CHAPTER 4

■ ■ ■

Generics

EXERCISES

1. What are generics (or generic types), parameterized types, and raw types? Give an example of a generic type and its parameterized type.

Answer:

Generics allows you to declare types (classes and interfaces) that use type parameters. Type parameters are specified when the generic type is used. The type when used with the actual type parameter is known a parameterized type. When a generic type is used without specifying its type parameters, it is called a raw type. Example let generic type be `Wrapper<T>`, then its parameterized type with Long as the type parameter is `Wrapper<Long>`.

2. The `Number` class is the superclass of the `Long` class. The following snippet of code does not compile. Explain.

```
List<Number> list1= new ArrayList<>();
List<Long> list2= new ArrayList<>();
list1 = list2;  // A compile-time error
```

Answer:

The supertype–subtype relationship does not exist with parameterized types. As `List<Long>` is not a subtype of `List<Number>`, the compiler flags an error for `list1 = list2`;

3. Write the output when the following `ClassNamePrinter` class is run. Rewrite the code for the `print()` method of this class after the compiler erases the type parameter `T` during compilation .

```
// ClassNamePrinter.java
```

```
package com.jdojo.generics.exercises;

public class ClassNamePrinter {
    public static void main(String[] args) {
        ClassNamePrinter.print(10);
        ClassNamePrinter.print(10L);
        ClassNamePrinter.print(10.2);
    }

    public static <T extends Number> void print(T obj) {
        String className = obj.getClass().getSimpleName();
       System.out.println(className);
    }
}
```

Answer:

Output is as follows:

```
Integer
Long
Double
```

Code for the `print()` method after type erasure is as follows:

```
public static void print(Number obj) {
    String className = obj.getClass().getSimpleName();
    System.out.println(className);
}
```

4. What are unbounded wildcards? Why does the following snippet of code not compile?

```
List<?> list = new ArrayList<>();
list.add("Hello"); // A compile-time error
```

Answer:

An unbounded wildcard type means "some unknown type". Using a question mark as a parameter type (`<?>`) specifies an *unbounded wildcard*.

Parameter type `T` for `List<T>` is unknown for list when it is declared as `List<? >`.. Compiler flags an error when input is `String`, as `String` is not same as type unknown (`?`).

5. Consider the following incomplete declaration of the `Util` class:

```
// Util.java
package com.jdojo.generics.exercises;
```

```
import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Util {
    public static void main(String[] args) {
        Integer[] n1 = {1, 2};
        Integer[] n2 = {3, 4};
        Integer[] m = merge(n1, n2);
        System.out.println(Arrays.toString(m));

        String[] s1 = {"one", "two"};
        String[] s2 = {"three", "four"};
        String[] t = merge(s1, s2);
        System.out.println(Arrays.toString(t));

        List<Number> list = new ArrayList<>();
        add(list, 10, 20, 30L, 40.5F, 50.9);
        System.out.println(list);
    }

    public static <T> T[] merge(T[] a, T[] b) {

    }

    public static /* Add type parameters here */ void add(List<T> list, U...
elems) {

        /* Your code to add elems to list goes here */
    }
}
```

Complete the body of the `merge()` method, so it can concatenate the two arrays passed in as its parameters and return the concatenated array.

Complete the `add()` method by specifying its type parameters and adding the code in its body. The first parameter to the method is a parameterized `List<T>` and the second parameter is a varargs parameter of the type `T` or its descendant. That is the second parameter type is any type whose objects can be added to the `List<T>`.

Running the `Util` class should produce the following output:

```
[1, 2, 3, 4]
[one, two, three, four]
[10, 20, 30, 40.5, 50.9]
```

Solution:

```
// Util.java
package com.jdojo.generics.exercises;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Util {
   public static void main(String[] args) {
       Integer[] n1 = {1, 2};
       Integer[] n2 = {3, 4};
       Integer[] m = merge(n1, n2);
       System.out.println(Arrays.toString(m));

       String[] s1 = {"one", "two"};
       String[] s2 = {"three", "four"};
       String[] t = merge(s1, s2);
       System.out.println(Arrays.toString(t));

       List<Number> list = new ArrayList<>();
       add(list, 10, 20, 30L, 40.5F, 50.9);
       System.out.println(list);
   }

   public static <T> T[] merge(T[] a, T[] b) {
       T[] arr = Arrays.copyOf(a, a.length + b.length);
       int i = a.length;
       for (T t : b)
           arr[i++] = t;
       return arr;
   }

   public static <T, U extends T> void add(List<T> list, U... nums) {
       list.addAll(Arrays.asList(nums));
   }

}
```

6. Create a generic `Stack<E>` class. Its objects represent a stack that can store elements of its type parameter `E`. The following is a template for the class. You need to provide implementation for all its methods. Write test code to test all methods. Method names are standard method names for a stack. Any illegal access to the stack should throw a runtime exception.

```
// Stack.java
package com.jdojo.generics.exercises;

import java.util.LinkedList;
import java.util.List;
```

```
public class Stack<E> {
    // Use LinkedList instead of ArrayList
    private final List<E> stack = new LinkedList<>();

    public void push(E e) {}
    public E pop() { }
    public E peek() { }
    public boolean isEmpty() { }
    public int size() { }
}
```

Solution:

```
// Stack.java
package com.jdojo.generics.exercises;

import java.util.LinkedList;
import java.util.List;

public class Stack<E> {
    // Use LinkedList instead of ArrayList
    private final List<E> stack = new LinkedList<>();

    public void push(E e) {
        stack.add(0, e);
    }

    public E pop() {
        if (stack.isEmpty())
            throw new RuntimeException("Stack is empty.");
        return stack.remove(0);
    }

    public E peek() {
        if (stack.isEmpty())
            throw new RuntimeException("Stack is empty.");
        return stack.get(0);
    }

    public boolean isEmpty() {
        return stack.isEmpty();
    }

    public int size() {
        return stack.size();
    }
}

// StackTest.java
package com.jdojo.generics.exercises;
```

```
public class StackTest<E> {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        System.out.println("Initial stack size: " + stack.size());
        System.out.println("Initial stack isEmpty: " + stack.isEmpty());

        try {
            System.out.println("Pop an element from empty stack: " + stack.pop());
        } catch (RuntimeException ex) {
            System.out.println("Exception during pop() from empty stack: " +
ex.getMessage());
        }
        // add an element
        stack.push(1);
        System.out.println("Stack size after first push: " + stack.size());
        System.out.println("Stack isEmpty after first push: " + stack.isEmpty());

        // add 3 elements
        stack.push(10);
        stack.push(100);
        stack.push(1000);
        System.out.println("Stack size after adding 4 ints: " + stack.size());

        System.out.println("Top element: " + stack.peek());
        System.out.println("Pop an element: " + stack.pop());
        System.out.println("Stack size after pop(): " + stack.size());
    }

}
```

7. What is heap pollution? What types of warnings does the compiler generate when it detects a possibility of heap pollution. How do you print such warnings during compilation. How do you suppress such warnings?

Answer:

Heap pollution is a situation that occurs when a variable of a parameterized type refers to an object not of the same parameterized type. The compiler issues an unchecked warning if it detects possible heap pollution. Use `-Xlint:unchecked` option with `javac` command to see unchecked warnings. Unchecked warnings can be suppressed by using `@SuppressWarnings` annotation.

8. Describe the reasons that the following declaration of the `Test` class does not compile.

```
public class Test {
    public <T> void test(T t) {
       // More code goes here
    }

    public <U> void test(U u) {
       // More code goes here
    }
 }
```

Answer:

Both type parameters T & U are same after type erasure, which will be replaced with `Object`. So, both the `test()` method signatures are same after type erasure – both take in one generic type parameter and return `void`. In fact, the `test(U u)` method is a duplicate of `test(T t)`, which causes the compiler to flag an error.