```java
public class MySimpleGenerics {

    public static void main(String a[]){

        //we are going to create SimpleGeneric object with String as type parameter
        SimpleGeneric<String> sgs = new SimpleGeneric<String>("JAVA2NOVICE");
        sgs.printType();
        //we are going to create SimpleGeneric object with Boolean as type parameter
        SimpleGeneric<Boolean> sgb = new SimpleGeneric<Boolean>(Boolean.TRUE);
        sgb.printType();
    }
}

/**
 * Here T is a type parameter, and it will be replaced with
 * actual type when the object got created.
 */
class SimpleGeneric<T>{

    //declaration of object type T
    private T objReff = null;

    //constructor to accept type parameter T
    public SimpleGeneric(T param){
        this.objReff = param;
    }

    public T getObjReff(){
        return this.objReff;
    }

    //this method prints the holding parameter type
    public void printType(){
        System.out.println("Type: "+objReff.getClass().getName());
    }
}
```

Output:

```
Type: java.lang.String
Type: java.lang.Boolean
```

```java
public class MySimpleTwoGenerics {

    public static void main(String a[]){

        SimpleGen<String, Integer> sample
                    = new SimpleGen<String, Integer>("JAVA2NOVICE", 100);
        sample.printTypes();
    }
}

/**
 * Simple generics class with two type parameters U, V.
 */
class SimpleGen<U, V>{

    //type U object reference
    private U objUreff;
    //type V object reference
    private V objVreff;

    //constructor to accept object type U and object type V
    public SimpleGen(U objU, V objV){
        this.objUreff = objU;
        this.objVreff = objV;
    }

    public void printTypes(){
        System.out.println("U Type: "+this.objUreff.getClass().getName());
        System.out.println("V Type: "+this.objVreff.getClass().getName());
    }
}
```

Output:

```
U Type: java.lang.String
V Type: java.lang.Integer
```

PROGRAM: HOW IMPLEMENT BOUNDED TYPES IMPLEMENTS AN INTERFACE) WITH GENERICS?

As of now we have seen examples for only one type parameter. What happens in case we want to access group of objects comes from same family, means implementing same interface? You can restrict the generics type parameter to a certain group of objects which implements same interface. You can achieve this my specifying extends <interface-name> at class definitions, look at the example, it gives you more comments to understand. You can also specify multiple interfaces at the definision. you can do this by specifying mulitple interfaces seperated by "&". You can also specify class which implements an interface and the interface together. For example:

class MyClass<T extends TestClass & TestInterface> {

```
public class MyBoundedInterface {

    public static void main(String a[]){

        //Creating object of implementation class X called Y and
        //passing it to BoundExmp as a type parameter.
        BoundExmp<Y> bey = new BoundExmp<Y>(new Y());
        bey.doRunTest();
        //Creating object of implementation class X called Z and
        //passing it to BoundExmp as a type parameter.
        BoundExmp<Z> bez = new BoundExmp<Z>(new Z());
        bez.doRunTest();
        //If you uncomment below code it will throw compiler error
        //becasue we restricted to only of type X  implementation classes.
        //BoundExmp<String> bes = new BoundExmp<String>(new String());
        //bea.doRunTest();
    }
}
```

```
class BoundExmp<T extends X>{

    private T objRef;

    public BoundExmp(T obj){
        this.objRef = obj;
    }

    public void doRunTest(){
        this.objRef.printClass();
    }
}

interface X{
    public void printClass();
}

class Y implements X{
    public void printClass(){
        System.out.println("I am in class Y");
    }
}

class Z implements X{
    public void printClass(){
        System.out.println("I am in class Z");
    }
}
```

Output:

```
I am in class Y
I am in class Z
```

PROGRAM: HOW IMPLEMENT BOUNDED TYPES (EXTEND SUPERCLASS) WITH GENERICS?

As of now we have seen examples for only one type parameter. What happens in case we want to access group of objects comes from same family, means extending same super class? You can restrict the generics type parameter to a certain group of objects which extends same super class. You can achieve this my specifying extends <super-class> at class definitions, look at the example, it gives you more comments to understand.

```
package com.java2novice.generics;

public class MyBoundedClassEx {

    public static void main(String a[]){
        //Creating object of sub class C and
        //passing it to BoundEx as a type parameter.
        BoundEx<C> bec = new BoundEx<C>(new C());
        bec.doRunTest();
        //Creating object of sub class B and
        //passing it to BoundEx as a type parameter.
        BoundEx<B> beb = new BoundEx<B>(new B());
        beb.doRunTest();
        //similarly passing super class A
        BoundEx<A> bea = new BoundEx<A>(new A());
        bea.doRunTest();
        //If you uncomment below code it will throw compiler error
        //becasue we restricted to only of type A and its sub classes.
        //BoundEx<String> bes = new BoundEx<String>(new String());
        //bea.doRunTest();
    }
}
```

```
/**
 * This class only accepts type parametes as any class
 * which extends class A or class A itself.
 * Passing any other type will cause compiler time error
 */
class BoundEx<T extends A>{

    private T objRef;

    public BoundEx(T obj){
        this.objRef = obj;
    }

    public void doRunTest(){
        this.objRef.printClass();
    }
}

class A{
    public void printClass(){
        System.out.println("I am in super class A");
    }
}

class B extends A{
    public void printClass(){
        System.out.println("I am in sub class B");
    }
}

class C extends A{
    public void printClass(){
        System.out.println("I am in sub class C");
    }
}
```

Output:

```
I am in sub class C
I am in sub class B
I am in super class A
```

PROGRAM: WHAT IS GENERICS WILDCARD ARGUMENTS? GIVE AN EXAMPLE.

Below example exmplains what is wildcard arguments and how it helps us to solve problem. In the example, we have two classes called CompAEmp and CompBEmp extending Emp class. We have a generic class called MyEmployeeUtil, where we have utilities to perform employee functions irrespective of which comapany emp belogns too. This class accepts subclasses of Emp. Incase if we want to compare salaries of two employees, how can we do using MyEmployeeUtil class? U can think that below sample code might work, but it wont work.

```
//this logic wont work
public boolean isSalaryEqual(MyEmployeeUtil<T> otherEmp){

    if(emp.getSalary() == otherEmp.getSalary()){
        return true;
    }
    return false;
}

```

Because once you create an object of MyEmployeeUtil class, the type argument will be specific to an instance type. So you can compare between only same object types, ie, you can comapare either objects of CompAEmp or CompBEmp, but not between CompAEmp and CompBEmp. To solve this problem, wildcard argument will helps you. Look at below sample code, which can solve your problem.

```
public boolean isSalaryEqual(MyEmployeeUtil<?> otherEmp){

    if(emp.getSalary() == otherEmp.getSalary()){
        return true;
    }
    return false;
}

```

```java
public class MyWildcardEx {

    public static void main(String a[]){

        MyEmployeeUtil<CompAEmp> empA
                    = new MyEmployeeUtil<CompAEmp>(new CompAEmp("Ram", 20000));
        MyEmployeeUtil<CompBEmp> empB
                    = new MyEmployeeUtil<CompBEmp>(new CompBEmp("Krish", 30000));
        MyEmployeeUtil<CompAEmp> empC
                    = new MyEmployeeUtil<CompAEmp>(new CompAEmp("Nagesh", 20000));
        System.out.println("Is salary same? "+empA.isSalaryEqual(empB));
        System.out.println("Is salary same? "+empA.isSalaryEqual(empC));
    }
}
```

```java
class MyEmployeeUtil<T extends Emp>{

    private T emp;

    public MyEmployeeUtil(T obj){
        emp = obj;
    }

    public int getSalary(){
        return emp.getSalary();
    }

    public boolean isSalaryEqual(MyEmployeeUtil<?> otherEmp){

        if(emp.getSalary() == otherEmp.getSalary()){
            return true;
        }
        return false;
    }

    //// create some utility methods to do employee functionalities
    //
    //
    //
}
```

```java
class Emp{

    private String name;
    private int salary;

    public Emp(String name, int sal){
        this.name = name;
        this.salary = sal;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary(int salary) {
        this.salary = salary;
    }
}
```

```java
class CompAEmp extends Emp{

    public CompAEmp(String nm, int sal){
        super(nm, sal);
    }
}

class CompBEmp extends Emp{

    public CompBEmp(String nm, int sal){
        super(nm, sal);
    }
}
```