number>

T can Number,Integer,Float,Double etc.,

T cannot be String,because String is not a subclass of Number

Lowerbound : to accept superclass and its subclasses

<T ***super*** subclass>

T can be subclass or super classes of the subclass

Example : <T super Integer>

T can Integer, Number etc.,

```
public 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;
    }
}
public class BoundsDemo {
    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("double average is"+w);
    }
}
```

Because T is bounded by Number, the java compiler knowns that all objects of Type T can call **doubleValue()** method, which is available in Number class

Wildcard Arguments

- For suppose to check whether the average of two arrays (irrespective of data type) is same or not.

Arrays

```
Integer inums[ ]= {1,2,3,4,5};
Double dnums[ ] ={1.1,2.2,3.3,4.4,5.5};
```

Stats Object creation

```
Stats<Integer> iob=new Stats<Integer>(inums);
Stats<Double> dob=new Stats<Double>(dnums);
```

checking whether averages are same

```
If(iob.sameAvg(dob))
    System.out.println("averages are same");
else
    System.out.println("avarages differ.");
```

Definition of sameAvg( ) method in Stats<T> class

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

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

The above method is applicable for same types i.e., ***invoking object*** and ***ob*** must be of same type.

i.e., In statement ***iob.sameAvg(dob)*** ,the iob(of integer type) and dob(of double type) are different

So, to resolve the above problem use wildcard argument ' **?** ' (represents an unknown type)

```java
public 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(average()==ob.average())
                return true;
        return false;
    }
}
```

```java
public class BoundsDemo {
  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("double average is"+w);
    Float fnums[]= {1.0F,2.0F,3.0F,4.0F,5.0F};
    Stats<Float> fob=new Stats<Float>(fnums);
    double f=fob.average();
    System.out.println("float average is"+f);
    if(iob.sameAvg(fob))
        System.out.println("are the same..");
    else
        System.out.println("differ");
  }
}
```

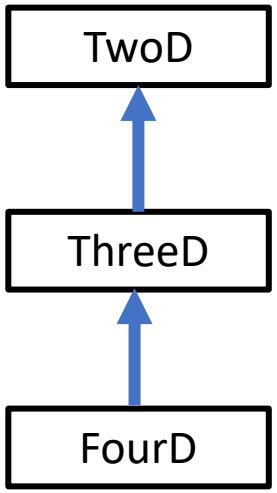
Wildcard Arguments (contd..)

A Wildcard describes a family of types

- <?> – the unknown(unbounded) type which represents family of all types
- <? extends ***Type>*** – a wild card with an upper bound . It supports a family of subtypes of ***Type***
- <? super ***Type>*** – a wild card with an lower bound . It supports a family of supertypes of ***Type***

- Wild cards are useful in situations where ***no or only partial knowledge about the types arguments*** of a parameterized type is required.
- Wild cards are also used to create the ***relationship between generic classes and interfaces***.

Example




```
Class Coords<T extends TwoD>{ T[] coords;
      Coords(T[] o){ cords = 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);
      }
      static void showXYZ(Coords<? extends ThreeD> c) { /*…*/ }
      static void showAll(Coords<? extends FourD> c)  { /*…*/ }
      public static void main(String[] args) {
            TwoD td[]= { new TwoD(0,0),new TwoD(2,3),new TwoD(3,4)};
            Coord<TwoD> tdlocs= new Coord<>(td);
            showXY(tdlocs);
            showXYZ(tdlocs); //error
            showAll(tdlocs); //error
            FourD fd[]={ new FourD(1,2,-3,4),new FourD(-3,4,22,10)};
            Coord<FourD> fdlocs= new Coord<>(fd);
            showXY(fdlocs);
            showXYZ(fdlocs);
            showAll(fdlocs);
      }
}
```

It accepts all types

It accepts ThreeD & FourD

It accepts FourD

it's upper bound is ThreeD, so it won't allow

it's upper bound is FourD, so it won't allow

Bounded types example

Example	Valid/Invalid
Class X0 <T extends int> {….}	Not accepted. Because, primitive types are not allowed
Class X1 <T extends Object[]> {….}	Not accepted. Because, Array types are not permitted
Class X2 <T extends Number> { …} Class X3 <T extends String> {….}	Classes
Class X4 <T extends Runnable> {….}	Interfaces
Class X5 <T extends Thread.State> {….}	Enum
Class X6 <T extends List>{ …}	Raw type
Class X7 <T extends List<String> {….}	Generic type
Class X8 <T extends <? extends Number>> {…..} Class X9 <T extends<? super Number>> {….}	Generic wildcard bounded type

Given the following two regular (non-generic) classes: **class A { /* ... */ }** **class B extends A { /* ... */ }**	It would be reasonable to write the following code: B b = new B(); A a = b;
List<B> lb = new ArrayList<>(); List<A> la = lb; // compile-time error	

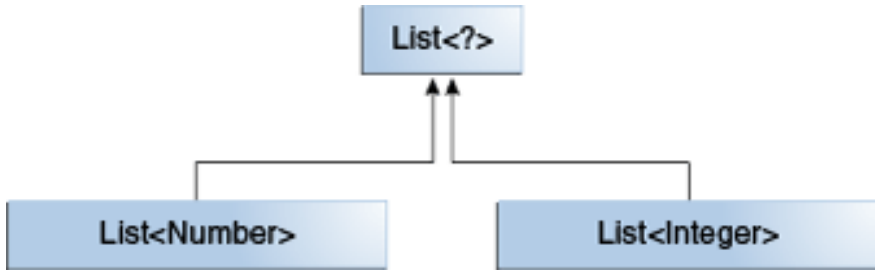
Integer is a subtype of Number , but List<Integer> is not a subclass List<Number>

To create relationship among these classes we can use Wildcard argument

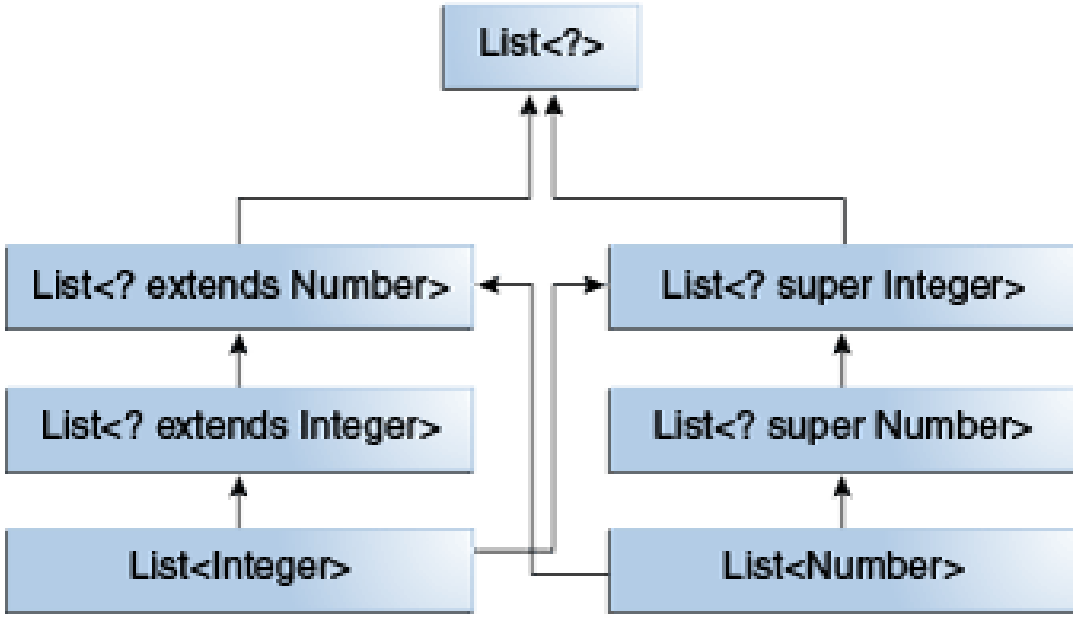
List<? extends Integer> intList = new ArrayList<>();

List<? extends Number> numList = intList; // OK.

Because List<? extends Integer> is a subtype of List<? extends Number>



Relationships between several List classes declared with both upper and lower bounded wildcards.



Generic Methods

- Similar to generic types(classes), we can define generic methods and generic interfaces.
- The type parameter section is specified in **angular brackets** and appears before the **method's return type.**
- Syntax

  access-modifiers ***<type parameter>*** return-type method-name<***formal-argument-list***>

Example

printList(T[] elements) is a generic method which prints any type of array

```
class Util{
        public static <T> void printList(T[] elements)
        {
                for(T element:elements)
                   System.out.print(element+" ");
                System.out.println();
        }
}
```

The type parameter is passed from type section to formal parameter

```
class Util{
        public static <T> void printList(T[] elements)
        {
                for(T element:elements)
                   System.out.print(element+" ");
                System.out.println();
        }
}
public class GenericMethod {
  public static void main(String... args) {
      Integer[] inums= {1,2,3,4,5};
      Float[] fnums= {1.1f,2.2f,3.3f,4.4f,5.5f};
      Double[] dnums= {1.5,2.5,3.5,4.5,5.5};
      System.out.println("Integer list");
      Util.<Integer>printList(inums);
      System.out.println("Float list");
      Util.printList(fnums);
      System.out.println("Double list");
      Util.printList(dnums);
  }
}
```

printList() is a generic method which prints any type of array

```
class Util1{
   public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
         int count = 0;
         for (T e : anArray)
            if (e.compareTo(elem) > 0)
               ++count;
         return count;
   }
}
public class GreaterThanDemo {
      public static void main(String[] args) {
            Integer inum[]= {15,2,10,14,32,7,3,21};
            Float fnum[]= {2.1f,4.3f,5.2f,5.7f,4.2f,6.7f};
            Double dnum[]= {2.3,4.5,6.1,4.2,5.7,8.9,9.0};
            System.out.println("count > 10 in "+Arrays.toString(inum)+"is"+
                                    Util1.countGreaterThan(inum,10));
            System.out.println("count > 5.1f in "+Arrays.toString(fnum)+"is"+
                                    Util1.countGreaterThan(fnum,5.1f));
            System.out.println("count > 4.2 in "+Arrays.toString(dnum)+"is"+
                                    Util1.countGreaterThan(dnum,4.2));
      }
}
```

- T is bounded by the class implemented by Comparable Interface(which is defined in java.lang package).
- T can be any class that implements comparable interface like Integer, Float , String, etc.

Generic Interfaces

```
interface MinMax<T extends Comparable<T>> {
        T min();
        T max();
}
public class MyClass<T extends Comparable<T>> implements MinMax<T>{
        T[] vals;
        MyClass(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;
        }
}
```

```
public class GenIfDemo {
	public static void main(String[] args) {
		Integer inums[]= {2,6,3,8,1};
		Character chs[]= {'b','x','p','w'};
		MyClass<Integer> iob=new MyClass<Integer>(inums);
		MyClass<Character> cob=new MyClass<>(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());
	}
}
```

Restrictions on Generics

- **Cannot Instantiate Generic Types with Primitive Types**

  creating a Pair object, you cannot substitute a primitive type for the type parameter K or V:

  Pair<**int, char**> p = new Pair<>(8, 'a'); // compile-time error

  You can substitute only non-primitive types for the type parameters K and V:

  Pair<**Integer, Character**> p = new Pair<>(8, 'a');

- **Cannot Create Instances of Type Parameters**

  An instance of a type parameter cannot be created. For example, the following code causes a compile-time error:

```
public static <E> void append(List<E> list) {
        E elem = new E(); // compile-time error
        list.add(elem);
 }
```

- **Cannot Declare Static Fields Whose Types are Type Parameters**

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
        private static T os;
        // ...
}
```

- **Cannot Use Casts or instanceof with Parameterized Types**

The Java compiler erases all type parameters in generic code, So the verification of parameterized type for a generic type at runtime is not possible:

```
public static <E> void rtti(List<E> list) {
        if (list instanceof ArrayList<Integer>) {
                // compile-time error
                // ...
        }
}
```

if (list instanceof ArrayList<?>) ***// it is ok***

- **Cannot Create Arrays of Parameterized Types**

  An arrays of parameterized types cannot created. For example, the following code does not compile:

  List<Integer>[] arrayOfLists = new List<Integer>[2]; // compile-time error

- **Cannot Create, Catch, or Throw Objects of Parameterized Types**

  A generic class cannot extend the Throwable class directly or indirectly. For example, the following classes will not compile:

```
// Extends Throwable indirectly
class MathException<T> extends Exception { /* ... */ } // compile-time error
// Extends Throwable directly
class QueueFullException<T> extends Throwable { /* ... */ } // compile-time error
```

  A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
        try {
                for (J job : jobs)
                    // ...
        } catch (T e) { // compile-time error
                // ...
          }
}
```

Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
        public void print(Set<String> strSet) { }
        public void print(Set<Integer> intSet) { }
}
```

Questions

1.Will the following class compile? If not, why?

```
public final class Algorithm {
    public static <T> T max(T x, T y) {
        return x > y ? x : y;
    }
}
```

2. Consider this class:

```
class Node<T> implements Comparable<T> {

    public int compareTo(T obj) { /* ... */ }

    // ...
}
```

Will the following code compile? If not, why?

```
Node<String> node = new Node<>();
Comparable<String> comp = node;
```

3.What is the following class converted to after type erasure?

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

```
public class Pair {
        public Pair(Object key, Object value) {
                this.key = key;
                this.value = value;
        }
        public Object getKey(); {
                return key;
        }
        public Object getValue(); {
                return value;
        }
        public void setKey(Object key) {
                this.key = key;
        }
        public void setValue(Object value) {
                this.value = value;
        }
        private Object key;
        private Object value;
}
```

4. What is the following method converted to after type erasure?

```
public static <T extends Comparable<T>> int findFirstGreaterThan(T[] at, T elem) {
                                        // ...
}
```

```
public static int findFirstGreaterThan(Comparable[] at, Comparable elem) {
                                        // ...
}
```

5. Will the following class compile? If not, why?

```
public class Singleton<T> {
        public static T getInstance()
        {
                if (instance == null)
                        instance = new Singleton<T>();
                        return instance;
        }
        private static T instance = null;
}
```