Program for all types without Generic. Using Object.

```
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());
}       }

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: java.lang.String cannot be
cast to java.lang.Integer
        at NonGenDemo.main(NonGenDemo.java:44)
```

- 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.

```
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());
        }
}
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;
                //GenDemo.java:31: error: incompatible types:
                //Gen<String> cannot be converted to Gen<Integer> iOb = strOb;
        }
}
```

OUTPUT:

```
Type of ob is java.lang.Integer
value: 88
Type of ob is java.lang.String
value: Generics Test
```

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;
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
    void showTypes() {
        System.out.println("Type of T is " +ob1.getClass().getName());
        System.out.println("Type of V is " +ob2.getClass().getName());
    }
    T getob1() {
        return ob1;
    }
```

```
	V getob2() {
		return ob2;
	}
}
class SimpGen {
	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;
}	}
```

```
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("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: for class Stats<T> {
iob average is 3.0
dob average is 3.3

BoundsDemo.java:30: error: type argument String is not within bounds of type-variable T
		Stats<String> strob = new Stats<String>(strs);
	where T is a type-variable:
	T extends Number declared in class Stats
```

Program2:

```
class Person{
	String name;
	public String toString(){
		return "Name  = "+name;
}	}
class Student extends Person{
	int rollno;
	public String toString(){
		return super.toString() + " Rollno = "+ rollno;
}	}
```

```
class Employee extends Person{
        int id;
        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);
}       }
class BoundsPerson {
        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
```

Using Wildcard arguments

- 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.
- For example, if one object contains the double values 1.0, 2.0, and 3.0, and the other object contains the integer values 2, 1, and 3, then the averages will be the same.

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

At first, you might think of a solution like this, in which T is used as the type parameter:

```
// This won't work!
// Determine if two averages are the same.
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.

- To create a generic sameAvg( ) method, you must use another feature of Java generics: 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;
        }
        // Determine if two averages are the same.
        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.

```
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;
}       }
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); //compile time error: cannot find symbol
                }
                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:	X Y Z Coordinates:	X Y Z T Coordinates:
1 2	1 2 3	1 2 3 4
6 8	6 8 14	6 8 14 8
22 9	22 9 4	22 9 4 9
3 -2	3 -2 -23	3 -2 -23 17

Create a generic method that is enclosed within a non-generic class

```
class GenMethDemo {
        static <T> boolean isIn(T x, T[] y) {
                for(int i=0; i < y.length; i++)
                        if(x.equals(y[i]))
                                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");
                System.out.println();

                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");
        }
}
```

It is possible for constructors to be generic, even if their class is not.

```
class GenCons {
        private double val;
        <T extends Number> GenCons(T arg) {
                val = arg.doubleValue();
        }
        void showval() {
                System.out.println("val: " + val);
        }
}
```

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

Generic Interfaces

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.

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

class MinMaxImpl<T extends Comparable<T>> implements MinMax<T> {
        T[] vals;
        MinMaxImpl(T[] o) {
                vals = o;
        }
        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;
        }
        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 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<Integer>(inums);
		MinMaxImpl<Character> cob = new MinMaxImpl<Character>(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());
		Student s [] = {	new Student(1, "abc", 9.0),
		new Student(3,"xyz",7.0),   new Student(2,"pqr",9.80)};
		MinMaxImpl<Student> ob = new MinMaxImpl<Student>(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());
}	}
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;
	}
	public String toString()	{
		return "Roll no = " + rollno + " Name = " + name + " CPI = "+cpi ;
	}
	public int compareTo(Student obj){
		//return this.name.compareTo(obj.name);  (comparing by name only)
		//comparing by marks and then name;
		if(this.cpi > obj.cpi)
			return 1;
		else if(this.cpi < obj.cpi)
			return -1;
		else
			return this.name.compareTo(obj.name);
}	}
```

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;
        Gen(T o) {
                ob = o;
        }
        T getob() {
                return ob;
        }
}

public class RawType {
        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(new Double(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(This assignment overrides type safety)

                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
        }
}
```

Using a Generic Superclass

```
class Gen<T> {
	T ob;
	Gen(T o) {
		ob = o;
	}
	T getob() {
		return ob;
	}
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
	Gen2(T o) {
		super(o);
	}
}

// A subclass of Gen that defines a second type parameter, called V.
class Gen2<T, V> extends Gen<T> {
	V ob2;
	Gen2(T o, V o2) {
		super(o);
		ob2 = o2;
	}
	V getob2() {
		return ob2;
	}
}

// Create an object of type Gen2.
class HierDemo {
	public static void main(String args[]) {
		Gen2<String, Integer> x =new Gen2<String, Integer>("Value is: ", 99);
		System.out.print(x.getob());
		System.out.println(x.getob2());
	}
}
```

A Generic Subclass

It is perfectly acceptable for a non-generic class to be the superclass of a generic subclass.

```
// A non-generic class can be the superclass of a generic subclass.
class NonGen {
        int num;
        NonGen(int i) {
                num = i;
        }
        int getnum() {
                return num;
        }
}

// A generic subclass.
class Gen<T> extends NonGen {
        T ob;
        Gen(T o, int i) {
                super(i);
                ob = o;
        }
        T getob() {
                return ob;
        }
}

// Create a Gen object.
class HierDemo2 {
        public static void main(String args[]) {
                // Create a Gen object for String.
                Gen<String> w = new Gen<String>("Hello", 47);
                System.out.print(w.getob() + " ");
                System.out.println(w.getnum());
        }
}
```

```
// Use the instanceof operator with a generic class hierarchy.
class Gen<T> {
        T ob;
        Gen(T o) {
                ob = o;
        }
        T getob() {
                return ob;
        }
}
// A subclass of Gen.
class Gen2<T> extends Gen<T> {
        Gen2(T o) {
                super(o);
        }
}

// Demonstrate run-time type ID implications of generic class hierarchy.
class HierDemo3 {
        public static void main(String args[]) {
                Gen<Integer> iOb = new Gen<Integer>(88);
                Gen2<Integer> iOb2 = new Gen2<Integer>(99);
                Gen2<String> strOb2 = new Gen2<String>("Generics Test");
                if(iOb2 instanceof Gen2<?>)
                        System.out.println("iOb2 is instance of Gen2");
                if(iOb2 instanceof Gen<?>)
                        System.out.println("iOb2 is instance of Gen");

                if(strOb2 instanceof Gen2<?>)
                        System.out.println("strOb2 is instance of Gen2");
                if(strOb2 instanceof Gen<?>)
                        System.out.println("strOb2 is instance of Gen");

                if(iOb instanceof Gen2<?>)
                        System.out.println("iOb is instance of Gen2");
                if(iOb instanceof Gen<?>)
                        System.out.println("iOb is instance of Gen");
}       }
```

The following can't be compiled because **generic type info does not exist at run time.**

```
        if(iOb2 instanceof Gen2<Integer>)
                System.out.println("iOb2 is instance of Gen2<Integer>");
```

OUTPUT:

```
iOb2 is instance of Gen2
iOb2 is instance of Gen
strOb2 is instance of Gen2
strOb2 is instance of Gen
iOb is instance of Gen
```

- **Typecasting in generics**

(Gen<Integer>) iOb2 // legal because iOb2 includes an instance of Gen<Integer>.
(Gen<Long>) iOb2 // illegal because iOb2 is not an instance of Gen<Long>.

- **A method in a generic class can be overridden just like any other method.**

- **Ambiguity Errors:**

```
class MyGenClass<T, V> {
        T ob1;
        V ob2;
        // These two overloaded methods are ambiguous and will not compile.
        void set(T o) {        ob1 = o;        }
        void set(V o) {        ob2 = o;        }
}
```

e.g. MyGenClass<String, String> obj = new MyGenClass<String, String>()

Some Generic Restrictions

1. Type parameters can’t 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.