Generics: It refers to a way of writing code that allowed types (Strings, integers, objects.) to be used as parameters while defining classes, interfaces or methods. We use generics to create classes, interfaces and methods that will work in a type safe manner with various kinds of data.

We can also say that the term generics means parameterized types. Parameterized types enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Here, T is the name of type parameter, which can be used As a placeholder for the actual type That will be passed to Gen class when an object is created.

```
// A simple generic class.
// Here, T is a type parameter that
// will be replaced by a real type
// when an object of type Gen is create
class Gen<T> {
  T ob; // declare an object of type T

  // Pass the constructor a reference t
  // an object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getOb() {
    return ob;
  }

  // Show type of T.
```

Gen uses a type parameter, Gen is a generic class, which is also called a parameterized type. In the declaration of generic class there is no Specific significance to the name T, any valid identifier could have been used, but is traditional.

The most commonly used type parameter identifiers are:

- E for Element (used extensively by the Java Collections Framework).
- K for Key (used for mapped types).
- N for Number.
- T for Type.
- V for Value.
- S, U, V etc. for 2nd, 3rd, 4th types.

Now consider Gen's constructor: Notice that its parameter, o, is of type T. This means that the actual type of o is determined by the type passed to T when a Gen object is created. Also, because both the parameter o and the member variable ob are of type T, they will both be of the same actual type when a Gen object is created.

When you use generic classes, either referencing them or instantiating them, it's definitely recommended that you include a type parameter. But you can still use them without specifying one. This is called the Raw Use of the reference type.

- Generics allow the compiler to do compile-time type checking, when adding and processing elements in the list.
- Generics simplify code, because we don't have to do our own type checking and casting, as we would, if the type of our elements was Object.

Note: **Generics only work with reference type** (String, Integer, Boolean, Double). With generic it is possible to pass any class type to T, but you cannot pass a primitive type (int, double, float, char) to a type parameter.

```
Gen<int> intOb = new Gen<int>(53); // Error,
```

```
// Create a Gen object for Strings.
Gen<String> strOb = new Gen<String> ("Generic

// Show the type of data used by strOb.
strOb.showType();

// Get the value of strOb. Again, notice
// that no cast is needed.
String str = strOb.getOb();
System.out.println("value: " + str);
```

```
class GenDemo {
  public static void main(String[] args) {
    // Create a Gen reference for Integers.
    Gen<Integer> iOb;

    // Create a Gen<Integer> object and assign its
    // reference to iOb. Notice the use of autoboxing
    // to encapsulate the value 88 within an Integer object
    iOb = new Gen<Integer>(88);

    // Show the type of data used by iOb.
    iOb.showType();

    // Get the value in iOb. Notice that
    // no cast is needed.
    int v = iOb.getOb();
    System.out.println("value: " + v);

    System.out.println();
```

Generic types differ based on their type of argument:

```
iOb = strOb; // Wrong!
```

Even though both iOb and strOb are of type Gen<T>, they are references to different types because their type arguments differ. This is part of the way that generics add type safety and prevent errors.

How generics improve type safety.?

- With generics, type information is checked at compile time. This means that if you try to use a data type that is not compatible with the one expected by the generic code, the compiler will catch it and generate an error. This prevents type-related errors from occurring at runtime, which can be harder to debug.
- By using generics, we eliminate the need for manual typecasting, which simplifies the code and reduces the more chances of errors. You can simply specify the data type when creating an instance of the generic class, and the compiler ensures that only compatible types are used. If the parameterized type is not used for the class Gen, the return type of getOb( ) is Object, the cast to
- Integer is necessary to enable that value to be auto-unboxed and stored in v. If you remove the cast, the program will not compile.

```
// This compiles, but is conceptually wrong!
iOb = strOb;
v = (Integer) iOb.getOb(); // run-time error
```

```
int v = (Integer) iOb.getOb();
```

Generic classes with two type parameters: You can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list. so we could do T1, T2, T3. But again t instead of using type parameters like this, it's easier to read the code with alternate letter selections. And these are usually S, U, and V, in that order. If we had three types, we'd probably want to declare this class as shown here, with T, S, and U.

```
public class Team<T1, T2, T3> {
public class Team<T, S, U> {
```

<*T* extends *superclass*>

Bounded Types: When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived.
We can achieve this by using extends clause when specifying the type parameter.

```java
class Box<T extends Number> {
    // T must be a subtype of Number
}
```

```java
class Box<T super Integer> {
    // T must be a supertype of Integer
}
```

- Here T extends superclass: This specifies that T can only be replaced by superclass, or subclasses of superclass. Thus, superclass defines an inclusive, upper limit (In this example, T can be any class that extends Number, such as Integer, Double, or BigDecimal.)
- Lower Bound: T can be Number, Object, or any other class that Integer extends.
- In addition to using a class type as a bond, you can also use an interface type. In fact, you can specify multiple interfaces as bounds.
- A bound can include both class type and one or more interfaces.
    - In this case class type must be first.
    - When specifying a bound that has class and an interface or multiple interfaces, use the & operator to connect them. This creates an intersection type.

```
class Gen<T extends MyClass & MyInterface> { // ...
```

Using wild card arguments: Denoted by ? extends Type, where Type is a specific type. This wildcard allows any type that is a subtype of Type or Type itself.

```java
public void process(List<? extends Number> list) {
    // Process list of any type that is a subtype of Number
}
```

Generic Exception Restriction: A generic class cannot extend Throwable. This means that you cannot create generic exception classes.