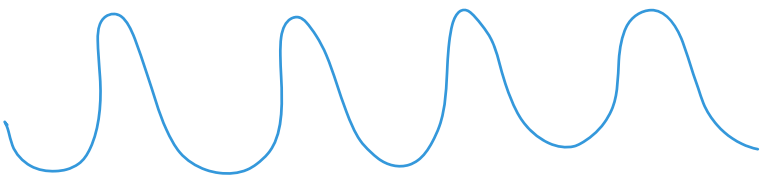
What are Design Patterns ?

Design → S/w Systems.

Patterns → Something that repeats very frequently.



⇒ Well established solutions to commonly occuring problems in S/w design.

⇒ to Design Patterns.

Type of Design Patterns :

↓

OOD

1. Creational Design Patterns :-

→ How an object is going to be created ?

→ How many objects can be created ?

2. Structural Design Patterns :-

→ How a class will be structured ?

→ What all the attrs & methods will be there inside the class.

3. Behavioural Design Patterns :—

↓

Methods | actions.

⇒ How to code methods | actions.

Creational Design Patterns :

- Singleton
- Builder
- factory
- Prototype

Singleton

Allows us to create only one object of a class.

```
X x = new X()
```

Why ?

① A class which has shared resource.

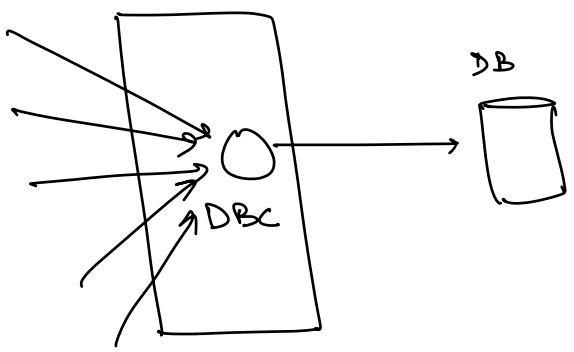
```
Class DatabaseConnection {
    Url;
    Port;
    username;
    password;
    TCPConnection
    =
}
```

```
UserService {
    DBC dbc = ——— ;
    dbc.save();
    ———
}
```

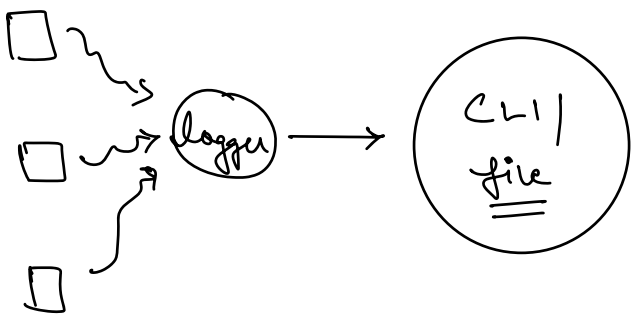
```
OrderService {
    DBC dbc = ——— ;
    dbc.save();
    ———
}
```

⇒ DBC is not easy to be maintained.

⇒ DBC takes lot of resources



Logger : Logging/Printing the important things in the log files.



② If creation of object is expensive

When NOT to use Singleton?

UserService	OrderService
dbc	dbc
dbc.changeURL()	

⇒ Singleton objects are Immutable

① Shared resource

② Object creation is expensive

How to Implement Singleton

⇒ Class DatabaseConnection {

```
    Url;
    Port;
    username;
    password;
    TCPConnection
    =
}
```

```
DBC dbc = new DBC();
DBC dbc2 = new DBC();
```

Till the time Constructor is available, Class can't be Singleton.

```
Class DatabaseConnection {
        Url;
        Port;
        username;
        password;
        TCPConnection

        private DatabaseConnection() { }
}
```

```
DBC dbc = new DBC();
```

If the constructor is private, then we can't create even (1) object of a class.

```
Class DatabaseConnection {
        Url;
        Port;
        username;
        password;
        TCPConnection

        private DatabaseConnection() { }
              static
        public ^ DBC getInstance() {
                DBC dbc = new DBC();
                return dbc;
        }
}
```

```
DBC dbc = DBC.getInstance();
DBC dbc1 = DBC.getInstance();

Class DatabaseConnection {
    private static DBC instance = null;

    url;
    Port;
    username;
    password;
    TCPConnection

    private DatabaseConnection() { }
        static
    public ^ DBC getInstance(){
        if( instance == null){
            instance = new DBC();
        }
        return instance;
    }
}
```

```
DBC dbc = DBC.getInstance();
      ↳@800
DBC dbc1 = DBC.getInstance();
      ↳@800
```

1. Make constructor private.
2. Make a public getInstance() method.
3. Create a static instance

⇒ This code will run fine in a Single threaded environment, but in multithreaded this won't work.

```
if (instance == null) {
    T2 T1 instance = new DBC();
}
return instance;
```

⇒ This implementation is prone to errors in Multithreaded environments.

SINGLETON in MULTITHREADED Environment

1. Eager / Early Initialization

```
Class DatabaseConnection {
    ⇒ private static final DBC instance = new DBC();   // App^n startup time.
      URL;
      Port;
      username;
      password;
      TCPConnection

      private DatabaseConnection() { }

      public static ^ DBC getInstance() {
            return instance;        T1 | T2 | T3
      }
}
```

Disadvantages

1) This might impact the App startup time.
2) We can't give a variable / attr in the constructor at start time.

2.

```
Class DatabaseConnection {
    private static DBC instance = null;

    Url;
    Port;
    username;
    password;
    TCP Connection

    private DatabaseConnection() { }

    public synchronized static DBC getInstance() {
        if( instance == null ) {
            instance = new DBC();   => Critical section
        }
        return instance;
    }
}
```

⇒ Only (1) thread can call getInstance() method at one time.

⇒ Performance impact.

Synchronized ≡ lock.

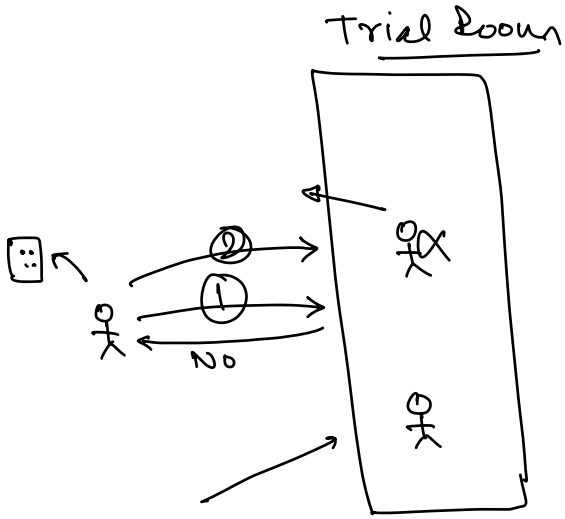
```
Public static DBC getInstance(){
    if (instance == null){
        lock()
        T2 instance = new DBC();
        unlock()
    }
T1  return instance;
}
```

@900

```
Public static DBC getInstance(){
1st  if (instance == null){ T2 T3...T100
        lock()
2nd     if(instance == null){
            instance = new DBC(); T1
        }
        unlock()
    }
T1 T2 T3 return instance;
}
```

⇒ Double Check locking

```
Public static DBC getInstance(){
T1 T4 T5 → lock()                                    @900
    T2 if (instance == null){
        instance = new DBC();
    }
    T3 unlock()
    }
    return instance;
}
```



Double Check locking.

⇒ Best way to implement Singleton design pattern in production env, where performance matters.

Pros & Cons of Singleton

Pros

1. Resource Efficiency
2. If an object is expensive.

Cons

=> Difficult to test.