Generics in java

Generics means parameterized types. The idea is to allow type (Integer, String, … etc, and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types.

An entity such as class, interface, or method that operates on a parameterized type is called generic entity.

Generic Class

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use 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'.

```
// A Simple Java program to show working of user defined
// Generic classes
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>  obj = new Test<Integer>(15);
                    System.out.println(obj.getObject());

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

We can also pass multiple Type parameters in Generic classes.

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

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

        // To print objects of T and U
        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>(“Hello", 15);
obj.print();
        }
}
```

Generic Functions:

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

```
// A Simple Java program to show working of user defined
// Generic functions
class Test
{           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("Hello");

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

Output

```
java.lang.Integer = 11
java.lang.String = Hello
java.lang.Double = 1.0
```

Programming Exercise

- Write a program to create a generic method which will allow us to swap object two Integers, two Floats and Two Characters wrapper classes.

Generics work only with Reference Types:

When we declare an instance of 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 by using type wrappers to encapsulate a primitive type.

Generics Types Differ Based on Their Type Arguments:

```
// A Simple Java program to show working
// of user-defined Generic classes

// We use < > to specify Parameter type
class Test<T>
{
        // An object of type T is declared
        T obj;
        Test(T obj) { this.obj = obj; } // constructor
        public T getObject() { return this.obj; }
}

// Driver class to test above
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>("GeeksForGeeks");
                System.out.println(sObj.getObject());
                iObj = sObj; //This results an error
        }
}
```

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

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.

Advantages of Generics:

1. Code Reuse: We can write a method/class/interface once and use for any type we want.
2. Type Safety : Generics make errors to appear compile time 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 store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime. Consider example on next slide

```
// A Simple Java program to demonstrate that NOT using
// generics can cause run time exceptions
import java.util.*;

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

				al.add("Sachin");
				al.add("Rahul");
				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);
		}
}
```

Program will throw runtime exception

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
	at Main.main(Main.java:17)
```

How generics solve this problem?

At the time of defining ArrayList, we can specify that this list can take only String objects.

```
// Using generics converts run time exceptions into
// compile time exception.
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("Sachin");
				al.add("Rahul");

				// 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);
		}
}
```

3. Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same they are type safe too.