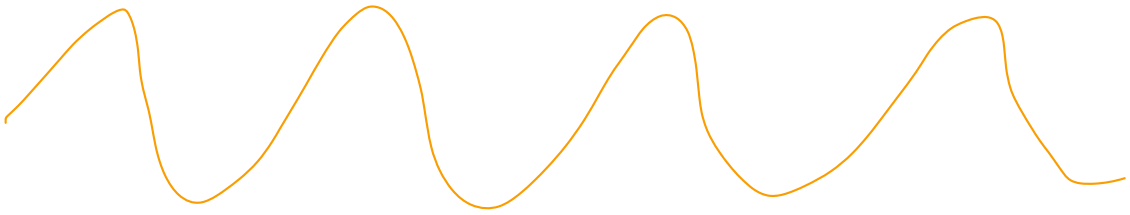
Agenda : ① Intro to Design Patterns
② Types of design patterns
③ Creational design pattern
↳ Singleton

Class starts at 9:05 PM

Ques ⇒ Patterns ⇒ Recurring thing



⇒ Basically referring to Software design.

⇒ Design Patterns ⇒ Well established solutions to common Software design problems.

⇒ Chances of finding a solution to a problem in S/W design on the internet is very high.

⇒ Design Patterns : Elements of Reusable

code

↳ Gang of 4 ⇒ Design Patterns

↳ 23 design patterns were introduced by this book, out of these we will be studying 10. ⇒ most important
(1) Interviews
(2) Actual work.

⇒ Why learn design patterns ?
(1) They give S/W a shared vocabulary.
(2) Saves a lot of time ⇒ helps you in covering a lot of edge cases.

Diff. types of Design Patterns

We have been dealing with OOP & the centre of OOP ⇒ Object.

⇒ Design Patterns are solutions to common problems in OOP.

1. Creational Design Patterns ⇒

↳ How will object be created

↳ How many objects are you going to create?

2. Structural Design Patterns

↳ How a class will be structured

↳ What all attributes are going to be there in class.

↳ How a class will interact with other classes.

3. Behavioral design patterns

↳ deal with how to code an action

Eg: if we are sorting, then we should design our application in such a way that we can change sorting behaviour in real time. basically code should

be extensible, maintainable with changing behaviour.

⇒ Creational Design Patterns ⇒

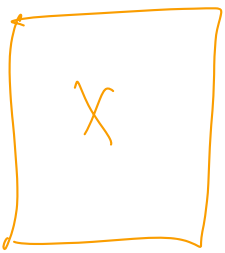
Singleton, Builder, Factory, Prototype

Simple-factory, Factory Method, Abstract Factory

(1) Singleton Design Pattern

↳ Defⁿ, Problem Statement, How to Implement, Pros & Cons

Defⁿ ⇒ Allows us to design/create a class for which only 1 object can be created.



⇒ only 1 object of X exists in the memory. in the

entire lifecycle of application.

Why do we need it ?

① A class which is having shared resources behind the scene.

DB ⇒ Database

DB Connection ⇒ establishing a connection to a database.

```
Server {
    DBConnection dbc;
    db.save();
    db.execute();
}
```

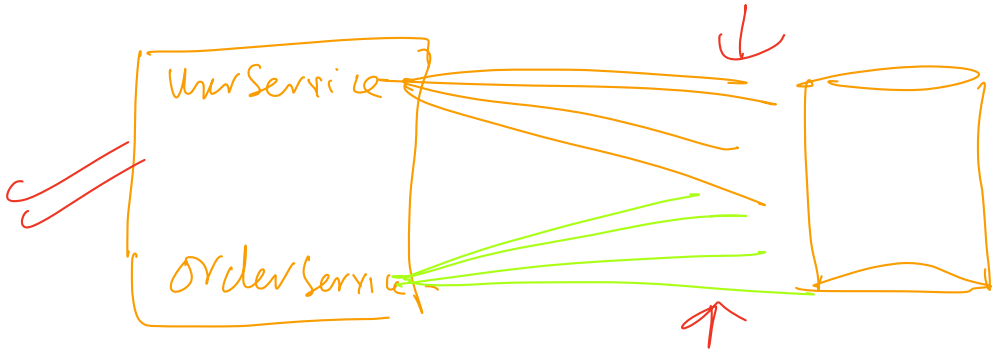
```
DBConnection {
    host;
    username;
    password;
    List<Connection> pool;
    Cache
}
```

IP Address,

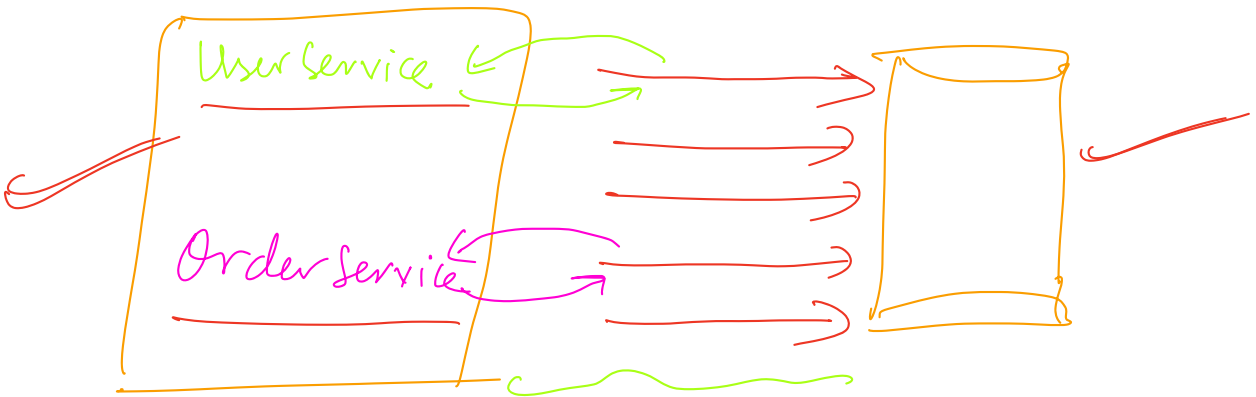
DNS

```
UserService {
    DBConnection db;
    db.save();
}
```

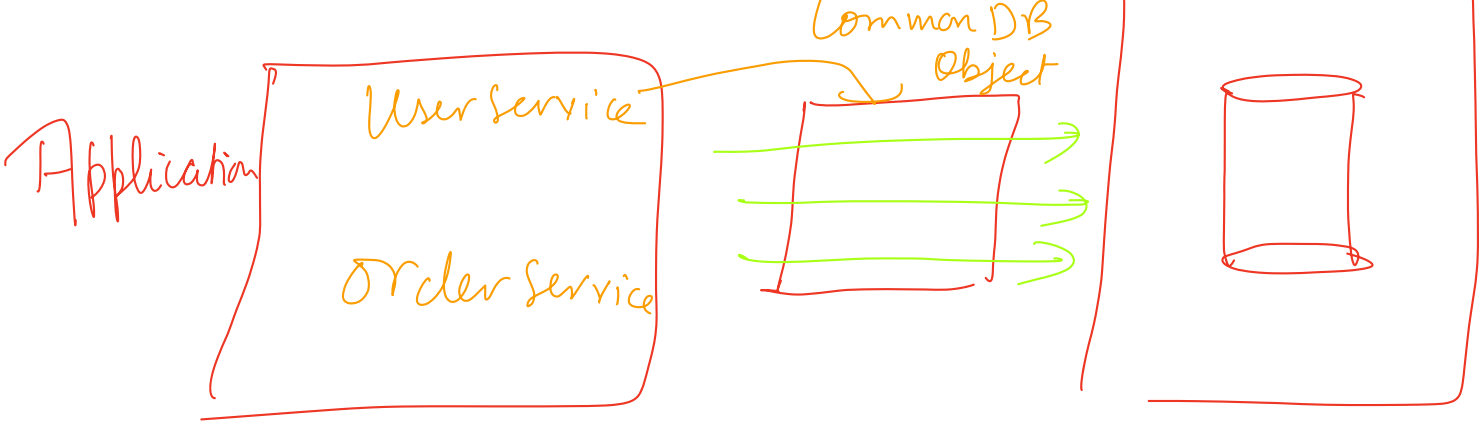
```
OrderService {
    DBConnection db;
    db.save;
}
```



⇒ It doesn't make any sense in creating multiple database connections.



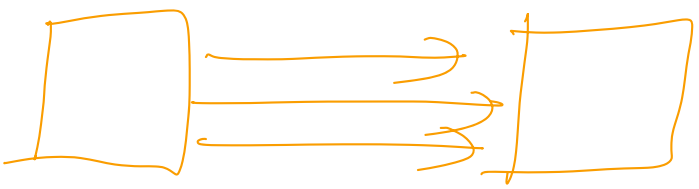

```
UserService {
    Connection conn. = getDBConnection();
    Conn. execute();
}
```



eg: logger ⇒ it prints the output to the cmd line

⇒ SOP ("Hello")
SOP ("Bye")

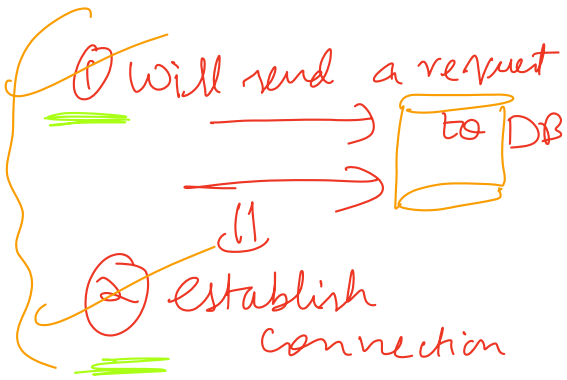
{ logger ⇒ another popular use case where it makes sense to have only 1 object.

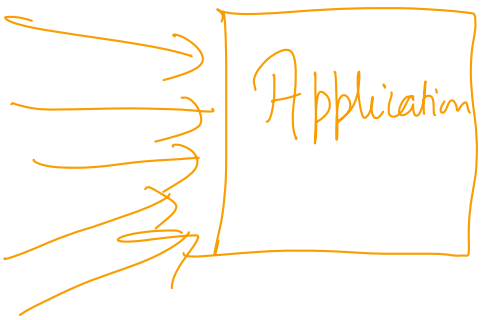
⇒ 

url, username, pswd }

② When creation of object is expensive

DB Connection {
host;
username;
password;
List<Connection> pool;
Cache
}





If for every request, we start creating a new DBConnection object, it would make our program/application very slow.

⇒ When do you think singleton design pattern cannot be followed.
⇒ When there are mutable objects.

DBConnection {
⇒ url,
⇒ username,
⇒ pswd
changeUrl() {

class A
dBC
dBC.changeUrl()

class B
dBC
shared

}

⇒ Singletons are ideally immutable else otherwise it may cause inconsistencies.

Summary: (1) Class for which only 1 object can be created

Why: (1) Shared resource
(2) Creation of object is expensive and only 1 object is sufficient.
(3) Common sense.

How do we implement a Singleton

(1) class DBConnection {
String url;
String username;
String pswd;
List<Connection> pool;
}

I can create any no. of objects

DBConnection db1 = new DBConnection();
DBConnection db2 = new DBConnection();

(2)

```
private DBConnection() {
}
```

⟹ I can't even create one object.

(3)

```
class DBConnection {

    private DBConnection() { }

    public static DBConnection getConnection() {

        return new DBConnection();
    }
```

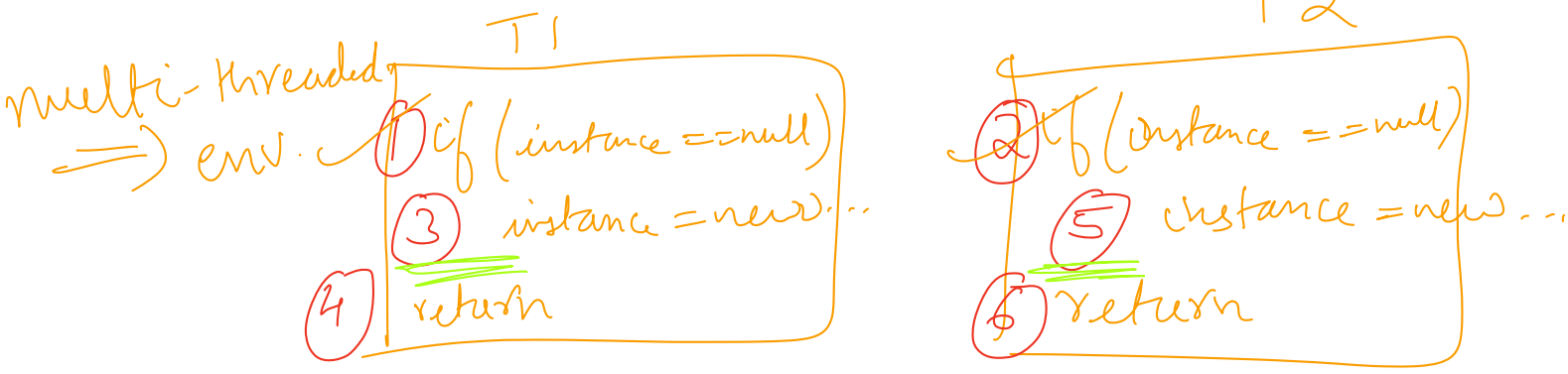
(4)

```
class DBConnection {
    private static DBConnection instance = null;
    private DBConnection() { }

    public static DBConnection getConnection() {
```

```
if (instance == null)
    instance = new DBConnection();
return instance;
}
```

Steps :
1. Make the constructor as private
2. Create a static method to create an instance.
3. Check if instance is already created
   - ⇒ if yes, then return the instance
   - ⇒ if NO, then create & return the instance.



This is prone to error only when object is being created for first time.

```
class DBConnection {
```

```
private static DBConnection instance = new DB
                                       - Connection();
private DBConnection() { }
public static DBConnection getConnection() {
}
}
```

problems

1. It will increase the startup time of your application.
2. What if the object has to be created based on an input that will be passed to the constructor.

So I need a way to

1. Create a Singleton
2. How to make it work in concurrent environment.

⇒ Most common soln. to concurrency
↳ Locks.

```
class DBConnection {
    private static DBConnection instance;
```

```
private DBConnection() { }

public synchronized static DBConnection getInstance(){

    if (instance == null)
        instance = new DBConnection();

    return instance;

}
}
```

⇒ Problem with this approach ⇒ performance always a thread will have to take a lock no matter at what time this thread is calling the function.

Let us see all the approaches

① LOCK
if (instance == null)
instance = new ...
return instance
UNLOCK

② DD if instance == null
LOCK
||
this is UNLOCK
not instance = new...
even return instance.

solving the problem.

double checking lock -

① if (instance == null) {

Will only be executed once, at the time when instance is null.

LOCK

if (instance == null) {

instance = new DBConnection();

}

UNLOCK

}

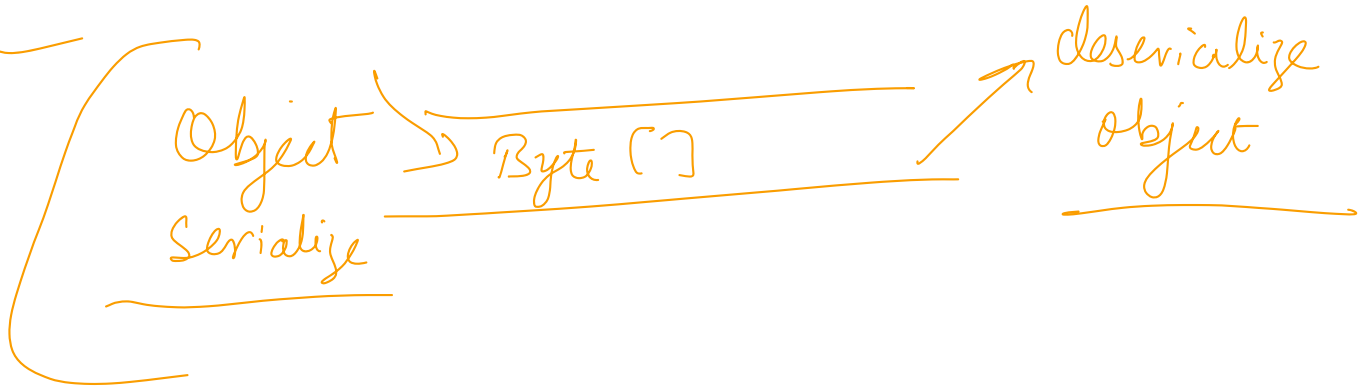
return instance

① checking if the object is null

② If the object is null, check with a lock if object is actually null or not.

The sum of double checking locks

works well on multi-threaded env. but it fails to handle serialization.



Pros : ① Resource efficiency, we aren't wasting resources

② Creating new object could be inefficient, rather we are sort of caching the object.

Cons : ① Difficult to test Singleton classes.

⇒ A(B1, C1)
⇒ A(B2, C2)

} A is Singleton class

X