Vishal Govathoti

S555276

Link to GitHub repo : https://github.com/VishalGovathoti/GovathotiAssignment03.git

1. **What are generics?**

Generics make it possible to design classes and methods that may be used with various forms of data without having to write unique code for each type.

**package** Question1;

**public** **class** genricMethod {

**public** **static** <Demo> **void** printArray(Demo[] input) {

**for**(Demo print: input) {

System.***out***.print(print + " ");

}

System.***out***.println();

}

**public** **static** **void** main(String[] args) {

String[] s = {"Vishal", "Govathoti"};

System.***out***.println("Method Generic has: ");

*printArray*(s);

}

}

**package** Question1;

**public** **class** genrics<s> {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

genrics vishal = **new** genrics();

vishal.setName("Govathoti");

System.***out***.println(vishal);

}

**private** s name;

**public** s getName() {

**return** name;

}

**public** **void** setName(s name) {

**this**.name = name;

}

@Override

**public** String toString() {

**return** "name= " + name;

}

}

1. **Can we change the scope of the overridden method in the subclass for private, public, default and protected? Explain how can it be changed for each scope?**

Yes, we can modify the overridden method's scope in the subclass. We must be aware, nevertheless, that we cannot make the technique less accessible.

1. Private scope: A private method in a superclass cannot be overridden since it is inaccessible to the subclass. As a result, it is not possible to alter the overridden method's scope in the subclass.

2. Default scope: If both classes are in the same package, a default (or package-private) function in a superclass can be accessed from the subclass. In the subclass, the scope of the overridden method can be modified to default, public, or protected, but not to private.

3. Protected scope: As long as the subclass is a subclass of the superclass, a protected method in a superclass can be accessed from a subclass, even if they are in separate packages. The overridden method's scope can be modified in the subclass to protected or public but not to default or private.

4. Public scope: There is no need to override the scope of a public method in a superclass because it can be accessed from anywhere in the application. The overridden method's scope can be modified to any of the four access modifiers (private, default, protected, or public) in the subclass, though.

**3.What is the covariant return type?**

The term "covariant return type" refers to a Java feature that enables a subclass method to return a type that is more specialized than the type returned by the method that was overridden in the superclass. In other words, it enables a subclass method to restrict the return type of the superclass method that it overrides.

**package** question3;

**import** question3.vishal;

**import** question3.vishalclass;

**public** **class** vishal **extends** vishalclass {

**public** vishal getDetails(){

System.***out***.println(" this is the vishal class");

**return** **this**;

}

**public** **void** student() {

System.***out***.println("vishal is a good student");

}

}

**package** question3;

**public** **class** vishalclass {

**public** vishalclass getDetails(){

System.***out***.println("Hello, this is a superclass");

**return** **this**;

}

}

**package** question3;

**public** **class** vishalDriver {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

**new** vishal().getDetails().student();

}

}

**4.Can we override the static and private methods? Why?**

The answer is negative, since method overriding in Java depends on dynamic binding at runtime, whereas static methods are bound with static binding at compile time. When it comes to overriding private methods, however, we cannot access them outside of the class because the scope of private methods is restricted to the class.

1. **Difference between String Buffer and StringBuilder?**

\*String Buffer is less effective and synchronized.

\*StringBuilder is more effective because it is not synced.

\*In order for two or more threads to call methods simultaneously, a string buffer must be thread safe.

\*A string builder must not be thread safe in order for two threads to call methods simultaneously.

**package** question5;

**public** **class** StringBuffer {

StringBuffer sb = **new** StringBuffer();

sb.append("Hello");

sb.append(" ");

sb.append("world!");

String message = sb.toString();

System.out.println(message);

}

**package** question5;

**public** **class** StringBuilder {

**public** StringBuilder(String input2) {

// **TODO** Auto-generated constructor stub

}

String input = "Hello, world!";

StringBuilder sb = **new** StringBuilder(input);

sb.reverse();

String reversed = sb.toString();

System.out.println(reversed);

}

**6. Difference between String class and String Buffer?**

**Class String:**

\*The String class cannot be changed.String Buffer is a changeable class.

\*The equals() method of the Object class is replaced by the String class. Therefore, you can use the equals() method to compare the contents of two strings.The equals() function of the Object class is not overridden by the String Buffer class.

\*String constant pool is used by the String class.Using heap memory is String Buffer.

\*When we concatenate too many strings, string is slow and uses more memory because it creates a new instance each time.When we concatenate t strings, String Buffer is quick and uses less memory.

\*While performing a concatenation operation, the String class is slower.Concatenation operations using the String Buffer class are faster.

**String Buffer:**

\*String Buffer is a changeable class.

\*The equals() function of the Object class is not overridden by the String Buffer class.

\*Using heap memory is String Buffer.

\*When we concatenate t strings, String Buffer is quick and uses less memory.

\*Concatenation operations are faster when done with the String Buffer class.

**7.Can we declare constructor as final?**

No, in Java, a constructor cannot be declared to be final. When a variable or procedure cannot be changed or overridden, it is indicated by the final keyword. Contrarily, constructors cannot be overridden in the conventional sense because they are used to build objects and initialize their states.

**package** question7;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** hashmap {

**public** **static** **void** main(String[] args) {

Map<String,Integer> vishal = **new** HashMap<String,Integer>();

vishal.put("Neha", 33);

vishal.put("teju", 65);

vishal.put("raja rao", 10);

vishal.put("munikumari", 56); Yes, by utilizing a finally block, we can have a try without a catch block.

vishal.put(**null**, **null**);

System.***out***.print("Elements in Hashtable are:\n " + vishal.entrySet());

System.***out***.println(vishal.get("Neha"));

System.***out***.println(vishal.get("teju"));

}

}

**8. Can we have try without catch block in java?**

Yes, by utilizing a finally block, we can have a try without a catch block.

**package** question8;

**public** **class** TryWithoutCatch {

**public** **static** **void** main(String[] args) {

**try** {

System.***out***.println("vishal ");

} **finally** {

System.***out***.println("Yes, by utilizing a finally block, we can have a try without a catch block. ");

}

}

}

1. **What is try with the resource?**

In Java, a try statement that declares one or more resources is known as a try-with-resources statement. When your program is finished utilizing an object, it must be closed, which is known as a resource. a File resource or a Socket connection resource, for instance. At the conclusion of the statement execution, the try-with-resources statement makes sure that each resource is closed. If we don't seal the resources, it might be a resource leak and the program might use up all of the resources at its disposal.

**10. Can we modify the throws clause of the superclass method while overriding it in the subclass?**

Yes, while overriding the superclass method in the subclass, we can change the throws clause. However, there are several guidelines that must be observed when overriding in the case of handling exceptions.

If the overridden method of the subclass declares an exception but the superclass method does not, the overridden method of the subclass may only declare the unchecked exception.

If the superclass method declares an exception, the overridden subclass method may do the same, declare a subclass exception, or declare none at all, but not a parent exception.

**package** question10;

**public** **class** ExceptionDemo {

**public** **void** sampleMethod() {

System.***out***.println("Subclass method");

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

ExceptionDemo vishal = **new** ExceptionDemo();

vishal.sampleMethod();

}

}

**package** question10;

**import** java.io.FileNotFoundException;

**public** **abstract** **class** OverridingSuper {

**public** **void** sampleMethod()**throws** FileNotFoundException

{

System.***out***.println("superclass method");

}

}

11.What is an association, aggregation, and composition in UML?

In a UML diagram, a relationship is used to depict the link between different objects.

**Association**: An association is a relationship between two classifiers that explains the factors that led to the link as well as the guidelines that control it, such as classes or use cases. An association is a structural connection between two classifiers. Similar to attributes, associations keep track of classifiers' characteristics. Using an arrow, association relationships are indicated.

**Aggregation:** A straight line with an empty arrowhead at one end is used to represent an aggregation relationship. A relationship of affiliation includes aggregation. A weak form of association is said to be aggregation. Objects that are related to one another can still be included in the scope of a system when there is an aggregate relationship between them. The linked things are independent of one another. It's also referred to as a "has-a" relationship.

**Composition**: An arrowhead is filled at one end of a straight line to indicate a composition relationship. A connection of association includes the composition. The composition is thought to be a powerful kind of association. Objects related with one another in a composition relationship cannot continue to be included in the scope without one another**.**

**12. Difference between final, finally and finalize()?**

\*The keyword "final" is used to stop a variable from being changed.

\* After a try-catch block, code is executed using the finally block.

\* Before an item is destroyed, the garbage collector calls a function called finalize().

**package** question12;

**public** **class** Final {

**public** **final** **int** pay = 1050;

**public** **void** display()

{

System.***out***.println("pay :" +pay);

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Final v = **new** Final();

v.display();

}

}

**package** question12;

**public** **class** Finalize {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Finalize v = **new** Finalize();

v = **null**;

System.*gc*();

System.***out***.println("Result from primary method");

}

@Override

**protected** **void** finalize() {

System.***out***.println("output from the finished technique");

}

}

**package** question12;

**public** **class** Finally {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

**try** {

System.***out***.println("During the try block");

**int** information=60/0;

System.***out***.println(information);

}

**catch** (ArithmeticException e){

System.***out***.println("Mathematical Error Occurred "+ e);

}

**finally** {

System.***out***.println("During the try block");

}

}

}

13.Difference between Vector and ArrayList?

**ArrayList:**

\* The array list is not synced.

\* If the array's capacity is reached, ArrayList increases the array size by 50%.

\* Multiple threads are permitted.

\* It is not a legacy class, ArrayList. It first appears in JDK 1.2.

\* Since ArrayList is not synchronized, it is quick.

\* High performance for ArrayList.

**Vector:**

\* The vector is coordinated.

\* If the total number of elements exceeds the array's limit, a vector increment of 100% signifies that the array size is doubled.

\* One thread only is permitted.

\* The legacy class Vector

\* Vector is sluggish because it synchronizes, keeping other threads in a runnable or non-runnable state in a multithreading environment until the current thread releases the lock of the object.

\* Low vector performance

**14. What are the different ways to make ArrayList methods synchronized?**

* The synchronized keyword, the Collections.synchronizedList() method, or the CopyOnWriteArrayList class can all be used to synchronize ArrayList functions.

**\*** A synchronized list supported by the supplied list is returned by the Collections.synchronizedList() function**.**

**package** questio14;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** ArraySync {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

List<String> l = **new** ArrayList<String>();

l.add("vishal");

l.add("govathoti");

l.add("Tenali");

l = Collections.*synchronizedList*(l);

**synchronized**(l) {

Iterator<String> itr = l.listIterator();

**while**(itr.hasNext()) {

System.***out***.println(itr.next());

}

}

}

}

**package** questio14;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** ArraySyncMethod2 {

**public** **static** **void** main(String[] args) {

ArrayList<String> data = **new** ArrayList<String>();

data.add("vishal");

data.add("govathoti");

data.add("tenali");

System.***out***.println("Syncronized arraylist elements are :");

Iterator<String> itr = data.iterator();

**while** (itr.hasNext())

System.***out***.println(itr.next());

}

}

**15.Difference between Hash table and Hash Map?**

**Hash Map:**

\*There is no synchronized technique.

\* Since multiple threads can run concurrently, the hashmap object is not thread-safe.

\* Since threads don't have to wait, performance is great in comparison.

\* Both the value and the key may be null.

\* It has no legacy.

**Hash Table:**

\*Each technique is synchronized.

\* Only one thread may manage the Hashtable's object at once. It is therefore thread-safe.

\* Performance suffers as a result of the thread's increased waiting time.

\* Keys and values cannot both be null. Otherwise, a null pointer exception will be thrown.

\* It has a history.

**16. In Java 8, explain how Hasp Map internally works?**

The HashMap class in Java 8 stores key-value pairs internally in an array of linked lists. A linked list of nodes is contained in each bucket that makes up an element of the array. In the linked list, each node is an item that has a key-value pair.

The hash function is used by the HashMap class to determine the bucket's index for a given key. The hash function uses a bitwise operation to transform the key's hash code into an index that falls inside the bounds of the array size. The bucket, which houses the linked list of nodes, is accessed using the resulting index.

**package** question16;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** HashMap1 {

**public** **static** **void** main(String[] args) {

Map<String,Integer> m1 = **new** HashMap<String,Integer>();

m1.put("vishal", 47);

m1.put("govathoti", 443);

m1.put("tenali", 12);

m1.put("guntur", 26);

m1.put(**null**, **null**);

System.***out***.print("Elements in Hashtable are:\n " + m1.entrySet());

System.***out***.println(m1.get("vishal"));

System.***out***.println(m1.get("govathoti"));

}

}

**17.Difference between fail fast and fail-safe iterator?**

Two methods for handling concurrent updates of collection objects during iteration using iterators are fail-fast and fail-safe in Java.

**Fail-Fast Iterator**: If a fail-fast iterator discovers that the collection has been updated while it is iterating, it instantly throws a ConcurrentModificationException. This means that an iterator will throw an exception to signal that the collection has been modified and is no longer acceptable if any structural changes are made to the collection (such as adding, removing, or changing elements) while an iterator is iterating over it.

**Fail-Safe Iterator:** If the collection is updated while the iteration is in progress, a fail-safe iterator does not raise a ConcurrentModificationException. Instead, it creates a copy of the collection before iterating, using the copy as the starting point for subsequent iterations.

**18.Can we start the thread twice?**

We are unable to begin a thread twice. A thread cannot be restarted once it has finished running. It will raise the IllegalThreadStateException error.

Graphical user interface, text, application

Description automatically generated

**package** question18;

**public** **class** ThreadsDemo **extends** Thread {

**public** **void** run(){

System.***out***.println("progress...");

}

**public** **static** **void** main(String args[]){

ThreadsDemo v1=**new** ThreadsDemo();

v1.start();

v1.start();

}

}

**19.What are the different ways to create a thread in java? Which one is preferred?**

**Extending the Thread class:** A class that extends the Thread class and overrides the run() method can be used to generate a thread. The thread execution can then be started by creating an instance of that class's object and calling its start() function**.**

**Implementing the Runnable interface**: By defining the run() method and implementing the Runnable interface, you can start a thread. The thread execution process can then be started by creating a Thread object, giving in an instance of the class that implements Runnable, and calling the start() method.

The Runnable interface should be implemented if you want to create a thread in Java. Due to Java's restriction on single inheritance, extending the Thread class prevents you from extending any other classes. Additionally, Runnable allows you to decouple the thread's activity from its thread identity, which can help your code be more modular and simpler to test.

**package** question19;

**public** **class** Runnablee **implements** Runnable{

**public** Runnablee(){

}

**public** **void** run() {

System.***out***.println("thread started.");

}

}

**package** question19;

**public** **class** ExtendsThread **extends** Thread {

**public** **void** run(){

System.***out***.println("thread started.");

}

}

**20. What are the different states a thread will go through?**

In java there are 6 states that the thread go through

**NEW**: The thread is in a new state, which means it has been created but has not yet started running.

**RUNNABLE:** The thread is in a runnable state, which means it has been started and is ready to run, but it may not be currently executing because the operating system has temporarily stopped it to give another thread a chance to run.

**BLOCKED:** The thread is in a blocked state, which means it is waiting for a lock to be released by another thread so it can enter a synchronized block or method.

**WAITING:** The thread is in a waiting state, which means it is waiting indefinitely for another thread to perform a specific action.

**TIMED\_WAITING**: The thread is in a timed waiting state, which means it is waiting for a specified period of time for another thread to perform a specific action.

**TERMINATED**: The thread is in a terminated state, which means it has completed execution and has either exited normally or thrown an exception.

**package** question20;

**import** question20.MyThread;

**public** **class** Threadstates {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

// **TODO** Auto-generated method stub

MyThread t = **new** MyThread();

Thread t1 = **new** Thread(t);

System.***out***.println("t1 state: " + t1.getState());//new

t1.start();

//runnable

System.***out***.println("t1 state before main sleep : " +t1.getState());

Thread.*sleep*(1000);

System.***out***.println("t1 state after main sleep : " +t1.getState());//TIMED\_WAITING

Thread.*sleep*(2000);

System.***out***.println("t1 state after 2 main sleep : " +t1.getState());//terminated

}

}

**class** MyThread **implements** Runnable{

**public** **void** run() {

**try** {

Thread.*sleep*(2500);

System.***out***.println("t1 in run(), state: " + Thread.*currentThread*().getState());

}**catch**(InterruptedException e) {

e.printStackTrace();

}

}

}

**21. What is Serialization? How do we achieve it?**

**\*** An object is transformed into a stream of bytes known as serialization so that it can be stored or communicated. By implementing the Serializable interface, it may be done in Java.

\* An object is transformed into a stream of bytes known as serialization, which can then be deserialized to recreate the original object and be written to a file or transferred over a network. Java's java.io library is used to implement serialization.interface with serialization.

\* To make an object serializable in Java, you simply need to implement the Serializable interface. This interface does not define any methods, but serves as a marker that the object can be serialize.

**22. What is immutable class? Is String class immutable?**

A class is considered immutable if its instances can never be changed after they are created. In other words, an object that is immutable maintains its state during the course of its existence. An immutable object's state cannot be altered once it has been created. Any effort to change its state will result in the creation of a new object with the altered state and leave the original object unaltered.

By having the class final and its fields private and final, we may build an immutable class. Public getters are also required in order to access the fields. Making the fields final ensures that they can only be given a value once when creating the object.

**package** question22;

**public** **final** **class** Immutablee {

**private** String name;

**private** **int** age;

**public** Immutablee(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** **void** Immutablee1(String name2, **int** age2) {

// **TODO** Auto-generated constructor stub

}

// getter method

**public** String getName() {

**return** name;

}

**public** **int** getAge() {

**return** age;

}

}

**package** question22;

**public** **class** vishalDriver {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Immutablee object1 = **new** Immutablee("vishal", 22);

System.***out***.println("Name: " + object1.getName());

System.***out***.println("Age: " + object1.getAge());

}

}

**23. Do immutable classes thread safe? If yes then how?**

Yes, immutable classes are by definition thread-safe. An immutable object's state cannot be changed after it is created, hence there is no chance of thread interference or synchronization problems.

A new instance is created whenever an immutable class is modified, making it safe to share them among many threads. This means that there is no danger of race situations or other thread-related problems when several threads read from the same immutable object.

**package** question23;

**public** **class** ImmutableThread {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

String name = "VISHAL";

name.concat("GOVATHOTI");

System.***out***.println("name : " +name);

Thread v1 = **new** Thread(**new** Runnable() {

**public** **void** run() {

name.concat("v1");

System.***out***.println("Thread: " +name);

}

});

Thread v2 = **new** Thread(**new** Runnable() {

**public** **void** run() {

name.concat("v2");

System.***out***.println("Thread1 : " +name);

}

});

v1.start();

v2.start();

}

}

**24. Can we call the garbage collector explicitly? Will it trigger the garbage collector?**

By using the System.gc() function in Java, you can ask for the garbage collector to start working, but there is no assurance that it will do so right away.

The System.gc() method only serves as a suggestion to the JVM that the garbage collector should start; the JVM is free to choose when the garbage collector should actually start. Depending on the system load and other variables, the JVM may decide to disregard the suggestion entirely or it may choose to delay the trash collection operation for a while.

**package** question24;

**public** **class** Garbagee {

**protected** **void** finalize() **throws** Throwable{

System.***out***.println("Finalize called,hence garbage collector triggered");

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Garbagee v1 = **new** Garbagee();

System.***out***.println("Calling garbage collector before making null");

System.*gc*();

v1 = **null**;

System.***out***.println("Calling garbage collector after making null");

System.*gc*();

}

}

**25.What are Java 8 features? Explain all of them with examples?**

Diagram

Description automatically generated

**An efficient technique to represent anonymous functions is with lambda expressions. The lambda phrase that follows, for instance, provides a function that accepts two numbers and returns their sum.**

**A potent method for carrying out bulk operations on data collections is offered by the Stream API. The code that follows, for instance, makes advantage of the Stream API to calculate the average of a list of values.**

**Interfaces with precisely one abstract method are said to be functional interfaces. They frequently work in tandem with lambda expressions. the java.util.function, as an illustration.A boolean-valued function of one argument is represented via the functional interface known as the predicate interface. Here is an illustration of how to use a lambda expression in a predicate.**

\* Interfaces with default methods offer a mechanism to expand an interface's functionality without disrupting existing implementations. Stream(), forEach(), and removeIf(), for instance, are only a few of the basic methods defined by the java.util.Collection interface.

\* An existing method can be referred to directly using method references. For instance, the code below employs a method reference to arrange a list of strings in alphabetical order:

\* A contemporary, type-safe method of representing dates, timings, and durations is offered by the Date-Time API. For instance, the code that follows uses the Date-Time API to produce a LocalDate object that represents the current date:

\* In JDK 8, the Nashorn JavaScript engine was released. Nashorn allows us to run JavaScript code on the Java Virtual Machine. In JDK 8, Nashorn is included to replace Rhino, the current JavaScript engine. In terms of performance, Nashorn outperforms Rhino. The Nashorn is more well-known in JDK 8 due to the uses of invoking dynamic features, conversion of JavaScript code into bytecode directly into memory, etc. JavaScript code can be run using the command-line tool or by being inserted into the Java source code.

**package** question25;

**public** **interface** Default {

**public** **default** **void** say(){

System.***out***.println("Hello, this is default method");

}

// Abstract method

**public** **void** sayMore(String msg);

}

**package** question25;

**public** **class** DefaultMethod **implements** Default{

@Override

**public** **void** sayMore(String msg) {

System.***out***.println(msg);

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

DefaultMethod assignment = **new** DefaultMethod();

assignment.say(); // calling default method

assignment.sayMore(" for practice "); // calling abstract method

}

}

**package** question25;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ForEachDemo {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

List<Integer> salaries = **new** ArrayList<Integer>();

salaries.add(1000);

salaries.add(5000);

salaries.add(9000);

salaries.add(9000);

salaries.add(1000);

System.***out***.println("forEach Method");

salaries.forEach(salary -> System.***out***.println(salary));

}

}

**26.How to make a pure singleton?**

The Singleton class's function is to regulate object creation and enforce a one-object maximum. The singleton class' main job is to control the creation of new instances or objects while limiting the overall number of objects to one. The simplest way to create a Singleton class is to generate an instance of it at the time the class is loaded. Making the constructor private prevents other classes from creating new instances of the class for which the singleton is intended.

**package** question26;

**public** **class** Singleeeton {

**private** **static** **volatile** Singleeeton *obj* = **null**;

**private** Singleeeton() {

}

**public** **static** Singleeeton getInstance() {

**if**(*obj*==**null**) {

**synchronized** (Singleeeton.**class**) {

**if**(*obj*==**null**) {

*obj* = **new** Singleeeton();

}

}

}

**return** *obj*;

}

}

**27. How to make a singleton synchronized?**

**A block of code cannot be run concurrently by multiple threads thanks to synchronization.The singleton class can be made threadsafe and synchronized using the getInstance() method, and the synchronized keyword must be added to the method definition.**

**More exactly, the first thread will use the Singleton class to get a lock, run the method, make an instance of the class, and then return the monitor. Once the Singleton class's key monitor is available for use, the second thread will accept it, execute the function, get the instance that has already beean generated, and then return the key monitor.**

**package** question27;

**public** **class** SingletonSynchronized {

**private** **static** SingletonSynchronized *instance* = **null**;

**private** SingletonSynchronized() {}

**public** **static** **synchronized** SingletonSynchronized getInstance() {

**if** (*instance* == **null**) {

*instance* = **new** SingletonSynchronized();

}

**return** *instance*;

}

}