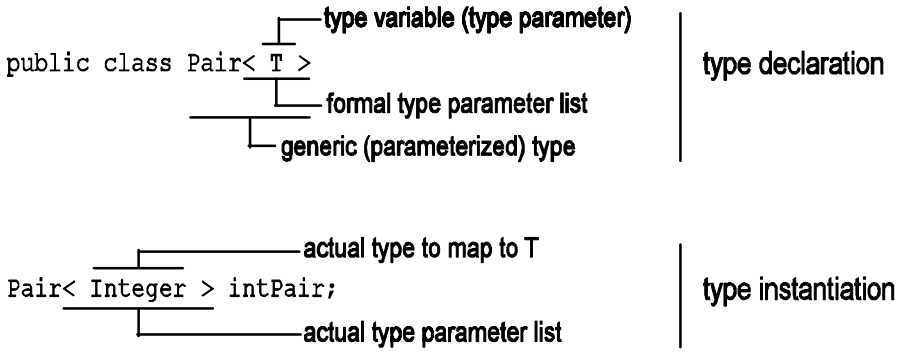
Java Generics

Ameya Joshi
ameya.joshi@vinsys.com
+91 9850676160

1

Generic Types in Java



Format for a generic (parameterized) type and instantiation of a generic type

1-2

2

Using the `Pair<T>` class

```
/**
 * Illustrate the use of a generic type.
 */
public class GenericsEx {
    public static void main( String args[] ) {
        Pair<String> stringPair = new Pair<String>( "string", "Pair" );
        Pair<Integer> intPair = new   Pair<Integer>( 1, 2 );

        System.out.println( "intPair is: " + intPair );
        System.out.println( "stringPair is: " + stringPair );

        intPair.swapElements();
        System.out.println("\nAfter swapping elements, intPair is " + intPair);

        intPair.setFirstElement( new Integer( -1 ) );
        stringPair.setSecondElement( "Generic types are useful" );

        System.out.println( "\nThe pairs after some resetting: " );
        System.out.println( "\tintPair is: " + intPair );
        System.out.println( "\tstringPair is: " + stringPair );
        Integer intElement1 = intPair.getFirstElement();
        String stringElement1 = stringPair.getFirstElement();

        System.out.println( "\nintElement1 is " + intElement1 +
                    " and stringElement1 is " + stringElement1 );
    }
}
```

Specifying the type parameter

stringPair can store a pair of `String` objects

intPair can store a pair of `Integer` objects

Program Output:

```
intPair is: < 1, 2 >
stringPair is: < string, Pair >
After swapping elements, intPair is < 2, 1 >
The pairs after some resetting:
intPair is: < -1, 1 >
stringPair is: < string, Generic types are useful
intElement1 is -1 and stringElement1 is string
```


3

The Pair<T> Class

```
/**
 * A pair consists of two elements of the same type, specified by type parameter T.
 */
public class Pair<T> {
    private T firstElement;
    private T secondElement;
    /**
     * Construct an instance of a <tt>Pair</tt> initialized to the given elements.
     * @param e1 the first element of this pair
     * @param e2 the second element of this pair
     * @throws NullPointerException if either <tt>e1</tt> or <tt>e2</tt> is <tt>null</tt>.
     */
    public Pair( T e1, T e2 ) {
       if ( ( e1 == null ) || ( e2 == null ) ) throw new NullPointerException();
       this.firstElement = e1;
       this.secondElement = e2;
    }

    /**
     * Return the value of the first element of this pair.
     * @return the first element of this pair
     */
    public T getFirstElement() {
       return this.firstElement;
    }

   /**
    * Swap the two elements.
    */
   public void swapElements() {
      T temp = this.firstElement;
      this.firstElement = this.secondElement;
      this.secondElement = temp;
   }
   . . . // other stuff
```

The generic type **T** can be use to declare the type of

- class variables,
- parameters,
- return type
- local variables

4

Generic Types and Erasure

- **Erasure** – the *compiler* will replace all occurrences in the class of type parameter with the *upper bound* of the formal type parameter

  The default upper bound is the class `Object`

1-5

5

Generics & Erasure: The Generic Class

After erasure, the declaration `Pair<Integer> intPair;` would class definition for `Pair` would look like this. Note that the places where T appeared have been replaced with `Integer`'s upper bound, **`Object`**.

```
/**
 * A pair consists of two elements of the same type. This class illustrates the
 * definition of a generic type with type parameter <tt>T</tt>.
 */
public class Pair {
   private Object firstElement;
   private Object secondElement;

   /**
    * Construct an instance of a <tt>Pair</tt> initialized to the given elements.
    * @param e1 the first element of this pair
    * @param e2 the second element of this pair
    * @throws NullPointerException if either <tt>e1</tt> or <tt>e2</tt> is <tt>null</tt>.
    */
   public Pair( Object e1, Object e2 ){
      if ( ( e1 == null ) || ( e2 == null ) )
          throw new NullPointerException();
      this.firstElement = e1;
      this.secondElement = e2;
}
. . .  // more stuff goes here
```

1-6

6

Generic & Erasure: The Client Class

```
/**
 * Illustrate use of a generic type. The client class after erasure.
 */
public class GenericsEx {
    public static void main ( String args[] ) {
        Pair stringPair = new Pair( "string", "Pair" );
        Pair intPair = new Pair( 1, 2 );

        System.out.println( "intPair is: " + intPair );
        System.out.println( "stringPair is: " + stringPair );

        intPair.swapElements();
        System.out.println("\nAfter swapping elements, intPair is "
                           + intPair);
        intPair.setFirstElement( new Integer( -1 ) );
        stringPair.setSecondElement( "Generic types are useful" );

        System.out.println( "\nThe pairs after some resetting: " );
        System.out.println( "\tintPair is: " + intPair );
        System.out.println( "\tstringPair is: " + stringPair );
        Integer intElement1 = (Integer)intPair.getFirstElement();
        String stringElement1 = (String)stringPair.getFirstElement();

        System.out.println( "\nintElement1 is " + intElement1 +
                    " and stringElement1 is " + stringElement1 );
    }
}
```

All occurrences of the formal type are removed

Remember, the ***compiler*** takes care of doing erasure

Type casts are need to convert the `Object` references returned by the get methods to their respective types; the *compiler* does this for you!

1-7

7

Java Generics: Printing a Collection

Goal: Create a method to print all the elements in a collection.
Problem: How to express the type in the parameter list?

```
10  // This method is specific to a collection of Shape objects.
11  // Note the variable type in the for loop.
12  static void printShapeCollection( Collection<Shape> collection ){
13      for ( Shape shape : collection )
14          System.out.println( shape );
15  }
 .  .  .
26  Collection<Shape> shapes = new BasicCollection<Shape>();
27  shapes.add( new Circle( 5.0 ) );
28  shapes.add( new Rectangle( 4.5, 21.2 ) );
29  shapes.add( new Cube( ) );
30  System.out.printf( "From printShapeCollection( shapes )\n");
31  printShapeCollection( shapes );
32
33  Collection<Circle> circles = new BasicCollection<Circle>();
34  circles.add( new Circle( 5.0 ) );
35  circles.add( new Circle( 15.0 ) );
36  circles.add( new Circle( 25.0 ) );
37  //printShapeCollection( circles );          // ERROR!
```

4-8

8

Lesson:

(Previous Slide continued)

While a Circle "is a kind of" Shape, a Collection<Circle> is *NOT* "a kind of" Collection<Shape>! Think of it this way. If Collection<Circle> "is a kind of" Collection<Shape>, then anything you can do to Collection<Shape>, you should be able to also do to Collection<Circle>, but I can add a Rectangle to Collection<Shape>, but not to Collection<Circle>!

9

The Unbounded Wildcard '?'

So, what to do? We need more flexibility in the type parameter. That is, instead of specifying the exact type of the collection, we want to accept a collection storing a *family* of types. Java's solution is the type wildcard, ?, which, because it matches anything, is called an **unbounded wildcard.** More formally, ? is said to be an "unknown type"; thus Collection<?> is a "collection of unknown type."

```
18  // The most general print method. It will print collections of
19  // any kind of type. Note the variable type in the for loop.
20  static void printAnyCollection( Collection<?> collection ){
21      for ( Object element : collection )
22          System.out.println( element );
23  }
```

4-10

10

Another Example: Collection.containsAll()

Note that the element type of the `Collection c` must match the element type of the `Iterator` over `c`.

```
1   public boolean containsAll( Collection<?> c ) {
2       Iterator<?> e = c.iterator();
3       while ( e.hasNext() )
4          // does this collection contain the
5          // next element from c?
6          if( !contains( e.next() ) )
7             // nope, c has an element we don't have
8             return false;
9       return true; // yep, we have all the elements c has
10  }
```

4-11

11

Bounded Wildcards

- There will be times when we don't want the broad inclusiveness of the unbounded wildcard and would prefer to put a *bound* on the family of types accepted. The bounded wildcard does this.
- **Example**: The `addAll()` method from `Collection`.
- Here is the class header for `AbstractCollection`

```
public abstract class AbstractCollection<E>
                      implements Collection<E> {
```

This means we can store elements of type E *or any of E's subclasses* in the collection.

4-12

12

Bounded Wildcard

```
public abstract class AbstractCollection<E>
                      implements Collection<E> {
```

What should be the parameter for `addAll()`?

1st attempt

```
public Boolean addAll(Collection<E> c)
```

Too restrictive. This would preclude adding a Collection<Circle> to a Collection<Shape> or a Collection<Integer> to a Collection<Number>.

2nd attempt

```
public Boolean addAll(Collection<?> c)
```

Not restrictive enough. Would let you *try* to add a Collection<Shape> to a Collection<Number>.

4-13

13

Bounded Wildcard: <? extends E>

What we need is a flexible mechanism that will allow us to specify a *family* of types *constrained* by some "*upper bound*" on the type family. Java's **bounded wildcard** does just this.

```
1  // method from AbstractCollection<E>
2  public boolean addAll(Collection<? extends E> c) {
3      boolean modified = false;
4      Iterator<? extends E> e = c.iterator();
5
6      while ( e.hasNext() ) {
7          if ( add( e.next() ) )
8              modified = true;
9      }
10     return modified;
11 }
```

The bounded wildcard in the parameter type means that the `Collection` type is unknown, but is bounded by type `E`. That is, the element type of `c` must be `E` or one of its subclasses.

4-14

14

Generic Methods: Two Examples

```
18  <T> Collection<T> reverseCopy( Iterator<T> source ) {
19     LinkedList<T> theCopy = new LinkedList<T>();
20     while ( source.hasNext() )
21        theCopy.addFirst( source.next() ); // add at front of list
22     return theCopy;
23   }
24
25   <E> Collection<E> reverseCopy( Collection<E> source ) {
26      LinkedList<E> theCopy = new LinkedList<E>();
27      for ( E element : source )
28        theCopy.addFirst( element ); // add at front of list
29      return theCopy;
30   }
```

```
33   Collection<String> pets = new BasicCollection<String>();
40   Collection<String> petsCopy = copy( pets );
45   petsCopy = reverseCopy( pets );
50   petsCopy = reverseCopy( petsCopy.iterator() );
```

4-15

15