Hello, Ubaldo Acosta greets you again. I hope you're ready to start with this lesson.

We are going to study the topic of Generics in Java.

Are you ready? Come on!

JAVA GENERICS

Definition of a generic class:

```
// We define a generic class with the diamond operator <>
public class GenericClass<T> {

    //We define a variable of generic type
    T object;

    // Constructor that initializes the type to be used
    public GenericClass(T object) {
        this.object = object;
    }

    public void getType() {
        System.out.println("The type T is: " + object.getClass().getName());
    }
}
```

Use of a generic type or class:

```
    public static void main(String[] args) {
        //We create an instance of Generic Class for Integer.
        GenericClass<Integer> intObject = new GenericClass<Integer>(15);
        intObject.getType();
    }
```

We will study in this lesson the generic types in Java. Generic or generic types were introduced in version 1.5 of Java SE, and it was one of the biggest changes for this version.

Previously we had to know exactly the type of data that we were going to use to be able to pass parameters to a function, for example, or to instantiate a class, however with the help of generics we can leave the type of data pending until instantiation of some kind generic, or the passage of a generic parameter.

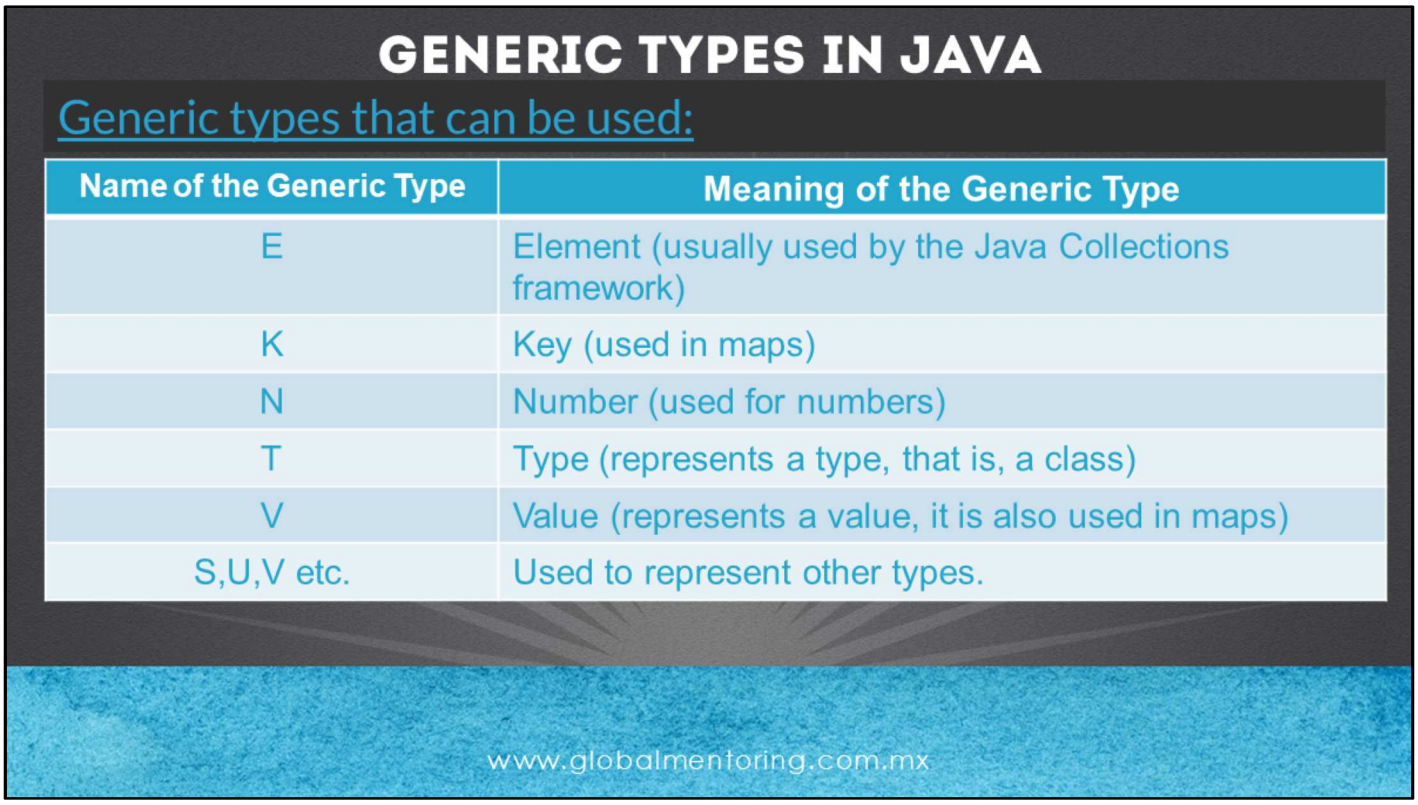
We can observe in the code shown a class that defines a generic type with the use of the diamond operator <> in the definition of the class. This means that the type will know until the moment this class is used, and therefore is known as a generic type.

The T basically means a generic type that will be replaced when using this class, so the T is simply the name we give to the generic type we are going to use. There are several names that we can use as we will see in the next sheet.

Later in the class we define an attribute of the same type T, which we will initialize through the constructor of this class, and we can see that when we use this class in the main method shown, at the moment of creating the object of this class, we specify The diamond operator means the type of data that we want to use.

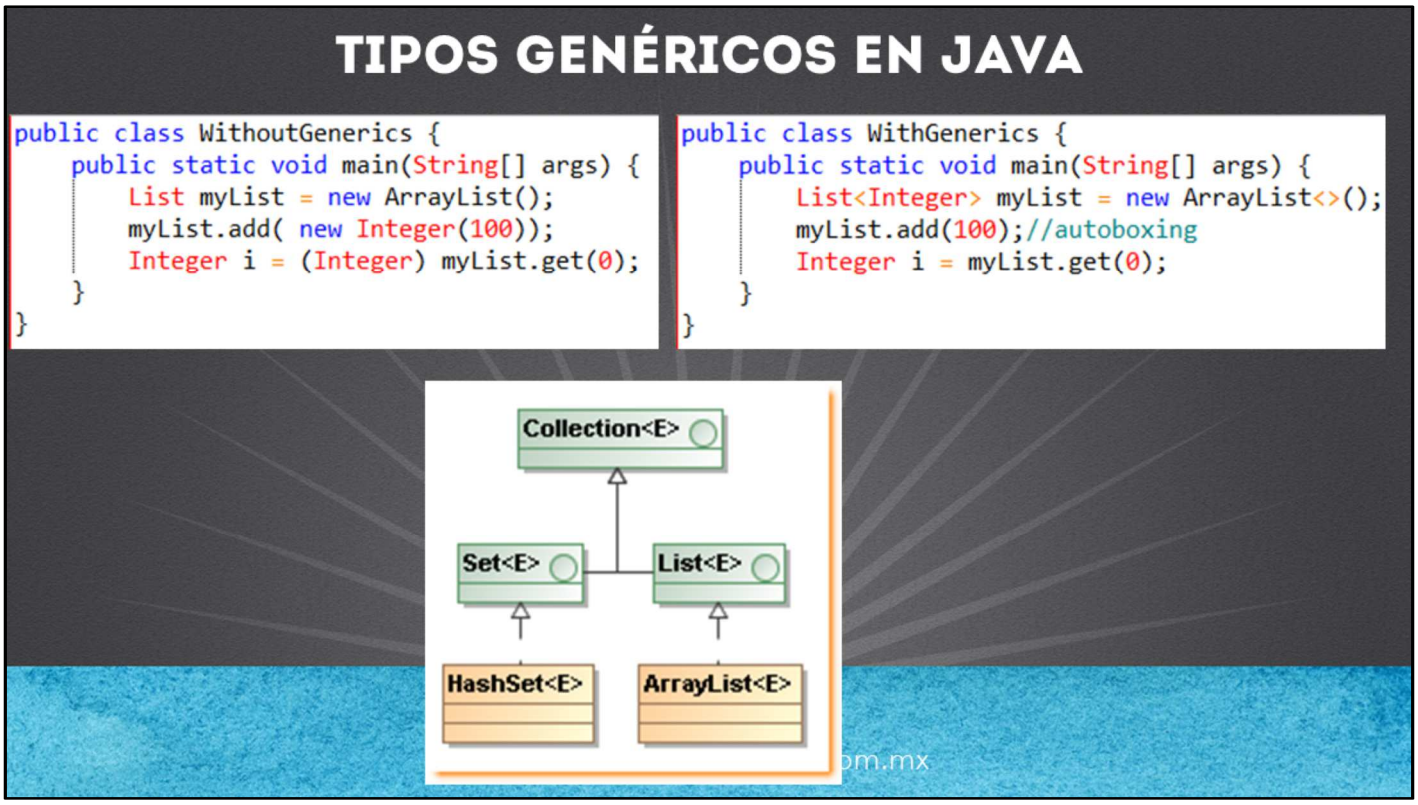
And here is where the use of generic classes is very useful, since it allows us to define the type of data we want to use up to the moment of instantiating an object of our class, and thus define increasingly generic functionality worth the redundancy, made by using interfaces, abstract classes and polymorphism we have already created several generic methods since they can be used by various types of data, however, this concept of Generics goes a step further, and allows us to define the type to use and create methods even more generic than we have created.

Finally, the getType () method allows us to send the type of data we have defined for the generic class to console, and just as we used the <Integer> type to instantiate our generic class, we can use any type of data, however we can not use primitive types, but only Object types.



When using and creating generic types, we can use several names, such as the T that we have already used. However, there are some conventions that we can follow when creating our generic classes and thus use good practices when creating them. Obviously, it is also useful when using these generic types in classes already created within the Java API, since we will know with greater certainty what each class and generic type that we are using serves.

Next we will see how to apply the generic types to the Java Collections framework.



The topic of generics was mostly used to change several of the Java APIs, but in particular the Framework Collections API was the most benefited, since as we can see, previously it was necessary to define a list type, however there was no certainty that it will store a single type and thus avoid problems of conversion of objects when extracting its elements. Therefore previously the lists stored an object type, and at the time of extracting the object from the list the compiler had an object, and therefore it was necessary to do a casting of the extracted object, as observed in the code of the SinGenerics class.

However, with the generic collections, we can declare a list of a specific type, and add to this list only compatible types or subtypes according to the downcasting rules that we mentioned in the topic of object conversion.

As we can see in the ConGenerics example, we observe that to define the list we are using the diamond operator and we are specifying that the objects that we can add to this list are exclusively Integer type or subclasses of the Integer type, in this way when we extract the values from this list we are sure that they will be of a type compatible with the Integer type, and therefore it is not necessary to do a casting or conversion of the extracted object.

Therefore, the generic collections offer the assurance that at compile time the data to be stored is only of the specified type, eliminating the need to make a casting (conversion of types). We just have to remember that version 1.5 or higher of the JDK is needed.

In the figure we can see how the definition of the APIs and classes of the Coleccions API was changed, and now generic types are used in its definition. Here are some examples of the use of generic collections.