Introduction:
Generics in Java are a powerful feature that allows you to create classes, interfaces, and methods that can work with different types while maintaining type safety. They enable us to write code that is more flexible, reusable, and less error-prone. Let's dive into the world of generics!

collection
Collections

Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

T - Type
E - Element
K - Key
N - Number
V - Value

Generic Classes:
Imagine you have a Box that can hold any type of object. With generics, you can create a Box class that can be used with different types.
Here's an example:

```
public class Box<T>
{
    private T content;

    public void setContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}
```

In the above code, the <T> inside the angle brackets is a placeholder for a specific type. It could be any type, such as String, Integer, or a custom class.

Let's create instances of the Box class with different types:

1. ```
   Box<String> stringBox = new Box<>();
   stringBox.setContent("Hello, Generics!");
   ```

2. ```
   Box<Integer> intBox = new Box<>();
   intBox.setContent(42);

   System.out.println(stringBox.getContent());
   System.out.println(intBox.getContent());
   ```

In the above code, we create a stringBox instance of type Box<String> and set its content to a string. Similarly, we create an intBox instance of type Box<Integer> and set its content to an integer. The type parameter ensures that only the specified type can be stored in the respective boxes.

Type-safety: We can hold only a single type of objects in generics. It doesn't allow to store other objects.

Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");          //0
list.add(5);                //1
list.add(“Hiii”);           //2
list.add(10);               //3
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

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

Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Syntax to use generic collection

1. ClassOrInterface<Type>

Example to use Generics in java

1. ArrayList<String>

Generic Methods:
Sometimes, you want to write a method that can work with different types. Generics allow you to do that. Here's an example of a generic method that swaps the positions of two elements in an array:

```
public class ArrayUtils {
    public static <T> void swap(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

In the above code, the <T> before the return type declares the type parameter for the method. It represents a placeholder for any type.

Let's use the swap method to swap two elements in an array of integers:

```
Integer[] numbers = {1, 2, 3, 4, 5};
ArrayUtils.swap(numbers, 0, 4);
System.out.println(Arrays.toString(numbers));
```

The output of the above code will be: [5, 2, 3, 4, 1].

The swap method works with any type of array, as long as the type supports the swapping operation.

Wildcards:
Wildcards in generics allow you to work with unknown types. There are two types of wildcards: ? (unbounded wildcard) and ? extends Type (bounded wildcard).

The unbounded wildcard (?) represents an unknown type. It can be used when you don't care about the specific type, but you still want to perform certain operations.

Here's an example of a method that prints the contents of a list using an unbounded wildcard:

```
public class ListUtils {
    public static void printList(List<?> list)
{
        for (Object element : list) {
            System.out.println(element);
        }
    }
}
```

In the above code, the printList method accepts a list of unknown type (List<?>) and prints its contents. The unbounded wildcard allows us to work with any type of list.

Let's use the printList method with different types of lists:

1. List<String> stringList = Arrays.asList("Hello", "Generics");

```
List<Integer> intList = Arrays.asList(1, 2, 3);
```

2. ```
   ListUtils.printList(stringList);
   ListUtils.printList(intList);
   ```

The above code will print the contents of both lists.

The bounded wildcard (? extends Type) represents an unknown type that is a subtype of the specified type. It allows you to work with a specific unknown type or its subclasses.

Here's an example of a method that calculates the sum of a list of numbers using a bounded wildcard:

```
public class MathUtils {
   public static double sum(List<? extends Number> numbers)
{
      double total = 0;
      for (Number number : numbers) {
         total += number.doubleValue();
      }
      return total;
   }
}
```

In the above code, the sum method accepts a list of numbers (List<? extends Number>) and calculates their sum. The bounded wildcard ensures that we can work with any type that extends Number.

Let's calculate the sum of a list of integers and a list of doubles:

```
List<Integer> intList = Arrays.asList(1, 2, 3);
List<Double> doubleList = Arrays.asList(1.5, 2.5, 3.5);

System.out.println("Sum of integers: " + MathUtils.sum(intList));
System.out.println("Sum of doubles: " + MathUtils.sum(doubleList));
```

The output of the above code will be:

```
Sum of integers: 6.0
Sum of doubles: 7.5
```

Example 1:

```
import java.util.*;
class Gnr1
{
        public static void main(String args[])
        {
                ArrayList<String> list=new ArrayList<String>();
                list.add("rahul");
                list.add("jai");
                list.add(32);               // this line will throw an error as the Generic type is String

                String s=list.get(1);//type casting is not required
                System.out.println("element is: "+s);
```
```

```
		Iterator<String> itr=list.iterator();
		while(itr.hasNext())
		{
			System.out.println(itr.next());
		}
	}
}
```

Real life examples of Generics in JAVA

Sample 1:

```
public class GenericsDemo
{

        public static void main(String[] args)
        {
      // Creating a generic box for storing any type of object
      Box<String> stringBox = new Box<>();
      stringBox.set("Hello, Generics!");

      // Creating another generic box for storing integers
      Box<Integer> intBox = new Box<>();
      intBox.set(42);

      // Retrieving and printing the values from the boxes
      String str = stringBox.get();
      System.out.println(str);

      int num = intBox.get();
      System.out.println(num);
   }

   // Generic class representing a box that can store any type of object
   public static class Box<T> {
      private T item;

      public void set(T item) {
         this.item = item;
      }

      public T get() {
         return item;
      }
   }
}
```

Sample 2:

```
public class GenericsDemo {

   public static void main(String[] args) {
      // Creating a generic list to store integers
      GenericList<Integer> integerList = new GenericList<>();
      integerList.add(10);
      integerList.add(20);
      integerList.add(30);
```

```
        // Creating a generic list to store strings
        GenericList<String> stringList = new GenericList<>();
        stringList.add("Apple");
        stringList.add("Banana");
        stringList.add("Orange");

        // Printing the elements in the integer list
        System.out.println("Integer List:");
        for (int i = 0; i < integerList.size(); i++) {
            int number = integerList.get(i);
            System.out.println(number);
        }

        // Printing the elements in the string list
        System.out.println("\nString List:");
        for (int i = 0; i < stringList.size(); i++) {
            String fruit = stringList.get(i);
            System.out.println(fruit);
        }
    }

    // Generic class representing a list that can store elements of any type
    public static class GenericList<T> {
        private Object[] elements;
        private int size;

        public GenericList() {
            elements = new Object[10];
            size = 0;
        }

        public void add(T element) {
            if (size < elements.length) {
                elements[size] = element;
                size++;
            }
        }

        public T get(int index) {
            if (index >= 0 && index < size) {
                return (T) elements[index];
            }
            return null;
        }

        public int size() {
            return size;
        }
    }
}
```

Practice Purpose

1. Write a Java function that takes two sorted ArrayLists of integers, merges them into a single sorted ArrayList, and returns the result.

Sample Input:

ArrayList 1: [1, 3, 5, 7]

ArrayList 2: [2, 4, 6, 8]

Sample Output: 1w

Merged ArrayList: [1, 2, 3, 4, 5, 6, 7, 8]