Advanced Java Programming

Topic: Generics

By

Ravi Kant Sahu

Asst. Professor, LPU

Contents

- Introduction
- Benefits of Generics
- Generic Classes and Interfaces
- Generic Methods
- Wildcard Generic Types
- Restrictions on Generics

Introduction

- Enables to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.
- Introduced in Java by jdk 1.5.
- Generics means parameterized types.
- Generics add the type-safety.
- Generics add stability to your code by making more of your bugs detectable at compile time.

Why Generics?

- The functionality of Gen class can be achieved without generics by specifying Object type and using proper casting whenever required.

Then why we use Generics?

- Java compiler does not have knowledge about the type of data actually stored in NonGen. So-
- Explicit casts must be employed to retrieve the stored data.
- Several type mismatch errors cannot be found until run time.

Why Generics?

- Stronger type checks at compile time
- Elimination of casts

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

- **Using generics:**

```
List<String> list = new ArrayList<String>(); list.add("hello");
String s = list.get(0); // no cast
```

- Enabling programmers to implement generic algorithms.

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

Advantage of Generics

- The ability to create type-safe code in which type-mismatch errors are caught at compile time is a key advantage of generics.

Example

```
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 GenDemo {
  public static void main(String args[]) {
        Gen<Integer> iOb;
        iOb = new Gen<Integer>(88);
        iOb.showType();
        int v = iOb.getob();
        System.out.println("value: " + v);
        Gen<String> strOb = new Gen<String>("Generics
        Test");
        strOb.showType();
        String str = strOb.getob();
        System.out.println("value: " + str);
 } }
```

Generics Work Only with Objects

- When declaring an instance of a generic type, the type argument passed to the type parameter must be a class type.

```
Gen<int> strOb = new Gen<int>(53);
```

- The above declaration is an error.

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

```
iOb = strOb;                    // Wrong!
```

Generic class

General Form of Generic Class

- The generics syntax for declaring a generic class:

```
        class class-name<type-param-list>
                { // ... }
```

- The syntax for declaring a reference to a generic class:

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

Generic Class with Multiple Type Parameters

```
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 SimpGen {
        public static void main(String args[]) {
                TwoGen<Integer, String> t =
                new TwoGen<Integer, String>(16920,  "Ravi
  Kant");
                t.showTypes();
                int v = t.getob1();
                System.out.println("value: " + v);
                String str = t.getob2();
                System.out.println("value: " + str);
        }
}
```

Problem

Create a generic class that contains a method that returns the average of an array of numbers of any type, including integers, floats, and doubles.

Possible Solution

```
class Stats<T> {
    T[] nums;
    Stats(T[] o) {
            nums = o;
            }
    double average() {
            double sum = 0.0;
            for(int i=0; i < nums.length; i++)
                    sum += nums[i].doubleValue(); // Error!!!
                    return sum / nums.length;
            }
}
```

Why Error?

- The compiler has no way to know that you are intending to create Stats objects using only numeric types.
- When we try to compile Stats, an error is reported that indicates that the doubleValue( ) method is unknown.
- We need some way to tell the compiler that we intend to pass only numeric types to T.

Bounded Types

- Used to limit the types that can be passed to a type parameter.

- When specifying a type parameter, we can create an upper bound that declares the super-class from which all type arguments must be derived.

```
<T extends superclass>
```

- A bound can include both a class type and one or more interfaces.

```
class Gen<T extends MyClass & MyInterface>
```

WILD CARD GeneRICS

Problem

- Create a generic class that contains a method sameAvg( ) that determines if two Stats objects contain arrays that yield the same average, no matter what type of numeric data each object holds.

- For example, if one object contains the double values 1.0, 2.0, and 3.0, and the other object contains the integer values 2, 1, and 3, then the averages will be the same.

Possible Solution

```
Integer inums[] = { 1, 2, 3, 4 }; Double dnums[] = { 1.1, 2.2, 3.3, 4.4 };
Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);
if(iob.sameAvg(dob))
        System.out.println("Averages are the same.");
else
        System.out.println("Averages differ.");

boolean same_Avg(Stats<T> ob)
        {
                if(average() == ob.average())
                        return true;
                return false;
        }
```

Why Error?

- It will work only with the objects of same type.
- If the invoking object is of type Stats<Integer>, then the parameter ob must also be of type Stats<Integer>.

WildCard Argument

- The wildcard simply matches the validity of object.

- The wildcard argument is specified by the ?, and it represents an unknown type.

```
boolean same_Avg(Stats<?> ob)
  {
      if(average() == ob.average())
              return true;
      return false;
  }
```

Generic Method

Generic Method

- It is possible to declare a generic method that uses one or more type parameters.

- Methods inside a generic class are automatically generic relative to the type parameters.

- It is possible to create a generic method that is enclosed within a non-generic class.

Generic Methods

- The type parameters are declared before the return type of the method.

- Generic methods can be either static or non-static.

```
<type-param-list> ret-type method-name(param-list) {…}
```

Example:

```
static <T, V extends T> boolean isIn (T x, V[] y)
```

Generic interfaces

Generic Interfaces

- Generic interfaces are specified just like generic classes.

Example:

```
interface MinMax<T extends Comparable<T>>
        {  T min();     T max(); }
```

- The implementing class must specify the same bound.

- Once the bound has been established, it need not to be specified again in the implements clause.

Generic Interface

```
interface MinMax<T extends Comparable<T>>
  {
 T min();
 T max();
  }

class My<T extends Comparable<T>> implements MinMax<T>
 {
        ...
 }
```

Generic constructors

Generic Constructors

- It is possible for constructors to be generic, even if their class is not.

Example:

```
class GenCons {
        private double val;
        <T extends Number> GenCons(T arg)
           {
                 val = arg.doubleValue();
           }
        void show_val() { System.out.println("val: " + val); } }
```

ERASURE

Erasure

- Generic code had to be compatible with pre-existing, non-generic code.

- Any changes to the syntax of the Java language, or to the JVM, had to avoid breaking older code.

- Java implements Generics using Erasures to avoid breaking older codes.

Working of Erasure

- When Java code is compiled, all generic type information is removed (erased).

- This includes replacing type parameters with their bound type, which is Object if no explicit bound is specified.

- Then applying the appropriate casts (as determined by the type arguments) to maintain type compatibility with the types specified by the type arguments.

Erasures

- The compiler enforces type compatibility.
- It means that no type parameters exist at run time.
- They are simply a source-code mechanism.

Generic restrictions

Generic Restrictions

1. Type Parameters Can’t Be Instantiated:

It is not possible to create an instance of a type parameter.

Example:

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

Because T does not exist at run time, how would the compiler know what type of object to create?

Generic Restrictions

2. Restrictions on Static Members :

No static member can use a type parameter declared by the enclosing class.

Example:

```
class Wrong<T> {
        static T ob;
        static T getob() { return ob; }
        static void showob() { System.out.println(ob);
        }   }
```

Note: We can declare static generic methods, which define their own type parameters

Generic Restrictions

3. Generic Array Restrictions:

- We cannot instantiate an array whose base type is a type parameter.

- We cannot create an array of type-specific generic references.

```
class Gen<T extends Number> {
   T ob;
   T vals[]; // OK
   Gen(T o, T[] nums) {
          ob = o;                    // This statement is illegal.
          // vals = new T[10];    // can't create an array of T
          vals = nums;    // OK to assign reference to existent array
   }
}
```

```
class GenArrays {
        public static void main(String args[]) {
                Integer n[] = { 1, 2, 3, 4, 5 };
                Gen<Integer> iOb = new Gen<Integer>(50, n);

// Can't create an array of type-specific generic references.
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
// This is OK.

                Gen<?> gens[] = new Gen<?>[10]; // OK
        }
  }
```

Generic Restrictions

4. Generic Exception Restrictions:

- A generic class cannot extend Throwable.
- This means that we cannot create generic exception classes.

Object Oriented Programming
in
Python

Session -3

Collection Framework concept in Java

Kumar Gaurav
k10gaurav@gmail.com

Java 8

λ Expressions

Scott
Leberknight

Java Programming

Semester- V
Course: CO/CM/IF

Tushar B Kute,
Assistant Professor,
Sandip Institute of Technology and Research Centre, Nashik.

Seminar Report
submit on the topic of

INHERITANCE

in JAVA

presented by :
POOJA JAISWAL

Collection Framework

Ankit Kumar Garg

Java Collections

Parag Shah

Adaptive Software Solutions

http://www.adaptivelearningonline.net
http://www.adaptivesoftware.biz

Java

w3codeblog