**WIA1002 TUTORIAL 2**

**GENERICS**

1. Create a generic class called Container that accepts one parameter, T. Create a no-arg constructor. Declare a private variable, t of type T. Create a method, add() that returns nothing, accepting a parameter of generic type. Initialize this parameter to the initially declared variable. Create a generic method called retrieve() that returns the initially declared variable.

Create a main method within the Container class. Create two containers of type Integer and String. Append two objects to the containers, one of type Integer having value 50, another of type

String having value Java. Display the Integer and String objects from the respective containers.

public class Container <**T**>{  
 private **T** value**;** Container(){  
 }  
 public void add(**T** value){  
 this.value=value**;** }  
 public **T** retrieve(){  
 return this.value**;** }  
 public static void main(String[] args) {  
 Container <Integer> integerContainer=new Container<Integer>()**;** integerContainer.add(**50**)**;** Container<String> strContainer = new Container<>()**;** strContainer.add("Java")**;** System.*out*.println("Integer value: " + integerContainer.retrieve())**;** System.*out*.println("String value: " + strContainer.retrieve())**;** }  
}

2. Create a class called MyArray that has two methods, a main method that creates 3 arrays of

a) integer containing the numbers 1,2,3,4 and 5

b) string containing names, Jane, Tom and Bob

c) character containing alphabet, a, b and c

and a generic method listAll that displays the list of arrays.

public class MyArray <**T**> {  
 private **T** []array**;** MyArray(){  
 }  
 public static <**T**> void listAll(**T** [] array){  
 for(**T** item : array){  
 System.*out*.print(item + " ")**;** }  
 System.*out*.println()**;** }  
  
 public static void main(String[] args) {  
 Integer[] intArray={**1,2,3,4,5**}**;** String[] stringArray={"Jane"**,**"Tom"**,**"Bob"}**;** Character[] charArray={'a'**,**'b'**,**'c'}**;** *listAll*(intArray)**;** *listAll*(stringArray)**;** *listAll*(charArray)**;** }  
}

3. What is a raw type? Why is a raw type unsafe? Why is the raw type allowed in Java?

Raw type is generic type that is used without specifying a type parameter for it. When a generic type is used without specifying its type parameter, it is considered a raw type.

List myList = new ArrayList(); // Raw type

myList.add("Hello"); // Valid, but not type-safe

myList.add(123); // Valid, but not type-safe

String firstElement = (String) myList.get(0); // Type cast required, may result in runtime error

List<String> list = new ArrayList<>(); //not raw type

Raw type is unsafe because they bypass the type-checking mechanisms of the compiler. This can lead to type errors at runtime, which can cause unexpected behavior or crashes in the program. When using raw types, the code can fail silently and cause hard-to-find bugs.

*Raw type allowed in Java to allow backward compatibilities for legacy code since generics are introduced only in JDK 5.*

*Backward compatibility is the ability of a system or a component to accept input or data from earlier versions of itself or from other systems, and to still function without any negative impact. In the context of programming languages, backward compatibility means that a newer version of a language can still run programs written in an older version of the same language. In Java, backward compatibility means that newer versions of Java can still run programs written in older versions of Java without any modification or recompilation.*

Raw type allowed in Java used to avoid raw types whenever possible and to use generics instead, as generics provide type safety and make the code easier to understand and maintain. When using raw type is necessary, it is important to be careful and to ensure that the code is type-safe by manually checking the types of the objects being used.

4. What is erasure? Why are Java generics implements using erasure?

*The implementation of generic where compiler erases the generic type during run time after using generic information during compile time.*

*Why? This approach enables the generic code to be backward-compatible with the legacy code that uses raw type.*

Erasure is the process by which Java’s generic type information is removed at compile time. When you define a generic class or method in Java, the compile generates bytecode that works with the generic type parameter specified in the source code.

The reason is to maintain backward compatibility with pre-Java 5 code that did not use generics. By erasing the generic type information at compile time, the JVM can still execute code that uses generic type in a way that is compatible with code that does not use generics.

Also, it used to reduce the size of generated bytecode and improve performance. By erasing generic type information at compile time, the generated bytecode is smaller and more efficient, as there is less overhead associated with dealing with the generic type.

5. Create a generic class named Duo that has two parameters, A and B. Declare a variable named first of type A, and the second variable named second of type B. Create a constructor that accepts these two parameters. In the constructor, assign these parameters respectively to the declared variables.

public class Duo<**A,B**>{  
 private **A** first**;** private **B** second**;** Duo(**A** first**, B** second){  
 this.first=first**;** this.second=second**;** }

* 1. 6. Use the Duo class in Question 5 to declare and create two objects as follows :
  2. a. First object called sideShape consist of respectively String type and Integer type.
  3. b. Second object called points consists of two Double types.

public static void main(String[] args) {  
 Duo<String**,**Integer> sideShape=new Duo<>("shape"**,1**)**;** Duo<Double**,**Double> points=new Duo<>(**1.0,1.0**)**;**}

1. 7. Assume that the following objects were created
2. ArrayList<String> vehicle = new ArrayList<>();
3. ArrayList<Object> transportation = new ArrayList<>();
4. Declare a method header for generic method,allTransportation that returns nothing,which accepts two ArrayList parameters using the wildcards.

public static void allTransportation(ArrayList<? extends T> list1**,** ArrayList<?> list2){  
  
}

be **static void** because the method does not need to return anything and it is not required to have an instance of the class to be executed.

1. 8. Assuming that two new object are created as follows
2. ArrayList<Integer> numOfCars = new ArrayList<>();
3. ArrayList<Double> milesPerHour = new ArrayList<>();
4. Using the <?> wildcard, implement a generic method that displays the list.

public static void display(ArrayList<? extends Number> list){  
 for(Number num: list){  
 System.*out*.print(num + " ")**;** }  
 System.*out*.println()**;**}

public static void main(String[] args){

ArrayList<Integer> numOfCars = new ArrayList<>();

ArrayList<Double> milesPerHour = new ArrayList<>();

for(int i=0; i<5; i++){

numOfCars.add((int)(Math.random()\*6));

milesPerHour.add(Math.random()\*100+20);

}

display(numOfCars);

display(milesPerHour);

}

1. 9. When the compiler encounters a generic class, interface, or method with an unbound type parameter, such as <T> or <E>, it replaces all occurrences of the type parameter with what type?

With Object.

Every class has an Object as superclass

public class MyGenericClass<T> {

private T data;

public void setData(T data) {

this.data = data;

}

public T getData() {

return data;

}

}

MyGenericClass<String> stringClass = new MyGenericClass<>();

stringClass.setData("Hello, world!");

String data = stringClass.getData();

那个就是stringclass=object

MyGenericClass<String> stringClass = new MyGenericClass<>();

stringClass.setData("Hello, world!");

String data = stringClass.getData();)

1. 10. When the compiler encounters a generic class, interface, or method with a bound type parameter, such as <T extends Number> or <E extends Comparable>, it replaces all occurrences of the type parameter with what type?

Replace with Bounded type, that is Number and Comparable.

the type parameter "T" is an unbounded type parameter, which means that it can be any class or interface.

type parameter "T" is bounded by the "extends Number" clause, which means that "T" can only be a type that is a subclass of the "Number" class or that implements the "Number" interface.

*So when we change T into Number, does we still need to write <T extends Number> ?*

*Yes, even if you change T to Number in the method signature, you still need to include the generic type parameter declaration <T extends Number> at the beginning of the class or method declaration. This is because the generic type parameter declaration specifies the upper bound for the type parameter T, which is necessary for the compiler to enforce the type constraints specified by the programmer.*

*Using* ***T extends Number*** *is more specific and allows us to enforce that the type parameter must be a subclass of* ***Number****. The* ***?*** *wildcard, on the other hand, represents any type, which means it is less specific and does not provide the same level of type safety.*

*In addition, using* ***T extends Number*** *allows us to access methods and fields specific to* ***Number*** *in our generic code, which would not be possible if we used the* ***?*** *wildcard.*