**Generics**

Generics allow us to create classes, to design them, in a general way, without really worrying about the specific details of elements it might contain. Java Array’s list is an example of a generic class. You can apply Arrays list for any type of object, because many methods on that class are applied to any type.

Java supports generic types, such as classes, records, Interfaces. It also supports generic methods.

For example:

Regular class: class ITellYou {

Private String field;

}

Generic class: class YouTellMe <T> {

Private T field;

}

Another example of Generic class: ArrayList< String>listOfString; where ArrayList is the reference type, <String> is the parameter, listOfString is variable name.

**Generic Type Parameter**

We can have mnore than one parameter <T, S, U> are the conventions. A few letters are reserved for special use cases.

**E** for Element. Use extensively by Java collections framework

**K** for Key. Used for mapped types

**N** for number

**T** for type

**V** for value

**S**, **U**, **V** 2nd, 3rd and 4th types.

**You cannot use primitive data type with generics, use wrapper class instead**

**Comparable vs Comparator interfaces**

In Java, both Comparable and Comparator interfaces are used for sorting objects, but they serve different purposes and have distinct characteristics:

Comparable:

Purpose:

Defines the natural ordering of objects within a class.

Method:

compareTo(Object o): Compares the current object with another object, returning a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Implementation:

Implemented by the class itself, indicating that the class has a natural ordering.

Use case:

When you want to define a default sorting order for your class, which can be used by sorting algorithms like Collections.sort() or Arrays.sort().

Example:

class Person implements Comparable<Person> {

private String name;

private int age;

// Constructor, getters, setters

@Override

Public int compareTo(Person other) {

return this.age - other.age; // Sort by age

}

}

Comparator:

Purpose:

Provides a flexible way to define custom sorting logic for objects, independent of the class implementation.

Method:

compare(Object o1, Object o2): Compares two objects, returning a negative integer, zero, or a positive integer as the first object is less than, equal to, or greater than the second object.

Implementation:

Implemented in a separate class, providing the flexibility to define multiple sorting strategies for the same class.

Use case:

When you need to sort objects based on different criteria or when the class doesn't have a natural ordering.

Example:

class NameComparator implements Comparator<Person> {

@Override

public int compare(Person p1, Person p2) {

return p1.getName().compareTo(p2.getName()); // Sort by name

}

}

Key Differences:

Number of sorting orders:

Comparable provides a single natural ordering, while Comparator allows multiple sorting strategies.

Class modification:

Comparable requires modifying the class itself, while Comparator can be used without modifying the class.

Flexibility:

Comparator is more flexible, allowing for custom sorting logic, while Comparable defines a single default order.

Choosing the right interface:

Use Comparable when your class has a natural ordering that makes sense for most use cases.

Use Comparator when you need multiple sorting strategies or when the class doesn't have a natural ordering.