1. What are generics?

Generics: Generics in Java is a powerful feature introduced in Java 5.0 that allows the developer to develop generic code. It enables the developer to write a single generic method or a class that can work with different kinds of data types. Generics provide the following benefits: - Type safety - Code reusability - Eliminates the risk of a ClassCastException at runtime Here's a sample code for generics:

``` public class Example<T>

{ private T value;

public Example(T value) {

this.value = value;

}

public T getValue() {

return value;

}

public void setValue(T value) {

this.value = value;

}

}

Example<Integer> example1 = new Example<>(1);

Example<String> example2 = new Example<>("Hello, World");

System.out.println(example1.getValue()); // Output: 1

System.out.println(example2.getValue()); // Output: "Hello, World" ```

2. Can we change the scope of the overridden method in the subclass for private, public,

Overriding method in subclass:

In Java, it is possible to override a method defined in the superclass by creating a method with the same name and signature in the subclass. However, the scope of the overridden method cannot be changed directly. Here are the rules for changing the scope of the overridden method: - Private methods cannot be overridden - The scope of the overridden method in the subclass should be the same or broader than the scope in the superclass.

Here's a sample code for overriding methods in a subclass:

``` class Superclass {

protected void print() {

System.out.println("Superclass");

}

}

class Subclass extends Superclass {

public void print() { // overridden method

System.out.println("Subclass");

}

}

Subclass s = new Subclass();

s.print(); // Output: Subclass ```

3. What is the covariant return type?

Covariant return type: Covariant return type is a feature introduced in Java 5.0 that allows the overridden method in the subclass to have a return type that is a subclass of the return type of the method in the superclass. The return type of the overriding method must be the subtype of the return type declared in the overridden method.

Here's a sample code for Covariant return type:

``` class Superclass {

Superclass show() {

System.out.println("Superclass method");

return this; }   
}

class Subclass extends Superclass {

Subclass show() {

// overridden method with covariant return type

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

return this; } }

Subclass s = new Subclass();

s.show(); // Output: Subclass method ```

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

Overriding static and private methods: Static methods cannot be overridden in Java. Private methods cannot be accessed outside the class, so it cannot be overridden.

Here's a sample code for overriding methods:

``` class Superclass {

public static void print() {

System.out.println("Superclass");

}

private void display() {

System.out.println("Private method"); }

}

class Subclass extends Superclass {   
public static void print() {

// overrides the superclass static method

System.out.println("Subclass");

}

//cannot override private method }

Subclass.print();

// Output: Subclass ```

5. Difference between String Buffer and StringBuilder?

StringBuffer vs StringBuilder: StringBuffer and StringBuilder classes are used to create mutable strings in Java. The main difference between them is that StringBuffer methods are synchronized, while StringBuilder methods are not synchronized. StringBuilder performs better than StringBuffer in non-threaded environments because of its non-synchronized methods.

Here's a sample code for StringBuffer and StringBuilder:

``` StringBuffer sb = new StringBuffer();

sb.append("Java")

.append(" is")

.append(" awesome");

System.out.println(sb);

// Output: Java is awesome

StringBuilder sb1 = new StringBuilder();

sb1.append("Java")

.append(" is")

.append(" awesome");

System.out.println(sb1); /

/ Output: Java is awesome ```

6. Difference between String class and String Buffer?

String class vs. StringBuffer class: String class is immutable in Java, meaning you cannot modify the string once it is created. StringBuffer class is mutable in Java, meaning you can modify the string once it is created. The following are the significant differences between String class and StringBuffer class: - String is immutable while StringBuffer is mutable - String is slow and consumes more memory if we do a lot of modifications while StringBuffer is fast and consumes less memory if we do a lot of modifications.

Here's a sample code for String vs StringBuffer:

``` String str = "Java";

str.concat(" is awesome");

System.out.println(str); /

/ Output: Java

StringBuffer sb = new StringBuffer();

sb.append("Java");

sb.append(" is");

sb.append(" awesome");

System.out.println(sb);

// Output: Java is awesome ```

7. Can we declare constructor as final?

Final constructor: A constructor can be declared as final in Java, which means we cannot inherit the class. This is because final methods cannot be overridden.

Here's a sample code for a final constructor:

``` class Test {

final Test() {

System.out.println("Test constructor"); }

}

// cannot inherit Test class

class Test1 extends Test {

Test1() { super();

// compiler error - cannot inherit final class }

} ```

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

Try without catch block: Java 7 introduced the try-with-resources statement, which allows us to declare one or more resources in a try statement. The try-with-resources statement is a better way to handle resources than the try-catch-finally block. However, we can use try block without a catch block in Java.

Here's a sample code for a try block without a catch block:

``` try (BufferedReader br = new BufferedReader(new FileReader("input.txt"))) {

String line = br.readLine();

System.out.println(line);

} catch(IOException e)

{ e.printStackTrace();

} ```

9. What is try with the resource?

Try with resources: try-with-resources is a new feature added in Java 7 that allows us to declare one or more resources in a try statement. The resources declared in the try-with-resources block are automatically closed when the try block finishes executing.

Here's a sample code for try-with-resources:

``` try (BufferedReader br = new BufferedReader(new FileReader("input.txt"))) {

String line = br.readLine();

System.out.println(line);

} catch(IOException e) {

e.printStackTrace(); } ```

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

subclass?

Modifying throws clause: In Java, when we override a method, we can only keep the same signature or a sub-set of the exception thrown by the parent method, but we cannot add more exceptions to what the parent method throws. An overridden method can throw fewer exceptions than the existing ones or none at all.

Here's a sample code for modifying the throws clause:

``` class Superclass {

void method() throws IOException {

// some code that may throw IOException }

}

class Subclass extends Superclass {   
void method() { // no exception is thrown

// some code

}

} ```

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

Association, Aggregation, and Composition: Association, Aggregation, and Composition are types of relationships in UML. The main difference between them is the ownership of objects. In UML, the following symbols represent different types of relationships:

- Association: An association is a relationship between two classes where both are dependent on each other. The association is represented by a line.

- Aggregation: Aggregation is a type of association where the child class can exist independently of the parent class. The parent class has a reference to the child class. The aggregation is represented by a line with an empty diamond at one end.

- Composition: Composition is a subtype of aggregation where the child class cannot exist independently of the parent class. The parent class owns the child class. The composition is represented by a line with a filled diamond at one end.

Here's an example of Association, Aggregation, and Composition in UML:

``` // Association

class Car { Engine engine = new Engine();

}

//Aggregation

class Car { Wheel[] wheels = new Wheel[4];

}

//Composition

class Car {

private Engine engine = new Engine();

private Wheel[] wheels = new Wheel[4];

} ```

13.Difference between Vector and ArrayList?

Final, Finally, and Finalize: Final, Finally, and Finalize are three different things in Java.

- Final is a keyword used to declare a variable, method, or class that cannot be changed or overridden.

- Finally is a block of code that is executed after the try block, regardless of whether an exception is thrown or not.

- Finalize is a method that is called by the garbage collector when it is about to destroy an object.

Here's a sample code for Final, Finally, and Finalize:

``` final class Example {

final int value = 10;

final void print() {

System.out.println("Final method"); }

}

try {

// some code that may throw an exception

} catch(Exception e) {

// exception handling

} finally {

// finally block will always execute

}

class Example {

protected void finalize() {

System.out.println("Finalize method"); }

} ```

13. Vector vs ArrayList:

Vector and ArrayList are both used to store a list of objects in Java, but they differ in synchronization, performance, and space requirements.

- Vector is a synchronized class, whereas ArrayList is not.

- Vector doubles its size when it reaches its capacity, whereas ArrayList increases its size by half.

Here's a sample code for Vector and ArrayList:

``` Vector<Integer> vector = new Vector<>();

vector.add(1);

vector.add(2);

vector.add(3);

System.out.println(vector);

// Output: [1, 2, 3]

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

arrayList.add(1);

arrayList.add(2);

arrayList.add(3);

System.out.println(arrayList);

// Output: [1, 2, 3] ```

14. Synchronize ArrayList:

Java provides three different ways to make ArrayList methods synchronized:

- Synchronized block

- Collections.synchronizedList()

- CopyOnWriteArrayList

Here's a sample code for synchronizing ArrayList:

``` // Synchronized block

List<String> list = new ArrayList<>();

List<String> synchronizedList = Collections.synchronizedList(list);

// Collections.synchronizedList()

synchronized(synchronizedList) {

// iterating the list

}

// CopyOnWriteArrayList

CopyOnWriteArrayList<String> cowList = new CopyOnWriteArrayList<>(); cowList.add("Java");

cowList.add("is");

cowList.add("awesome"); ```

15. Hashtable vs HashMap:

Hashtable and HashMap are both used to store and access data in key-value format. The following are the significant differences between Hashtable and HashMap:

- Hashtable is synchronized, whereas HashMap is not.

- Hashtable does not allow null keys or values, while HashMap allows null values and one null key.

Here's a sample code for Hashtable and HashMap:

``` Hashtable<Integer, String> hashtable = new Hashtable<>();

hashtable.put(1, "Java");

hashtable.put(2, "is");

hashtable.put(3, "awesome");

HashMap<Integer, String> hashMap = new HashMap<>();

hashMap.put(1, "Java");

hashMap.put(2, "is");

hashMap.put(3, "awesome"); ```

16. HashMap in Java 8: Java 8 introduced a new data structure called Internal HashMap or Node-based HashMap to support the efficient working of HashMap. This new data structure is designed to handle situations where multiple keys map to the same hash code. It uses a linked list approach to handling hash collisions.

Here's a sample code for HashMap in Java 8:

``` Map<Integer, String> map = new HashMap<>();

map.put(1, "Java");

map.put(2, "is");

map.put(3, "awesome");

//Iterating over Map using forEach() method

map.forEach((k, v) -> System.out.println("Key = " + k + ", Value = " + v)); ```

17. Fail-fast vs Fail-safe iterator: The main difference between fail-fast and fail-safe iterator is in how they handle concurrent modifications to the collection while iterating. Fail-fast iterator throws ConcurrentModificationException if the collection is modified while iterating, while fail-safe iterator creates a copy of the collection before iterating and iterates over the copy.

Here's a sample code for a fail-fast and fail-safe iterator:

``` // Fail-fast iterator

List<Integer> list = new ArrayList<>();

Iterator<Integer> iterator = list.iterator(); list.add(1);

// ConcurrentModificationException

while(iterator.hasNext())

{ System.out.println(iterator.next()); }

// Fail-safe iterator

CopyOnWriteArrayList<Integer> cowList = new CopyOnWriteArrayList<>();

Iterator<Integer> it = cowList.iterator();

list.add(1);

while(it.hasNext())

{ System.out.println(it.next()); } ```

18. Starting a thread twice:

No, a thread cannot be started twice. If we try to start a thread that has already been started, it will throw IllegalThreadStateException.

Here's a sample code for starting a thread more than once:

``` Thread t = new Thread();

t.start();

t.start(); // IllegalThreadStateException ```

19. Creating a thread:

Java provides three different ways to create a thread:

- Extending the Thread class

- Implementing the Runnable interface

- Using a lambda expression

Here's a sample code for creating a thread:

``` // Extending the Thread class

class Example extends Thread{

public void run() {

// some code to execute in this thread

}

} Example t1 = new Example(); t1.start();

// Implementing the Runnable interface

class Example1 implements Runnable {

public void run() {   
// some code to execute in this thread

}

}

Thread t2 = new Thread(new Example1()); t2.start();

//Using a lambda expression

Thread t3 = new Thread(() ->

{

// some code to execute in this thread });

t3.start(); ```

20. Thread States: Threads in Java can be in one of the following states:

- New

- Runnable

- Blocked

- Waiting

- Timed Waiting

- Terminated

Here's a sample code for thread states:

``` Thread t = new Thread(() -> {

// some code to execute in this thread

});

System.out.println(t.getState()); // NEW

t.start(); System.out.println(t.getState()); // RUNNABLE ```

21. Serialization:

Serialization is a process of converting an object into a stream of bytes so that it can be stored in a file or transmitted over the network. Deserialization is the opposite process of creating an object from a stream of bytes.

Here's a sample code for serialization in Java:

``` class Example implements Serializable {

int id;

String name;

}

Example obj = new Example();

obj.id = 1;

obj.name = "Java";

try {

FileOutputStream fos = new FileOutputStream("example.ser");

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(obj);

oos.close(); fos.close();

} catch(IOException e) {

e.printStackTrace(); } ```

22. Immutable Class:

An immutable class in Java is a class that cannot be modified once it is created. String class is an immutable class in Java. Immutable classes provide the following benefits:

- Thread safe

- Simple to use

- High performance

Here's a sample code for an immutable class:

``` final class Example {

final int value;

Example(int value) {

this.value = value;

}

int getValue()

{

return value; }

}

Example obj = new Example(10);

System.out.println(obj.getValue());

// Output: 10 ```

23. Immutable class thread-safe:

Yes, immutable classes are thread-safe in Java because the state of an immutable object cannot be changed once it is created. Immutable objects are safe to use in a multi-threaded environment because they cannot be modified.

Here's an example of how immutable classes are thread-safe

public final class Person {

private final String name;

private final int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

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

Yes, we can call the garbage collector explicitly using the System.gc() method, but it is not recommended as it does not guarantee that the garbage collector will run. The decision to run the garbage collector is up to the JVM, and it may or may not choose to run it based on the current state of the heap.

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

Java 8 introduced several new features, including lambda expressions, method references, functional interfaces, default methods, streams, and the new date/time API. Here are some examples of these features:

Lambda expressions:  
  
List names = Arrays.asList("John", "Jane", "Joe");

Collections.sort(names, (s1, s2) -> s1.compareTo(s2));

In this example, we are sorting a list of strings using a lambda expression that defines the comparison logic.

Method references:

List names = Arrays.asList("John", "Jane", "Joe");

Collections.sort(names, String::compareTo);

This is an alternative way of sorting the list using a method reference instead of a lambda expression.

Functional interfaces:

@FunctionalInterface

public interface Calculator {

int operate(int a, int b);

}

Calculator add = (a, b) -> a + b;

Calculator subtract = (a, b) -> a - b;

System.out.println(add.operate(2, 3)); // Output: 5

System.out.println(subtract.operate(5, 3)); // Output: 2

In this example, we define a functional interface called Calculator that has a single abstract method called operate. We then create two instances of this interface using lambda expressions to define the implementation of the operate method.

Default methods:

public interface Animal {

void makeSound();

default void walk() {

System.out.println("Walking...");

}

}

public class Dog implements Animal {

public void makeSound() {

System.out.println("Bark!");

}

}

Dog dog = new Dog();

dog.makeSound(); // Output: Bark!

dog.walk(); // Output: Walking...

In this example, we define an interface called Animal that has a default method called walk. We then implement this interface in a class called Dog and call both the makeSound and walk methods.

Streams:

List names = Arrays.asList("John", "Jane", "Joe");

names.stream()

.filter(name -> name.startsWith("J"))

.map(String::toUpperCase)

.forEach(System.out::println);

26.How to make a pure singleton?

A pure singleton is a class that can only be instantiated once, and provides a global point of access to that instance.

Example:

public class Singleton {

private static final Singleton INSTANCE = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return INSTANCE;

}

}

27.How to make a singleton synchronized?

A synchronized singleton ensures that only one thread can access the instance at a time, preventing potential concurrency issues.

Example:

public class Singleton {

private static Singleton INSTANCE;

private Singleton() {}

public static synchronized Singleton getInstance() {

if (INSTANCE == null) {

INSTANCE = new Singleton();

}

return INSTANCE;

}

}

GIthub link:- https://github.com/venky4144/Guttikonda\_Assignment03