Chapter – 8

Generics

Introduction to Generics:

Generics was first introduced in jdk 5 and generic in simple term refers general. By using generics, it is possible to create a single class that automatically works with different types of data. A class, interface or method that operates on a parameterized type is called generic or generic class or generic method.

Parameterized types are important because they enables us to create classes, interface and methods in which the type of data upon which they operates is specified as a parameter.

It is important to understand that java has always given us the ability to create generalized classes, interface and methods by operating through references of type object. Because object is the super class of all other classes, and object reference can refer to any type of object. Thus, in pre-generics code, generalized classes, interface and methods used object reference to operate on various type of object. Generics expands our ability to reuse code.

Generics programs have the following properties:

1. **Generics Works Only with Object:**

When declaring an instance of a generics type, the type of argument passed to the type of parameter must be a class type. We cannot use a primitive type such as int, char, double, float.

2. **Generics Type Differ Based on Their Type Argument:**

A reference of one specific version of generics type is not type compatible with another version of the same generic type.

3. **Stronger Type Check at Compile Time:**

A java compiler applies strong type checking to generics code and issue errors if the code violets type safety. Fixing compile time error is easier than fixing runtime error, which can be difficult to find.

4. **Enabling Programmers to Implement Generics Algorithms:**

By using generics programmer implement generics algorithms that works on collection of different types, can be customized and are type safe and easier to read.

Generics Class with Parameters:

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

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example:

```
class Gen <T> {
   T obj;
   Gen(T o){
      obj = o;
   }
   T getObj(){
      return obj;
   }
   void showType(){
      System.out.println("Type of T is: "+obj.getClass().getName());
   }
}
public class GenericsDemo{
   public static void main(String [] args){
      Gen <Integer> obj1 = new Gen <Integer>(88);
      obj1.showType();
     System.out.println("Input Data: "+obj1.getObj());
     System.out.println();
     Gen <String> obj2 = new Gen <String>("Ramesh");
      obj2.showType();
     System.out.println("Input Data: "+obj2.getObj());
   }
}
```

General Form of a Generic Class:

Syntax:

```
class classname <type parameter list>{
     statements;
}
```

Syntax for declaring a reference to a generic classes:

classname <type-argument-list> variable = new classname <type-argument-list>(constructor-argument-list);

Creating a Generic Method, Constructor, and Interface:

1. Generic Method:

Methods inside a generic class can make use of class type parameter and are automatically generic relative to the type parameter. It is possible to declare a generic method that uses one or

more type parameters of its own. It is also possible to create a generic method that is enclosed with non-generic class.

Example:

```
public class GenericMethodTest {
  // generic method printArray
  public static < E > void printArray( E[] inputArray ) {
    // Display array elements
      int i;
    for(i=0;i<inputArray.length;i++) {
      System.out.print(inputArray[i]+" ");
    }
    System.out.println();
  }

  public static void main(String args[]) {
    // Create arrays of Integer, Double and Character
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

    System.out.println("Array integerArray contains:");
    printArray(intArray);   // pass an Integer array

    System.out.println("\nArray doubleArray contains:");
    printArray(doubleArray);   // pass a Double array

    System.out.println("\nArray characterArray contains:");
    printArray(charArray);   // pass a Character array
  }
}
```

2. Generic Constructor:

A constructor can be declared as generic, independently of whether the class that the constructor is declared in is itself generic. A constructor is generic if it declares one or more type variables. These type variables are known as the formal type parameters of the constructor. The form of the formal type parameter list is identical to a type parameter list of a generic class or interface.

Example:

```
class Test {
        //Generics constructor
        public <T> Test(T item){
                System.out.println("Value of the item: " + item);
                System.out.println("Type of the item: " + item.getClass().getName());
```

```
                System.out.println();
        }
    }
    public class GenericsTest {
        public static void main(String args[]){
                Test test1 = new Test("Test String.");
                Test test2 = new Test(100);
                Test test3 = new Test(100.45f);
                Test test4 = new Test(100.45d);
        }
    }
```

3. Generic Interface:

Generic interfaces are specified just like generic classes. It offers two benefits:

a. It can be implemented for different types of data.
b. It allows us to put constraints on the type of data for which interface can be implemented.

General syntax for generic interface:

interface interface-name <type-parameter-list>

Here, type-parameter-list is a comma separated list of type parameters.

When generic interface is implemented we must specify the type arguments as shown below:

class class-name <type-parameter-list> implements interface-name <type-argument-list>

Example:

```
interface Exmaple<T>{
  public void add(T t);
}
public class Box implements Example <Integer> {

  int t;
  public void add(Integer t) {
    this.t = t;
  }

  public int get() {
    return t;
  }

   public static void main(String[] args) {
    Box a = new Box();
    a.add(10);
    System.out.print(a.get());
```

```
    }
}
```

POLYMORPHISM IN GENERICS:

To understand generics programming we need to understand about the base type and generics type. We can make polymorphic in base type while it is not supported in generic type. The general form is:

List<String> strings = new ArrayList<String>(); // This is valid
List<Object> objects = new ArrayList<String>(); // This is not valid

- List is the base type.
- String is the generic type
- ArrayList is the base type.

There is a simple rule. The type of declaration must match the type of object we are creating. We can make polymorphic references for the base type NOT for the generic type. Hence the first statement is valid while second is not.

- ArrayList<String> list = new ArrayList<String>(); // Valid. Same base type, same generic type
- List<String> list1 = new ArrayList<String>(); // Valid. Polymorphic base type but same generic type
- ArrayList<Object> list2 = new ArrayList<String>(); // Invalid. Same base type but different generic type.
- List<Object> list3 = new ArrayList<String>(); // Invalid. Polymorphic base type different generic type

Example:

Bike extends Vehicle
Car extends Vehicle
Every class has service() method

Vehicle class

```
public class Vehicle {
    public void service() {
            System.out.println("Generic vehicle servicing");
    }
}
```

Bike Class

```
public class Bike extends Vehicle {
    @Override
    public void service(){
            System.out.println("Bike specific servicing");
    }
```

```
}
```

Car Class

```
public class Car extends Vehicle {
    @Override
    public void service() {
            System.out.println("Car specific servicing");
    }
}
```

Main Class

```
import java.util.ArrayList;
import java.util.List;

public class Mechanic {
    public void serviceVehicles(List<Vehicle> vehicles){
      for(Vehicle vehicle: vehicles){
          vehicle.service();
      }
    }

    public static void main(String[] args) {

     List<Vehicle> vehicles = new ArrayList<Vehicle>();
     vehicles.add(new Vehicle());
     vehicles.add(new Vehicle());

     List<Bike> bikes = new ArrayList<Bike>();
     bikes.add(new Bike());
     bikes.add(new Bike());

     List<Car> cars = new ArrayList<Car>();
     cars.add(new Car());
     cars.add(new Car());

     Mechanic mechanic = new Mechanic();
     mechanic.serviceVehicles(vehicles); // This works fine.
     mechanic.serviceVehicles(bikes); // Compiler error.
     mechanic.serviceVehicles(cars); //Compiler error.
   }
}
```

mechanic.serviceVehicles(vehicles) works fine because we are passing ArrayList <Vehicle> to the method that takes List <Vehicle>. mechanic.serviceVehicles (bikes) and

mechanic.serviceVehicles (cars) does not compile because we are passing ArrayList <Bike> and ArrayList<Car> to a method that takes List<Vehicle>.