Unit - 4

Generic Programming

* Parametric Polymorphism : ability for a function or a class to be written in such a way that it handles values identically without depending on the knowledge of their types.

Such a function or type is called a generic function or class or data type.

* Type Safety : compiler validates data types during compilation

Parametric polymorphism ensures type safety.

* Why use generics?

→ ensure type safety

→ many algorithms are logically the same no matter what type of data they are applied to. (stacks)

* Generic classes

PROGRAM 1

```
class GenTrialClass <T> {
    T obj;
    GenTrialClass (T obj) {
        this.obj = obj;
    }
```

```
    // getter
    public T getGenObj () {
            return obj;
    }

    // setter
    public void setGenObj (T newobj) {
            obj = newobj;
    }
}

class GenericMain {
    public static void main (String[] args) {
GenTrialClass <String> obj1 = new GenTrialClass<String>
                                        ("Pooja");
S.O.P (obj1.getGenObj());
obj1.setGenObj("POOJAA");

S.O.P (obj1.getclass());
S.O.P (obj1.getclass().getName());


GenTrialClass <Integer> obj2 = new GenTrialClass<Integer
                                        (25);
S.OP (obj2.getGenObj());
obj2.setGetObj(100);
S.OP (obj2.getclass());
S.O.P (obj2.getclass().getName());
```

Output

Pooja
class GenTrialClass
java.lang.String

100
class GenTrialClass
java.lang.Integer

Note: While passing types to generics, primitive classes like int, char, double, float, cannot be used. Only their wrapper classes like Integer, String, Float, Double etc. are allowed.

* Type Erasure : Different versions of the generic class, for different datatypes are not created separately. Instead, the compiler removes all generic type information, substituting the necessary casts, to make the code behave as if a specific version of the class was created. This is called Type Erasure

* Classes with multiple type parameters

PROGRAM 2

```
class Gen2 <T, U> {
    public T first;
    public U second;
}
}
```

```
class TwoGen {
    public static void main (String [ ] args) {
        Gen2 < String , Integer > obj = new Gen2
                                    < String , Integer > ("Mouse", 4);
        s.o.p(obj.first);
        s.o.p(obj.second);
        s.o.p(obj.first.getClass().getName());
        s.o.p(obj_second.getclass().getName());
    }
}
```

Output

```
Mouse
4
java.lang.String
java.lang.Integer
```

* Generic Methods

→ Rather than making the class generic, the methods can be made generic.

Advantage

→ If a class is made is made generic, different class objects have to be made for every data type.

→ With generic methods, a single class object, can be used with multiple function calls for different datatypes

PROGRAM 3

```
class listOperation {

    //syntax:
    public <Type> returntype fnname (<Type> param)
    public <T> T findFirstElement(T [] arr) {
            return arr[0];
    }
}

public class GenMethod {
    public static void main(String [] args) {
        listOperations obj = new listOperation();
        String [] string_list = {"apple", "cat"};
        Integer [] integer_list = {25, 36}
        S.O.P(obj.findFirstElement(string_list));
        S.O.P(obj.findFirstElement(integer_list));
    }
}
```

OUTPUT

```
apple
25
```

* Generic Interfaces

```
interface INTERFACENAME <T> {
    fn name (T a, Tb);
}
class ____ <E> implements INTEFACENAME <T> {
```

Program 4

```
interface myInterface <T> {
        public <T> T showElement (T ele);
}

class myClass <T> implements myInterface <T> {
        public <T> T showElement (T ele) {
                return ele;
        }
}

public class genInterface {
        public static void main (String [] aras){
                myclass obj = new myClass();
                S.o.p(obj.showElement ("Cat"));
                S.o.p(obj.showElement(5));
        }
}
```

OUTPUT

```
Cat
5
```

* Manipulation on generic data

PROGRAM 5

(1) Arithmetic Operations

```
class   addition < T extends Number > {

        T a, b;
        addition ( T a, T b) {
                this. a = a;
                this . b = b;
        }
        void addNums ( ) {
                s.o.p( a . intValue ( ) + b . intValue ( ));
        }
}
```

} an example for upper bound as well

```
public class genArithmetic {
        public static void main ( String [ ] args ) {
                addition obj = new addition (5.5 , 6.6 );
                obj . addNums ( );
        }
}
```

OUTPUT

11

(2) Comparing Equality of Objects

PROGRAM 6

```
class genEquality {
        public < T > void equalityCheck ( T a, T b) {
```

```
        if (a.equals(b)) {
                s.o.p ("Equal");
        }
        else {
                s.o.p("Not equal");
        }
    }
}
class Equality {
    public static void main (String [] args ) {
        gen Equality obj = new genEquality();
        obj.equalityCheck (5,3);
        obj. equalityCheck ("Pooja", "Pooja");
```

OUTPUT

Not equal
Equal

(3) Comparing >, <, = of objects

PROGRAM 7

```
class comparisonClass <T extends Comparable> {
    T max;
    T [] arr;
    comparisonClass (T [] arr) {
        this. arr = arr;
    }
```

```
void maxFind ( ){
        max = arr [0];
        for (int i=0; i< arr. length; i++ ){
                if (arr[i]. compareTo (max) >0 ){
                        max = arr [i];
        }       }
}
Class maximumProgram{
    public static void main ( String [] args ) {
            Integer [] = {2, 4, 3};
            String [ ] = { "Pooja", "Zebra", "Octopus" };
            comparisonClass <Integer > obj1 = new
                                comparisonClass < Integer> (arr1);
            comparisonclass < String > obj2 = new
                                comparisonclass< String > (arr2);
            obj1. maxFind ( );
            obj2 . maxFind ( );
}
```

PROGRAM 8 → Generic Stack Operations

```
class  Stack <T> {
        int top =0;
        T [] arr;
        Stack ( T [] arr ){
        }       this. arr = arr;
```

```
    // stack full
Boolean  stack_full(){
    if (top == arr.length) {

        return true;
    }

    else {
        return false;
    }
}.

// stack empty
Boolean  stack_empty(){
    if (top==0) {

        return true;
    }
    else {
        return false.
    }     }

//push
void push (T ele){
   if (stack_full() ==false){
      arr[top] = ele;

  }}    top++;

//pop
T pop(){
    if (stack_empty() == true) {

      return arr[--top];
    }
   else {
      return null;
    }
}
```

* Generic Inheritance

PROGRAM 9

```
class parent <T> {
        T parent_val;
        parent (T parent_val){
                this parent-val = parent_val;
        }
        void display(){
                S.O.P ( parent_val);
        }
}

class child <T, U> extends parent <T> {
        U child_val;
        child (T parent_val, U child_val) {
                super (parent_val);
                this. child_val= child_val;
        }
        void display( ) {
                ~~super(parent~~
                super. display( );
                S.OP (child_val);
        }
}
//main
class genInheritance {
        public static void main (String[ ] args ) {
                child < String, String> obj = new child <String, String>
                                                ("Preetha", Pooja)
                obj. display( );
        }
}
```

O/P Preetha
Pooja

Bounded Types

→ To restrict the kinds of types that are allowed to be passed to a type parameter

Upper bound : <Type extends SuperType>

Lower bound : <Type super SubType>

Multiple Bounds : <Type extends ClassA & InterfaceB & Interface C>

Wild Cards : A wild card ? is a special kind of type argument that controls the type safety of the usage of generic types

PROGRAM 10

To check if 2 lists have the same average.

```
class Average <T extends Number>{
    T[] numbers;
    Average(T[] numbers){
        this.numbers = numbers;
    }

    double avg(){
        double sum_val;
        for(int i=0; i< numbers.length; i++){
            sum_val += numbers[i].doubleValue();
        }
        return (sum_val)/numbers.length;
    }
```

```
boolean sameAvg(Stats<?> dobj){
    if avg() == dobj.avg(){
        return true;
    }
    else
        return False;
    }
```

```
class wildcardProg {
    public static void main (String [] args) {
        Integer [] i_list = {1, 2, 3, 4, 5}
        Average < Integer > iobj = new Average < Integer > ( i_list),

        Double [] d_list = {1.0, 2.0, 3.0, 4.0, 5.0},

        Average < Double > dobj = new Average < Double > (d_list),

        sud(iobj.someAvg(dobj));
    }
}
```

* Generic Constructor

say you want the value of a variable to be of a particular type → can change type in constructor

PROGRAM 11

```
class genConst {
    int num;
    <T extends Number> genConst (T num) {
        this.num = num.intValue();
    }
    void display() {
        s.o.p(num);
    }
}

class myProgram {
    public static void main (String [] args) {
        gen obj1 = new gen (100);
        gen obj2 = new gen (200.5);
        obj1.display();
```

output
100
200

Limitations of Generic Programming

(1) Use wrapper classes, not primitive types

(2) Type parameters cannot be instantiated.

```
class gen<T> {
        T ob;
        Gen() {
            ob = new T(); // wrong
        }
}
```

(3) Generic arrays cannot be created or instantiated

```
T[] arr = new arr[10] = wrong
```

rather, create a var within the class, and assign the array to that var, by passing it as a parameter.

(4) Static fields & static methods not allowed.

(5) Generic classes do not extend Throwable => cannot make generic exception classes