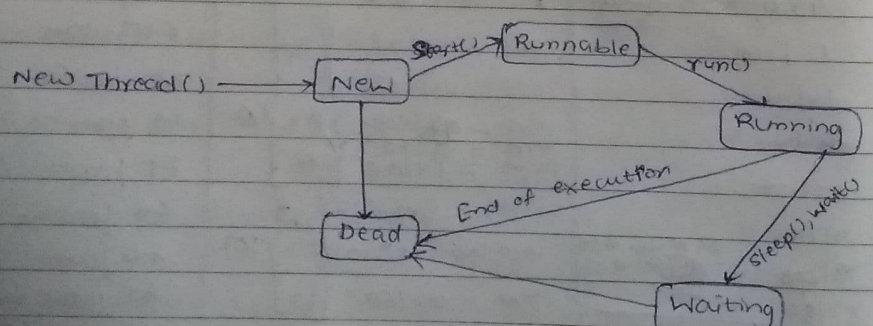
Problem 3 :-

Q 3) Threads with their life cycle (with code).
Singleton Classes (With code).
Uses of Static and Final classes and Methods.
Looping basic concepts, break, continue and return.
Data structure (List, Set, Map)

Ans) 1.) Threads with their life cycle (with code) :-

Thread :- A thread is the smallest unit of processing that can be performed in an OS. Recently, a thread exists within a process - that is, a single process may contain multiple threads. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improve performance in O.S. by reducing the overhead thread is equivalent to a classical process. Each thread belongs to exactly one process and no thread can exist outside a process.

A thread goes through various stages in its life cycle. For eg:- a thread is born, started, runs and then dies.



Following are the stages of the life cycle :-

1.) New :- A thread begins its life cycle in the new state. It remains in the state until the program starts the thread. Its also referred to a born thread.

2.) Runnable :- After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

3.) Waiting :- Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

4.) Timed Waiting :- A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when the event it is waiting for occurs.

5.) Terminated (Dead) :- A runnable thread enters the terminated state when it completes its task or otherwise terminates.

```
class RunnableDemo implements Runnable
{
  private Thread t;
  private String threadName;
  RunnableDemo(String name)
  {
    threadName = name;
    System.out.println("Creating" + threadName);
  }
```

P.T.O.

```
public void run()
{
System.out.println("Running" + threadName);
try
{
for(int i=4; i>0; i--)
{
System.out.println("Thread:" + threadName + "," + i);
Thread.sleep(50);
}
}
catch(InterruptedException e)
{
System.out.println("Thread" + threadName + "interrupted");
}
System.out.println("Thread" + threadName + "exiting.");
}
public void start()
{
System.out.println("Starting" + threadName);
if(t==null)
{
t=new Thread(this, threadName);
t.start();
}
}
}
public class TestThread
{
public static void main(String args[])
{
RunnableDemo R1=new RunnableDemo("Thread-1");
R1.start();
RunnableDemo R2=new RunnableDemo("Thread-2");
R2.start();
}
}
```

A #1) The singleton design pattern is used to restrict the instantiation of a class and ensures that only one instance of the class exists in JVM. In other words, a singleton class that can have only one object (an instance of the class) at a time per JVM instance. There are various ways to design/code a singleton class

1.) Class-level Member (Eager Initialization Method):-

1.) Make constructor private

2.) Make a private constant static instance (class-member) of this Singleton class.

3.) Write a static/factory method that returns the object of the singleton class that we have created as a class-member-instance.

4.) We can also mark a static member as public to access ~~class/instance via methods only~~ constant static instance directly. But, I like to access class/instance members via methods only.

5.) So, the singleton class is different from a normal Java class in terms of instantiation. For a normal class, we use a constructor, whereas for singleton class we use the getInstance() method

P.T.O.

```
public class SingletonClass
{
private static final SingletonClass
SINGLE_INSTANCE = new SingletonClass();

private SingletonClass() {}
public static SingletonClass getInstance()
{
return SINGLE_INSTANCE;
}
}
```

2.) Class-Level Member (Lazy Initialization Method):-

1.) Make constructor as private.

2.) Make a private static instance (class-member) of this singleton class. But DO NOT instantiate it.

3.) Write a static/factory method that checks the static instance-member for null and creates the instance. It returns an object of the singleton class.

```
public class SingletonClass{
private static SingletonClass
SINGLE_INSTANCE = null;
private SingletonClass() {}
public static SingletonClass getInstance() {
if (SINGLE_INSTANCE == null) {
synchronized (SingletonClass.class) {
SINGLE_INSTANCE = new SingletonClass();
} }
return SINGLE_INSTANCE;
}}
```

3) Class-level Member (Lazy Initialization with double lock Method):

1.) Here, we run into a problem. Suppose that there are two threads running. Both can get inside of the the if statement concurrently when the instance is null. Then, one thread enters the synchronized block to initialize the instance, while the other is blocked. When the first thread exists in the synchronized block, the waiting thread enter and creates another singleton object. When the second thread enter the synchronized block, it does not check to see if the instance is non-null.

```
public class SingletonClass {
private static SingletonClass
SINGLE_INSTANCE = null;
private SingletonClass() {}
public static SingletonClass getInstance() {
if (SINGLE_INSTANCE == null) {
SINGLE_INSTANCE = new SingletonClass();
} } }
return SINGLE_INSTANCE;
}}
```

4.) By using nested Inner class (Lazy load method):-

1.) In this method is based on the Java language Specification (JLS). Java Virtual Machine loads static data-members only on-demand. So, here the class SingletonClass loads at first by the JVM. Since there is no static data members in the class SingletonClassHolder does not loads or creates SINGLE_INSTANCE

2.) This will happen only we invoke getInstance() method. Loading and initialization. Here, since the initialization is the static variable SINGLE_INSTANCE in a sequential way, so

concurrent invocations of the getInstance() will return the ILS guaranteed the sequential execution of the class initialization that means thread-safe. So, we actually do not need to provide explicit synchronization on static getInstance() method for loading and initialization. Here, since the initialization creates the static variable SINGLE_INSTANCE in a sequential way, all concurrent invocations of the getInstance() will return the same correctly initialized SINGLE_INSTANCE without synchronization overhead.

```
public class SingletonClass {
private SingletonClass() {}
private static class SingletonClassHolder {
static final Something SINGLE_INSTANCE = new SingletonClass();
}
public static SingletonClass getInstance() {
return SingletonClassHolder.SINGLE_INSTANCE;
}
}
```

5.) By using Enums :- All above the previous approaches are not full-proof in all the cases. We can still create multiple instances of the above implementations by using serialization or reflection. In both of the cases, we can bypass the private constructor and hence can easily create multiple instances. So, the new approach is to create singleton class by using enums since enums fields are compiled time constants, but they are instances of their enum type. And, they are constructed when the enum type is referenced for the first time.

```
public enum SingletonClass {
SINGLE_INSTANCE;
}
```

(2.) Ans.3) 1.) Using a static class helps the compiler for checking to make sure that no instance members are added suddenly. The compiler guarantees that instances of this class cannot be created. Static classes are sealed and therefore they cannot be inherited. They cannot inherit from any class except Object.

2.) A final class is a class that can't be extended. Also methods could be declared as final to indicate that cannot be overridden by subclasses. Preventing the class from being subclassed could be particularly useful if you write APIs or libraries and want to avoid being extended to alter base behaviour.

3.) A method is a set of code which is referred by name and can be called (invoked) at any point in a program simply by utilizing the method's name. Think of a method as a subprogram that acts on data and often returns a value. Each method has its own name.

(3.) Ans.4) Looping basic concepts, break, continue and return

1.) Looping basic concepts :- A loop executes the sequence of statements many times until the stated condition becomes false. A loop consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop to execute until the specified condition becomes false. The purpose of the loop is to repeat the

same code a number of times.

Types of Loops:-

Depending upon the position of a control statement in a program a loop is classified into 2 types:

1) Entry controlled loop [pre checking loop]
2) Exit controlled loop. [post checking loop]

1) Entry Controlled Loop:- A condition is checked before executing the body of a loop.

2) Exit Controlled Loop:- A condition is checked before executing the body of a loop.

① * While Loop:- A while loop is the most straightforward looping structure. The basic format of while loop is as follows:

```
while (condition) {
        statements;
        }
```

② * do-while loop:- A do-while loop is similar to the while loop except that the condition is always executed after the body of a loop. It is also called an exit-controlled loop. The basic format of while loop is as follows:-

```
do {
    statements;
} while (expression);
```

③ * For Loop:- A for loop is a more efficient loop structure in 'C' programming. The general structure of for loop is as follows:-

```
for (initial value; condition; incrementation or decrementation)
{
 statements;
}
```

④ * Break Statement:- The break statement is used mainly in the switch statement. It is also useful for immediately stopping a loop.

Eg:-
```
#include <stdio.h>
int main()
{
 int num = 5;
 while (num > 0)
 {
  if (num == 3)
  break;
  printf("%d\n", num);
  num--;
 } }
```

O/P:-
```
5
4
```

⑤ * Continue Statement:- When you want to skip to the next iteration but remain in the loop, you should use the continue statement.

Eg:-
```
#include <stdio.h>
int main() {
 int nb = 7;
```

```
while (nb>0)
{
nb--;
if (nb==5)
continue;
printf ("%d\n", nb);
}}
```

O/p :-

6
4
3
2
1

⑥ * A return statement ends the execution of a function and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. A return statement can return a value to the calling function.

(3) Ans 5) 1.) List :- i) List allows duplicate elements. Any number of duplicate elements can be inserted into the list without affecting the same existing values and their index.

ii.) List allows any number of null values.

iii.) List and all of its implementation classes maintain insertion order.

iv) ArrayList, LinkedList, etc. are some of the commonly used classes.

2.) Set :- i) Set doesn't allow duplicates. Set and all of the classes which implements Set interface should have unique elements.

ii) Set allows single null value at most.

iii.) Set doesn't maintain any order; still few of its classes sort the elements in an order such as LinkedHashSet maintains the elements in insertion order.

iv.) HashSet, LinkedHashSet, TreeSet, SortedSet, etc are the commonly used classes.

3.) Map :- i.) Map stored the element as keys & value pair. Map doesn't allow duplicate keys while it allows duplicate values.

ii.) Map can have single null key at most and any number of null values.

iii.) HashMap, TreeMap, WeakHashMap, LinkedHashMap, IdentityHashMap, etc.