Unit-3: Generics

22UCSC401 – OBJECT ORIENTED PROGRAMMING

4TH SEMESTER B DIVISION ACADEMIC YEAR: 2023 – 24

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, SDMCET, DHARWAD-2

Generics

CHAPTER-14: JAVA THE COMPLETE REFERENCE NINTH EDITION, HERBERT SCHILDT

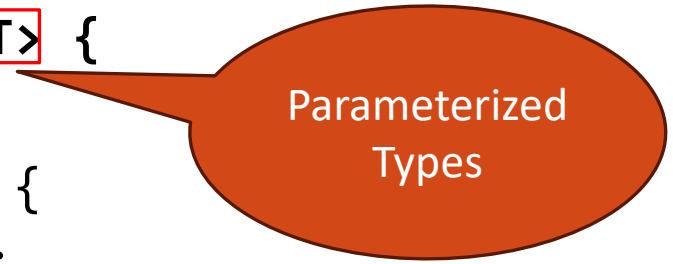
Preamble & Introduction

- ✓Generics was introduced in JDK5
- ✓It added a new syntactical element to Java
- ✓It caused changes to many of the classes and methods in the **core API**
- ✓Through generics, it is possible to create classes, interfaces, and methods that ensure **type-safety** with various kinds of data
- ✓With generics, an algorithm can be defined once and apply it to a wide variety of data types **without any additional effort**

What are Generics?

- ✓ Generics means ***parameterized types***
- ✓ Parameterized types enable creation of classes, interfaces, and methods in which the type of data upon which they operate is *specified* as a parameter
- ✓ Using generics, it is possible to create a single class that automatically works with different types of data

```
class Gen<T> {
  T ob;
  Gen(T o) {
    ob = o;
  }
  T getob() {
    return ob;
  }
  void showType() {
    System.out.println("Type of T is " +
ob.getClass().getName());
  }
}
```




```
class GenericsDemo {
  public static void main(String args[]) {
    Gen<Integer> iOb = new Gen<Integer>(88);
    iOb.showType();
    int v = iOb.getob();
    System.out.println("value: " + v);
    Gen<String> strOb = new Gen<String>("Hi");
    strOb.showType();
    String str = strOb.getob();
    System.out.println("value: " + str);
    Gen<Double> dOb = new Gen<Double>(34.5);
    dOb.showType();
    double d = dOb.getob();
    System.out.println("value: " + d);
  }
}
```

Generics Work Only with Reference Types

- ✓When declaring an instance of a generic type, the type argument passed to type parameter **must be** a reference type
- ✓Primitive types cannot be used
- ✓Gen<int> intOb = new Gen<int>(53); // Error, can't use primitive type

Two Parameter Generic Class

- ✓ We can declare more than one type parameter in a generic type
- ✓ To specify two or more type parameters, simply use a comma-separated list

```
class TwoGen<T, V> {
  T ob1;
  V ob2;
  TwoGen(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }
  void showTypes() {
    System.out.println("Type of T is " +
ob1.getClass().getName());
    System.out.println("Type of V is " +
ob2.getClass().getName());
  }
  T getob1() {
    return ob1;
  }
  V getob2() {
    return ob2;
  }
}
```

```
class TwoGenDemo {
  public static void main(String args[])
{
    TwoGen<Integer, String> tgObj = new
TwoGen<Integer, String>(88, "Generics");
    tgObj.showTypes();
    int v = tgObj.getob1();
    System.out.println("value: " + v);
    String str = tgObj.getob2();
    System.out.println("value: " + str);
  }
}
```

Bounded Types

- ✓ In the previous examples, the type parameters could be replaced by any class type
- ✓ Sometimes it is useful to limit the types that can be passed to a type parameter
- ✓ To handle such situations, Java provides ***bounded types***
- ✓ When specifying a type parameter, we can create an upper bound that declares the superclass from which all type arguments must be derived
- ✓ **<T extends *superclass*>**
- ✓ This specifies that T can only be replaced by superclass, or subclasses of superclass
- ✓ Thus, superclass defines an ***inclusive, upper limit***

```
class Stats<T extends Number> {
  Stats(T[] o) {
    nums = o;
  }
  double average() {
    double sum = 0.0;
    for (int i = 0; i < nums.length; i++)
      sum += nums[i].doubleValue();
    return sum / nums.length;
  }
}
class BoundedTypesDemo {
  public static void main(String args[])
{
    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new
Stats<Integer>(inums);
    double v = iob.average();
    System.out.println("iob average is "
+ v);
    Double dnums[] = { 1.1, 2.2, 3.3,
4.4, 5.5 };
    Stats<Double> dob = new
Stats<Double>(dnums);
    double w = dob.average();
    System.out.println("dob average is "
+ w);
     String strs[] = { "1", "2", "3",
"4", "5" }; //!!!
     Stats<String> strob = new
Stats<String>(strs); //!!!
     double x = strob.average(); //!!!
     System.out.println("strob average is
" + v); //!!!
  }
}
```

Wildcard Argument

- ✓The wildcard argument is specified by **?**
- ✓**?** represents an ***unknown type***
- ✓Ex: **Stats<?> ob**
- ✓Here**, Stats<?>** matches any **Stats** object, allowing any two **Stats** objects to have their averages compared

```
class StatsWild<T extends Number> {
  T[] nums;
  StatsWild(T[] o) {
    nums = o;
  }
  double average() {
    double sum = 0.0;
    for(int i=0;i<nums.length;i++)
      sum += nums[i].doubleValue();
    return sum / nums.length;
  }
  boolean sameAvg(StatsWild<?> ob) {
    if (average()==ob.average())
      return true;
    return false;
  }
}
```

```
class WildCardDemo {
  public static void main(String args[])
{
    Integer inums[]={1,2,3,4,5};
    StatsWild<Integer> iob = new
StatsWild<Integer>(inums);
    double v = iob.average();
    System.out.println("iob average is "
+ v);
    Double dnums[]={1.1,2.2,3.3,4.4,5.5};
    StatsWild<Double> dob = new
StatsWild<Double>(dnums);
    double w = dob.average();
    System.out.println("dob average is "
+ w);
```

```
    Float fnums[]={1.0F,2.0F,3.0F,4.0F, 5.0F};
    StatsWild<Float> fob = new StatsWild<Float>(fnums);
    double x = fob.average();
    System.out.println("fob average is " + x);
    System.out.print("Averages of iob and dob ");
    if (iob.sameAvg(dob))
      System.out.println("are the same.");
    else
      System.out.println("differ.");
    System.out.print("Averages of iob and fob ");
    if (iob.sameAvg(fob))
      System.out.println("are the same.");
    else
      System.out.println("differ.");
  }
}
```

Bounded Wildcards

- ✓Wildcard arguments can be bounded in the same way that a type parameter can be bounded
- ✓A bounded wildcard will operate on a class hierarchy

```java
class TwoD {
  int x, y;
  TwoD(int a, int b) {
    x = a;
    y = b;
  }
}

class ThreeD extends TwoD {
  int z;
  ThreeD(int a, int b, int c) {
    super(a, b);
    z = c;
  }
}
```

```java
class FourD extends ThreeD {
  int t;
  FourD(int a, int b, int c, int d) {
    super(a, b, c);
    t = d;
  }
}

class Coords<T extends TwoD> {
  T[] coords;
  Coords(T[] o) {
    coords = o;
  }
}
```

```
class BoundedWildcardDemo {
  static void showXY(Coords<?> c) {
    System.out.println("X Y Coordinates:");
    for(int i=0;i<c.coords.length;i++)
      System.out.println(c.coords[i].x+" "+c.coords[i].y);
    System.out.println();
  }
  static void showXYZ(Coords<? extends ThreeD> c) {
    System.out.println("X Y Z Coordinates:");
    for(int i=0;i<c.coords.length;i++)
      System.out.println(c.coords[i].x + " " + c.coords[i].y + " " +c.coords[i].z);
    System.out.println();
  }
  static void showAll(Coords<? extends FourD> c) {
    System.out.println("X Y Z T Coordinates:");
    for(int i=0;i<c.coords.length;i++)
      System.out.println(c.coords[i].x+" "+c.coords[i].y+" "+c.coords[i].z+" "+c.coords[i].t);
    System.out.println();
  }
  public static void main(String args[])
{
    TwoD td[]={new TwoD(0, 0),new TwoD(7, 9),new TwoD(18, 4),new TwoD(-1, -23)};
    Coords<TwoD> tdlocs = new Coords<TwoD>(td);
    System.out.println("Contents of tdlocs.");
```

```
    showXY(tdlocs); // !!!
    showXYZ(tdlocs); // !!!
    showAll(tdlocs); // !!!
    FourD fd[] = { new FourD(1, 2, 3, 4),
                   new FourD(6, 8, 14, 8),
                   new FourD(22, 9, 4, 9),
                   new FourD(3, -2, -23, 17) };
    Coords<FourD> fdlocs = new Coords<FourD>(fd);
    System.out.println("Contents of fdlocs.");
    showXY(fdlocs);
    showXYZ(fdlocs);
    showAll(fdlocs);
  }
}
```

Generic Method

- ✓ Methods inside a generic class are automatically generic relative to the type parameter of the generic class
- ✓ However, it is possible to declare a generic method that uses type parameters of its own
- ✓ It is also possible to create a generic method that is defined within a non-generic class
- ✓ The type parameters are declared ***before*** the return type of the method
- ✓ Ex: **public <T extends Comparable<T>> void sort(T[] a)**

```
public class GenericSortDemo {
  private <T> void swap(T[] a, int i, int j) {
    if (i != j) {
      T temp = a[i];
      a[i] = a[j];
      a[j] = temp;
    }
  }
  public <T extends Comparable<T>> void sort(T[] a) {
    for(int i=0;i<a.length-1;i++) {
      int smallest = i;
      for(intj=i+1;j<a.length;j++) {
        if (a[j].compareTo(a[smallest]) <= 0) {
          smallest = j;
        }
      }
      swap(a, i, smallest);
    }
  }
```

```
  public static void main(String[] args) {
    GenericSortDemo gs = new GenericSortDemo();
    Integer[] arr = { 3, 4, 1, 5 };
    System.out.println("before sorting int: " + Arrays.toString(arr));
    gs.sort(arr);
    System.out.println("After sorting int : " + Arrays.toString(arr));
    Double[] d = { 3.0, 4.0, 1.0, 5.0 };
    System.out.println("before sorting int: " + Arrays.toString(d));
    gs.sort(d);
    System.out.println("After sorting int : " + Arrays.toString(d));
    String[] str={"acd","ded","dal","bad", "cle" };
    System.out.println("before sorting String: " + Arrays.toString(str));
    gs.sort(str);
    System.out.println("After sorting String : " + Arrays.toString(str));
    Character[] ch = { 'c', 'e', 'a', 'd', 'c' };
    System.out.println("before sorting char: " + Arrays.toString(ch));
    gs.sort(ch);
    System.out.println("After sorting char : " + Arrays.toString(ch));
  }
}
```

Thank You