Java Theory Digital Assignment.

Mansi D. Singh
19BIT0069
G2+TG2.

Q1. Make a study on Generics in Java. Give one example for Generic class and Generic Method:

Ans: What are Generics in Java?

Generics in Java is a language feature that allows the use of specified types and methods for not creating anomalies and preventing subsequent errors by predefining datatypes. These differ from the regular datatypes and methods by forcing a person to store a specified type of object.

These allow types Integer, String, or even user defined types to be passed as a parameter to classes, methods and interfaces. Any entity such as class, interface or method that operates on a parametrized type is a called a generic entity.

Introduction to Generics in Java:

Java Generics were introduced in JDK 5.0 with the aim of reducing bugs and adding an extra layer of abstraction.

Before Generics, we could store any type of objects in the collection, i.e. non-generic too. Now this can give unpreecendented errors.

Why do we need Generics?

For this, let us start off with a simple code snippet where in we want an Integer to be returned:

```
import java.util.Arraylist;

public class mansi_genericsexplanation {

    // Creating a class
public static void main (String[]
    args) {

    Arraylist arraylist = new Arraylist();

// Here Arraylist is the name of the class.
Visual Studio generally assigns the
name arraylist to another class.

    arraylist.add("mansi");
    arraylist.add(2);
    arraylist.add("02/10/2000");

    int retrieve = arraylist.get(1);

// This shows that you want to retrieve
```

the element at the position 1, (which is an integer).

// However, when we try running this, the compiler complains about the last line and gives a redline error. It states that it is returning an object wherein we need an integer. We, on the other hand as a programmer believe that we gave in an integer only..

// Now we can solve the problem by changing the last line by performing typecasting on it

```
int retrieve = (int) arrayList.get(1);
```

// But redundantly specifying the int was not really needed here. Now if by some means, we tell the compiler that Integer is specified for this beforehand, it will be explicitly introduced.

This is where Generics come into picture.

~~This is the~~

Explaining this in a detailed format:

```
Integer retrieve = (Integer) arrayList.get
                                    (1);
```

There is nothing here to guarantee that the return type of this list is an Integer. It will give an output at

runtime, but the defined list could hold any object. We only know that we are retrieving a list by inspecting the context. When looking at types it can only guarantee that it is an Object and thus requires an explicit cast to ensure that the type is safe.

This cast can be cluttering the code and even has the redundant value.

It is easier if programmer can express his/her intention of using specific types and compiler can ensure correctness of such type. This could also help one in avoiding runtime errors if programmer makes a mistake with the explicit casting.

If we modify the previous code snippet to a diamond operator <> containing the type we narrow the specialization ~~list<Integer>~~ of the list only to Integer type i.e. we specify the type to be held in the list. This is similar to c++ type creation.

What are Types of Generics in Java?

1. Generic Method: Generic Java method takes a parameter and returns some value after performing a task. This is exactly like a normal function. However a generic method has type parameters which are cited by the actual type and the scope is limited to the method where it is declared.

Properties of Generic Methods:

1. Generic methods have a type parameter (the diamond enclosing the type) before the return type of the method declaration. For static generic methods, the type parameter section must appear before the method's return type.
2. Type parameters can be bounded (explained further).
3. Generic Methods can have different type parameters separated by commas in the method signature.
4. Method body for a generic method is just like a normal method.

~~counter (elements in~~

Writing this for searching the presence of a number in an array. This can work for a collection of sets (Int, Float, Double). Although we try searching specifically for Int here.

```
public class Sample jawantd
{
    public static <T> boolean count(
T[] array, T item){
    int count = 0;
    boolean res = false;
    if (item == null){
    } for (T element : array){

    if (element == null)
        res = true;
    }
}
else { for (T element : array){
    if (item.equals(element)){
        res = true;
    }
}
return res;
}
    public static void main (String args[])
    {
    Integer [] integers = { 1, 2, 34, 864, 9
100, 32};
    boolean ans = count (integers, 9);
```

System.out.println("Occ. of "+g+" in array"+ans);

Output: Occurence of 9 in array: false.

Boundedness in Generic Methods.

In general, the type parameter can accept any data types except primitive types.

However, if we want to use generics for some specific types only, we can use bounded types. In case of bound types, we use the extends keyword, i.e. we can specify that a method accepts a type and all its subclasses (upper bound) or a type and all its superclasses

Eg: <T extends A> indicates that T can only accept data that are subtypes of A. For example:

Considering the widely used example of invoking and implementing the method that counts the no. of elements in an array T[] that are greater than a specified element elem.

```
public static <T> int
countGreaterThan(T[] anArray,
T elem){
    int count = 0;
    for(T e: anArray)
        if (e > elem)
            ++count;
        return count;
}
```

The above method gives out a compiler error. This is because the greater than operator applies only to primitive types (such as short, int, double, long, float, byte and char. We cannot use this to compare objects. For this, we use a type parameter bounded by Comparable<T>.

```
public interface Comparable <T> {
    public int compareTo (T o);
}
```

The resulting code will be:

```
public static <T extends comparable
<T>> int
countGreaterThan(T[] anArray, T
elem) {
    int count=0;
    for (T e: anArray)
    if (e.compareTo(elem)>0)
        ++count;
    return count;
}
```

~~2. Generic Classes in Java.~~

2. Multiple Bounds in Java.

A type can also multiple upper bounds as follows:

<T extends Comparable and Number>.

2. Generic Classes in Java-

A class that can refer to any type is a generic class. These variables are known as parameters of the Java class.

Taking a famous example to understand this which can be standardized for our upcoming example also:

```
class Generic class {
private Object x;
public void set (Object x) { this.x=x;}
public Object get () { return x; }
}
```

Here one class is created with property x and the type of the property is object.
Once we initialize the class with a certain type, the class should be used with that particular type only. For example, if you want 1 instance of a class to hold a value of type string then programmer should set and get only of that type
Since the property type Object is declared, there is no way to set a restriction. A programmer can set any object and expect any return value from the get method since all

Java types are subtypes of object class. The this keyword here is used for reference and when x is returned it could be of any of the aforementioned types.

Formulating an example for a Generic Class.

```
class generic_grade_company <T, V>
{
  T t;
  V u;
  generic (T x, V y)
  {
    t = x;
    u = y;
  }
  public void print()
  {
    System.out.println(t);
    System.out.println(u);
  }
}
class self.
  { public static void main (String[] args)
    {
    generic_grade_company g1 = new genric_
grade_company (1, "Mansi");
generic_grade_company g2 = new
genric_grade_company ("Nokia", 5.8);
    g1.print();
    g2.print();
  }
}
```

output: 1 Mansi Nokia 5.8.

Page No.
Date

Advantages of Generics in Java:

1. Code Reusability: A strategy or class or interface can be composed once and be used for any type / way that one requires. For eg: A programmer writes a generic method for sorting an array of objects. Generics allows the programmer to use the same method for Integer Arrays, Double and String arrays.

2. Type Safety: Whenever we used different types of arguments and whenever we casted them in different types of arguments they were safe. There was no hampering of the code. Compile-time safety is ensured as the programme can catch invalid types while compiling. Also, individual type casting is not required. The programmer defines the type and then lets the code do its job.

3. It allows us to implement non-generic algorithms:

Q2. Make a study on Annotations in Java

What are Java Annotations?

Java Annotations are tags that represent the metadata attached with class, interface, methods or fields to indicate some additional information which can be used by Java compiler and JVM. These have no effect on the execution of the program and were added in

What are some Built-In Java Annotations?

Built-In Java Annotations used in Java Code

1. @ Override
2. @ SuppressWarnings
3. @ Deprecated

Built-In Java Annotations used in other annotations

1. @ Target
2. @ Retention
3. @ Inherited
4. @ Documented

What is the use of the aforementioned Annotations.

1. Instructions to the Compiler: The 3 built-in annotations available in Java

(used in the Java Code) are used for giving certain instructions to the compiler.

2. Compile-time instructors: Annotations can provide compile-time instructions to the compiler that can be further used by software build tools for generating code, XML, Files etc.

3. Runtime instructions: We can define annotations at runtime which we can access using java reflection and can be used to give instructions at runtime.

Understanding annotations:

@ Override: Indicates that a method overrides/replaces an inherited method. This information is not strictly necessary but helps to reduce mistakes. If we want an override method but have a simple type in signature, or the wrong argument type, the error might go unnoticed. But if we provide an @ override annotation, the compiler makes sure we actually override a method and not just accidentally add/overload it.

```
public class Vehicle {
    public void javaoverriding() {
        System.out.println("Parent class");
    }
}
public class Bike extends Vehicle {
    @Override
    public void javaoverriding() {
        System.out.println("Child Class");
    }
}
```

@Deprecated:
Another compile-only annotation. A code is marked as deprecated, and the compiler / IDE can access this information to tell us the code isn't supposed to be used anymore. Since Java 9, this previous marker annotation has become a configuration annotation. The values String since() default ""; and boolean forRemoval() default false were provided with even more info for compilers and IDE to work with.

Eg.

```
/**
 * @deprecated
 * reason
 */
@Deprecated
public void Method() {
```

// task.

6.

Now whenever any program would have to use this method the compiler would generate a warning.

@ Suppress Warnings:
A configuration annotation, accepting an array of warning names that should be disabled during compilation.

@ suppress warnings ("deprecation")
void myMethod() {
myobject.deprecatedMethod();

Built-In Java Annotations:

@Target: Not every annotation makes sense on every available target and thus the acceptable targets can be explicitly set. The 8 available targets defined are:

→ Element Type. PACKAGE.
→ Element Type. TYPE
→ Element Type. TYPE_PARAMETER.
→ Element Type. TYPE_USE.
→ Element Type. ANNOTATION_TYPE.
→ Element Type. CONSTRUCTOR.
→ Element Type. FIELD.
→ Element Type. METHOD.
→ Element Type. LOCAL_VARIABLE.

Syntax:

@ Target ({ ElementType.FIELD, ElementType.TYPE }).

@ Inherited
Annotations are not inherited by default. By adding @ Inherited to an annotation type, we allow it to be inherited.
This only applies to annotated types declared as which we will pass down to their subtypes.

@ MyAnnotation (value = 2)
public class MyType { ... }.

// Any annotation check at runtime

public class MySubType extends MyType { ... }.

@ Documented
Java's default behaviour is to ignore any annotation. With @ documented we can change this, making metadata and values accessible through documentation.

@ Retention.
With the help of @ Retention, we can now declare an annotation repeatable by providing an

intermediate annotation:

```
public @interface MyRetentionAnno{
    MyRetentionAnno[] value();
}
```

```
@Retention(MyRetentionAnno.class)
public @interface MyRetention
    Annotation{
    String value() default "";
}
```

How to Create Custom Annotations?

→ Annotations are created by using @ interface, followed by annotation name.
→ It can have elements known as metho
→ All annotations extend java.lang.annotation. Annotations cannot include any extends clause.