Generics in Java

Generics means parameterized types. They allow us to write code that works with different data types using a single class, interface or method. Instead of creating separate versions for each type, we use type parameters (like <T>) to make the code reusable and type-safe.

Why Use Generics?

- Before Generics, Java collections like ArrayList or HashMap could store any type of object, everything was treated as an Object. It had some problems.
- If you added a String to a List, Java didn't remember its type. You had to manually cast it when retrieving. If the type was wrong, it caused a runtime error.
- With Generics, you can specify the type the collection will hold like ArrayList<String>. Now, Java knows what to expect and it checks at compile time, not at runtime.

Types of Java Generics

1. Generic Class: A generic class is like a regular class but uses type parameters (like <T>). It can accept one or more types, making the class reusable for different data types. Such classes are called parameterized classes.

2. Generic Method: A generic method is a method that can work with different data types using a type parameter. It lets you write one method that works for all types, instead of repeating the same logic.

Generic Class

A generic class is a class that can operate on objects of different types using a type parameter. Like C++, we use <> to specify parameter types in generic class creation. To create objects of a generic class, we use the following syntax:

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()
```

Note: In Parameter type, we can not use primitives like "int", "char" or "double". Use wrapper classes like Integer, Character, etc.

Example:

```java
// We use < > to specify Parameter type
class Test<T> {
    T obj;
    Test(T obj) {
        this.obj = obj;
    }

    public T getObject() { return this.obj; }
}

class Main {
    public static void main(String[] args) {
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test<String> sObj = new Test<String>("Test");
        System.out.println(sObj.getObject());
    }
}

Output
15
Test
```

We can also pass multiple Type parameters in Generic classes.

Example: Generic Class with Multiple Type Parameters

```java
class Test<T, U> {
    T obj1;  // An object of type T
    U obj2;  // An object of type U

    Test(T obj1, U obj2) {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    public void print() {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}
```

```
class Main {
    public static void main (String[] args){
        Test <String, Integer> obj =
            new Test<String, Integer>("Test", 15);

        obj.print();
    }
}

Output
Test
15
```

Generic Method
We can also write generic methods that can be called with different types of arguments based on the type of arguments passed to the generic method. The compiler handles each method.

Example:

```
class Main {
    // A Generic method example
    static <T> void genericDisplay(T element) {
        System.out.println(element.getClass().getName()
                           + " = " + element);
    }

    public static void main(String[] args){
        // Calling generic method with Integer argument
        genericDisplay(11);

        // Calling generic method with String argument
        genericDisplay("Test");

        // Calling generic method with double argument
        genericDisplay(1.0);
    }
}

Output
java.lang.Integer = 11
java.lang.String = Test
java.lang.Double = 1.0
```

Limitations of Generics

1. Generics Work Only with Reference Types

When we declare an instance of a generic type, the type argument passed to the type parameter must be a reference type. We cannot use primitive data types like int, char.

```
Test<int> obj = new Test<int>(20);
```

The above line results in a compile-time error that can be resolved using type wrappers to encapsulate a primitive type.

But primitive type arrays can be passed to the type parameter because arrays are reference types.

```
ArrayList<int[]> a = new ArrayList<>();
```

2. Generic Types Differ Based on their Type Arguments

During compilation, generic type information is erased which is also known as type erasure.

Example:

```
class Test<T> {
    T obj;
    Test(T obj) { this.obj = obj; } // constructor
    public T getObject() { return this.obj; }
}

class Main {
    public static void main(String[] args){
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test<String> sObj
            = new Test<String>("Test");
        System.out.println(sObj.getObject());
        iObj = sObj; // This results an error
    }
}

Output:

error:
 incompatible types:
 Test cannot be converted to Test
```

Explanation: Even though iObj and sObj are of type Test, they are the references to different types because their type parameters differ. Generics add type safety through this and prevent errors.

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

- T: Type
- E: Element
- K: Key
- N: Number
- V: Value

Benefits of Generics
Programs that use Generics have many benefits over non-generic code.
1. Code Reuse: We can write a method/class/interface once and use it for any type we want.

2. Type Safety: Generics make errors appear at compile time rather than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time).

Suppose you want to create an ArrayList that stores the name of students and if by mistake the programmer adds an integer object instead of a string, the compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

Example: Without Generics

```java
import java.util.*;

class Test {
    public static void main(String[] args) {
        // Creating an ArrayList without any type specified
        ArrayList al = new ArrayList();

        al.add("Test1");
        al.add("Test2");
        al.add(10); // Compiler allows this

        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);

        // Causes Runtime Exception
        String s3 = (String)al.get(2);
    }
}
Output :

Exception in thread "main" java.lang.ClassCastException:
   java.lang.Integer cannot be cast to java.lang.String
    at Test.main(Test.java:19)
```
Here, we get a runtime error.

How do Generics Solve this Problem?
When defining ArrayList, we can specify that this list can take only String objects.

Example: With Generics
```
import java.util.*;

class Test {
    public static void main(String[] args) {
        // Creating a an ArrayList with String specified
        ArrayList <String> al = new ArrayList<String> ();

        al.add("Test");
        al.add("Test2");

        // Now Compiler doesn't allow this
        al.add(10);

        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);
        String s3 = (String)al.get(2);
    }
}
Output:
15: error: no suitable method found for add(int)
        al.add(10);
```

3. Individual Type Casting is not needed: If we do not use generics, then, in the above example, every time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data, we need not typecast it every time.

Example:

```java
import java.util.*;

class Test {
    public static void main(String[] args) {
        // Creating a an ArrayList with String specified
        ArrayList<String> al = new ArrayList<String>();

        al.add("Test1");
        al.add("Test2");

        // Typecasting is not needed
        String s1 = al.get(0);
        String s2 = al.get(1);
    }
}
```

4. Generics Promotes Code Reusability: With the help of generics in Java, we can write code that will work with different types of data. For example, let's say we want to Sort the array elements of various data types like int, char, String etc. Basically we will be needing different functions for different data types. For simplicity, we will be using Bubble sort.

Example: Generic Sorting

```java
public class Main {
    public static void main(String[] args) {
        Integer[] a = { 100, 22, 58, 41, 6, 50 };

        Character[] c = { 'v', 'g', 'a', 'c', 'x', 'd', 't' };

        String[] s = { "Test2", "Test1", "Test5", "Test1","Test7",
"Test3", "Test4" };

        System.out.print("Sorted Integer array:  ");
        sortGenerics(a);

        System.out.print("Sorted Character array:  ");
        sortGenerics(c);

        System.out.print("Sorted String array:  ");
        sortGenerics(s);
    }
public static <T extends Comparable<T> > void sortGenerics(T[] a) {
        //Bubble Sort logic
        for (int i = 0; i < a.length - 1; i++) {

            for (int j = 0; j < a.length - i - 1; j++) {

                if (a[j].compareTo(a[j + 1]) > 0) {

                    swap(j, j + 1, a);
                }
            }
        }
        // Printing the elements after sorted
        for (T i : a) {
            System.out.print(i + ", ");
        }
        System.out.println();

    }

    public static <T> void swap(int i, int j, T[] a) {
        T t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
}

Output
Sorted Integer array:  6, 22, 41, 50, 58, 100,
Sorted Character array:  a, c, d, g, t, v, x,
Sorted String array: Test1, Test2, Test3, Test5, Test5, Test6, Test7,
```

Here, we have created a generics method. This same method can be used to perform operations on integer data, string data and so on.