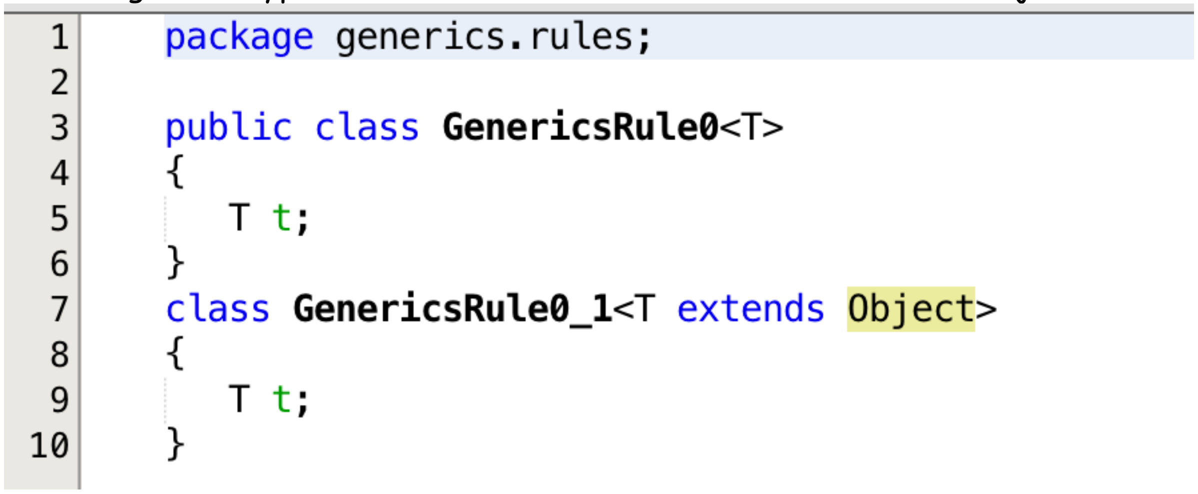
The Rules of Generics

Rule 0

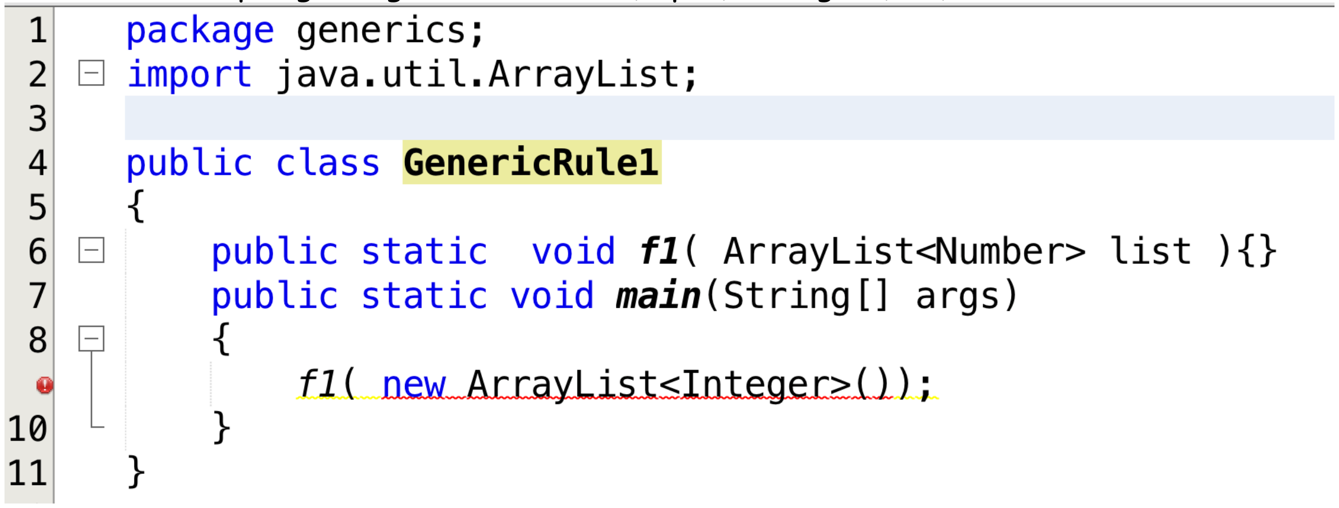
To declare a generic type <T> for a class it is the same as <T extends Object>.



```java
package generics.rules;

public class GenericsRule0<T>
{
    T t;
}
class GenericsRule0_1<T extends Object>
{
    T t;
}
```

Rule1

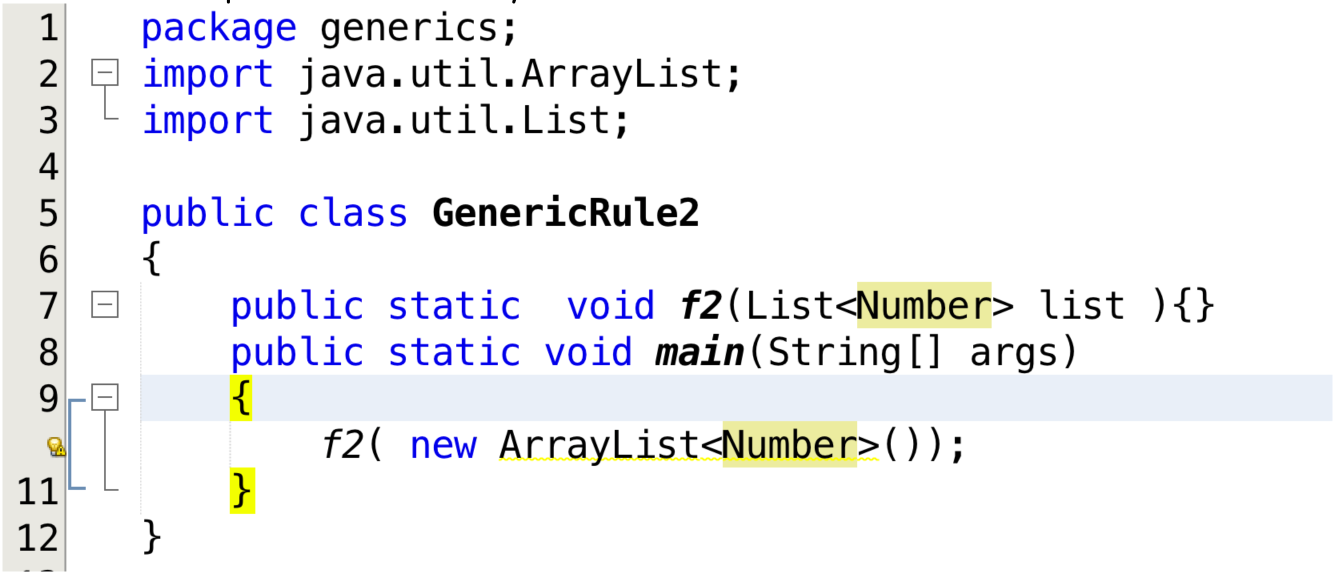
There is **no inheritance** for what is **inside** the diamond even if the class types in the diamond is a super getting a sub. Number(super), Integer (sub).



```java
package generics;
import java.util.ArrayList;

public class GenericRule1
{
    public static  void f1( ArrayList<Number> list ){}
    public static void main(String[] args)
    {
        f1( new ArrayList<Integer>());
    }
}
```

Rule 2

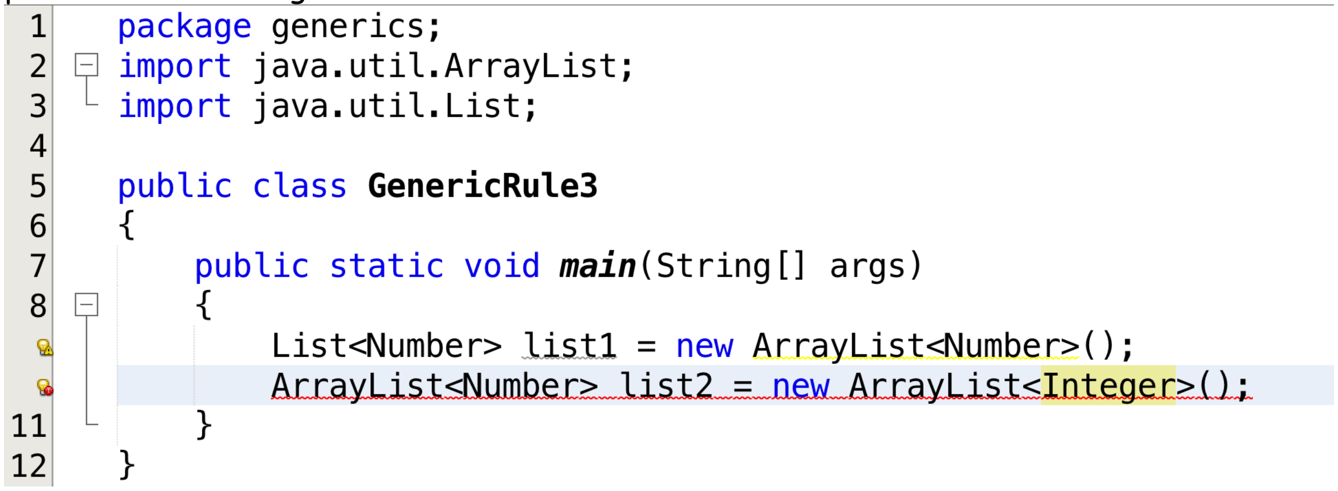
There **IS** inheritance for what is **outside** the diamond when the classes outside the diamond are super sub. List ArrayList



```java
package generics;
import java.util.ArrayList;
import java.util.List;

public class GenericRule2
{
    public static  void f2(List<Number> list ){}
    public static void main(String[] args)
    {
        f2( new ArrayList<Number>());
    }
}
```

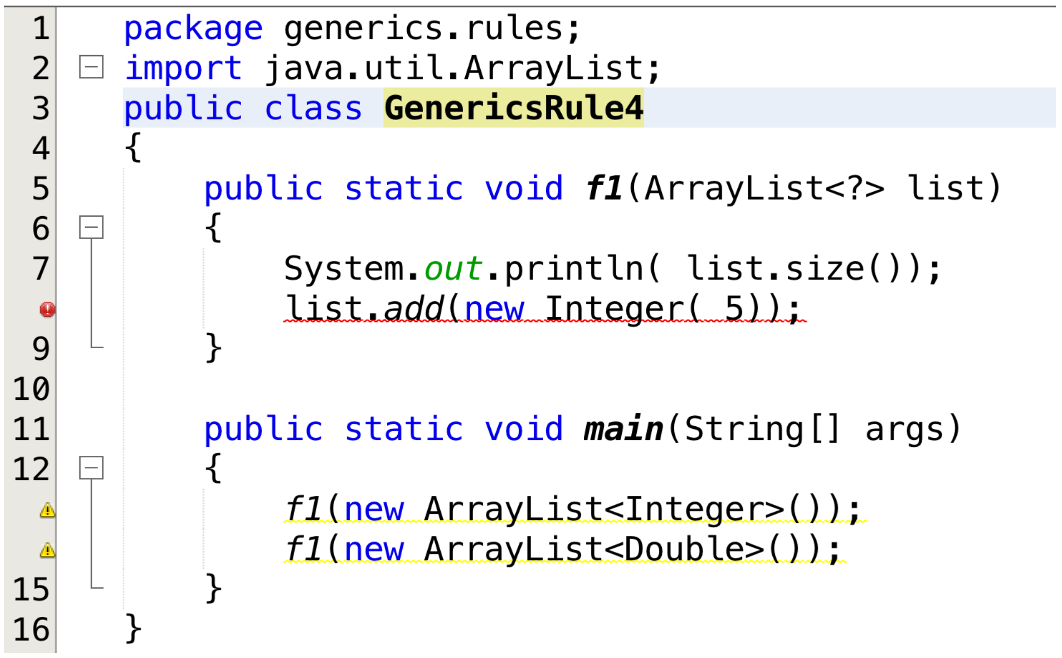
Rule 3

Rule 1 and rule 2 apply for assignment as well, as in method calls passing arguments to parameters IS assignment.



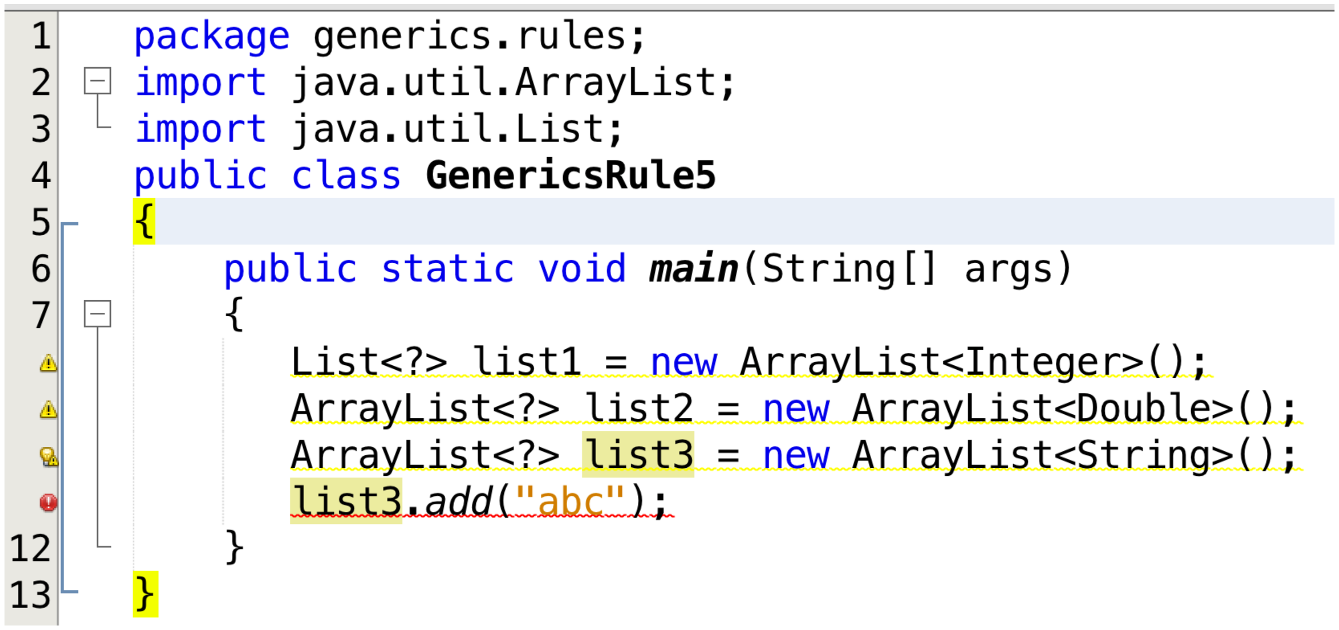
```java
package generics;
import java.util.ArrayList;
import java.util.List;

public class GenericRule3
{
    public static void main(String[] args)
    {
        List<Number> list1 = new ArrayList<Number>();
        ArrayList<Number> list2 = new ArrayList<Integer>();
    }
}
```

Rule 4
The wildcard ?, fixes the inheritance problem inside the diamonds. However, we cannot modify the ArrayList<?> (line 8) because we (the compiler), do not know what the exact type of the wild card is.



```
 1  package generics.rules;
 2  import java.util.ArrayList;
 3  public class GenericsRule4
 4  {
 5      public static void f1(ArrayList<?> list)
 6      {
 7          System.out.println( list.size());
            list.add(new Integer( 5));
 9      }
10
11      public static void main(String[] args)
12      {
            f1(new ArrayList<Integer>());
            f1(new ArrayList<Double>());
15      }
16  }
```

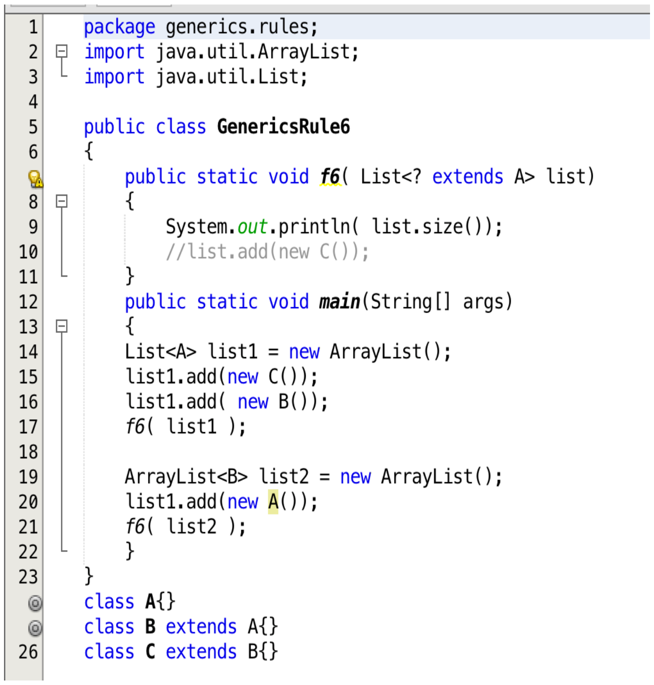
Rule 5
When we have a wildcard on the LHS of assignment then on the in RHS of the assignment, what is **outside** the diamond must be sub (or same class) of LHS and what is **inside** the diamond can be any type. We still cannot modify the variable as in Rule 4.



```
package generics.rules;
import java.util.ArrayList;
import java.util.List;
public class GenericsRule5
{
    public static void main(String[] args)
    {
        List<?> list1 = new ArrayList<Integer>();
        ArrayList<?> list2 = new ArrayList<Double>();
        ArrayList<?> list3 = new ArrayList<String>();
        list3.add("abc");
    }
}
```

Rule 6

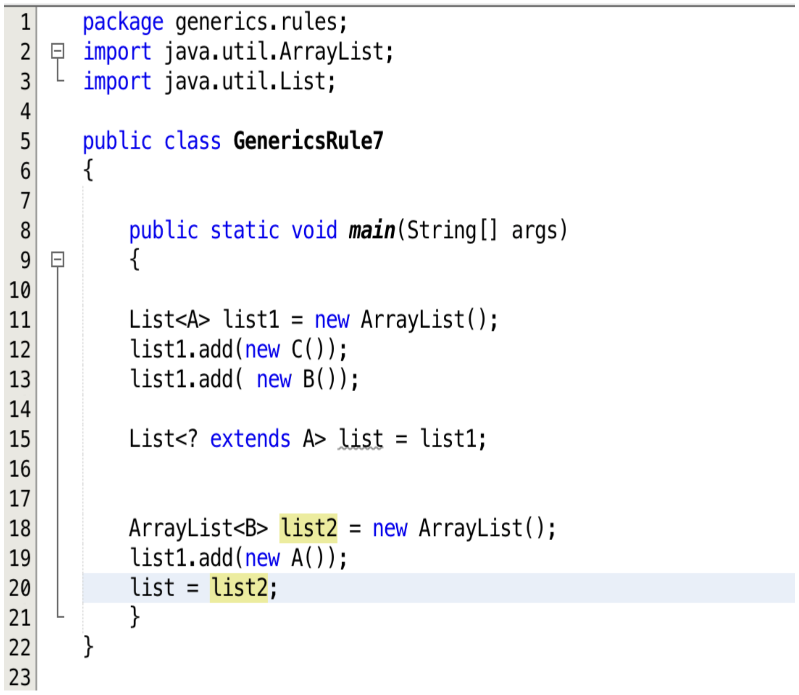
Bounded Inheritance for what is inside the diamond with <? extends *TYPE*>. The *TYPE (A)* is the UPPER BOUND of the ?. The method f6() says that the List can have in the diamond anything that extends type A. A is the super. Upper bound means superclass. Lower bound means subclass.



```java
package generics.rules;
import java.util.ArrayList;
import java.util.List;

public class GenericsRule6
{
    public static void f6( List<? extends A> list)
    {
        System.out.println( list.size());
        //list.add(new C());
    }
    public static void main(String[] args)
    {
    List<A> list1 = new ArrayList();
    list1.add(new C());
    list1.add( new B());
    f6( list1 );

    ArrayList<B> list2 = new ArrayList();
    list1.add(new A());
    f6( list2 );
    }
}
class A{}
class B extends A{}
class C extends B{}
```

Rule 7

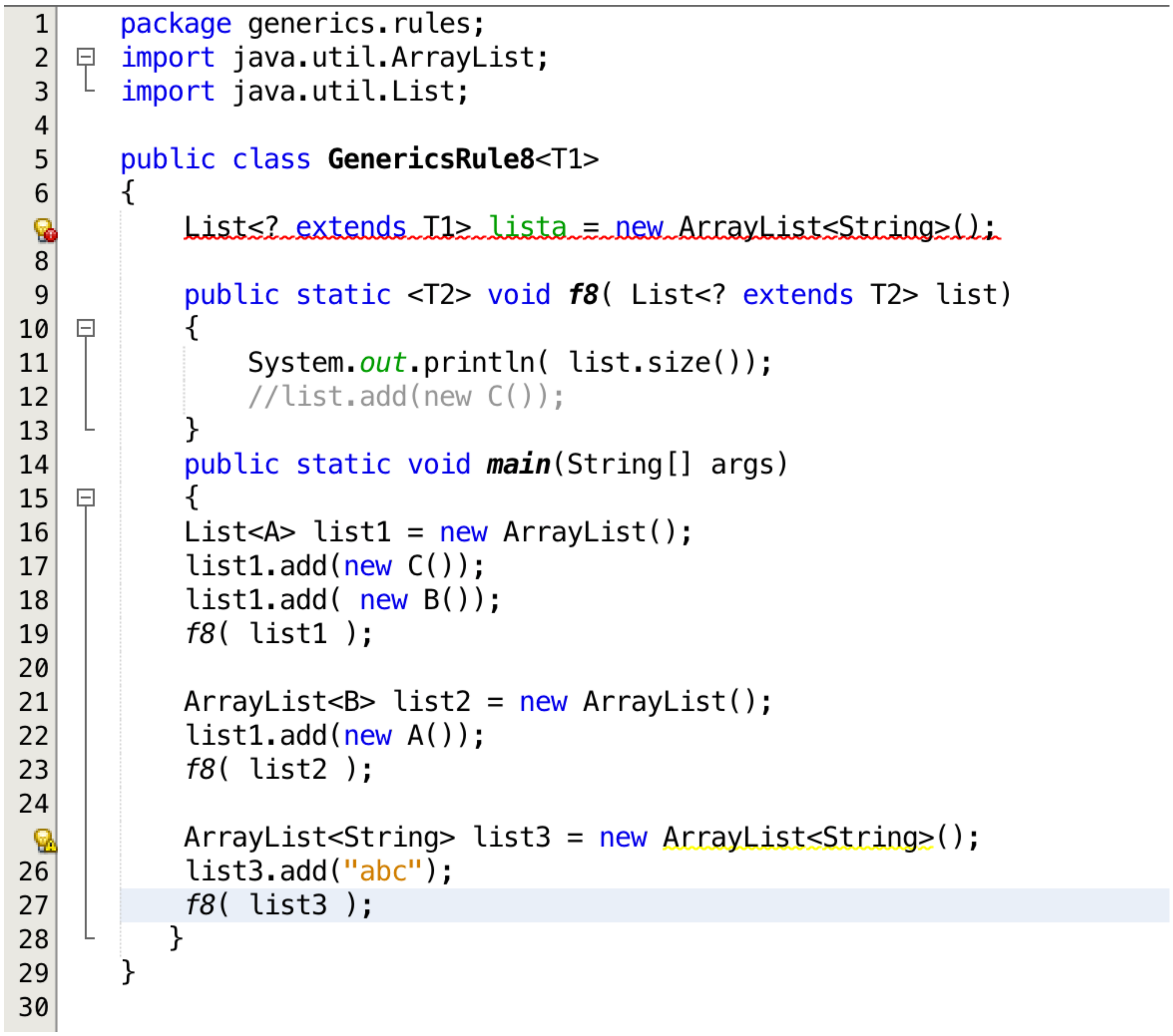
The same as rule 6 with the difference that we do the assignment *List<? extends A> list = list1;* line 15 and line 20.



```java
1  package generics.rules;
2  import java.util.ArrayList;
3  import java.util.List;
4
5  public class GenericsRule7
6  {
7
8      public static void main(String[] args)
9      {
10
11     List<A> list1 = new ArrayList();
12     list1.add(new C());
13     list1.add( new B());
14
15     List<? extends A> list = list1;
16
17
18     ArrayList<B> list2 = new ArrayList();
19     list1.add(new A());
20     list = list2;
21     }
22 }
23
```

Rule 8

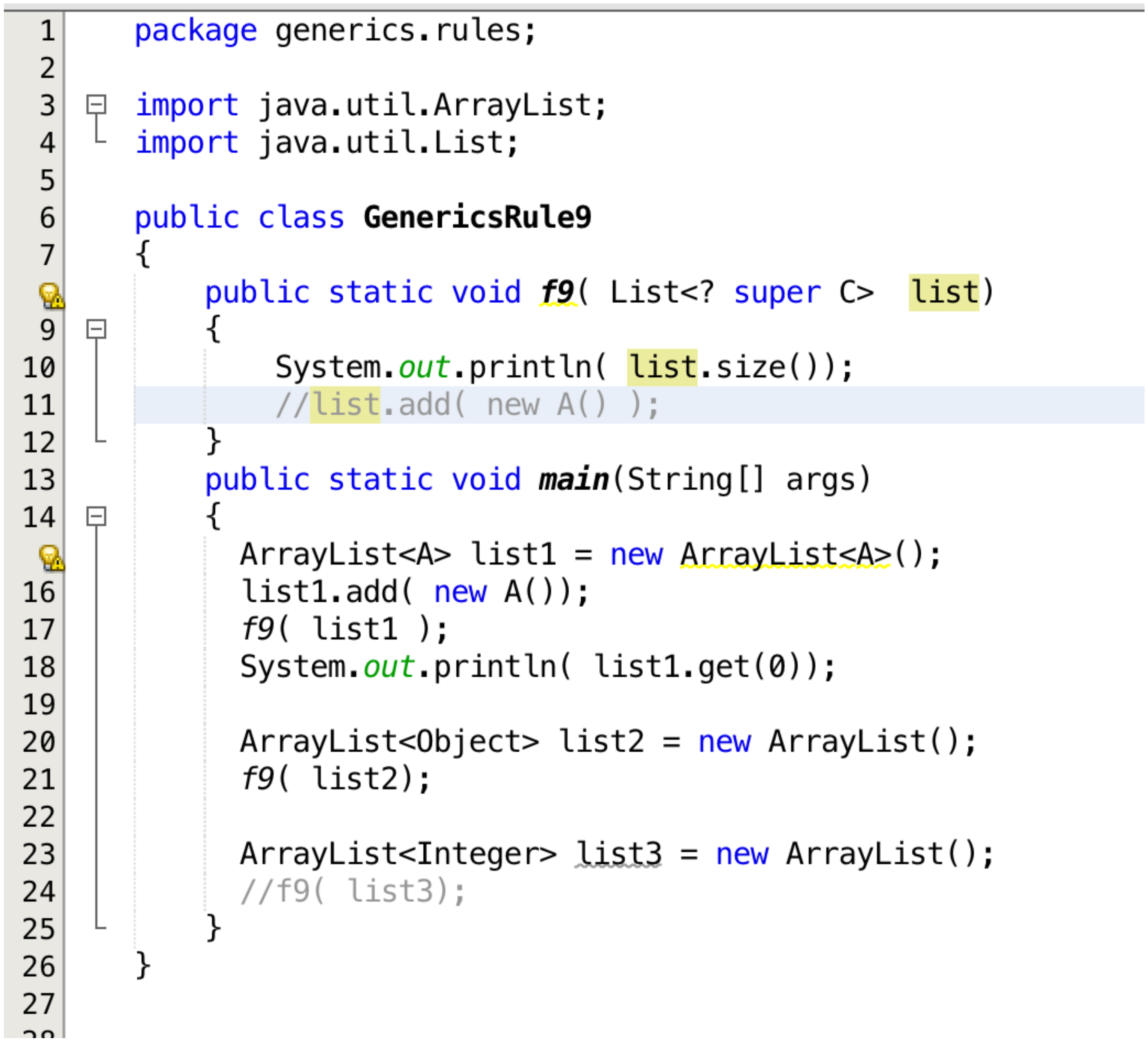
Upper bounded inheritance that is applicable in Rule 6 and Rule 7 does not work when it is an instance variable with an upper bound. Line 7 below. So line 9 (parameter is good). Line 7 is no good.



```
 1   package generics.rules;
 2   import java.util.ArrayList;
 3   import java.util.List;
 4
 5   public class GenericsRule8<T1>
 6   {
         List<? extends T1> lista = new ArrayList<String>();
 8
 9       public static <T2> void f8( List<? extends T2> list)
10       {
11           System.out.println( list.size());
12           //list.add(new C());
13       }
14       public static void main(String[] args)
15       {
16       List<A> list1 = new ArrayList();
17       list1.add(new C());
18       list1.add( new B());
19       f8( list1 );
20
21       ArrayList<B> list2 = new ArrayList();
22       list1.add(new A());
23       f8( list2 );
24
         ArrayList<String> list3 = new ArrayList<String>();
26       list3.add("abc");
27       f8( list3 );
28      }
29   }
30
```

Rule 9

Wildcards with a *lower bound* have the **syntax** `? super T` denotes an unknown type that is a supertype of `T`. In line 8, type class C is the lower bound of type class B, type class A (line 17) and type class Object (line 20). Type class Integer is NOT a super of type class C, and line 24 won't compile.

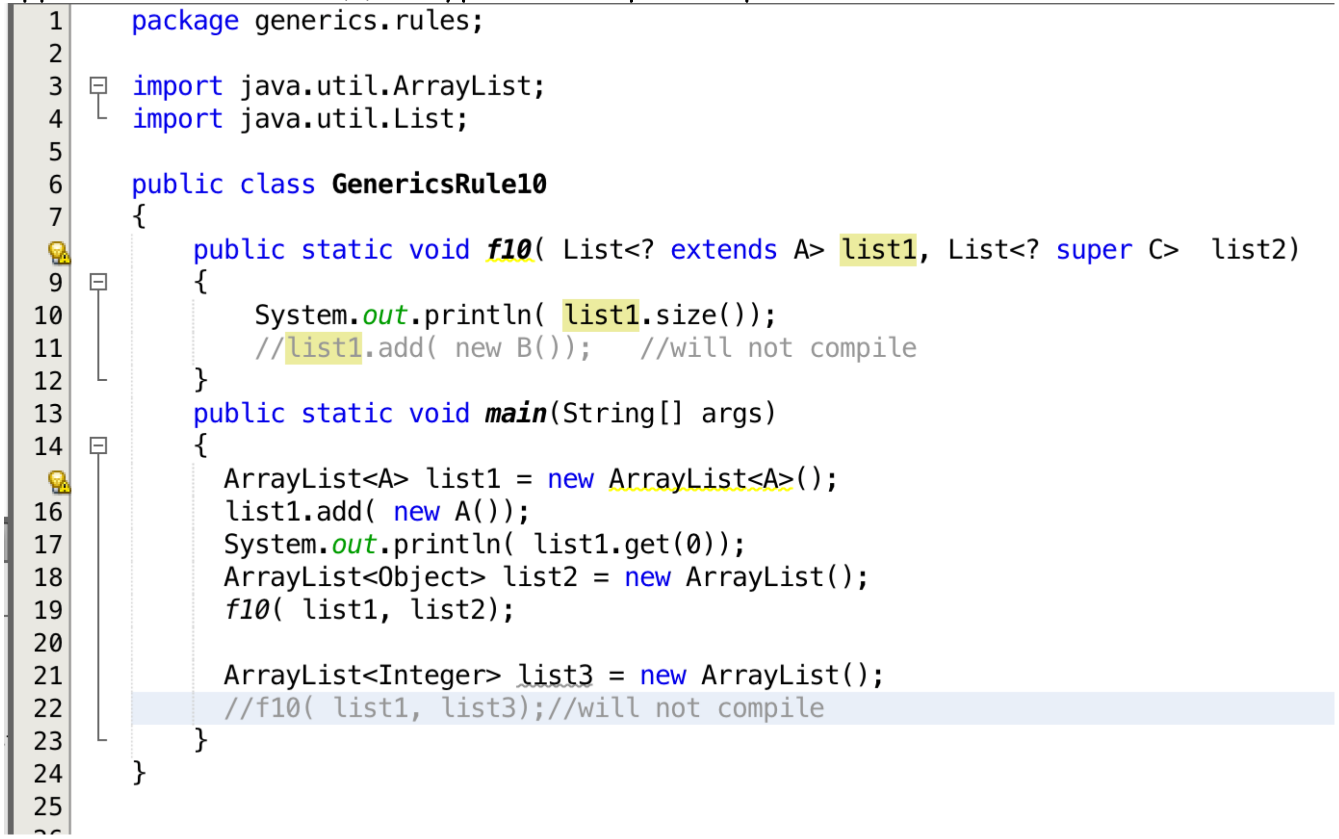


```
 1   package generics.rules;
 2
 3   import java.util.ArrayList;
 4   import java.util.List;
 5
 6   public class GenericsRule9
 7   {
         public static void f9( List<? super C>  list)
 9       {
10           System.out.println( list.size());
11           //list.add( new A() );
12       }
13       public static void main(String[] args)
14       {
           ArrayList<A> list1 = new ArrayList<A>();
16         list1.add( new A());
17         f9( list1 );
18         System.out.println( list1.get(0));
19
20         ArrayList<Object> list2 = new ArrayList();
21         f9( list2);
22
23         ArrayList<Integer> list3 = new ArrayList();
24         //f9( list3);
25       }
26   }
27
```

Rule 10

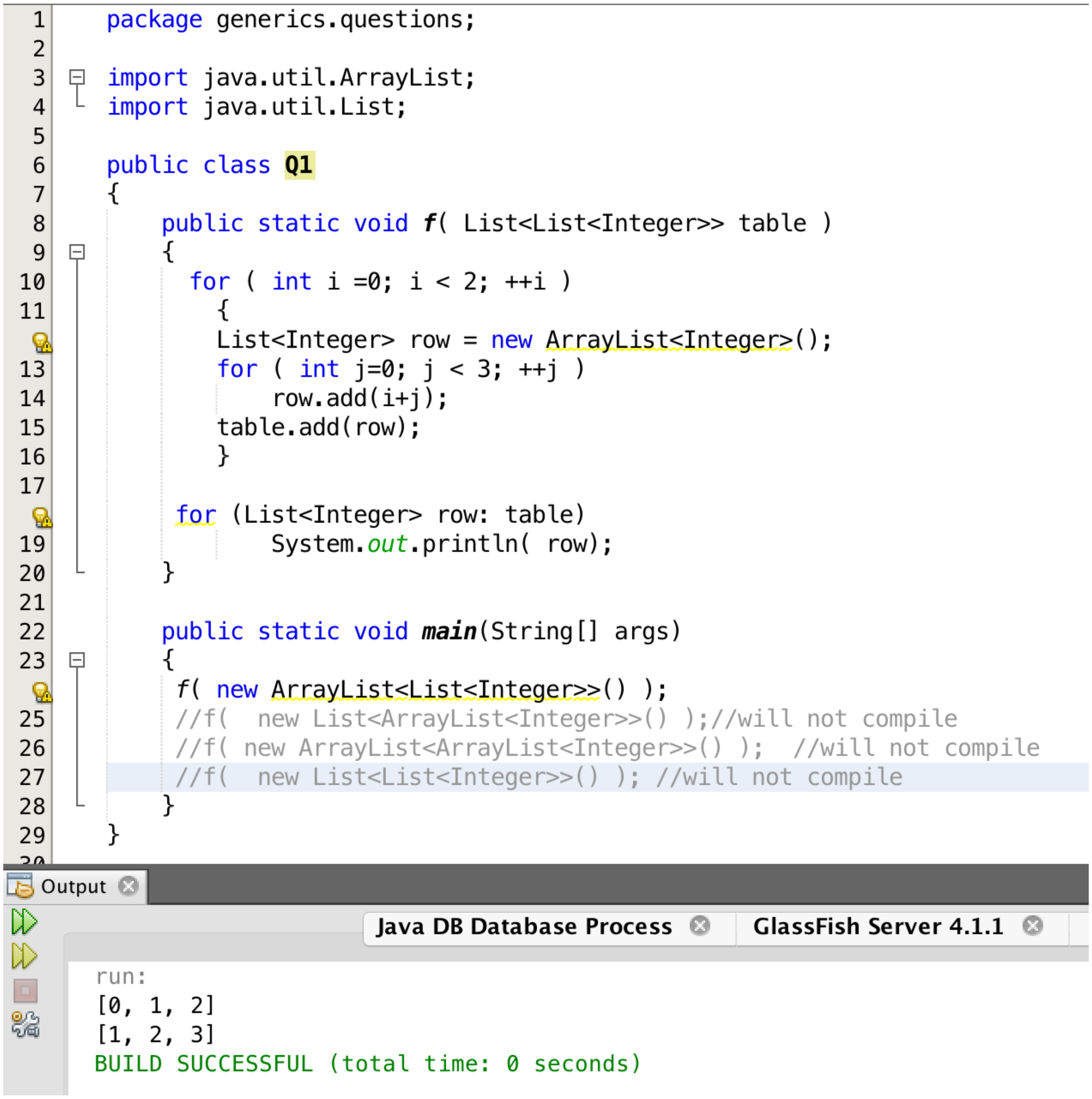
Upper bounded list1 and lower bounded list2 cannot modify list1 (line 11) or list2 inside the method which creates the bounds because inside the method we do not know exactly what the wildcard ? is. The wildcard ? can be A, B, C or Object type. But inside main we can modify the list as you know precisely (line 16) the type as there is no wildcard.

The bounded wildcards **allow inheritance in the diamond** and at the same time provide upper and lower bound(s) of types we can pass as parameters of the diamond.



```
 1   package generics.rules;
 2
 3   import java.util.ArrayList;
 4   import java.util.List;
 5
 6   public class GenericsRule10
 7   {
         public static void f10( List<? extends A> list1, List<? super C>  list2)
 9       {
10           System.out.println( list1.size());
11           //list1.add( new B());   //will not compile
12       }
13       public static void main(String[] args)
14       {
             ArrayList<A> list1 = new ArrayList<A>();
16           list1.add( new A());
17           System.out.println( list1.get(0));
18           ArrayList<Object> list2 = new ArrayList();
19           f10( list1, list2);
20
21           ArrayList<Integer> list3 = new ArrayList();
22           //f10( list1, list3);//will not compile
23       }
24   }
25
```

Rule 11
There is no inheritance in the diamond when we nest <<>>diamonds. The diamond of the diamond is of type List<Integer> here. We **cannot** pass as arguments Array<Integer> (line 26), but **can** pass the exact type of the inner diamond which is of type List<Integer> (line 24). Line 25 and 27 won't compile because we use new with abstract class List.



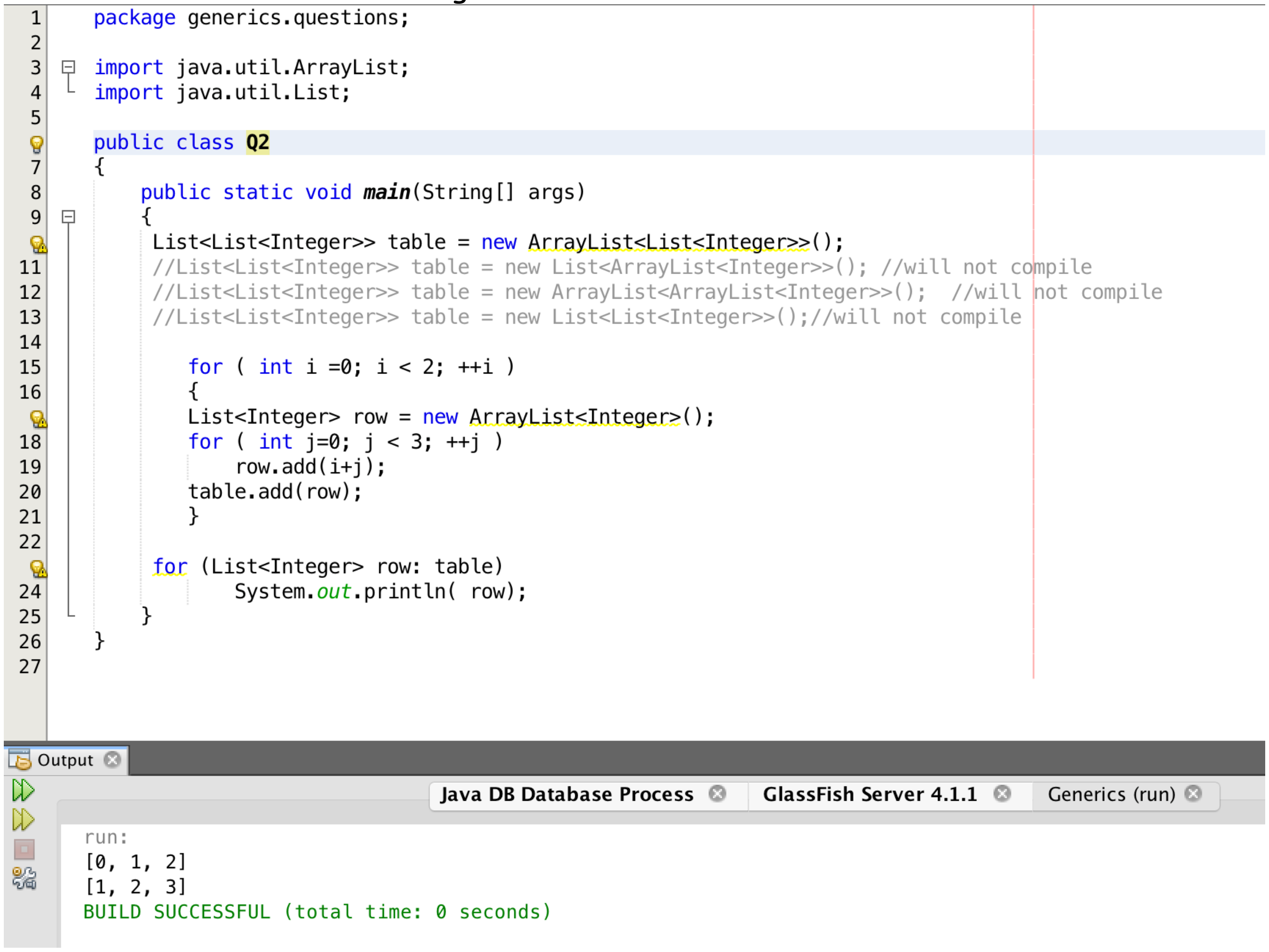
```
 1   package generics.questions;
 2
 3   import java.util.ArrayList;
 4   import java.util.List;
 5
 6   public class Q1
 7   {
 8       public static void f( List<List<Integer>> table )
 9       {
10         for ( int i =0; i < 2; ++i )
11           {
             List<Integer> row = new ArrayList<Integer>();
13           for ( int j=0; j < 3; ++j )
14               row.add(i+j);
15           table.add(row);
16           }
17
          for (List<Integer> row: table)
19               System.out.println( row);
20       }
21
22       public static void main(String[] args)
23       {
          f( new ArrayList<List<Integer>>() );
25        //f(  new List<ArrayList<Integer>>() );//will not compile
26        //f( new ArrayList<ArrayList<Integer>>() );  //will not compile
27        //f(  new List<List<Integer>>() ); //will not compile
28       }
29   }
```

Output | Java DB Database Process | GlassFish Server 4.1.1

```
run:
[0, 1, 2]
[1, 2, 3]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Rule 12

The same as rule 11 but for assignment. No inheritance in inner diamond.

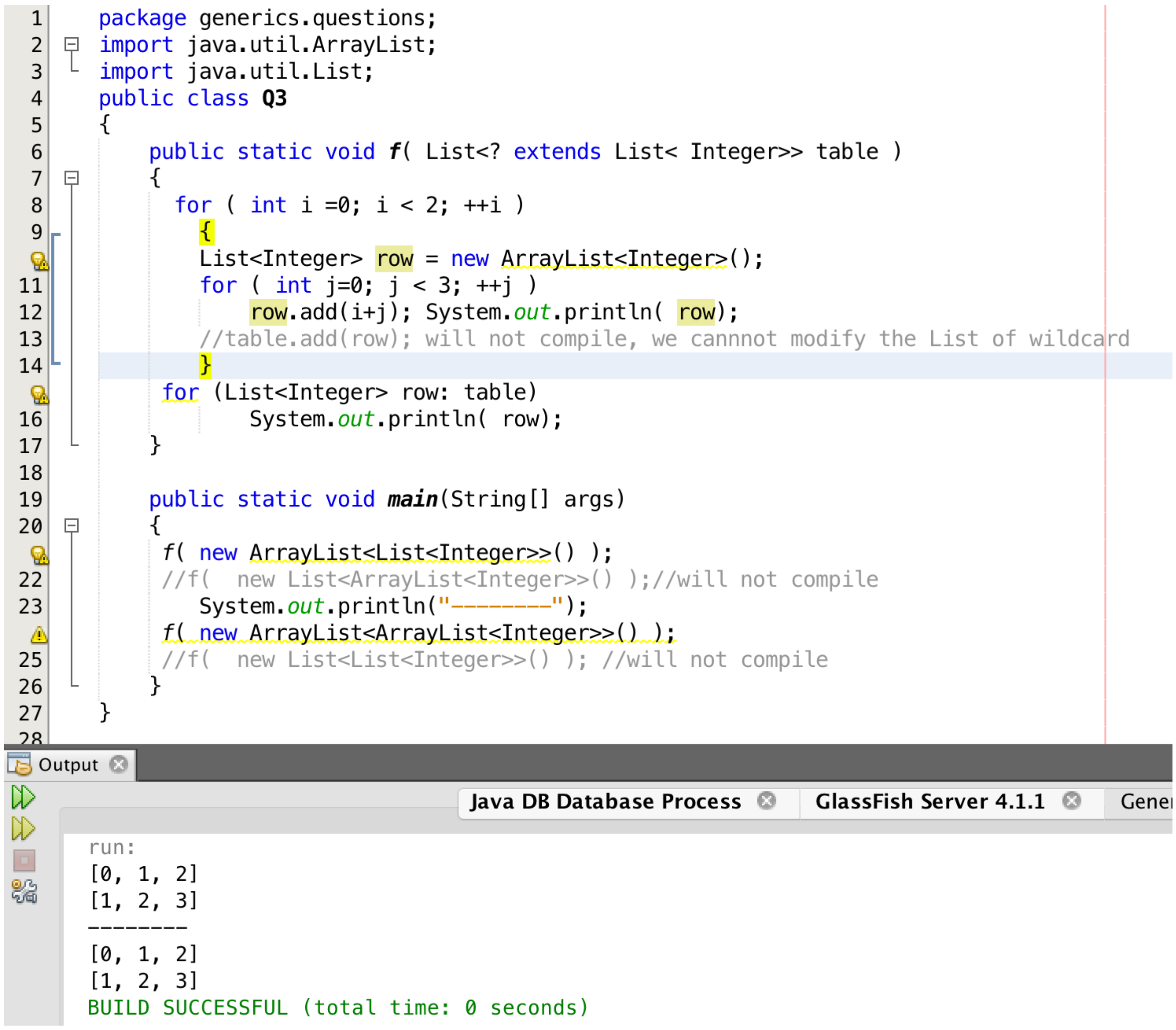


```java
package generics.questions;

import java.util.ArrayList;
import java.util.List;

public class Q2
{
    public static void main(String[] args)
    {
     List<List<Integer>> table = new ArrayList<List<Integer>>();
     //List<List<Integer>> table = new List<ArrayList<Integer>>(); //will not compile
     //List<List<Integer>> table = new ArrayList<ArrayList<Integer>>();  //will not compile
     //List<List<Integer>> table = new List<List<Integer>>();//will not compile

        for ( int i =0; i < 2; ++i )
        {
        List<Integer> row = new ArrayList<Integer>();
        for ( int j=0; j < 3; ++j )
            row.add(i+j);
        table.add(row);
        }

     for (List<Integer> row: table)
            System.out.println( row);
    }
}
```

```
run:
[0, 1, 2]
[1, 2, 3]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Rule 13

To allow inheritance in inner diamonds we use wild card ? in inner diamonds, as upper bound(? extends), lower bound (? super) or just ?. The example shows the upper bound fix of inner diamond inheritance. You can see that line 24 compiles as ArrayList<Integer> of line 24 is a subtype of parameter List<Integer> of line 6. Still we cannot modify the parameter (line 13).

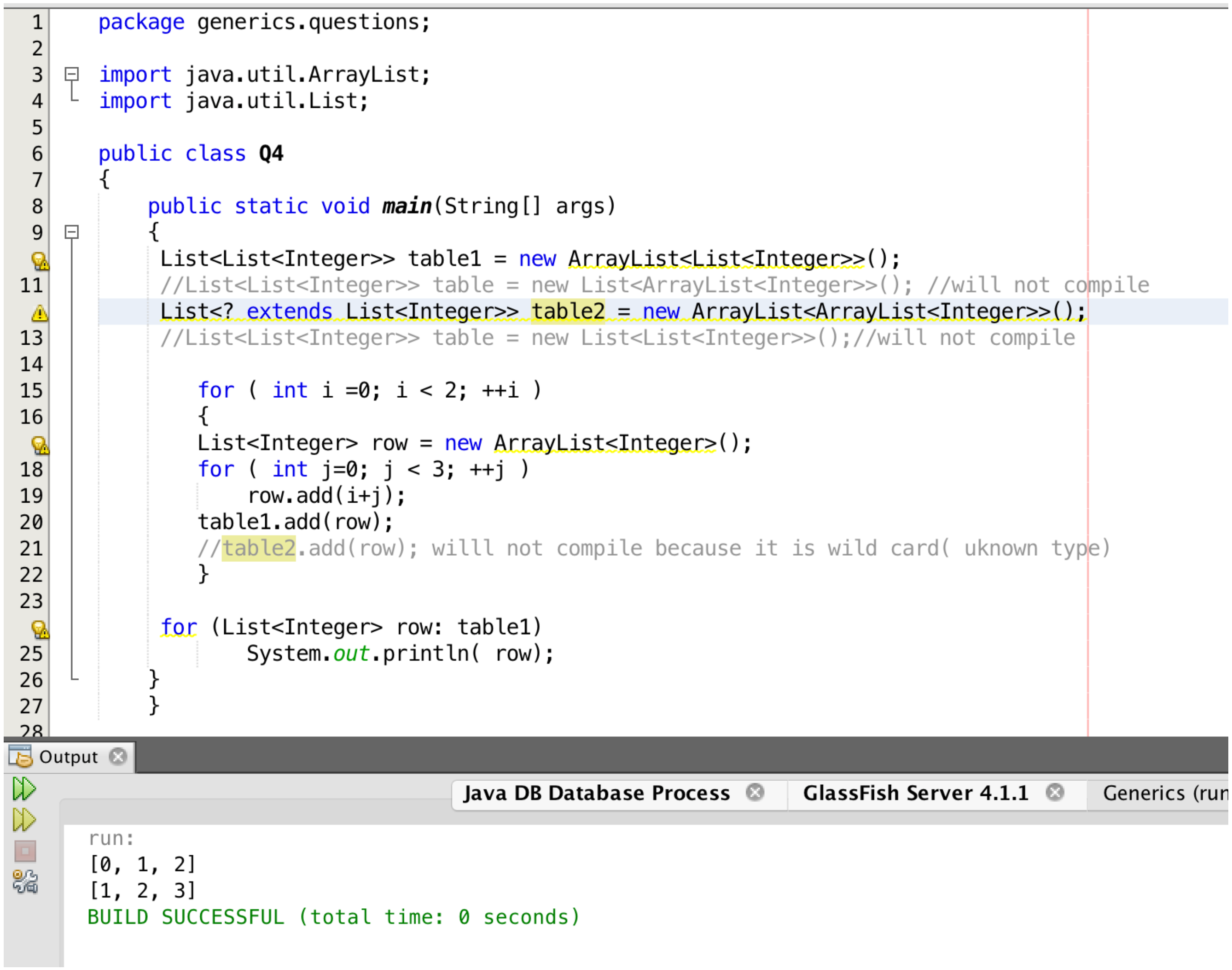


```
 1  package generics.questions;
 2  import java.util.ArrayList;
 3  import java.util.List;
 4  public class Q3
 5  {
 6      public static void f( List<? extends List< Integer>> table )
 7      {
 8        for ( int i =0; i < 2; ++i )
 9          {
           List<Integer> row = new ArrayList<Integer>();
11          for ( int j=0; j < 3; ++j )
12              row.add(i+j); System.out.println( row);
13          //table.add(row); will not compile, we cannnot modify the List of wildcard
14          }
         for (List<Integer> row: table)
16              System.out.println( row);
17      }
18
19      public static void main(String[] args)
20      {
         f( new ArrayList<List<Integer>>() );
22       //f(  new List<ArrayList<Integer>>() );//will not compile
23          System.out.println("--------");
         f( new ArrayList<ArrayList<Integer>>() );
25       //f(  new List<List<Integer>>() ); //will not compile
26      }
27  }
28
```

```
run:
[0, 1, 2]
[1, 2, 3]
--------
[0, 1, 2]
[1, 2, 3]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Rule 14

Same as Rule 13 but for assignment. Line 12 now compiles as on the RHS of the assignment we have ArrayList<Integer> extends the List<Integer> of the LHS of the assignment.

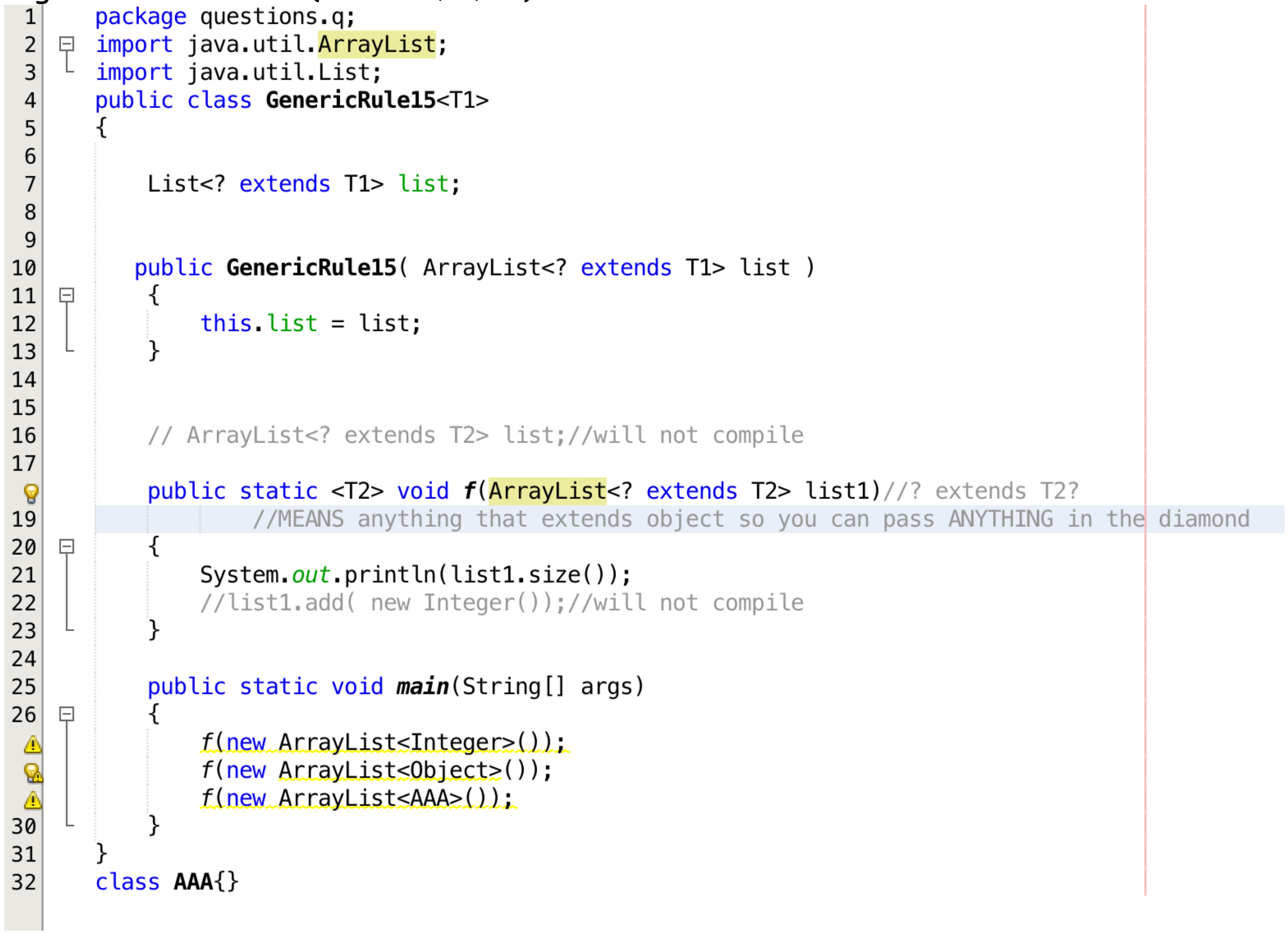


```
 1  package generics.questions;
 2
 3  import java.util.ArrayList;
 4  import java.util.List;
 5
 6  public class Q4
 7  {
 8      public static void main(String[] args)
 9      {
         List<List<Integer>> table1 = new ArrayList<List<Integer>>();
11       //List<List<Integer>> table = new List<ArrayList<Integer>>(); //will not compile
         List<? extends List<Integer>> table2 = new ArrayList<ArrayList<Integer>>();
13       //List<List<Integer>> table = new List<List<Integer>>();//will not compile
14
15          for ( int i =0; i < 2; ++i )
16          {
            List<Integer> row = new ArrayList<Integer>();
18          for ( int j=0; j < 3; ++j )
19              row.add(i+j);
20          table1.add(row);
21          //table2.add(row); willl not compile because it is wild card( uknown type)
22          }
23
         for (List<Integer> row: table1)
25              System.out.println( row);
26      }
27  }
28
```

```
run:
[0, 1, 2]
[1, 2, 3]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Rule 15

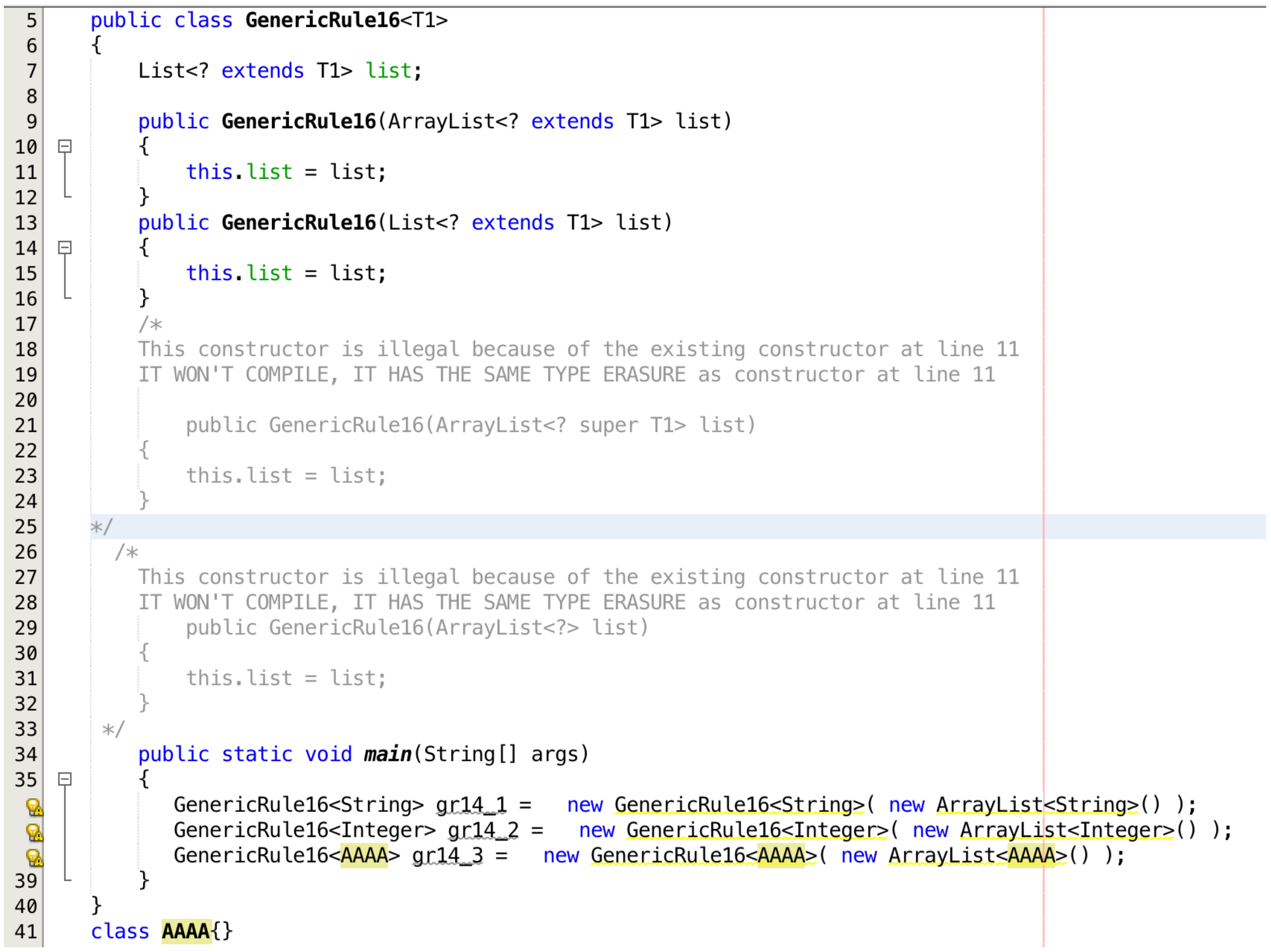
When a parameter has *wildcard* in the diamond and *extends* a generic type (such as T2), then it means anything that extends "Object" which implies that we can pass as argument ANY TYPE (lines 17, 8, 19).



```
 1   package questions.q;
 2   import java.util.ArrayList;
 3   import java.util.List;
 4   public class GenericRule15<T1>
 5   {
 6
 7       List<? extends T1> list;
 8
 9
10      public GenericRule15( ArrayList<? extends T1> list )
11       {
12           this.list = list;
13       }
14
15
16       // ArrayList<? extends T2> list;//will not compile
17
         public static <T2> void f(ArrayList<? extends T2> list1)//? extends T2?
19               //MEANS anything that extends object so you can pass ANYTHING in the diamond
20       {
21           System.out.println(list1.size());
22           //list1.add( new Integer());//will not compile
23       }
24
25       public static void main(String[] args)
26       {
             f(new ArrayList<Integer>());
             f(new ArrayList<Object>());
             f(new ArrayList<AAA>());
30       }
31   }
32   class AAA{}
```

Rule 16

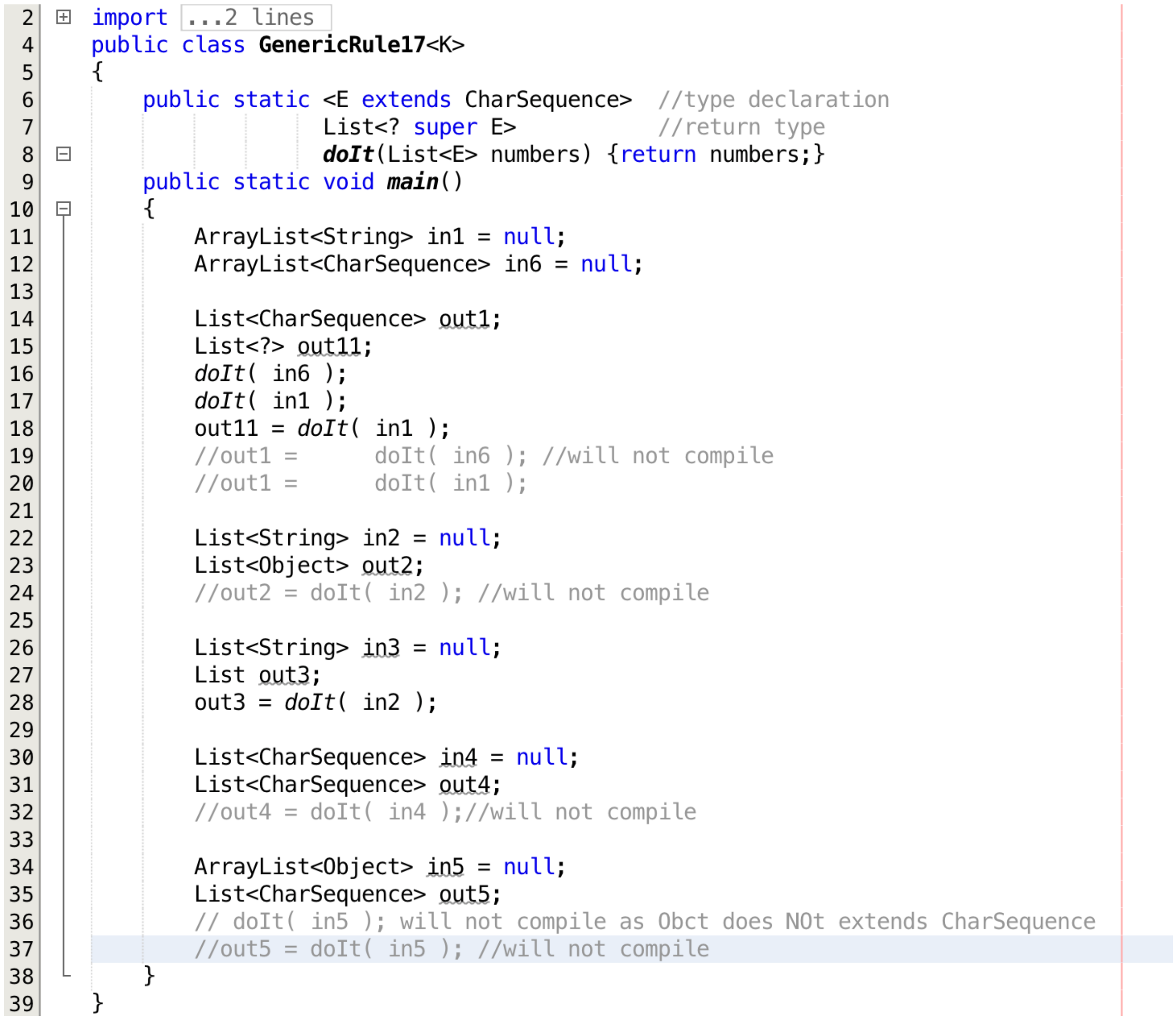
When a parameter of a constructor or method has *wildcard* in the diamond and *extends* a generic type (such as T1), line9, then it means anything that extends “Object” which implies that it **cannot be overloaded** with same signatures(line 21 uses <? super>, line 29 uses <?> BECUASE they would have the same type erasure after compilation (same signatures). Howerver, we can overload by using something different OUTISDE the out side the diamond (constructor overloading line 13, has different signatures than line 9)



```
 5  public class GenericRule16<T1>
 6  {
 7      List<? extends T1> list;
 8
 9      public GenericRule16(ArrayList<? extends T1> list)
10      {
11          this.list = list;
12      }
13      public GenericRule16(List<? extends T1> list)
14      {
15          this.list = list;
16      }
17      /*
18      This constructor is illegal because of the existing constructor at line 11
19      IT WON'T COMPILE, IT HAS THE SAME TYPE ERASURE as constructor at line 11
20
21          public GenericRule16(ArrayList<? super T1> list)
22      {
23          this.list = list;
24      }
25  */
26    /*
27      This constructor is illegal because of the existing constructor at line 11
28      IT WON'T COMPILE, IT HAS THE SAME TYPE ERASURE as constructor at line 11
29          public GenericRule16(ArrayList<?> list)
30      {
31          this.list = list;
32      }
33   */
34      public static void main(String[] args)
35      {
           GenericRule16<String> gr14_1 =   new GenericRule16<String>( new ArrayList<String>() );
           GenericRule16<Integer> gr14_2 =   new GenericRule16<Integer>( new ArrayList<Integer>() );
           GenericRule16<AAAA> gr14_3 =   new GenericRule16<AAAA>( new ArrayList<AAAA>() );
39      }
40  }
41  class AAAA{}
```

Rule 17

When the return type of a method contain wildcard we can assign the return type to a var that has either wilds card (line 15 and line 18), or no type lines(27,2*) f(or older Java,versions backward compatibly)



```
 2  import ...2 lines
 4  public class GenericRule17<K>
 5  {
 6      public static <E extends CharSequence>  //type declaration
 7                    List<? super E>           //return type
 8                    doIt(List<E> numbers) {return numbers;}
 9      public static void main()
10      {
11          ArrayList<String> in1 = null;
12          ArrayList<CharSequence> in6 = null;
13
14          List<CharSequence> out1;
15          List<?> out11;
16          doIt( in6 );
17          doIt( in1 );
18          out11 = doIt( in1 );
19          //out1 =      doIt( in6 ); //will not compile
20          //out1 =      doIt( in1 );
21
22          List<String> in2 = null;
23          List<Object> out2;
24          //out2 = doIt( in2 ); //will not compile
25
26          List<String> in3 = null;
27          List out3;
28          out3 = doIt( in2 );
29
30          List<CharSequence> in4 = null;
31          List<CharSequence> out4;
32          //out4 = doIt( in4 );//will not compile
33
34          ArrayList<Object> in5 = null;
35          List<CharSequence> out5;
36          // doIt( in5 ); will not compile as Obct does NOt extends CharSequence
37          //out5 = doIt( in5 ); //will not compile
38      }
39  }
```