SPRING BOOT 2

Singleton Class
-How
-Why (establishing one connection to the database instead of 10)

In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time.
After first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.
Remember the key points while defining class as singleton class that is while designing a singleton class:

1.Make the constructor private.

2.Write a static method that has a return type object of this singleton class.

Here, the concept of Lazy initialization is used to write this static method.

Lets code :

Create a public Person class with the following properties :

private int id;

private String name;

private int age;

Create Getters and Setters and private constructor .

Strive for progress, not perfection.

SPRING BOOT 2

```
private Person(){

       System.out.println("In the constructor");

   }

public static final Person person = new Person(); // we can use parameterized constructor too.
```

Now create a function getPerson

```
Public Person getPerson(){

Return person;

}
```

Now lets create a Main class where we will have a psvm main method .We can create the objects of Person as follows:

```
      Person p1 = Person.getPerson();
      Person p2 = Person.getPerson();
      Person p3 = Person.getPerson();
      Person p4 = Person.getPerson();
```

We can see the print statement inside the private constructor of Person is executed only once . Also , lets set age of p1 and check age of p2 .

Strive for progress, not perfection.

SPRING BOOT 2

```
p1.setAge(20);
System.out.println(p2.getAge());
```

Interface
-abstract and default methods
-overriding default methods
-calling interface default method instead of overridden method of class
- Access Modifiers
-Anonymous Inner Class
-Multiple Inheritance through Interfaces
-Abstract Class vs Interface
-Basics of Generics

Lets code :
Let's create a public interface called MathOperations.

```
public interface MathOperations{

    int var = 5; // In interfaces, all the variables are final and static

    int add(int a, int b); // In interfaces, all the methods are public and abstract automatically

    int subtract(int a, int b);

    int multiply(int a, int c);

    default int divide(int a, int b){
        return a / b;
    }

    default double power(int a, int b){
        return Math.pow(a, b);
    }

    default void testFunc() {
        System.out.println("I am in testFunc of MathOperations");
    }
}
```

Lets create another interface

Strive for progress, not perfection.

SPRING BOOT 2

```
public interface MathOperations2 {

    int add(int a, int b);

    int addParent();

    default double power(int a, int b){
        return Math.pow(a, b) + 1;
    }

    default void testFunc(){
        System.out.println("I am in testFunc of MathOperations2");
    }
```

Lets override the abstract and double methods in a class called MathOps which implements the above 2 interfaces .
In the main function on the class , lets see what are anonymous inner classes :

```
public static void main(String[] args) {
        MathOperations mathOperations = new MathOperations() {
            @Override
            public int add(int a, int b) {
                return a + b + 1;
            }

            @Override
            public int subtract(int a, int b) {
                return a - b + 1;
            }

            @Override
            public int multiply(int a, int b) {
                return a * b + 1;
            }

            @Override
            public int divide(int a, int b) {
                return a / b + 1;
            }
        };

        MathOperations mathOperations2 = new MathOps();
```

Strive for progress, not perfection.

SPRING BOOT 2

```
        mathOperations2.power(4, 3);

        System.out.println(mathOperations2.add(2, 3) + " " +
                mathOperations.add(2, 3));
    }
```

Now lets understand Abstract Classes vs Interfaces .Create an abstract class AbstractClass.

```
public abstract class AbstractClass {

    public String testFunc(){
        return "Hey";
    }

    public abstract int add(int a, int b);

    public abstract int subtract(int a, int b);

    public String toString(){
        return "";
    }

}
```

Now let's create a child class of the AbstractClass .

```
public class ChildClass extends AbstractClass{


    @Override
    public int add(int a, int b) {
        return a + b;
    }

    @Override
    public int subtract(int a, int b) {
        return a - b;
    }

    public static void main(String[] args) {
        ChildClass obj = new ChildClass();

        System.out.println(obj.add(2, 3));
```

Strive for progress, not perfection.

SPRING BOOT 2

```
    }
}
```

Functional Interface
-Anonymous Class vs Lambdas
-Iterative Approach vs Streams (ArrayList with City Names starting with vowel)
-Intermediate vs Terminal methods in Stream

Lets understand the Functional Interfaces now using Runnable Interface And lets do the same using Annonymous Inner Class and Lambda.

https://www.geeksforgeeks.org/functional-interfaces-java/
https://www.geeksforgeeks.org/lambda-expressions-java-8/

Lets understand about Streams now :
https://www.geeksforgeeks.org/stream-in-java/
Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

The features of Java stream are –

- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
- Streams don't change the original data structure, they only provide the result as per the pipelined methods.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-

Intermediate Operations:

1. **map:** The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

Strive for progress, not perfection.

List number = Arrays.asList(2,3,4,5);

List square = number.stream().map(x->x*x).collect(Collectors.toList());

2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.

   List names = Arrays.asList("Reflection","Collection","Stream");

   List result =

   names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());

3. **sorted:** The sorted method is used to sort the stream.

   List names = Arrays.asList("Reflection","Collection","Stream");

   List result = names.stream().sorted().collect(Collectors.toList());

Terminal Operations:

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.

   List number = Arrays.asList(2,3,4,5,3);

   Set square = number.stream().map(x->x*x).collect(Collectors.toSet());

2. **forEach:** The forEach method is used to iterate through every element of the stream.

   List number = Arrays.asList(2,3,4,5);

   number.stream().map(x->x*x).forEach(y->System.out.println(y));

3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.

   The reduce method takes a BinaryOperator as a parameter.

   List number = Arrays.asList(2,3,4,5);

SPRING BOOT 2

int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

Here ans variable is assigned 0 as the initial value and i is added to it .

PRACTICE:

// Q1. You have a list of integers, you need to find the sum of squares of even numbers
[1, 2, 3, 4, 5, 6, 7, 8] = [4 + 16 + 36 + 64 = 120]

// Q2. You have a list of string, you need to concatenate them

Generics :

Generics mean **parameterized types**. The idea is to allow type (Integer, String, … etc, and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types.

An entity such as class, interface, or method that operates on a parameterized type is called a generic entity.

Why Generics?

The **Object** is the superclass of all other classes and Object reference can refer to any type object. These features lack type safety. Generics add that type safety feature. We will discuss that type of safety feature in later examples.

Generics in Java are similar to templates in C++. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. There are some fundamental differences between the two approaches to generic types.

Generic Class

Like C++, we use <> to specify parameter types in generic class creation. To create objects of a generic class, we use the following syntax.

https://www.geeksforgeeks.org/generics-in-java/

Strive for progress, not perfection.