Chapter 8

Generics

CSC 113
King Saud University
College of Computer and Information Sciences
Department of Computer Science

Dr. S. HAMMAMI

OBJECTIVES

In this chapter you will learn:

- **To create generic methods that perform identical tasks on arguments of different types.**
- **To create a generic `Stack` class that can be used to store objects of any class or interface type.**
- **To understand how to overload generic methods with non-generic methods or with other generic methods.**
- **To understand raw types and how they help achieve backwards compatibility.**
- **To use wildcards when precise type information about a parameter is not required in the method body.**
- **The relationship between generics and inheritance.**

OUTLINE

- 1 Introduction
- 2 Motivation for Generic Methods
- 3 Generic Methods: Implementation and Compile-Time
- Translation
- 4 Additional Compile-Time Translation Issues:
- 5 Overloading Generic Methods
- 6 Generic Classes
- 7 Raw Types
- 8 Wildcards in Methods That Accept Type Parameters
- 9 Generics and Inheritance: Notes

1. Introduction

Generics

- Provide compile-time type safety
  - Catch invalid types at compile time
- Generic methods
  - A single method declaration
  - A set of related methods
- Generic classes
  - A single class declaration
  - A set of related classes

Generic methods and classes are among Java's most powerful capabilities for software reuse with compile-time type safety.

2. Motivation for Generic Methods

- Overloaded methods
  - Perform similar operations on different types of data
  - Overloaded `printArray` methods
    - `Integer` array
    - `Double` array
    - `Character` array
  - Only reference types can be used with generic methods and classes

2. Motivation for Generic Methods: Example

```
1  // OverloadedMethods.java
2  // Using overloaded methods to print array of different types.
3
4  public class OverloadedMethods
5  {
6     // method printArray to print Integer array
7     public static void printArray( Integer[] inputArray )
8     {
9        // display array elements
10       for ( Integer element : inputArray )
11          System.out.printf( "%s ", element );
12
13       System.out.println();
14    } // end method printArray
15
16    // method printArray to print Double array
17    public static void printArray( Double[] inputArray )
18    {
19       // display array elements
20       for ( Double element : inputArray )
21          System.out.printf( "%s ", element );
22
23       System.out.println();
24    } // end method printArray
25
```

Method `printArray` accepts an array of `Integer` objects

Method `printArray` accepts an array of `Double` objects

2. Motivation for Generic Methods: Example

```
26    // method printArray to print Character array
27    public static void printArray( Character[] inputArray )
28    {
29       // display array elements
30       for ( Character element : inputArray )
31          System.out.printf( "%s ", element );
32
33       System.out.println();
34    } // end method printArray
35
36    public static void main( String args[] )
37    {
38       // create arrays of Integer, Double and Character
39       Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
40       Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
41       Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
42
```

Method `printArray` accepts an array of `Character` objects

2. Motivation for Generic Methods: Example

```
43        System.out.println( "Array integerArray contains:" );
44        printArray( integerArray ); // pass an Integer array
45        System.out.println( "\nArray doubleArray contains:" );
46        printArray( doubleArray ); // pass a Double array
47        System.out.println( "\nArray characterArray contains:" );
48        printArray( characterArray ); // pass a Character array
49     } // end main
50  } // end class OverloadedMethods
```

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

At compile time, the compiler determines argument integerArray's type (i.e., Integer[]), attempts to locate a method named printArray that specifies a single Integer[] parameter (lines 7-14)

At compile time, the compiler determines argument `doubleArray`'s type (i.e., `Double`[]), attempts to locate a method named `printArray` that specifies a single `Double`[] parameter (lines 17-24)

At compile time, the compiler determines argument characterArray's type (i.e., Character[]), attempts to locate a method named printArray that specifies a single Character[] parameter (lines 7-14)

2. Motivation for Generic Methods

- Study each `printArray` method
  - Array element type appears in two location
    - Method header
    - `for` statement header
- Combine three `printArray` methods into one
  - Replace the element types with a generic name `E`
  - Declare one `printArray` method
    - Display the string representation of the elements of any array

2. Motivation for Generic Methods

```
1  public static < E > void printArray( E[] inputArray )
2  {
3     // display array elements
4     for ( E element : inputArray )
5        System.out.printf( "%s ", element );
6
7     System.out.println();
8  } // end method printArray
```

Replace the element type with a single generic type E

Replace the element type with a single generic type E

`printArray` method in which actual type names are replaced by convention with the generic name `E`.

3. Generic Methods: Implementation and Compile-Time Translation

- Using a generic method
  - Method calls are identical
  - Outputs are identical
- Generic method declaration
  - Type parameter section
    - Delimited by angle brackets (< and >)
    - Precede the method's return type
    - Contain one or more type parameters
      - Also called formal type paramters

3. Generic Methods: Implementation and Compile-Time Translation

- Type parameter
  - Also known as type variable
  - An identifier that specifies a generic type name
  - Used to declare return type, parameter types and local variable types
  - Act as placeholders for the types of the argument passed to the generic method
    - Actual type arguments
  - Can be declared only once but can appear more than once
    - `public static < E > void printTwoArrays( E[] array1, E[] array2 )`

Common Programming Error

- When declaring a generic method, failing to place a type parameter section before the return type of a method is a syntax error—the compiler will not understand the type parameter name when it is encountered in the method.

3. Generic Methods: Implementation and Compile-Time Translation

```
1  // GenericMethodTest.java
2  // Using generic methods to print array of different types.
3
4  public class GenericMethodTest
5  {
6     // generic method printArray
7     public static < E > void printArray( E[] inputArray )
8     {
9        // display array elements
10       for ( E element : inputArray )
11          System.out.printf( "%s ", element );
12
13       System.out.println();
14    } // end method printArray
15
16    public static void main( String args[] )
17    {
18       // create arrays of Integer, Double and Character
19       Integer[] intArray = { 1, 2, 3, 4, 5 };
20       Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
21       Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
22
```

Use the type parameter to declare method `printArray`'s parameter type

Type parameter section delimited by angle brackets (< and >)

Use the type parameter to declare method `printArray`'s local variable type

3. Generic Methods: Implementation and Compile-Time Translation

```
23        System.out.println( "Array integerArray contains:" );
24        printArray( integerArray ); // pass an Integer array
25        System.out.println( "\nArray doubleArray contains:" );
26        printArray( doubleArray ); // pass a Double array
27        System.out.println( "\nArray characterArray contains:" );
28        printArray( characterArray ); // pass a Character array
29     } // end main
30  } // end class GenericMethodTest
```

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

Invoke generic method `printArray` with an `Integer` array

Invoke generic method `printArray` with a `Double` array

Invoke generic method `printArray` with a `Character` array

Good Programming Practice

It is recommended that type parameters be specified as individual capital letters. Typically, a type parameter that represents the type of an element in an array (or other collection) is named `E` for “element.”

Common Programming Error

- If the compiler cannot match a method call to a non-generic or a generic method declaration, a compilation error occurs.

Common Programming Error

If the compiler does not find a method declaration that matches a method call exactly, but does find two or more generic methods that can satisfy the method call, a compilation error occurs.

4. Additional Compile-Time Translation Issues:
Methods That Swapping Types Parameter

```
public class Swapping
{
   public static <X> void swap(X a, X b)
   {
     X c = a;
     a = b;
     b = c;
   }

  public static void main (String args[])
  {
    Integer a = 10;
    Integer b =5;
    swap(a,b);
    System.out.println(a +" " + b);
   }
}
```

5. Generic Class.

```
public class BaseClassGeneric<T>
{
  T ob; // declare an object of type T

  // Pass the constructor a reference to an object of type T.
  public BaseClassGeneric(T o)
  {
    ob = o;
  }

  public void display()
  {
    System.out.println("\nmessage form Base Class: Assalamo Alykom " );
    System.out.println("Type of T is " + ob.getClass().getName());
  }
}
```

A simple generic class.

```
//Demonstrate the generic class.
public class BCGDemo {
  public static void main(String args[]) {

 // Create a  BaseClassGeneric   object for Strings.
    BaseClassGeneric<Integer> iOb;
    iOb = new BaseClassGeneric<Integer>(88);

    // Show the type of data used by iOb.
    iOb.showType();

    int v = iOb.getob();
    System.out.println("value: " + v);

    // Create a  BaseClassGeneric  object for Strings.
    BaseClassGeneric<String> strOb = new BaseClassGeneric<String>("Generics Test");

    // Show the type of data used by strOb.
    strOb.showType();

    // Get the value of strOb. Again, notice
    // that no cast is needed.
    String str = strOb.getob();
    System.out.println("value: " + str);
  }
}
```

A simple generic class.

```
public class MyGenericClass<T>
{
  private T var1; // here, T will be replaced by Object

  public MyGenericClass(T v)
  {
    var1=v;
  }

  public T getVar()
  {
    return var1;
  }

  public void set(T v1)
  {
      var1=v1;
  }
  public void display()
  {
      System.out.println("The value is :"+var1);
  }

}
```

A simple generic class.

```
public class MyGenericClassTest
{
   public static void main(String [] args)
   {
      MyGenericClass<Double> h;
      h = new MyGenericClass<Double>(4.5);

      MyGenericClass<Character> p;
      p = new MyGenericClass<Character>('b');

      h.display();
      h.set(3.44);
      System.out.println(" For double : "+h.getVar());

      p.display();
      p.set('k');
      System.out.println(" For Charecter "+p.getVar());
   }
}

/*   run
The value is gg :4.5
 For double : 3.44
The value is gg :b
 For Charecter k

*/
```

A simple generic class with two type parameters

```
/ A simple generic class with two type parameters: T and V.

public class ClassTwoTypes<T, V>
{
  private T ob1;
  private V ob2;

  // Pass the constructor a reference to an object of type T.
  public ClassTwoTypes(T o1, V o2)
  {
    setValues(o1,o2);
  }

  // Show types of T and V.
  public void showTypes()
  {
    System.out.println("Type of T = " +
                       ob1.getClass().getName());

    System.out.println("Type of V =  " +
                       ob2.getClass().getName());
  }
```

```
public T getob1()
  {
    return ob1;
  }

  public V getob2()
  {
    return ob2;
  }

  public void setValues(T v1, V v2)
  {
     ob1=v1;
     ob2=v2;
  }

  public void display()
  {
    System.out.println("attribute of T is " + ob1);
    System.out.println("attribute of V is " + ob2);
  }
}
```

A simple generic class with two type parameters

```
public class ClassTwoTypesTest
{
   public static void main(String []args)
   {
      ClassTwoTypes<Double,Character> g;
      g=new ClassTwoTypes<Double,Character>(4.5, 'k');
      g.showTypes();
      g.setValues(6.77, 'h');
      g.display();
   }

}

/*  run
Type of T = java.lang.Double
Type of V =  java.lang.Character
attribute of T is 6.77
attribute of V is h
*/
```

A simple generic class hierarchy

```
class AA<T> {
  T ob;

  AA(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
  public void display()
  {
    System.out.println("\nmessage form Base
                Class: Assalamo Alykom " );
  }
}

// A subclass of AA that defines a second
// type parameter, called V.
```

```
class BB<T, V> extends AA<T> {
  V ob2;

  BB(T o, V o2) {
    super(o);
    ob2 = o2;
  }

  V getob2() {
    return ob2;
  }
 public void display()
  {
    System.out.println("\nmessage form Sub class: Salam ");
  }
}
```

A simple generic class hierarchy

```
public class InheritanceGenericTest
{
  public static void main(String args[])
  {
    // Create a Gen object for Integers.
    BaseClassGeneric<Integer> x = new
BaseClassGeneric<Integer>(88);
    x.display();
    // Create a Gen2 object for Strings.
    SubClassGeneric<String> z = new
    SubClassGeneric<String>("Generics Test");
    z.display();
    BaseClassGeneric<Character> w = new
SubClassGeneric<Character>('c');
    w.display();
 }
}
```

Bounds for Type Parameters

- A bound on a type may be a class name (rather than an interface name)
  - Then only descendent classes of the bounding class may be plugged in for the type parameters

```
public class ExClass<T extends Class1>
```

- A bounds expression may contain multiple interfaces and up to one class
- If there is more than one type parameter, the syntax is as follows:

```
public class Two<T1 extends Class1, T2 extends Class2 & Comparable>
```

Bounds for Type Parameters

```
public class MaxGeneric {

   public static <T extends Comparable<T>> T maximum(T x, T y, T z)
   {
                              T max=x;
                              if (y.compareTo(max) > 0)
                                        max =y;
                              if (z.compareTo(max) > 0)
                                        max=z;
                              return max;
    }

   public static void main(String[] args)
   {
     System.out.println("The maximum of 3, 4, and 6 is : "+ maximum(3,4,6));
     System.out.println("The maximum of 3.3, 5.1, and 2.5 is : "+ maximum(3.3,5.2,2.5));

   }
}

/*
 The maximum of 3, 4, and 6 is : 6
 The maximum of 3.3, 5.1, and 2.5 is : 5.2

 */
```

Bounds for Type Parameters

A Bounded Type Parameter

```
public class Pair<T extends Comparable>
{
    private T first;
    private T second;

    public T max()
    {
        if (first.compareTo(second) >= 0)
            return first;
        else
            return second;
    }
    public Pair(T f, T s)
    {
      first = f;
      second = s;
    }
```

Bounds for Type Parameters

```
public void setFirst(T newFirst)
{
    first = newFirst;
}

public void setSecond(T newSecond)
{
    second = newSecond;
}

public T getFirst()
{
    return first;
}


public T getSecond()
{
    return second;
}

public String toString()
{
    return ( "first: " + first.toString() + "\n"
            + "second: " + second.toString() );
}
}
```

6. A generic interface example

```
// A Min/Max interface.
interface MinMax<T extends Comparable<T>> {
  T min();
  T max();
}
// Now, implement MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
  T[] vals;
  MyClass(T[] o) { vals = o; }

  // Return the minimum value in vals.
  public T min() {
    T v = vals[0];
     for(int i=1; i < vals.length; i++)
      if(vals[i].compareTo(v) < 0) v = vals[i];
     return v;
  }

  // Return the maximum value in vals.
  public T max() {
    T v = vals[0];
     for(int i=1; i < vals.length; i++)
      if(vals[i].compareTo(v) > 0) v = vals[i];
     return v;
  }
}
```

```
public class GenIFDemo {

   public static void main(String args[]) {
    Integer inums[] = {3, 6, 2, 8, 6 };
    Character chs[] = {'b', 'r', 'p', 'w' };

    MyClass<Integer> iob = new MyClass<Integer>(inums);
    MyClass<Character> cob = new MyClass<Character>(chs);

    System.out.println("Max value in inums: " + iob.max());
    System.out.println("Min value in inums: " + iob.min());

    System.out.println("Max value in chs: " + cob.max());
    System.out.println("Min value in chs: " + cob.min());
  }
}
```