Creating Generic Array in Java

studytonight.com/java-examples/creating-generic-array-in-java

An array is used to store a collection of similar types of data. Creating a generic array in Java could be a little complex. Arrays use the type information of their elements at runtime for memory allocation. But in the case of generics, this information is not available due to type erasure.

Generics check for type information at the compile time, and no information is available at runtime.

For arrays, the type information is checked at runtime because they are covariant. It means that we can assign a child-type array to a parent type(like String[] to Object[]).

```
String[] strArr = new String[5];
Object[] objArr = strArr; //No Error
```

The above code will not return any compile-time error. We can also add different objects to the above array without worrying about any compilation errors. But type-checking for all elements takes place at runtime for arrays. If any discrepancies are found at runtime, then we will get an `ArrayStoreException`.

```
public class Demo
{
        public static void main(String[] args)
        {
                String[] strArr = new String[5];
                Object[] objArr = strArr; //No Error
                objArr[0] = new Object();
                objArr[1] = new Exception();
        }
}
```

Exception in thread "main" java.lang.ArrayStoreException: java.lang.Object at Demo.main(Demo.java:7)

It is recommended to use Collection Frameworks like the List instead of generic arrays. However, we can create a generic array if we have some information about the type at compile time. Let's learn how to create a generic array in Java.

Creating a Generic Array Using Object Array

We can create a generic class with an Object array to mimic a generic array. We will use a `get()` method and a `set()` method in this class. The get() method will use an explicit cast.

```
import java.util.Arrays;
class GenericArray<T>
{
        private Object[] genericArr;

        GenericArray(int size)
        {
                genericArr = new Object[size];
        }
        public T get(int index)
        {
                return (T) genericArr[index];
        }
        public void set(int index, T element)
        {
                genericArr[index] = element;
        }
        @Override
    public String toString()
        {
        return Arrays.toString(genericArr);
    }
}
public class Demo
{
        public static void main(String[] args)
        {
                GenericArray<String> strArr = new GenericArray(3);
                strArr.set(0, "five");
                strArr.set(1, "ten");
                strArr.set(2, "twenty");
                GenericArray<Integer> intArr = new GenericArray(3);
                intArr.set(0, 5);
                intArr.set(1, 10);
                intArr.set(2, 20);
                GenericArray<Double> doubleArr = new GenericArray(3);
                doubleArr.set(0, 5.0);
                doubleArr.set(1, 10.0);
                doubleArr.set(2, 20.0);
                System.out.println("Integer Array: " + intArr);
                System.out.println("String Array: " + strArr);
                System.out.println("Double Array: " + doubleArr);
        }
}
```

Integer Array: [5, 10, 20] String Array: [five, ten, twenty] Double Array: [5.0, 10.0, 20.0]

The above approach works fine if we only use the get() and set() methods and do not provide users with direct access to the object array. A thing to note here is that even the `ArrayList` uses an object array to store elements. ArrayLists do not give direct access to this object array.

Creating a Generic Array Using Reflection

This approach is similar to the one discussed in the previous section. The only difference is that the constructor will take the type information, and then we will use the `Array.newInstance()` method to initialize our array. The complete implementation is shown below.

```
import java.lang.reflect.Array;
import java.util.Arrays;
class GenericArray<T>
{
        private T[] genericArr;

        GenericArray(Class<T> classType, int size)
        {
                genericArr = (T[]) Array.newInstance(classType, size);
        }
        public T get(int index)
        {
                return genericArr[index];
        }
        public void set(int index, T element)
        {
                genericArr[index] = element;
        }
        @Override
    public String toString()
        {
        return Arrays.toString(genericArr);
    }
}
public class Demo
{
        public static void main(String[] args)
        {
                GenericArray<String> strArr = new GenericArray(String.class, 3);
                strArr.set(0, "five");
                strArr.set(1, "ten");
                strArr.set(2, "twenty");

                GenericArray<Integer> intArr = new GenericArray(Integer.class, 3);
                intArr.set(0, 5);
                intArr.set(1, 10);
                intArr.set(2, 20);

                GenericArray<Double> doubleArr = new GenericArray(Double.class, 3);
                doubleArr.set(0, 5.0);
                doubleArr.set(1, 10.0);
                doubleArr.set(2, 20.0);

                System.out.println("Integer Array: " + intArr);
                System.out.println("String Array: " + strArr);
                System.out.println("Double Array: " + doubleArr);
        }
}
```

Integer Array: [5, 10, 20] String Array: [five, ten, twenty] Double Array: [5.0, 10.0, 20.0]

Summary

We can implement generic arrays in Java with the help of generic classes. We can use an Object array and appropriate methods for the user to interact with this generic array. The concept of an object array is also used by the ArrayList class. We can also use the Array.newInstance() method and make the user pass the type information to the constructor. A similar approach is used by generic methods of some Collections like the LinkedList. Overall, it is recommended to use ArrayLists or LinkedList in place of creating our own generic lists.