JAVA PROGRAMMING

Week 5: Generics

Lecturers:

- HO Tuan Thanh, M.Sc.

Plan

1. Generics fundamentals
2. Bounded types
3. Using wildcard arguments
4. Bounded wildcards
5. Generic methods
6. Generic constructors
7. Generic interfaces
8. Raw types and legacy code
9. Type inference with the diamond operator
10. Local variable type inference and generics

Generics fundamentals

- At its core: generics means parameterized types.
  - Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.
  - A class, interface, or method that operates on a type parameter is called generic, as in generic class or generic method.
- Advantage:
  - Automatically work with the type of data passed to its type parameter.
  - Generics expand your ability to reuse code and let you do so safely and reliably.

Example


```
1.  class Gen<T> {
2.      T ob; // declare an object of type T
3.      // Pass the constructor a reference to
4.      // an object of type T.
5.      Gen(T o) { ob = o; }
6.      // Return ob.
7.      T getob() { return ob; }
8.      // Show type of T.
9.      void showType() {
10.         System.out.println("Type of T is " +
11.                                 ob.getClass().getName());
12.     }
13. }
```

```
1.  class GenDemo {
2.      public static void main(String args[]) {
3.          Gen<Integer> iOb;
4.          iOb = new Gen<Integer>(88);
5.          iOb.showType();
6.          int v = iOb.getob();
7.          System.out.println("value: " + v);
8.          System.out.println();
9.          Gen<String> strOb = new
10.                                 Gen<String>("Generics Test");
11.         strOb.showType();
12.         String str = strOb.getob();
13.         System.out.println("value: " + str);
14.     }
15. }
```

Generics work only with reference types

- When declaring an instance of a generic type, the type argument passed to the type parameter must be a reference type.
- You cannot use a primitive type, such as int or char.
- Example: the following declaration is illegal:

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

- Not being able to specify a primitive type is not a serious restriction because you can use the type wrappers to encapsulate a primitive type.
- Java's autoboxing and autounboxing mechanism makes the use of the type wrapper transparent.

Generic types differ based on their type arguments

- A reference of one specific version of a generic type is not type-compatible with another version of the same generic type.
- Example: the following line of code is in error and will not compile:

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

  - Both iOb and strOb are of type Gen<T>, they are references to different types because their type arguments differ.
  - This is part of the way that generics add type safety and prevent errors.

A generic class with two type parameters


```
1.  class TwoGen<T, V> {
2.      T ob1; V ob2;
3.      TwoGen(T o1, V o2) { ob1 = o1; ob2 = o2; }
4.      void showTypes() {
5.          System.out.println("Type of T is " +
6.                                  ob1.getClass().getName());
7.          System.out.println("Type of V is " +
8.                                  ob2.getClass().getName());
9.      }
10.     T getob1() { return ob1; }
11.     V getob2() { return ob2; }
12. }
```

```
1.  class SimpGen {
2.      public static void main(String args[]) {
3.          TwoGen<Integer, String> tgObj = new
4.                  TwoGen<Integer, String>(88, "Generics");
5.          // Show the types.
6.          tgObj.showTypes();
7.          // Obtain and show values.
8.          int v = tgObj.getob1();
9.          System.out.println("value: " + v);
10.         String str = tgObj.getob2();
11.         System.out.println("value: " + str);
12.     }
13. }
```

General form

- To declare a generic class

```
class classname<typeparamlist> {
    // ...
}
```

- To declaring a reference to a generic class and creating a generic instance

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

Bounded type

- The type parameters could be replaced by any class type.
- This is fine for many purposes,...
  - but sometimes it is useful to limit the types that can be passed to a type parameter.

  → Java provides bounded types
- Syntax:

  <T extends superclass>
  - T can be replaced only by superclass, or subclasses of superclass.

Example [1]


```
1.  class NumericFns<T> {
2.      T num;
3.      NumericFns(T n) { num = n; }
4.      // Return the reciprocal.
5.      double reciprocal() {
6.          return 1 / num.doubleValue(); // Error!
7.      }
8.      // Return the fractional component.
9.      double fraction() {
10.         return num.doubleValue() - num.intValue(); // Error!
11.     }
12.     // ...
13. }
```

Example [2]


```
1.  class NumericFns<T extends Number> {
2.      T num;
3.      NumericFns(T n) { num = n; }
4.      double reciprocal() {
5.          return 1 / num.doubleValue();
6.      }
7.      double fraction() {
8.          return num.doubleValue() - num.intValue();
9.      }
10.     // ...
11. }
```

```
1.  //Demonstrate NumericFns.
2.  class BoundsDemo {
3.      public static void main(String args[]) {
4.          NumericFns<Integer> iOb = new NumericFns<Integer>(5);
5.          System.out.println("Reciprocal of iOb is "
6.                             + iOb.reciprocal());
7.          System.out.println("Fractional component of iOb is ”
8.                             + iOb.fraction());
9.          System.out.println();
10.         NumericFns<Double> dOb = new
11.                                  NumericFns<Double>(5.25);
12.         System.out.println("Reciprocal of dOb is "
13.                            + dOb.reciprocal());
14.         System.out.println("Fractional component of dOb is ”
15.                             + dOb.fraction());
16.         /* This won't compile because String is not a
17.         subclass of Number. */
18.         /* NumericFns<String> strOb = new
19.                             NumericFns<String>("Error");  */
20.     }
21. }
```

Using wildcard arguments

- Specified by the **?**, and it represents an unknown type.

```
1.  class NumericFns<T extends Number> {
2.      T num;
3.      NumericFns(T n) { num = n; }
4.      double reciprocal() { return 1 / num.doubleValue(); }
5.      double fraction() {
6.          return num.doubleValue() - num.intValue();
7.      }
8.      boolean absEqual(NumericFns<?> ob) {
9.          if (Math.abs(num.doubleValue()) ==
10.                 Math.abs(ob.num.doubleValue())) return true;
11.         return false;
12.     }
13.
14.     // ...
15. }
```

```
1.  //Demonstrate a wildcard.
2.  class WildcardDemo {
3.      public static void main(String args[]) {
4.          NumericFns<Integer> iOb = new NumericFns<Integer>(6);
5.          NumericFns<Double>dOb = new NumericFns<Double>(-6.0);
6.          NumericFns<Long> lOb = new NumericFns<Long>(5L);
            System.out.println("Testing iOb and dOb.");
7.          if (iOb.absEqual(dOb))
8.              System.out.println("Absolute values are equal.");
9.          else
10.             System.out.println("Absolute values differ.");
11.         System.out.println();
12.         System.out.println("Testing iOb and lOb.");
13.         if (iOb.absEqual(lOb))
14.             System.out.println("Absolute values are equal.");
15.         else
16.             System.out.println("Absolute values differ.");
17.     }
18. }
```

Bounded wildcards

- Wildcard arguments can be bounded in the same way that a type parameter can be bounded.
- A bounded wildcard is especially important when you are creating a method that is designed to operate only on objects that are subclasses of a specific superclass.
- To establish an upper bound for a wildcard:

<? **extends** superclass>

  - superclass is the name of the class that serves as the upper bound.
- To specify a lower bound for a wildcard :

<? **super** subclass>

  - Only classes that are superclasses of subclass are acceptable arguments.

Example


```
1.  class A {
2.      // ...
3.  }
4.  class B extends A {
5.      // ...
6.  }
7.  class C extends A {
8.      // ...
9.  }
10. // Note that D does NOT
11. // extend A.
12. class D {
13. // ...
14. }
```

```
//A simple generic class.
class Gen<T> {
    T ob;

    Gen(T o) {
        ob = o;
    }
}
```

```
1.  class UseBoundedWildcard {
2.      // Here, the ? will match A or any class type that
3.      // that extends A
4.      static void test(Gen<? extends A> o) {
5.          // ...
6.      }
7.      public static void main(String args[]) {
8.          A a = new A(); B b = new B();
9.          C c = new C(); D d = new D();
10.         Gen<A> w = new Gen<A>(a);
11.         Gen<B> w2 = new Gen<B>(b);
12.         Gen<C> w3 = new Gen<C>(c);
13.         Gen<D> w4 = new Gen<D>(d);
14.         // These calls to test() are OK.
15.         test(w); test(w2); test(w3);
16.         /* Can't call test() with w4 because it is not an object of a class that inherits A. */
17.         test(w4);  // Error!
18.     }
19. }
20.
```

Generic methods

- Previous examples:
  - Methods inside a generic class can make use of a class' 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 within a non-generic class.

```
1.  class GenericMethodDemo {
2.      // Determine if the contents of two arrays are same.
3.      static <T extends Comparable<T>, V extends T>
4.                                      boolean arraysEqual(T[] x, V[] y) {
5.          // If array lengths differ, then the arrays differ.
6.          if (x.length != y.length) return false;
7.          for (int i = 0; i < x.length; i++)
8.              if (!x[i].equals(y[i])) return false;
9.          return true; // contents of arrays are equivalent
10.     }
11.     public static void main(String args[]) {
12.         Integer nums[] = { 1, 2, 3, 4, 5 };
13.         Integer nums2[] = { 1, 2, 3, 4, 5 };
14.         Integer nums3[] = { 1, 2, 7, 4, 5 };
15.         Integer nums4[] = { 1, 2, 7, 4, 5, 6 };
16.         // ....
```

```
1.              // ....
2.              if (arraysEqual(nums, nums))
3.                  System.out.println("nums equals nums");
4.              if (arraysEqual(nums, nums2))
5.                  System.out.println("nums equals nums2");
6.              if (arraysEqual(nums, nums3))
7.                  System.out.println("nums equals nums3");
8.              if (arraysEqual(nums, nums4))
9.                  System.out.println("nums equals nums4");
10.             // Create an array of Doubles
11.             Double dvals[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
12.             // This won't compile because nums and dvals
13.             // are not of the same type.
14.             //  if(arraysEqual(nums, dvals))
15.             //    System.out.println("nums equals dvals");
16.         }
17.     }
```

Generic constructors


```
1.  class Summation {
2.      private int sum;
3.      <T extends Number> Summation(T arg) {
4.          sum = 0;
5.          for (int i = 0; i <= arg.intValue(); i++) sum += i;
6.      }
7.      int getSum() { return sum; }
8.  }
9.  class GenConsDemo {
10.     public static void main(String args[]) {
11.         Summation ob = new Summation(4.0);
12.         System.out.println("Summation of 4.0 is "
13.                                             + ob.getSum());
14.     }
15. }
```

Generic interfaces


```
1.  interface Containment<T> {
2.      boolean contains(T o);
3.  }
4.  class MyClass<T> implements Containment<T> {
5.      T[] arrayRef;
6.      MyClass(T[] o) {
7.          arrayRef = o;
8.      }
9.      public boolean contains(T o) {
10.         for (T x : arrayRef)
11.             if (x.equals(o)) return true;
12.         return false;
13.     }
14. }
```

```
1.  class GenIFDemo {
2.      public static void main(String args[]) {
3.          Integer x[] = { 1, 2, 3 };
4.          MyClass<Integer> ob = new MyClass<Integer>(x);
5.          if (ob.contains(2))System.out.println("2 is in ob");
6.          else System.out.println("2 is NOT in ob");
7.          if (ob.contains(5)) System.out.println("5 is in ob");
8.          else System.out.println("5 is NOT in ob");
9.          /* The follow is illegal because ob is an Integer
10.           Containment and 9.25 is a Double value.*/
11.         //    if(ob.contains(9.25)) // Illegal!
12.         //      System.out.println("9.25 is in ob");
13.     }
14. }
```

RAW TYPES AND LEGACY CODE

- Generics did not exist prior to JDK 5 → it was necessary for Java to provide some transition path from old, pre-generics code.
- Simply put, pre-generics legacy code had to remain both functional and compatible with generics.
- To handle the transition to generics: Java allows a generic class to be used without any type arguments.
  - This creates a raw type for the class.
  - This raw type is compatible with legacy code, which has no knowledge of generics.
  - The main drawback to using the raw type is that the type safety of generics is lost.

```
1.  class Gen<T> {
2.      T ob; // declare an object of type T
3.      Gen(T o) { ob = o; }
4.      T getob() { return ob; }
5.  }
6.  //Demonstrate raw type.
7.  class RawDemo {
8.      public static void main(String args[]) {
9.          Gen<Integer> iOb = new Gen<Integer>(88);
10.         Gen<String> strOb = new Gen<String>("Generics Test");
11.         /* Create a raw-type Gen object and give it a Double
12.           value. When no type argument is supplied, a raw
13.           type is created */
14.         Gen raw = new Gen(98.6);
```

```
1.              // Cast here is necessary because type is unknown.
2.              double d = (Double) raw.getob();
3.              System.out.println("value: " + d);
4.              /* The use of a raw type can lead to run-time
5.                exceptions. Here are some examples. The following
6.                cast causes a run-time error! */
7.              // int i = (Integer) raw.getob(); // run-time error
8.              // This assigment overrides type safety.
9.              strOb = raw; // OK, but potentially wrong
10.             // String str = strOb.getob(); // run-time error
11.             // This assingment also overrides type safety.
12.             raw = iOb; // OK, but potentially wrong
13.             // d = (Double) raw.getob(); // run-time error
14.         }
15.     }
```

Type inference with the diamond operator

- Previous example:

```
class TwoGen<T, V> {
        T ob1; V ob2;
        TwoGen(T o1, V o2) { ob1 = o1; ob2 = o2; }
        …
}
```

- Prior to JDK 7, to create an instance of TwoGen:

```
TwoGen<Integer, String> tgOb = new TwoGen<Integer, String>(42, “Testing”);
```

- From JDK 7:

```
TwoGen<Integer, String> tgOb = new TwoGen<>(42, “Testing”);
```

Type inference with the diamond operator

- Syntax:

class-name<type-arg-list> var-name =

new class-name< >(cons-arg-list);

Ambiguity errors


```
1.  //Ambiguity caused by erasure on overloaded methods.
2.  class MyGenClass<T, V> {
3.      T ob1;
4.      V ob2;
5.      // ...
6.      // These two overloaded methods are ambiguous
7.      // and will not compile.
8.      void set(T o) {
9.          ob1 = o;
10.     }
11.     void set(V o) {
12.         ob2 = o;
13.     }
14. }
```

Some generic restrictions

- Type parameters can’t be instantiated
  - It is not possible to create an instance of a type parameter.
- Restrictions on static members
  - No static member can use a type parameter declared by the enclosing class.
- Generic array restrictions
  - You cannot instantiate an array whose element type is a type parameter.
  - You cannot create an array of type-specific generic references.
- Generic exception restriction
  - A generic class cannot extend throwable.
  - This means that you cannot create generic exception classes.

QUESTION ?