GENERIC PROGRAMMING

Generic programming is a style of computer programming in which algorithms are written in terms of "**to-be-specified-later"** types that are then instantiated when needed for specific types provided as parameters.

Generic programming refers to writing code that will work for many types of data.

NON-GENERICS:

In java, there is an ability to create generalized classes, interfaces and methods by operating through Object class.

Example:

```
class NonGen
{
Object ob; NonGen(Object o)
   {

   }
   Object getob()
   {
ob=o;

     return ob;
   }
   void showType()
   {
      System.out.println("Type of ob is "+ob.getClass().getName());
   }
}
public class NonGenDemo
{
   public static void main(String[] arg)
   {
      NonGen integerObj;
      integerObj=new NonGen(88);
      integerObj.showType();


      int v=(Integer)integerObj.getob(); // casting required
      System.out.println("Value = "+v);
      NonGen strObj=new NonGen("Non-Generics Test");
      strObj.showType();
      String str=(String)strObj.getob(); // casting required
      System.out.println("Vlaue = "+str);
   }
}
```

Output:

```
Type of ob is java.lang.Integer
Value = 88
Type of ob is java.lang.String
```

Value = Non-Generics Test

Limitation of Non-Generic:

1) Explicit casts must be employed to retrieve the stored data.
2) Type mismatch errors cannot be found until run time.

Need for Generic:

1) It saves the programmers burden of creating separate methods for handling data belonging to different data types.
2) It allows the code reusability.
3) Compact code can be created.

Advantage of Java Generics (Motivation for Java Generics):

1) **Code Reuse:** We can write a method/class/interface once and use for any type we want.
2) **Type-safety :** We can hold only a single type of objects in generics. It doesn't allow to store other objects.
3) **Elimination of casts:** There is no need to typecast the object.

The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

4) **Stronger type checks at compile time:**

A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);                    //Compile Time Error
```

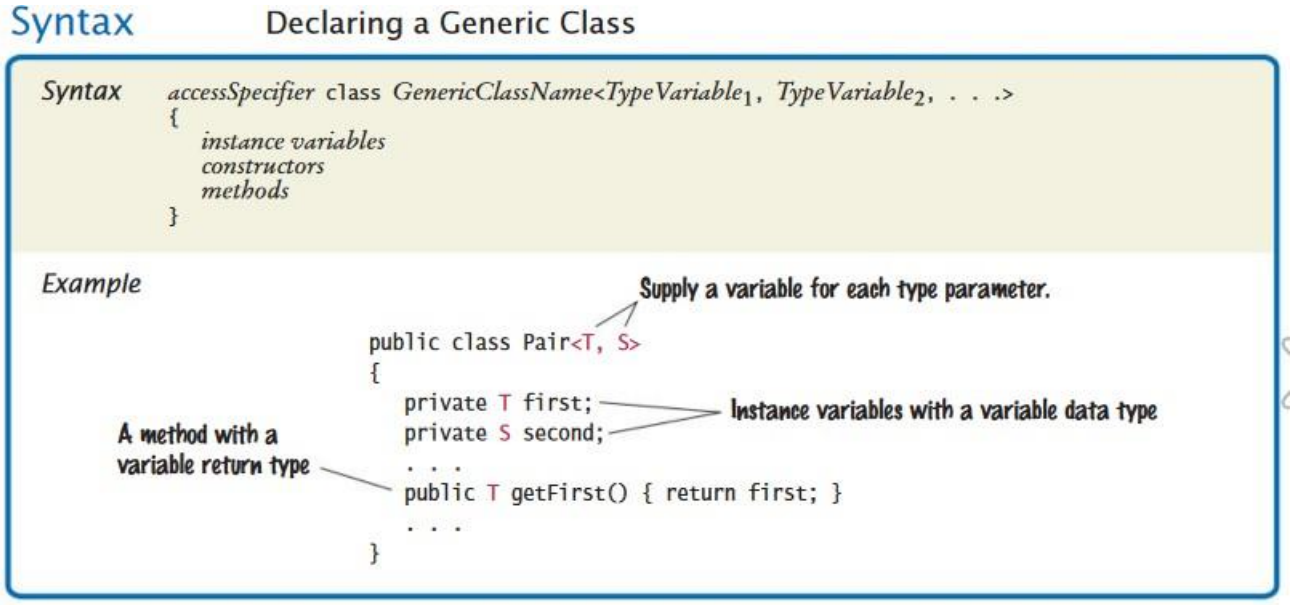
5) **Enabling programmers to implement generic algorithms.**

By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

4.5: GENERIC CLASSES

A class that can refer to any type is known as generic class. Here, we are using T type parameter to create the generic class of specific type.

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.



Where, the type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*)

Example:

```
public class Pair<T, S>
{
. . .
}
```

Purpose: To define a generic class with methods and fields that depends on type variables.

Class reference declaration:

To instantiate this class, use the new keyword, as usual, but place **<type_parameter>** between the class name and the parenthesis:

class_name<type-arg-list> var-name=new class_name<type-arg-list>(cons-arg-list);

Type Parameter Naming Conventions:

- ✓ Type parameter is a place holder for a type argument.
- ✓ By convention, type parameter names are single, uppercase letters.

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

Example: Generic class with single type parameter

```
class Gen <T>
{
T obj;
Gen(T x)
{
    obj= x;
}

T show()
{
    return obj;
}
void disp()
{
    System.out.println(obj.getClass().getName());
}
}
public class Test
{
public static void main (String[] args)
{
 Gen < String> ob = new Gen<>("java programming with Generics");
 ob.disp();
 System.out.println("value :  " +ob.show());

 Gen < Integer> ob1 = new Gen<>(550);
 ob1.disp();
 System.out.println("value  :" +ob1.show());
}
}
```

Output:

```
java.lang.String
value :  java programming with Generics
java.lang.Integer
value  :550
```

Example: Generic class with more than one type parameter

In Generic parameterized types, we can pass more than 1 data type as parameter. It works the same as with one parameter Generic type.

```
class Gen <T1,T2>
{
T1 obj1;
T2 obj2;
```

```
Gen(T1 o1,T2 o2)
{
  obj1 = o1;
  obj2 = o2;
}
 T1 get1()
{
  return obj1;
}
T2 get2()
{
    return obj2;
}
void disp()
{
   System.out.println(obj1.getClass().getName());
   System.out.println(obj2.getClass().getName());
}
}

public class Test
{
public static void main (String[] args)
{
  Gen < String, Integer> obj = new Gen<>("java programming with Generics",560);
  obj.disp();
  System.out.println("value 1 : " +obj.get1());
  System.out.println("value 2:  "+obj.get2());

Gen < Integer, Integer> obje = new Gen<>(1000,560);
  obje.disp();
  System.out.println("value 1 : " +obje.get1());
  System.out.println("value 2:  "+obje.get2());
}
}
```

Output:

```
java.lang.String
java.lang.Integer
value 1 :  java programming with Generics
value 2:  560
java.lang.Integer
java.lang.Integer
value 1 :  1000
value 2:  560
```

4.6: GENERIC METHODS

A Generic Method is a method with type parameter. We can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Rules to define Generic Methods

- ✓ All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type.
- ✓ Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- ✓ The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- ✓ A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Syntax Declaring a Generic Method

Syntax

```
modifiers <TypeVariable1, TypeVariable2, . . .> returnType methodName(parameters)
{
   body
}
```

Example

```
public static <E> void print(E[] a)
{
   for (E e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

Supply the type variable before the return type.

Local variable with a variable data type

Example: (To iterate through the list and display the element using generic method)

```
class a < T >
{
        <T> void show(T[] el)
        {
          for(T x:el)
      System.out.println(x);
  }
 }
public class GenMethod
{
  public static void main(String arg[])
  {
            System.out.println("Integer array");
            a<Integer> o1=new a<Integer>();
            Integer[] ar={10,67,23};
            o1.show(ar);
```

```
            System.out.println("String array");
            a<String> o2=new a<String>();
            String[] ar1={"Hai","Hello","Welcome","to","Java programming"};
            o2.show(ar1);

          System.out.println("Boolean array");
          a<Boolean> o3=new a<Boolean>();
          Boolean[] ar2={true,false};
          o3.show(ar2);
          System.out.println("Double array");
          a<Double> o4=new a<Double>();
          Double[] ar3={10.234,67.451,23.90};
          o4.show(ar3);
  }
}
```

Output:

```
Integer array10
67
23
String arrayHai
Hello Welcometo
Java  programming
Boolean array
true
false
Double array10.234
67.451
23.9
```

4.7: GENERICS WITH BOUNDED TYPES

GENERICS WITH BOUNDED TYPE PARAMETERS:

Bounded Type Parameter is a type parameter with one or more bounds. The bounds restrict the set of types that can be used as type arguments and give access to the methods defined by the bounds.

For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.

Syntax:

```
<T extends superclass>
```

Example:

The following example creates a generic class that contains a method that returns the average of array of any type of numbers. The type of the numbers is represented generically using Type Parameter.

```
public class GenBounds<T extends Number>
{
  T[]  nums;
  GenBounds(T[] obj)
```

```
  {
     nums=obj;
   }
  double average()
  {
     double sum=0.0;
     for(int i=0;i<nums.length;i++)
        sum+=nums[i].doubleValue();
     double avg=sum/nums.length;
     return avg;
   }
  public static void main(String[] args)
  {
     Integer inum[]={1,2,3,4,5};
     GenBounds<Integer> iobj=new GenBounds<Integer>(inum);
     System.out.println("Average of Integer Numbers : "+iobj.average());
     Double dnum[]={1.1,2.2,3.3,4.4,5.5};
     GenBounds<Double> dobj=new GenBounds<Double>(dnum);
     System.out.println("Average of Double Numbers : "+dobj.average());

      /* Error: java,lang.String not within bound
     String snum[]={"1","2","3","4","5"};
     GenBounds<String> sobj=new GenBounds<String>(snum);


     System.out.println("Average of Integer Numbers : "+iobj.average()); */
  }
}
```

Output:

```
F:\>java GenBounds
Average of Integer Numbers : 3.0
Average of Double Numbers : 3.3
```

Wild Card Arguments:

Question mark (?) is the wildcard in generics and represents an unknown type. The wildcard can be used as the type of a parameter, field, or local variable and sometimes as a return type.

Name	Syntax	Meaning
Wildcard with lower bound	? extends B	Any subtype of B
Wildcard with upper bound	? super B	Any supertype of B
Unbounded wildcard	?	Any type

Example: BOUNDED WILDCARDS:-

A bounded wildcard is a wildcard with either an upper or a lower bound.

The following program illustrates the use of wildcards with upper bound. In below method we can use all the methods of upper bound class Number.

```
import java.util.ArrayList;
import java.util.List;
public class GenericsWildcards
{
    public static void main(String[] args)
    {
        List<Integer> ints = new ArrayList<Integer>();
                        ints.add(3);
      ints.add(5);
      ints.add(10);
        double sum = sum(ints);
        System.out.println("Sum of ints="+sum);
    }
    // here Number is the upper bound for the type parameter
    public static double sum(List<? extends Number>  list)
    {
        double sum = 0;
        for(Number n : list)
     {
                        sum += n.doubleValue();
        }
        return sum;
    }
}
```

Output:

```
F:\>java GenericsBounds
Sum of ints=18.0
```

Example: UNBOUNDED WILDCARD:-

Sometimes we have a situation where we want our generic method to be working with all types; in this case unbounded wildcard can be used. The wildcard "?" simply matches any valid objects.

- ✓ Its same as using <? extends Object>.

```
import java.util.*;
public class GenUBWildcard
{
   public static void main(String[] args)
   {
     List<Integer> ints = new ArrayList<Integer>();
```

```
        ints.add(3);
        ints.add(5);
        ints.add(10);
      printData(ints);

     List<String> str = new ArrayList<String>();
        str.add("\nWelcome");
        str.add(" to ");
        str.add(" JAVA ");
      printData(str);
   }
   public static void printData(List<?> list)
   {
         for(Object obj : list)
           {
                 System.out.print(obj + "\n");
         }}}
```

Output:

```
F:\>java GenUBWildcard3
5
10
Welcometo
 JAVA
```

4.8: RESTRICTIONS AND LIMITATIONS OF GENERICS

1) In Java, generic types are compile time entities. The runtime execution is possible only if it is used along with raw type.
2) Primitive type parameters are not allowed for generic programming.
   For example:
   Stack<int> is not allowed.
3) For the instances of generic class throw and catch keywords are not allowed.
   For example:

```
public class Test<T> extends Exception
{
// code                // Error: can't extend the Exception class
}
```

4) Instantiation of generic parameter T is not allowed.
   For Example:

```
new T();       // Error
new T[10];   // Error
```

5) Arrays of parameterized types are not allowed.
   For Example:

```
New Stack<String>[10]; // Error
```