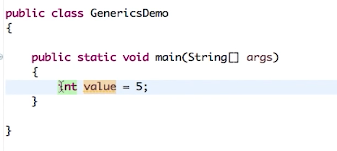
**Generics**

One of the best things of java is type safety. Means that, whenever we want to work with any variable that should be declared first.

While declaring the variable, it would expect the type of value that we are going to store.

Ex:



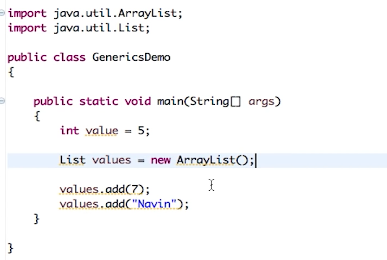
So here we are declaring variable “value” and assigning an integer value. This is how we are achieving type safety.

Here the type of the value is known at the compile time itself, and we don’t wait till run time to pick the value and assign it to the variable.

When it comes to collections, we can add any items (objects) without providing the type.

Note: Collection is list of objects and not primitive types. (Object is the base class of all the other classes in java)

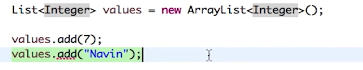
Ex:



Here in the list, we are adding different type of objects not provided any type for this.

So we are not achieving type safety here.

Solution: Using generics, we can achieve the type safety.

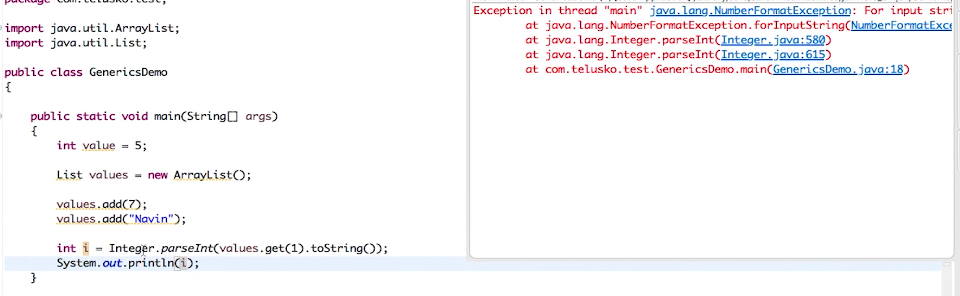


When we specify that we are going to work with Integer, at compile time itself we will get to know that strings can’t be added to the list.

So without generics, we can’t achieve type safety while working with collections.

While working with collections, the intention is not just to add the elements but we want to retrieve the values. We might face the issues at run time, if we don’t use concept of generics.

Ex:



In the above example, we can add any objects to the list, but while retrieving we are expecting the integer value at index 1 and unfortunately we have added string which results in “**NumberFormatException**” at run time.

To avoid this kind of issues, we go for generics.

\*\* Handling the errors at compile time is way better than handling at run time.

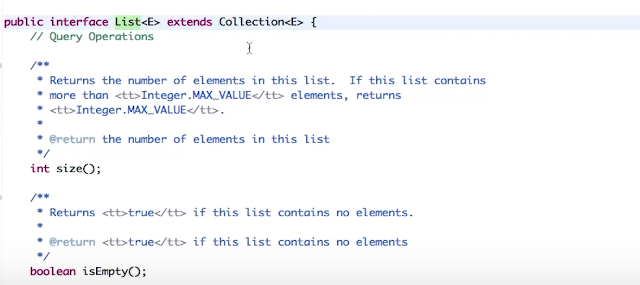
**How generics are implemented at the backend?**

For an example, if we declare a list and check the implementation of it, it looks like below.

Declaration:



**Internal implementation:**

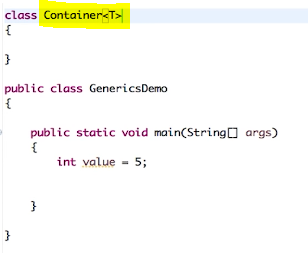


List is an interface, which is having the angular brackets with E inside it. E is nothing but the element representation.

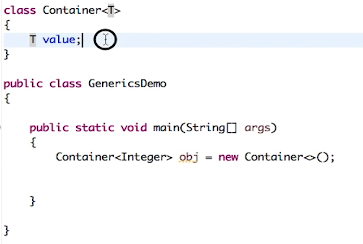
This is how we define the generic.

{Classname}/ {interfacename} <generictype>

Creating our own generic.



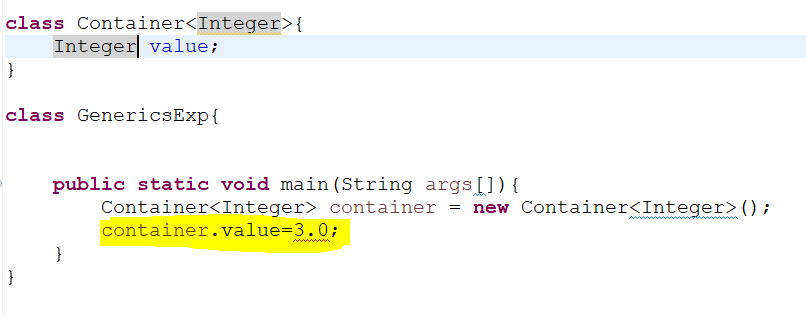
Here we have created a generic with type ‘T’ (T represents type here). So this container will work with this type ‘T’.



Here we are declaring a variable inside Container, which is of type ‘T’. So when we create an object of type Integer, value becomes integer, and whatever type that we create the object with, value becomes that type. It’s generic now, and works with all the types.

Defining value as Integer will work with only Integers and not with any other classes.

Ex:



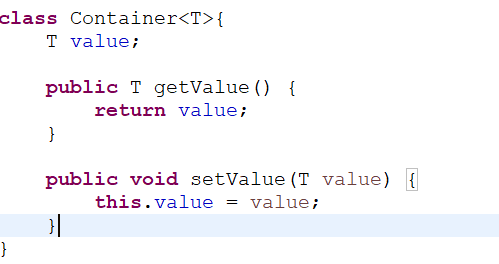
Here when we try to set the value as 3.0, it is showing error at compile time itself. To avoid this we can go with generic type ‘T’.

So that T is replaced with the type that we create the object with.

Even when we work with getters and setters, the methods return type will also be T.

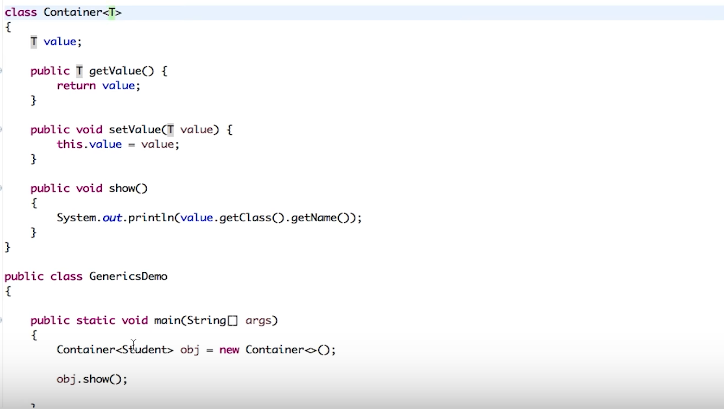
Ex:

Select the “T value” -> Sources -> generate getters and setters -> Output looks like below.



Note: Working with generics, it only supports classes. We can’t use primitive data types.

We can even have our custom class, like shown below. Student class is mentioned here. Which is still acceptable. In this case, T will be replaced with Student class.



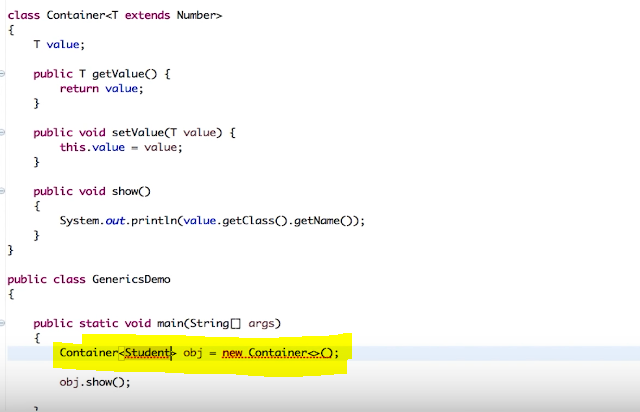
**Restricting the generics to specific classes**

Integer, Float, Double will extend Number class.

If we want to restrict the generic to support only these classes, we should do like below.

Providing Number, Integer, Double and Float will work now but not the Student class, as we are specifying that T extends Number. So only those classes which extends Number will be supported.

Even Object will not work here in this case.

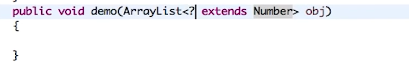


One more thing here is using ‘?’

Lets say we have declared a method, which would accept arraylist of type Integer (this is within container generic).

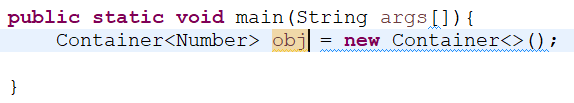


It would accepts only Integer type. But if we want it to support other Number extended classes, then we should do the following.



We can even replace this Number with T. So the T will be replaced with whatever type that we provide while creating an object. And ‘?’ is replaced with the type that is passed for ArrayList.

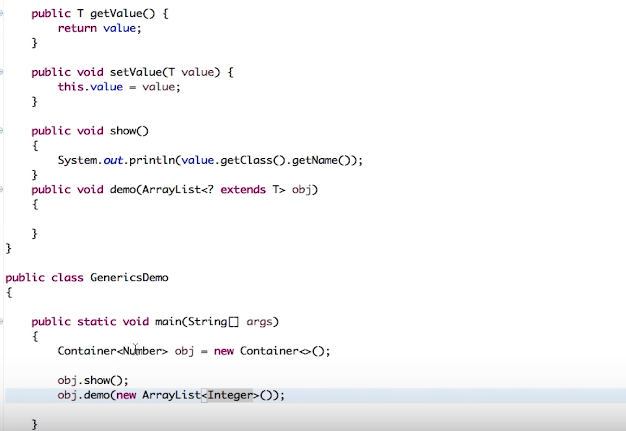
In the below example,



We are creating container of type Number, so T will be replaced with Number, and

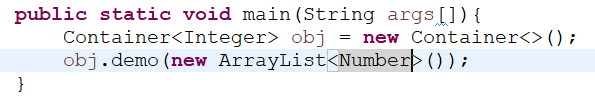


Here ‘?’ will be replaced with Integer.



Use of ‘super’





Here T will be replaced with Integer and ‘?’ is replaced with Number.

So the final syntax would be Number super Integer, which is valid. Number is the super type of Integer.

But ‘Integer super Number’ is not valid and show an error at run time itself.

