

# Module 3

---

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

## ■ Lecture 3 – Generics

## Learning Objectives

---

After successfully completing the module, students will be able to do the following:

1. Use and develop generic methods and generic classes.
2. Use bounded generic types.
3. Examine the benefits of generics.
4. Determine when wildcard generic types are necessary.
5. Analyze generic type erasure.
6. Analyze the limitations of generics.
7. Find out the relationship among interfaces and classes in Java Collections Framework.
8. Differentiate between Collection and Collections.
9. Use the common methods in the Collection interface.
10. Use the common methods in the Collections class.
11. Use the common methods in the Arrays class.
12. Develop applications using ArrayList, LinkedList, Stack, Queue, Set, Map.

### Module 3 Study Guide and Deliverables

January 30 – February 5

**Topics:** Lecture 3: Collections and Generics

**Readings:**

- Module 3 online content
- Deitel & Deitel, Chapter 16 and Chapter 20
- Instructor's live class slides

**Assignments:** Assignment 3 due **Tuesday, February 13 at 6:00 AM ET**  
(Access at "Assignments" on the left-hand course menu).

- Live** Join the live classroom and the facilitator's live office hour
- Classrooms:** session at "Live Classroom/Offices" > "Live Classroom" > Launch Meeting.
- **Wednesday, January 31, 6:00 – 8:00 PM ET**
  - Live Office: Schedule with facilitators as long as there are questions

## Module Welcome and Introduction

---

met\_cs622\_18\_su1\_ebraude\_mod3 video cannot be displayed here

## Generics

---

Generics make code reuse easier and enable detection of errors at compile time rather than at run time. As we will see shortly, generics eliminate or minimize the use of casts and make code easier to read and modify. This module discusses generic methods, generic classes, and generic collections.

### Need for Generic Methods

Consider the following example where the task is to print the elements of each of three different arrays. The arrays are different because they hold items of different types—Integer, Double, and String. Three different methods—

showIntArray(), showDoubleArray(), and showStringArray()—are needed to print the arrays, since one method prints elements of only one type.

```
// Not generic

public class NotGeneric1
{
    public static void main(String[] args)
    {
        Integer[] iarr = {2, 5, 8};           // autobox
        Double[] darr = {28.67, 5.05, 8.3};   // autobox
        String[] sarr = {"Twelve", "Angry", "Men"};

        showIntArray(iarr);
        showDoubleArray(darr);
        showStringArray(sarr);
    }

    public static void showIntArray(Integer[] a)
    {
        for (Integer elem: a)
            System.out.print(elem + " ");
        System.out.println();
    }

    public static void showDoubleArray(Double[] a)
    {
        for (Double elem: a)
            System.out.print(elem + " ");
        System.out.println();
    }

    public static void showStringArray(String[] a)
    {
        for (String elem: a)
            System.out.print(elem + " ");
        System.out.println();
    }
}
```

The output of the program is shown below.

Output of the program – non-generic example

```
Administrator: Command Prompt
C:\Users\uday\UdayJava\generics>javac NotGeneric1.java
C:\Users\uday\UdayJava\generics>java NotGeneric1
2 5 8
28.67 5.05 8.3
Twelve Angry Men
C:\Users\uday\UdayJava\generics>
```

Now, it is not difficult to see that the three methods `showIntArray()`, `showDoubleArray()`, and `showStringArray()` are almost identical except for the type of the elements they handle. If the same method could handle different types, the code would be much cleaner (and shorter). This is possible if the method is generic. A generic method can handle arbitrary non-primitive or reference types; it is not confined to a single type.

The following example illustrates the use of a generic method `showGeneric()` that is capable of printing arrays of arbitrary (reference) types. A generic method is specified by providing the generic type name in angle brackets before the return type in the method's header. Note that the argument to `showGeneric` is of type `T[]`, or an array of type `T`, where `T` stands for type (`T` is used by convention; we could have used, say, `A` or `X` as well).

```
// Generic method

public class Generics1
{
    public static void main(String[] args)
    {
        Integer[] iarr = {2, 5, 8};           // autobox
        Double[] darr = {28.67, 5.05, 8.3}; // autobox
        String[] sarr = {"Twelve", "Angry", "Men"};

        showGeneric(iarr);
        showGeneric(darr);
        showGeneric(sarr);
    }

    public static <T> void showGeneric(T[] a) // generic method
    {
        for (T elem: a)
            System.out.print(elem + " ");
        System.out.println();
    }
}
```

The output of the above program is the same as that of its non-generic counterpart.

Let us repeat that primitive types (e.g., `int`, `double`, `char`) cannot be used as the actual type for a generic type name `T`. For example, it would be wrong to expect `showGeneric(myarr)` to work when `myarr` is an array of `int`'s (or `double`'s or `char`'s). (Note the autoboxing in the above code.)

As another example of a situation where a generic method is useful, consider the problem of finding the maximum of three given elements. The non-generic version (class NotGeneric2 below) has three different methods—`intFindMax`, `doubleFindMax`, and `stringFindMax`—for the three types of elements, whereas a single generic method `genericFindMax` (in class `Generics2`) handles all the different types.

```
// Not generic

public class NotGeneric2
{
    public static void main(String[] args)
    {
        int imax = intFindMax(2, 5, 8);
        double dmax = doubleFindMax(28.67, 5.05, 8.3);
        String smax = stringFindMax("good", "bad", "ugly");

        System.out.printf("imax = %d, dmax = %f, smax = %s\n", imax, dmax, smax);
    }

    public static int intFindMax(int a, int b, int c)
    {
        int largest = a > b ? a : b;
        largest = c > largest ? c : largest;
        return largest;
    }

    public static double doubleFindMax(double a, double b, double c)
    {
        double largest = a > b ? a : b;
        largest = c > largest ? c : largest;
        return largest;
    }

    public static String stringFindMax(String a, String b, String c)
    {
        String largest = a.compareTo(b) > 0 ? a : b;
        largest = c.compareTo(largest) > 0 ? c : largest;
        return largest;
    }
}

// Generic method

public class Generics2
{
    public static void main(String[] args)
    {
        int imax = genericFindMax(2, 5, 8); // autobox, autounbox
```

```

        double dmax = genericFindMax(28.67, 5.05, 8.3); // autobox, autounbox
        String smax = genericFindMax("good", "bad", "ugly");

        System.out.printf("imax = %d, dmax = %f, smax = %s\n", imax, dmax, smax);
    }

    public static <T extends Comparable<T>> T genericFindMax(T a, T b, T c)
    {
        T largest = a.compareTo(b) > 0 ? a : b;
        largest = c.compareTo(largest) > 0 ? c : largest;
        return largest;
    }
}

```

The generic version of the Comparable interface, Comparable<T>, is used in the above program. Note that the return type of method genericFindMax() is T. Note also the autoboxing and autounboxing that take place at the invocation of genericFindMax() and in storing the method's return object.

Why not use "Object" all through? Let us see what the problem is with the following:

```

public static void showObject(Object[] a)
{
    for (Object elem: a)
        System.out.println(elem);
}

```

We object to "Object" because Object is too wide (general), and offers no guarantee that type compatibility will be maintained. For instance, a part of a program could put an Integer element in the array while another part could expect a String as an element. The use of generics catches such errors at compile time.

## Generic Class

A generic type can be defined for a class. A generic class is a parameterized class. When using the generic class to create objects, we must supply a concrete type.

Consider (non-generic) class Item:

```

class Item          // non-generic class
{
    private Object object;
    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}

```

A better alternative is a generic class:

```

class GenericItem<T>          // T stands for "Type"

```

```

{
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}

```

Study the definition and use of the generic class `GenericItem<T>` in the following example.

```

// Generic class

public class GenericClass1
{
    public static void main(String[] args)
    {
        GenericItem<Integer> myint = new GenericItem<Integer>();
        System.out.printf("myint: %s\n", myint);

        GenericItem<Double> mydouble = new GenericItem<Double>(3.1416);
        System.out.printf("mydouble: %s\n", mydouble);

        GenericItem<String> mystring = new GenericItem<String>("Hello");
        System.out.printf("mystring: %s\n", mystring);

        GenericItem<Alpha> myalpha = new GenericItem<Alpha>(new Alpha(91));
        System.out.printf("myalpha: %s\n", myalpha);
    }
}

class GenericItem<T>
{
    private T t;

    public GenericItem()
    {
        t = null;
    }

    public GenericItem(T t)
    {
        this.t = t;
    }

    public String toString()
    {
        return t == null ?
            "GenericItem<T> has no object" :
            "GenericItem<T> object: " + t.toString();
    }
}

```

```

class Alpha
{
    private int id;

    public Alpha(int number)
    {
        id = number;
    }

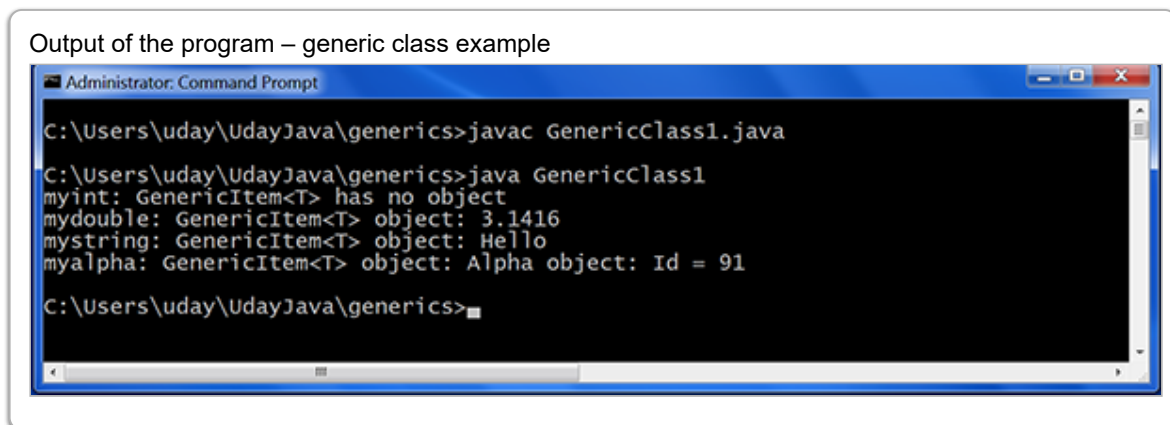
    public String toString()
    {
        return "Alpha object: Id = " + id;
    }
}

```

The above example demonstrated that by instantiating the same generic class with the types Integer, Double, String, and Alpha, we obtained four different classes:

- GenericItem<Integer>,
- GenericItem<Double>,
- GenericItem<String>,
- GenericItem<Alpha>.

Note that the constructor of class GenericItem<T> has method name GenericItem, not GenericItem<T>. The output of the program is shown below.



To instantiate a parameterized class, the T must be replaced with a reference type, that is:

- a non-primitive type (e.g., Integer)
- a class type (e.g., Alpha where Alpha is a user-defined class)
- an interface (e.g., Comparable or Comparable<String> or a user-defined interface)
- an array type (e.g., double[] or Double[])

A further example of generic class instantiation is given below. Note how Double[] is used as the type, defining GenericItem<Double[]>.

```
// Generic class
```



```

public class Generics3
{
    public static void main(String[] args)
    {
        GenericItem<Student> stu = new GenericItem<Student>();
        stu.set(new Student(172, "Vishnu", 4.99));
        System.out.printf("Student: %s%n", stu.get());

        GenericItem<Double[]> doublearrayItem = new GenericItem<Double[]>();
        Double Darr[] = {1.1, 2.2, 3.3};
        doublearrayItem.set(Darr);
        System.out.println("doublearrayItem's internal array's second element: " +
            (doublearrayItem.get())[1]);
    }
}

class GenericItem<T>          // T stands for "Type"
{
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}

class Student
{
    private int id;
    private String name;
    private double gpa;

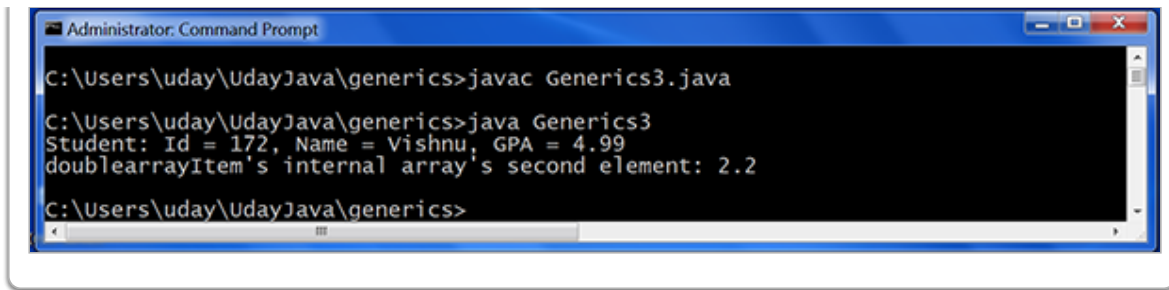
    public Student(int serialno, String nm, double grade)
    {
        id = serialno;
        name = nm;
        gpa = grade;
    }

    public String toString()
    {
        return "Id = " + id + ", Name = " + name + ", GPA = " + gpa;
    }
}

```

The program's output is shown below.

Output of the program – generic class example



```
Administrator: Command Prompt
C:\Users\uday\UdayJava\generics>javac Generics3.java
C:\Users\uday\UdayJava\generics>java Generics3
Student: Id = 172, Name = Vishnu, GPA = 4.99
doublearrayItem's internal array's second element: 2.2
C:\Users\uday\UdayJava\generics>
```

Generics eliminate or minimize the use of casts. Consider the following piece of code:

```
List mylist = new ArrayList();
mylist.add("hello");
String s = (String) mylist.get(0);
```

When re-written to use generics, the code does not require casting:

```
List<String> mylist = new ArrayList<String>();
mylist.add("hello");
String s = mylist.get(0);    // no cast
```

An extremely useful generic class is a generic stack. The following example demonstrates how a generic stack can be defined and used. (Java already provides a generic `Stack<T>` class; we show the following example only for pedagogical purposes.) In particular, note the use of `GenStack<GenericItem<Double>>`.

```
// Generic stack

import java.util.*;

public class GenericStack
{
    public static void main(String[] args)
    {
        GenStack<String> s2 = new GenStack<String>();
        s2.push("one");
        s2.push("two");
        s2.push("three");
        while (!s2.isEmpty()) System.out.printf("%s\n", s2.pop());

        GenericItem<Double> g1 = new GenericItem<Double>();
        g1.set(3.1416);

        GenericItem<Double> g2 = new GenericItem<Double>();
        g2.set(2.718);

        GenStack<GenericItem<Double>> s3 = new GenStack<GenericItem<Double>>();
        s3.push(g1);
        s3.push(g2);
        while (!s3.isEmpty()) System.out.printf("%f\n", s3.pop().get());
    }
}
```

```

    }

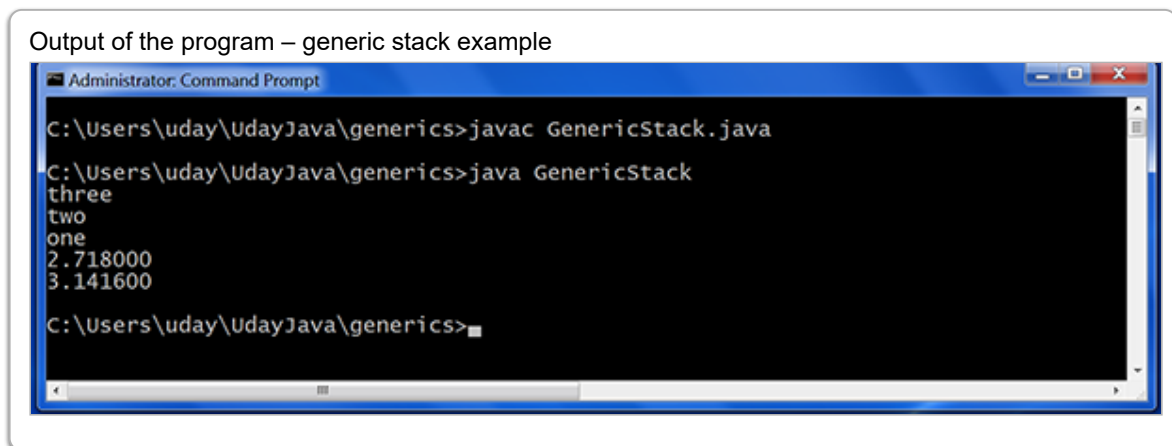
    class GenStack<T>
    {
        private ArrayList<T> a;

        public GenStack() { a = new ArrayList<T>(); }
        public void push(T t) { a.add(t); }
        public T pop() { return isEmpty() ? null : a.remove(a.size() - 1); }
        public boolean isEmpty() { return a.size() == 0; }
    }

    class GenericItem<T>
    {
        private T t;
        public void set(T t) { this.t = t; }
        public T get() { return t; }
    }

```

The output is shown below:



## Bounded Generic Type

Consider the following program, which is reduced to the bare outlines in order to focus on an important idea.

```

// Bounded type - wrong - doesn't compile

public class BoundedTypeWrong
{
    public static void main(String[] args)
    {
        GenericItem<Student> stu = new GenericItem<Student>();

        GenericItem<GradStudent> grad = new GenericItem<GradStudent>();
    }
}

```

```

        processItem(stu);
        processItem(grad);
    }

    public static void processItem(GenericItem<Student> item)
    {

    }

}

class GenericItem<T>
{
}

class Student
{
}

class GradStudent extends Student
{
}

```

The above program is wrong; it doesn't compile, let alone run. The problem with this program is that the statement

```
processItem(grad);
```

is erroneous. Even though GradStudent is a subtype of Student, GenericItem<GradStudent> is *not* a subtype of GenericItem<Student>. That is, the object grad is not an instance of GenericItem<Student>.

The solution to the above problem is to use what is known as a "bounded generic type," or a generic type that can be specified as a subtype of another type. We illustrate this concept of bounded type in the following (correct) program which is a modification to the wrong program.

```

// Bounded type

public class BoundedTypeCorrect
{
    public static void main(String[] args)
    {
        GenericItem<Student> stu = new GenericItem<Student>();

        GenericItem<GradStudent> grad = new GenericItem<GradStudent>();

        processItem(stu);
        processItem(grad);
    }

    public static <E extends Student> void processItem(GenericItem<E> item)
    {

```

```

    }
}

class GenericItem<T>
{
}

class Student
{
}

class GradStudent extends Student
{
}

```

In the above code, `<E extends Student>` represents any subtype of `Student` (including, of course, the `Student` type itself). This makes `GenericItem<GradStudent>` a valid type for use as the argument in method `processItem`, and therefore the statement

```
processItem(grad);
```

is now okay. `<E extends Student>` is an example of a bounded generic type. Note that the generic type `<E>` is equivalent to `<E extends Object>`.

## Wildcard Generic Types

Wildcard generic types, indicated with a question mark (?), are often useful. Three types of wildcard generic types exist:

### 1. Upper-bounded wild card: **Class<? extends T>**

`<? extends T>` is a wildcard type that represents `T` or its subtypes.

### 2. Lower-bounded wild card: **Class<? super T>**

`<? super T>` is a wildcard type that represents `T` or its supertypes.

### 3. Unbounded wild card: **Class<?>**

`<?>` is the same as `<? extends Object>`, a wildcard type that represents any type.

Study the following example. You should be able to realize that this program will not work because of the type-subtype issue discussed earlier.

```

// Wrong program
// Upper-bounded wildcard

import java.util.*;

```

```

public class WildcardWrong
{
    public static void main(String[] args)
    {
        GenStack<Integer> s = new GenStack<Integer>();
        s.push(2);
        s.push(7);
        System.out.println(maxInStack(s)); // wrong
    }

    public static double maxInStack(GenStack<Number> inputStack)
    {
        double max = inputStack.pop().doubleValue();
        while (!inputStack.isEmpty())
        {
            double value = inputStack.pop().doubleValue();
            max = value > max ? value : max;
        }
        return max;
    }
}

class GenStack<T>
{
    private ArrayList<T> a;

    public GenStack() { a = new ArrayList<T>(); }
    public void push(T t) { a.add(t); }
    public T pop() { return isEmpty() ? null : a.remove(a.size() - 1); }
    public boolean isEmpty() { return a.size() == 0; }
}

```

Note the following issues with types in generics in the above program:

- Integer is a subtype of Number, but Stack<Integer> is not a subtype of Stack<Number>
- Stack<Integer> can be used where Stack<? extends Number> is expected

A correct version of the same task follows.

```

// Upper-bounded wildcard: correct version

import java.util.*;

public class WildcardCorrect
{
    public static void main(String[] args)
    {
        GenStack<Integer> s = new GenStack<Integer>();
        s.push(2);
    }
}

```

```

        s.push(7);
        s.push(5);
        System.out.println(maxInStack(s));
    }

    public static double maxInStack(GenStack<? extends Number> inputStack)
    {
        double max = inputStack.pop().doubleValue();
        while (!inputStack.isEmpty())
        {
            double value = inputStack.pop().doubleValue();
            max = value > max ? value : max;
        }
        return max;
    }
}

class GenStack<T>
{
    private ArrayList<T> a;

    public GenStack() { a = new ArrayList<T>(); }
    public void push(T t) { a.add(t); }
    public T pop() { return isEmpty() ? null : a.remove(a.size() - 1); }
    public boolean isEmpty() { return a.size() == 0; }
}

```

An alternative version of the above program is given below.

```

// Upper-bounded wildcard: alternative version

import java.util.*;

public class WildcardCorrect2
{
    public static void main(String[] args)
    {
        GenStack<Integer> s = new GenStack<Integer>();
        s.push(2);
        s.push(7);
        s.push(5);
        System.out.println(maxInStack(s));
    }

    public static <E extends Number> double maxInStack(GenStack<E> inputStack)
    {
        double max = inputStack.pop().doubleValue();
        while (!inputStack.isEmpty())

```

```

        {
            double value = inputStack.pop().doubleValue();
            max = value > max ? value : max;
        }
        return max;
    }
}

class GenStack<T>
{
    private ArrayList<T> a;

    public GenStack() { a = new ArrayList<T>(); }
    public void push(T t) { a.add(t); }
    public T pop() { return isEmpty() ? null : a.remove(a.size() - 1); }
    public boolean isEmpty() { return a.size() == 0; }
}

```

To see the advantage of the style

```
public static <E extends Number> double maxInStack(GenStack<E> inputStack)
```

over the one with a ? in the header,

```
public static double maxInStack(GenStack<? extends Number> inputStack)
```

consider the following example where instead of a stack an arraylist is used.

```

// Upper-bounded wildcard
import java.util.*;

public class WildcardCorrect3
{
    public static void main(String[] args)
    {
        ArrayList<Integer> arr = new ArrayList<Integer>();
        arr.add(2);
        arr.add(7);
        arr.add(5);
        System.out.println(maxInArrayList(arr));
    }

    public static <E extends Number> double maxInArrayList(ArrayList<E> inputlist)
    {
        double max = inputlist.get(0).doubleValue();
        for (E elem: inputlist)
        {
            double val = elem.doubleValue();
            max = val > max ? val : max;
        }
        return max;
    }
}

```



The advantage of the above style is that the generic type E can be used in the body of the method, as in the statement

```
for (E elem: inputlist)
```

Note that "? elem: inputlist" is invalid.

## Unbounded Wildcard

The following program demonstrates the effect of misusing an unbounded wildcard.

```
// Unbounded wildcard

import java.util.*;

public class UnboundedWildcard
{
    public static void printListObject(ArrayList<Object> list)
    {
        for (Object elem : list)
            System.out.println(elem);
    }

    public static void printListWild(ArrayList<?> list)
    {
        for (Object elem: list)
            System.out.println(elem);
    }

    public static void main(String[] args)
    {
        ArrayList<Integer> intlist = new ArrayList<Integer>();
        intlist.add(1); intlist.add(2); intlist.add(3);

        printListWild(intlist); // ok
        printListObject(intlist); // error
    }
}
```

This program doesn't compile because `ArrayList<Integer>` is not a subtype of `ArrayList<Object>`. However, `ArrayList<Integer>` can be used where `ArrayList<?>` is expected.

Let us reiterate the salient points:

- `<?>` is a wildcard that represents any object type.
- `ArrayList<?>` is equivalent to `ArrayList<? extends Object>`
- `ArrayList<?>` is NOT the same as `ArrayList<Object>`
- `printListObject(ArrayList<Object>)` cannot print `ArrayList<Integer>`, `ArrayList<String>`, `ArrayList<Double>`, because they are not subtypes of `ArrayList<Object>`.

# Lowerbounded Wildcard

Study the following example. The code works fine, because `<? super Integer>` means `Integer` or its supertype.

```
// Lower-bounded wildcard

import java.util.*;

public class LowerboundedWildcard
{
    public static void addNumbers(ArrayList<? super Integer> list)
        // Integer or its supertype
    {
        for (int i = 1; i <= 10; i++)
            list.add(i);
    }

    public static void main(String[] args)
    {
        ArrayList<Integer> ai = new ArrayList<Integer>();
        ArrayList<Number> an = new ArrayList<Number>();

        addNumbers(ai);
        addNumbers(an);
    }
}
```

## Type Erasure

---

While the use of generics is good for compile-time detection of errors, it is important to understand that generic information is not available after compilation. This ensures backward compatibility with legacy code that uses raw types.

### Example #1

Code with generics:

```
ArrayList<String> list = new ArrayList<String>();
list.add("Boston");
String city = list.get(0);
```

Equivalent raw code after compilation:

```
ArrayList list = new ArrayList();
```

```
list.add("Boston");  
String city = (String) list.get(0);
```

## Example #2

```
public static <E> void print(E[] list)  
{  
    for (E elem: list)  
        System.out.println(elem);  
}
```

Equivalent raw code after compilation:

```
public static void print(Object[] list)  
{  
    for (Object elem: list)  
        System.out.println(elem);  
}
```

## Example #3

```
public static <E extends Stu> boolean checkStudents(E s1, E s2)  
{  
    return s1.getScore() == s2.getScore();  
}
```

Equivalent code after type erasure:

```
public static boolean checkStudents(Stu s1, Stu s2)  
{  
    return s1.getScore() == s2.getScore();  
}
```

# Limitations of Generics

---

Some important limitations:

- No instanceof generic-type
- No new E()
- No new E[]
- No static field of generic type of the same class
- No static method of generic type of the same class

## Example #1

A generic class is shared by all its instances regardless of their actual concrete types.

Given

```
ArrayList<String> list1 = new ArrayList<String>();
ArrayList<String> list2 = new ArrayList<String>();
list1 instanceof ArrayList    is true,
list2 instanceof ArrayList    is true, but

list1 instanceof ArrayList<String>
```

does not compile. At run-time, there is no `ArrayList<String>` class.

## Example #2

- `E object = new E();` is wrong because `new E()` is executed at run-time
- `E[] myarray = new E[10];` is wrong

## Example #3

A generic type not allowed in a static field of a generic class because there is only one memory location of static field for class Alpha:

```
class Alpha<E>
{
    private E field;
    private static E staticField;    // wrong

    private static int count = 0;    //ok
}
```

In the above class, the (non-generic) static field `count` is OK, but `Alpha<String>`, `Alpha<Double>`, etc. all use the same `count`.

## Example #4

Observe what happens to the (non-generic) static field count of class `Alpha<E>` in the program below:

```
// Limitations of generics

public class Limitation1
{
    public static void main(String[] args)
    {
        String s = "hi";
```

```

        Alpha<String> a = new Alpha<String>(s);
        Alpha<Number> b = new Alpha<Number>(172);
        System.out.println("b.getCount() = " + b.getCount());           // count is 2
        System.out.println("Alpha.getCount() = " + Alpha.getCount());    // count is 2
    }
}

class Alpha<E>
{
    private E field;
    private static int count = 0; // Only one memory location of static field
                                // for class Alpha

    public Alpha(E e)
    {
        field = e;
        count++;
    }

    public static int getCount() {return count;}
}

```

Output of the program – limitations of generics example

.

## Example #5

For further illustration of the issue of static fields and methods in a generic context, study the following example carefully; the comments in the code explain the issue.

```

// Limitations of generics

public class Limitation2
{
    public static void main(String[] args)
    {
        Alpha<String> a = new Alpha<String>();
    }
}

class Alpha<E>
{
    public static void illegalStaticMethod (E e)
    // Illegal; static method cannot involve generic type E
    {
        E obj = null; // Illegal; no generic type in a static context
    }
}

```

```

    }

    public static <T> void okStaticMethod1(T t)
    // OK; this method is generic on type T but NOT generic on type E
    {
        T obj = t;
        System.out.printf("%s\n", obj.toString());
    }

    public static <E> void okStaticMethod2 (E t)
    // OK; this method is NOT generic on the same type on which class Alpha is generic;
    // The E here and the E in Alpha<E> are different
    {
        E obj = t;
        System.out.printf("%s\n", obj.toString());
    }
}

```

## Generic Collections

---

The "Java Collections Framework" includes two types of containers:

- Collection objects (e.g., lists, stacks, queues, sets), created from the **Collection** interface,
- Map objects (for storing key-value pairs), created from the **Map** interface

This module discusses the following types of objects from the Java Collections Framework:

- ArrayList
- LinkedList
- Vector
- Stack
- Queue
- Set
- Map

The Collection interface is the root (main) interface for operations on a collection of objects.

The **Collections** class (note the plural; this is not to be confused with the Collection – singular - interface) and the **Arrays** class (note the plural again) provide useful static methods for working with collection objects.

The collections discussed in this module are all generic.

## ArrayList

Two classes, ArrayList and LinkedList, are used for creating lists or sequential collections of elements of the same (reference) type. The List interface extends the Collection interface, and both ArrayList and LinkedList implement the List interface. We repeat that ArrayList<E> and LinkedList<E> are classes, while List<E> is an interface.

An ArrayList stores elements in an array; the array grows in size dynamically. A LinkedList stores elements in a linked list of a dynamically changing size.

The following example demonstrates several important methods: add() for inserting an element, get() to retrieve the element at the given index, size() for retrieving the current size of the ArrayList.

```
// ArrayList

import java.util.*;

public class ArrayList1
{
    public static void main(String[] args)
    {
        String[] subjects = {"physics", "chemistry", "maths", "biology"};

        List<String> a = new ArrayList<String>();
        for (String sub: subjects)
            a.add(sub);

        for (String elem: a)
            System.out.print(elem + " ");
        System.out.println();

        for (int i = 0; i < a.size(); i++)
            System.out.print(a.get(i) + " ");
        System.out.println();

        Iterator<String> iter = a.iterator();
        while (iter.hasNext())
            if (iter.next().length() > 7)
                iter.remove();

        System.out.println("Remaining elements: ");
        for (String elem: a)
            System.out.print(elem + " ");
        System.out.println();
    }
}
```

Output of the program – ArrayList

.

The above code illustrates two different ways of printing the elements of the ArrayList: using a traditional for loop with explicit indexing of the elements, and using the enhanced-for (for-each) loop (with no explicit indexing).

The above example also demonstrates the use of an Iterator. An Iterator object is used to traverse the elements of a collection. Iterable is an interface that defines a method called iterator() that returns an Iterator object. Each collection is Iterable. For instance, in the above code, a.iterator() returns an Iterator object of type Iterator<String>.

Iterator's boolean method hasNext() returns true if there's at least one more element to be examined, false otherwise. The next() method returns the next element in the iteration; if no elements remain, the method throws NoSuchElementException. The remove() method deletes from the collection the last element that was returned by next(). Corresponding to a given call to next(), there can be exactly one remove() call; a second remove() corresponding to the same next() is an error.

## Removing Elements from an ArrayList

The following program is an attempt to selectively remove some elements from an ArrayList. It does not work. Why?

```
// Remove elements from ArrayList - program doesn't work

import java.util.*;

public class ArrayList2
{
    public static void main(String[] args)
    {

        List<Integer> a = new ArrayList<Integer>();

        a.add(-2); //autobox
        a.add(-5);
        a.add(-7);
        a.add(-3);
        a.add(2);

        for (int i = 0; i < a.size(); i++)
            if (a.get(i) < 0) // auto-unbox
                a.remove(i);

        for (Integer elem: a)
            System.out.print(elem + " ");
        System.out.println();
    }
}
```

The output of this program is:

```
-5, -3, 2
```

even though the negative numbers were supposed to be removed from the ArrayList.



The reason is that the index *i* varies from 0 to 4 (note that the call to `a.size()` returns 5), but after each removal takes place, the array size goes down by 1. A fix is provided by the following version, where the looping starts from the end of the list:

```
// Remove elements from ArrayList - correct version

import java.util.*;

public class ArrayList3
{
    public static void main(String[] args)
    {

        List<Integer> a = new ArrayList<Integer>();

        a.add(-2); //autobox
        a.add(-5);
        a.add(-7);
        a.add(-3);
        a.add(2);

        for (int i = a.size() - 1; i >= 0; i--)
            if (a.get(i) < 0) // auto-unbox
                a.remove(i);

        for (Integer elem: a)
            System.out.print(elem + " ");
        System.out.println();
    }
}
```

The output of the above program is 2, as expected. In this context, keep in mind that a for-each (enhanced-for) loop cannot be used to remove elements from a list; for that purpose, we have to use an iterator.

## LinkedList

While `ArrayList` is a resizable-array implementation of the `List` interface, `LinkedList` provides a linked list implementation. An `ArrayList` or `LinkedList` can have a no-argument constructor and can also be constructed from an existing collection object. For the constructors and other methods, see the documentation: [Class ArrayList<E>](#) and [Class LinkedList<E>](#).

The following example illustrates a `LinkedList<Double>` object and also a `ListIterator`. The `listIterator()` or `listIterator(startIndex)` method returns an instance of `ListIterator`. The `ListIterator` interface extends the `Iterator` interface and adds bidirectional traversal of the list.

```
// LinkedList: ListIterator

import java.util.*;
```

```

public class LinkedList1
{
    public static void main(String[] args)
    {
        Double[] numbers = {2.3, 4.9, 0.01, 5.5, 3.9, 1.5};

        List<Double> a = new LinkedList<Double>();
        for (Double num: numbers)
            a.add(num);

        for (Double elem: a)
            System.out.print(elem + " ");
        System.out.println();

        for (int i = 0; i < a.size(); i++)
            System.out.print(a.get(i) + " ");
        System.out.println();

        Iterator<Double> iter = a.iterator(); // forward only
        while (iter.hasNext())
            if (iter.next() > 3.0)
                iter.remove();

        System.out.println("Remaining elements: ");
        for (Double elem: a)
            System.out.print(elem + " ");
        System.out.println();

        System.out.print("Elements in reverse order: ");
        ListIterator<Double> it = a.listIterator(a.size()); // bi-directional
        while (it.hasPrevious())
            System.out.print(it.previous() + " ");
    }
}

```

In the above example, the `ListIterator<Double>` object named "it" is made to start at the end of the list; the `hasPrevious()` and `previous()` methods are counterparts to the `hasNext()` and `next()` methods discussed earlier. The output of the program is shown below.

Output of the program – LinkedList

.