**Generics:**

Java Generics deals with type-safe objects.

Before Generics, we can store any type of collection i.e., non-generic. Now generics, forces the programmer to store specific type of objects.

**Advantages of Generics:**

1. **Type-safety:** We can hold only a single type of objects in Generics.
2. **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(“Sravan”);

String s = (String) list.get(0); //Typecasting to String

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

List<String> list = new ArrayList<String>();

list.add(“Sravan”);

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(“Sravan”);

list.add(32); //throws compile time error

**Generic Class:**

A class that can refer to any type is known as generic class. Here, we are using T type parameter to create generic class of specific type.

**Type Parameters:**

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

1. T – Type
2. E – Element
3. K – Key
4. N – Number
5. V – Value

**Generic Method:**

Like Generic class, we can create a generic method which can accept any type of argument.

**Wildcard in Java Generics:**

The ? (question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number class e.g. Integer, Float, double etc., Now we can call the method of Number class through any child class object.

List<? super Numer>

**The extends Wildcard boundary:**

List<? extends A> means a List of objects that are instances of the Class A, or subclasses of A.

**The super Wildcard boundary:**

List<? super A> means that the list is typed to either Class A , or a superclass of A.

**Java Custom Annotations:**

Java Annotations provide information about the code and they have no direct effect on the code they annotate.

Java Custom Annotations or Java user-defined annotations are easy to create and use. The @interface element is used to declare an annotation.

For Example:

@interface MyAnnotation{}

Here, MyAnnotation is a custom name.

**Points to remember:**

* Method should not have any throws clauses
* Method should return one of the following: primitive data types, String, Class, enum or array of these datatypes.
* Method should not have any parameter.
* We should attach @ just before interface keyword to define annotation.
* It may assign a default value to the method.

**Types of Annotations:**

1. **Marker Annotation:**

An annotation that has no method, is called marker annotation. For example:

@interface MyAnnotation{}

The @Override and @Deprecated are marker annotation.

1. **Single-Value Annotation:**

An annotation that has one method, is called single-value annotation. For Example:

|  |
| --- |
| **@interface** MyAnnotation{  **int** value();  } |

We can provide default values also.

|  |
| --- |
| **@interface** MyAnnotation{  **int** value() **default** 0;  } |

**How to apply Single-Value Annotation?**

Let’s see the code to apply the single value annotation

@MyAnnotation(value=10)

1. **Multi-Value Annotation:**

An annotation that has more than one method, is called Multi-Value annotation. For example:

|  |
| --- |
| **@interface** MyAnnotation{  **int** value1();  String value2();  String value3();  }  } |

We can provide default values also. For example:

|  |
| --- |
| **@interface** MyAnnotation{  **int** value1() **default** 1;  String value2() **default** "";  String value3() **default** "xyz";  } |

**How to apply Multi-Value Annotation?**

Let’s see the code to apply multi-value annotation.

|  |
| --- |
| 1. @MyAnnotation(value1=10,value2="Arun Kumar",value3="Ghaziabad") |