ASDV 2520
Data Structures and Algorithms
Generics II – **Raw Types, Wild Cards**

1. Use the same project you have the classes GenericStack and create a a package rawTypes

Raw Types and Backward Compatibility

A generic class or interface used without specifying a concrete type, called a raw type, enables backward compatibility with earlier versions of Java.

You can use a generic class without specifying a concrete type like this:

GenericStack stack = new GenericStack(); // raw type
This is roughly equivalent to
GenericStack<Object> stack = new GenericStack<Object>();

2. Create a class raw type class Max and test it as shown below:

```
1   package rawTypes;
2   public class Max
3   {
4       public static Comparable max(Comparable o1, Comparable o2)
5       {
6           if (o1.compareTo(o2) > 0)
7             {
8               return o1;
9             }
10          else
11            {
12              return o2;
13            }
14      }
15
16      public static void main(String[] args)
17      {
18          System.out.println(max(1, 2));
19          try
20            {
21              System.out.println(
22                      "This line compiles but crushes the program" + max(1, "two"));
23            }
24          catch (ClassCastException e)
25            {
27              System.err.println("RAW TYPES ARE UNSAFE " + e.getMessage());
27            }
28      }
29  }
```

3. In the code below we ADD method maxSafe, lines 15-25 which is the remedy for raw types. Also, inside main() add lines 32-35 to test maxSafe. As you see line 35 DOES NOT COMPILE as line 38 compiled. Comment out line 35 and run the class Max.

```
1  package rawTypes;
2  public class Max
3  {
4      public static Comparable max(Comparable o1, Comparable o2)
5      {
6          if (o1.compareTo(o2) > 0)
7            {
8              return o1;
9            }
10         else
11           {
12             return o2;
13           }
14     }
15     public static <E extends Comparable<E>>  E maxSafe(E e1, E e2)
16     {
17         if (e1.compareTo(e2) > 0)
18           {
19             return e1;
20           }
21         else
22           {
23             return e2;
24           }
25     }
26     public static void main(String[] args)
27     {
28         System.out.println(max(1, 2));
29
30         try
31           {
32             System.out.println( maxSafe(1, 2));
33             System.out.println( maxSafe("abc", "ABC"));
34             System.out.println("");
35             System.out.println( maxSafe(1, "two"));
36
37             System.out.println(
38                     "This line compiles but crushes the program" + max(1, "two"));
39
40
41           }
42         catch (ClassCastException e)
43           {
44               System.err.println("RAW TYPES ARE UNSAFE " + e.getMessage());
45           }
46     }
47 }
48
```

4. Add lines 37-41 and **replicate** the SAFE and UNSAFE types by using the class GenericStack as shown below. UNDESTAND RAW TYPES (Old Java) and HOW WE HAVE SAFE types now. The stack of line 37 is unsafe. The stack of Line 38 is safe.

```
1   package rawTypes;
2   public class Max
3   {
4       public static Comparable max(Comparable o1, Comparable o2)
5       {
6           if (o1.compareTo(o2) > 0)
7             {
8               return o1;
9             }
10          else
11            {
12              return o2;
13            }
14      }
15      public static <E extends Comparable<E>>  E maxSafe(E e1, E e2)
16      {
17          if (e1.compareTo(e2) > 0)
18            {
19              return e1;
20            }
21          else
22            {
23              return e2;
24            }
25      }
26      public static void main(String[] args)
27      {
28          System.out.println(max(1, 2));
29
30          try
31            {
32              System.out.println( maxSafe(1, 2));
33              System.out.println( maxSafe("abc", "ABC"));
34              System.out.println("");
35              //System.out.println( maxSafe(1, "two"));
36
37              generics.GenericStack stackUnsafe = new generics.GenericStack();
38              generics.GenericStack<Integer> stackSafe = new generics.GenericStack();
39              stackSafe.push( 1 ); stackSafe.push( 2 );
40              System.out.println( stackSafe );
41              stackUnsafe.push(1); stackUnsafe.push("two");
42
43              System.out.println(
44                      "This line compiles but crushes the program" + max(1, "two"));
45            }
46          catch (ClassCastException e)
47            {
48              System.err.println("RAW TYPES ARE UNSAFE " + e.getMessage());
49            }
50      }
51  }
```

Wild Cards

5. Create a new package wildCards. Create the class NoWildCard under that package and type the code shown below:

The class NoWildCard has a compile error in line 25 because intStack of type GenericStack<Integer>is not an instance of GenericStack<Number> even though class Integer is a subclass of class Number(impements interface Number). Thus, you **cannot** invoke **max(intStack) .** **The fact is that Integer is a subtype of Number , but GenericStack<Integer> is not a subtype of GenericStack<Number>** .

```
1   package wildCards;
2
3   public class NoWildCard
4   {
5       public static double max(generics.GenericStack<Number> stack)
6       {
7           double max = -888;
8
9           while (!stack.isEmpty())
10            {
11              double value = stack.pop().doubleValue();
12              if (value > max)
13                {
14                  max = value;
15                }
16            }
17          return max;
18      }
19  public static void main(String[] args)
20      {
21          generics.GenericStack<Integer> intStack = new generics.GenericStack<>();
22          intStack.push(1);
23          intStack.push(2);
24          intStack.push(-2);
            System.out.print("The max number is " + max( intStack ));
26      }
27  }
```

6. To remedy this problem, use **wildcard** generic types.

A wildcard generic type has three forms, where T is a generic type.:

1. **?**

   ? is called an unbounded wildcard , is the same as ? extends Object .

2. **? extends T ,**

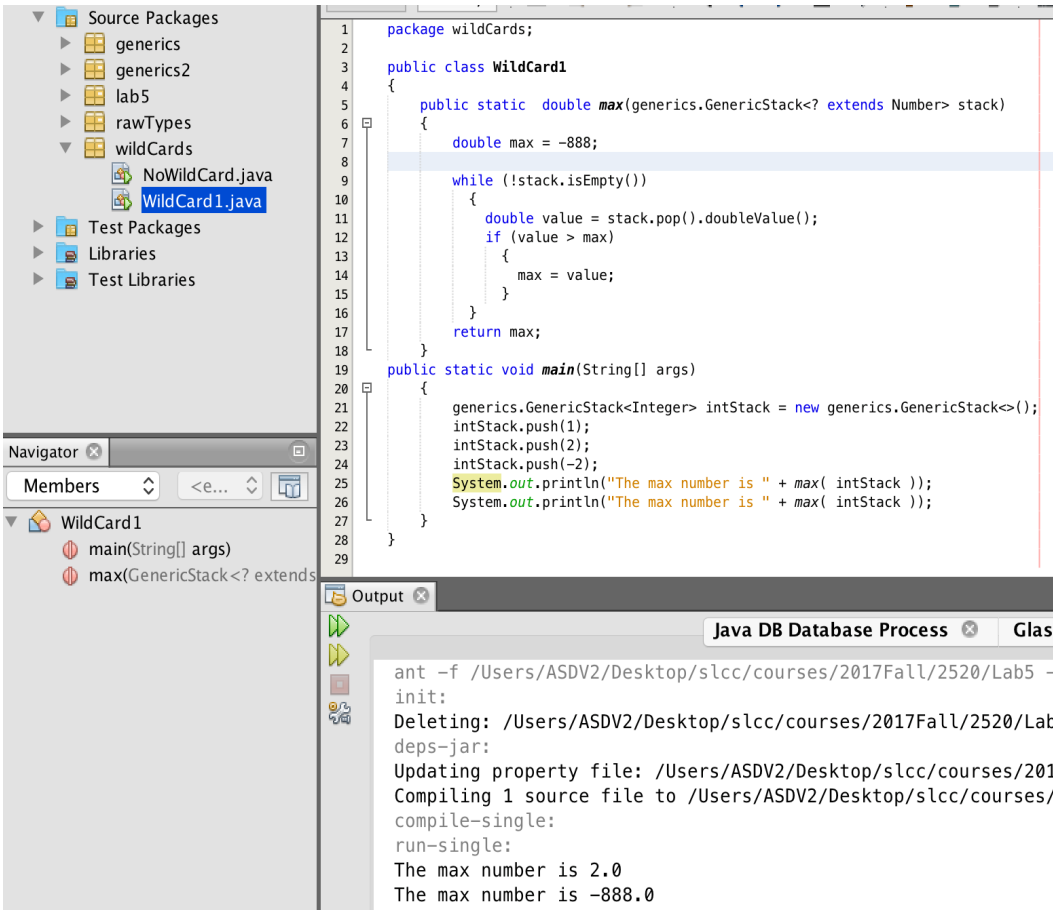
   ? extends T , is called a bounded wildcard , the ? is T or a subtype of T .

3. **? super T**

   ? super T , is called a lower-bound wildcard , the ? is T or a supertype of T .

You can fix the compiling error error by replacing line 5 i as follows:

7. Copy class NoWildcard and paste it (refactor to class WildCard) under package wildCards as shown below. REPLACE line 5 with the WILD CARD line 5 shown below. Test it as shown in main.



<? extends Number> of line 5 is a wildcard type that represents Number or a subtype of Number, so it is legal to invoke max(new GenericStack<Integer>()) or max(new GenericStack<Double>()).

8. Under package wildcard create class WildCard2 and run it.

WildCard2 s using the ? wildcard in the print method that prints objects in a stack and empties the stack. <?> is a wildcard that represents any object type. It is equivalent to <? extends Object> . What happens if we replace GenericStack<?> with GenericStack<Object> ? It would be wrong to invoke print(intStack) , because intStack is not an instance of GenericStack<Object> . Please note that GenericStack<Integer> is not a subtype of GenericStack<Object> , even though Integer is a subtype of Object .

```
1   package wildCards;
2   import generics.GenericStack;
3
4   public class WildCard2
5   {
6       public static void print(GenericStack<?> stack)
7       {
8           while (!stack.isEmpty())
9             {
10              System.out.print(stack.pop() + " ");
11            }
12      }
13      public static void main(String[] args)
14      {
15          GenericStack<Integer> intStack = new GenericStack<>();
16          intStack.push(1);
17          intStack.push(2);
18          intStack.push(-2);
19
20          print(intStack);
21      }
22  }
```

9. **When is the wildcard <? super T> needed?** Under package wildCards create the class **WildCardWithSuper**, **and run it.**

WildCardWithSuper creates a stack of strings in stack1 (line 14) and a stack of objects in stack2 (line 15), and invokes add(stack1, stack2) (line 20) to add the strings in stack1 into stack2 .
GenericStack<? super T> is used to declare stack2 in line 5. If <? super T> is replaced by <T> in line 5, a compile error will occur on add(stack1, stack2) in line 15, because stack1 's type is GenericStack<String> and stack2 's type is GenericStack<Object> .
<? super T> MEANS type T or a supertype of T . Object is a supertype of String .

```
1   package wildCards;
2   import generics.GenericStack;
3   public class WildCardWithSuper
4   {
5       public static <T> void add(GenericStack<T> stack1, GenericStack<? super T> stack2)
6       {
7           while (!stack1.isEmpty())
8             {
9               stack2.push(stack1.pop());
10            }
11      }
12      public static void main(String[] args)
13      {
14          generics.GenericStack<String> stack1 = new generics.GenericStack<>();
15          generics.GenericStack<Object> stack2 = new generics.GenericStack<>();
16          stack2.push("one");
17          stack2.push(2);
18          stack1.push("one");
19
20          add(stack1, stack2);
21          WildCard2.print( stack2 );
22      }
23  }
```

**** VERY IMPORTANT NOTE 1****

Erasure and Restrictions on Generics

The information on generics is used by the compiler but is not available at runtime. This is called **type erasure**. Generics are implemented using an approach called type erasure: The compiler uses the generic type information to compile the code, but erases it afterwards.

When generic classes, interfaces, and methods are compiled, the compiler **replaces the generic type with the Object type**.

For example, the compiler would convert the following method in (a) into (b).

```
public static <E> void print(E[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(a)

```
public static void print(Object[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(b)

If a generic type is bounded, the compiler replaces it with the bounded type (its super). For example, the compiler would convert the following method in (a) into (b).

```
public static <E extends GeometricObject>
    boolean equalArea(
      E object1,
      E object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(a)

```
public static
    boolean equalArea(
      GeometricObject object1,
      GeometricObject object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(b)

**** VERY IMPORTANT NOTE 2****

Restriction 1
Cannot Use new E()
use E object = new E();

Restriction 2
Cannot Use new E[]
use E[] ar = (E[])new Object[SZE];

Restriction 3: *A Generic Type Parameter of a Class Is Not Allowed in a Static Context*

```
public class Test<E> {
  public static void m(E o1) { // Illegal
  }

  public static E o1; // Illegal

  static {
    E o2; // Illegal
  }
}
```

Restriction 4: Exception Classes Cannot Be Generic

A generic class may not extend java.lang.Throwable , so the following class declaration would be **illegal**:
public class MyException<T> extends Exception
{
}