An Approach to Generics & Collections

Nethravathi H[1], Megharaja D S[2], Mangesh S Pai[3], Syed Asim K[4], Rakshitha H J[5], Priyanka G M[6], Nikhitha S[7]

[1] *Assistant Professor & Head of Department, DVS College of Arts and Science, Shimoga (KAR)*

[2,3,4,5,6,7] *Assistant Professor, DVS College of Arts and Science, Shimoga (KAR)*

Abstract **- The Java Generics concept can be used only for storing objects but not for primitive values. It is possible to create classes, interfaces and methods that will work in a type - safety manner. The Generics can be called type erasers since the generic information is existing only up to compilation, once compilation is done then all the generic information will be erased.**

The Java Collection interface represents the operations possible on a generic collection, like on a List, Set, Stack, Queue and Deque. For instance, methods to access the elements based on their index are available in the Java Collection interface.

Index Terms **- Bounded Type, Collections, Generics, Hash Se, Maps, Template, Wildcards**

INTRODUCTION

The Generics concept is introduced in Java 5.0 Version which is used to achieve generic programming and resolving the problems of type safety and need for typecasting. Generics can also be called as generic parameter types. Using the generics concept we can achieve compile-time polymorphism. This generic concept looks like a template concept in C++. We can apply the generics concept for classes, interfaces, and for methods.

A collection is a general term that means "a bunch of objects stored in a structured manner" and it is known as collection elements. A collection is an object that holds multiple elements into a single unit. Collections are used to store data, retrieve, manipulate, and communicate aggregate data with certain methods provided.

A Collection mainly contains the following 3 parts:

1. Set of interfaces:

Interfaces allow collections to be manipulated independently of the details of their representation.

2. Concrete class implementation of the interfaces:

They are reusable.

3. Standard utility methods and algorithms:

These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

Advantages of Generics in Java:

1. We can write a method/class/interface once and use it for any type we want.
2. We can hold only a single type of object in generics. It does not allow to store other objects.
3. Individual Type Casting is not needed.
4. By using generics, we can implement algorithms that work on different types of objects and at the same, they are type-safe too.
5. It is checked at compile time so the problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Advantages of Java Collection:

1. Reduced programming effort:

By providing useful data structures and algorithms,

2. Increase quality and speed:

Because you're freed from writing hell lot of your own data structures, you'll have more time to devote to improving programs' quality and performance.

3. Reduces effort to learn new APIs:

Many APIs naturally take collections on input and furnish them as output.

4. Reduces effort to design and implement new APIs:

Designers and implementers do not have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.

5. Software reuse:

New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Generic Class in Java:
A generic class is a class that can hold any kind of object. To create a Generic class we have to specify generic type <T> after the class name.
The syntax is given below.

```
class ClassName< T >
{   // members}
```

Collection Classes (Concrete classes):
We know that we need some concrete collection classes that implements Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract classes, providing skeletal implementations that are used as starting points for creating concrete collections.

ArrayList class:
ArrayList class is also concrete class for List interfaces implementation, in other words it can be treated as resizable-array implementation of the List interface. ArrayList supports dynamic arrays that can grow as needed. Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

HashSet class:
HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage. Hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code. HashSet is a generic class that has this declaration:

Map:
A map is an object that stores associations between keys and values, or key/value pairs. Given a key, you can find its value. Both keys and values are objects. The keys must be unique, but the values may be duplicated.

Generic Interfaces in Java
We can also create a generic interface by specifying the <T> after the interface name.

Syntax :

```
interface InterfaceName< T >
        {   // members }
```

Collection Interfaces: Collection interfaces are the foundation of the Java Collection.

1. Collection interfaces: This interface has methods to tell us how many elements are in the collection (size, isEmpty), to check whether a given object is in the collection (contains), to add and remove an element from the collection (add, remove), and to provide an iterator over the collection (iterator). Collection interface also provides operations (methods) that work on entire collection – containsAll, addAll, removeAll, retainAll, clear.
2. List interfaces: List is one of the most used collection type, the List interface extends the Collection interface and declares the behavior of a collection that stores a sequence of elements. List is an ordered collection and can contain duplicate elements. You can access any element from its index. ArrayList and LinkedList are implementation classes of List interface. Elements can be inserted or accessed by their position in the list.
3. Set interfaces: Set is a collection that cannot contain duplicate elements. The Java platform contains three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet. Set interface doesn't allow random-access to an element in the Collection.
4. SortedSet interfaces: SortedSet is a Set that maintains its elements in ascending order.
5. Map interfaces: A Map is an object that maps unique keys to values. A map cannot contain duplicate keys in its collection and each key can map to at most one value. The Java platform contains three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap. The basic operations of Map are put, get, containsKey, containsValue, size, and isEmpty.
6. SortedMap interfaces: The SortedMap interface extends Map. It ensures that the entries are maintained in ascending key.The Java platform contains three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap. The basic operations of

SortedMap are put, get, containsKey, containsValue, size, and isEmpty.

7. Enumeration interfaces: The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. The Enumeration interface defines the methods by which you can enumerate the elements in a collection of objects.

Generic Methods in Java:

Generic Method is a method that can take any kind of parameter. To create the generic method we have to specify the generic type <T> before the return of the method.

Syntax :

```
< T > returntype methodName(parameters)
        {   // statements  }
```

Type Parameters

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

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

Sample Generic Program:

```
public class generics
{
public static void main(String a[])
{
SimpleGen<String, Integer> sample = new SimpleGen<String, Integer>("JAVA2", 100);
sample.printTypes();
}
}
class SimpleGen<U, V>
{
        private U objUreff;
        private V objVreff;
          public SimpleGen(U objU, V objV)
        {
                this.objUreff = objU;
                this.objVreff = objV;
        }
        public void printTypes( )
        {
System.out.println("U Type: "+this.objUreff.getClass().getName());
System.out.println("V Type: "+this.objVreff.getClass().getName());
        }
}
```

Output :

```
U Type : java.lang.String
V Type : java.lang.Integer
```

NOTE: Generic programming means reusing the same code for storing different types of objects.

The wildcard in Java Generics

In generic code, the question mark (?), called the wildcard, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; occasionally as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method incantation, a generic class instance creation, or a supertype.

Unbounded Wildcards

The unbounded wildcard type represents the list of an unknown type such as List<?>. This approach can be useful in the following scenarios: -

- When the given method is implemented by using the functionality provided in the Object class.
- When the generic class contains the methods that don't depend on the type parameter.

Upper Bounded Wildcards in Java

You can use an upper bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on List<Integer>, List<Double>, and List<Number>; you can achieve this by using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its upper bound. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

Unbounded Wildcards in Java generics

The unbounded wildcard type is specified using the wildcard character (?), for example, List<?>. This is called a list of unknown type. There are two scenarios where an unbounded wildcard is a useful approach: If you are writing a method that can be implemented using functionality provided in the Object class.
When the code is using methods in the generic class that doesn't depend on the type parameter. For example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

Lower Bounded Wildcards in Java
The Upper Bounded Wildcards section shows that an upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the extends keyword. In a similar way, a lower bounded wildcard restricts the unknown type to be a specific type or a supertype of that type.

Generics Restriction
There are some restrictions that needed to remember while using generics. They involve creating objects of a type parameter, static members, exceptions and arrays.

Type Parameters Cannot Be Instantiated
Not possible to create an instance of a type parameter.

```
// Can't create an instance of T
class Gen1< T >
{
T b;
Gen1( )
{
 b = new T ( ); // Illegal;
}
}
```

T does not exist at run time, due to that it is not possible to create an instance of type parameter.

Restrictions on Static Members
A type parameter can not used by static member of a class.

```
class  Gen2 < T >
static T o;  //  No static variable of type T
static T getob() // No static method can use T
{
 return o;
}
static void showob( ) // Can't access object of type T
{
 System.out.println(o);
}
}
```

static generic methods can define their own type parameters.

Generic Exception Restriction
Generic class cannot extend Throwable, so generic exception classes cannot be created.

CONCLUSION

Java Generics is a influential addition to the Java language as it makes the programmer's job easier and fewer error-prone. Generics impose type correctness at compile time and, most importantly, enable implementing generic algorithms without causing any extra overhead to our applications. Generic code will be a part of the future for all java programmers.
The Collections Framework gives the programmer a powerful set of well - engineered solutions to some programming's common task. The Collections Framework is generic, it can be used with type safety, which further contributes its value.

REFERENCES

[1] https://dotnettutorials.net/lesson/generics-in-java
[2] http://tutorials.jenkov.com/java-collections
[3] https://beginnersbook.com/java-collections-tutorials
[4] https://www.javatpoint.com/generics-in-java
[5] THE COMPLETE REFERENCE JAVA: Herbert Schildt, McGraw – Hill, SEVENTH EDITION .
[6] PROGRAMMING WITH JAVA A PRIMER - E. BALAGURUSWAMY, McGraw – Hill, 4TH EDITION