Softwares required

* JDK 17 or later
* IDE - Eclipse / IntelliJ
* Database - MySQL

Java - Platform independent programming

Fundamentals - Classes, Objects, Arrays, Encapsulation, Abstraction, Polymorphism, Inheritance, Exception Handling

Multithreading -

It allows an application to perform multiple tasks at the same time

ex: MS word

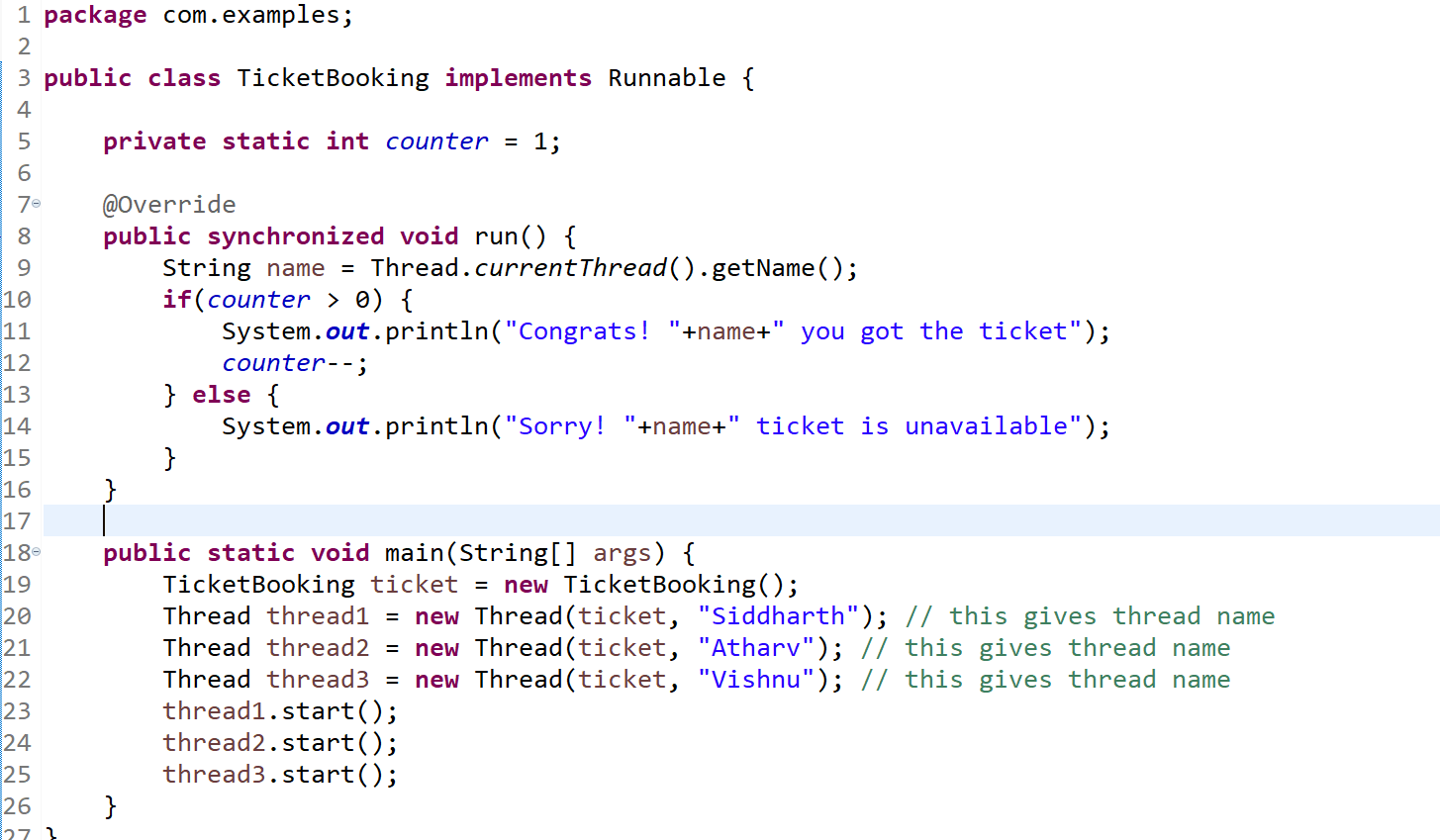
Two API’s used to create multithread programs

1. Thread class - it provides methods to manage threads - creating, starting, stopping
2. Runnable interface - it has a run method - it is an entry point for the threads

In Java application you will have a main thread which runs the main method

How the threads must work while sharing the data

* ticket booking application - tickets = 1 -> 3 users are trying to book the ticket

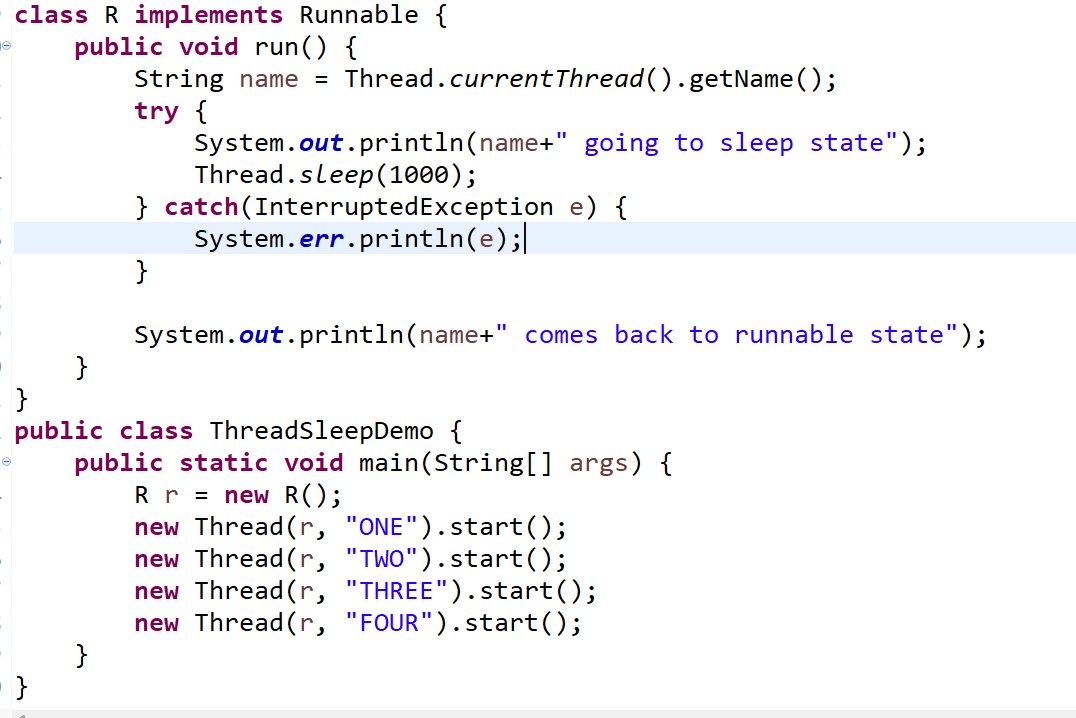


synchronized method allows only one thread to enter inside the method by acquiring the object lock, other threads should wait for the thread entered to release the lock to invoke the synchronized method

Note: You can also synchronize a particular block of code

synchronized(this) {  
 if(counter>0) { … } else {… }  
}

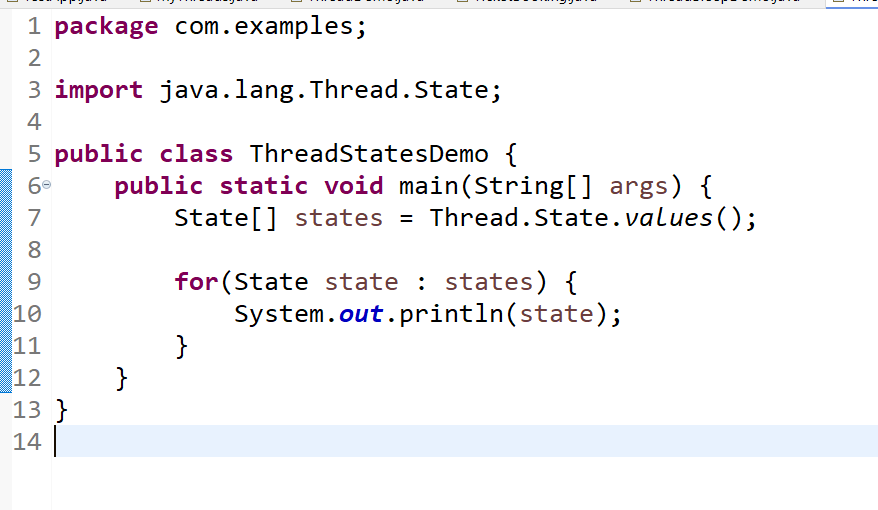
Thread.sleep(1000): Current thread goes to the sleep state allowing other idle threads to get the CPU time



Though all the threads goes to the sleep state, there will be a delay of ms in between the threads to come out of the sleep state, because once the thread goes to the sleep state, another waiting thread for the CPU time gets the CPU time & goes to the sleep state, there by with few ms all the threads will go to sleep state & come out to the runnable state after 1s delay

Thread states:

1. NEW: When you create a Thread
2. RUNNABLE: When thread gets the CPU time
3. WAITING: When thread goes to the wait state: wait() you can call to make a thread to go to wait state(), it comes out only when another thread calls notify() method, wait() & notify() are the methods present in the Object class
4. TIMED\_WAITING: When thread goes to the sleep state
5. TERMINATED: When thread is terminated



Thread pool:

It is a collection of worker threads that efficiently uses the threads, once the thread is created it will not be destroyed, it will be reused for the task

Thread t1 = new Thread();  
t1.start(); // once the thread task completes it will be destroyed, it wouldn’t be reused

In Java you have Executors & ExecutorService to use the thread pool, Executors allow you to create a thread pool of some threads which will reuse the threads for the next task without destoying, until you call the shutdown()

Design pattern

They are solutions for recurring problems

Factory pattern:

It abstracts object creation from the client which helps the client code to be loosely coupled

Developer1:  
Dao2 dao = new Dao2();  
dao.save(obj)

Developer2:  
Dao2 dao = new Dao2();  
dao.get(id);

Developer3:  
Dao2 dao = new Dao2();  
dao.get(id);

The above code is tightly coupled code, because changes done at the server side forces client side also to change, to avoid this problem we must go with loosely coupled approach, that is where factory design pattern comes

interface DBLogics {   
 save(); delete(); get(); update();  
}  
class DAO1 implements DBLogics { …. }  
class DAO2 implements DBLogics { …. }

Developer1  
Developer2  
Deverloper3

DBLogics dao = new DAO1();// still this is tightly coupled  
DBLogics dao = FactoryPattern.getDao(); // loosely coupled  
Developer who writes the factory pattern

static DBLogics getDao() {   
 return new DAO2(); //   
}

Builder pattern

It is a pattern to create a complex object step by step when you have many optional properties to be initialized and creating constructors for such properties will be cumbersome

Best example is we are going to discuss this object in Spring boot

an object called response must have following data like header, status, body and so on

ResponseEntity.body(“some content”): Only content will be sent to the client

ResponseEntity.status(200).body(“some content”): content & status both will be sent to the client

ResponseEntity.cookie(…).status(200).body(“some content”).

Observer pattern:

To define one to many relationship between objects, when one object changes its state, all dependent objects(observers) get automatically notified and updated.

object(publish a message) & multiple objects(subscribers) are connected to the publisher, then once the publisher publishes a message all the subscribers receive that message.

Ex: Youtube channel -> a new video is updated then all the subscribers are notified

Ex: Telecom services -> publish some offers then all the subscribers would receive the offers

JMS models - Java Messaging Service

Singleton pattern: It creates only one object of the class no matter how many times you request

Ex: Task manager

Activity:

Modify existing DBLogics & DAO program so that the factory pattern returns only one object when you call getDao() method multiple times, create a constructor and print the statement in the constructor when the object is created, so that when getDao() is invoked multiple times you shouldn’t see constructor print statements more than once

Java 8 or later features

* New Date & Time APIs
* Changes in the interface - default methods, static methods
* Functional interface -> you can pass function as an argument instead of an object -> functional interfaces will have only one abstract method
* Stream APIs
* JShell
* Record
* New HttpClients to work with Http2 protocol
* Sealed classes

Functional interface: It supports passing functions as a parameter instead of passing an object, it simplifies writing implementations to the interfaces with only one abstract method