1. Good Evening
2. We begin at 9:10pm
3. Topic
    - Code Singleton
    - Builder Design Pattern

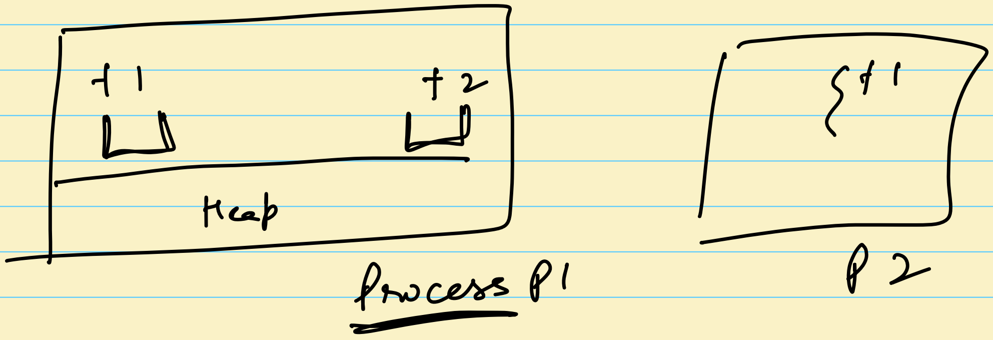
---

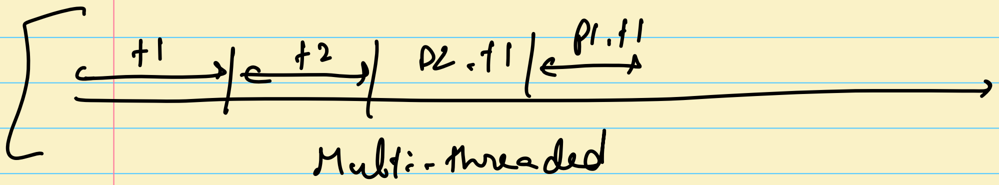
Agenda

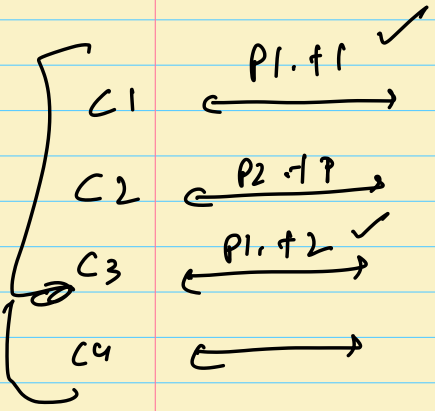
1. Finish Singleton → Coding
2. Builder Design Pattern
    - Discussion
    - Code
    - FireBase API
3. Next class
    - Prototype
    - Factory
        - simple
        - Factory Method
        - Abstract Factory

Concurrent [Multi-thread]

└ Parralled [Multi-core]







7.1 7.3

```
DB Conn {
    url —
    userN —
    pwd —
    =
    private DBConn () {
    }
    private static DBConn inst;
public static DBConn getInstance() {     → synchronized
    if (inst == null) {
        inst = new DBConn()
    }
    return inst;
}
```

url, uN, pwd

→ Performance Degradation after the instance has been created

Eager loading Singleton implementation

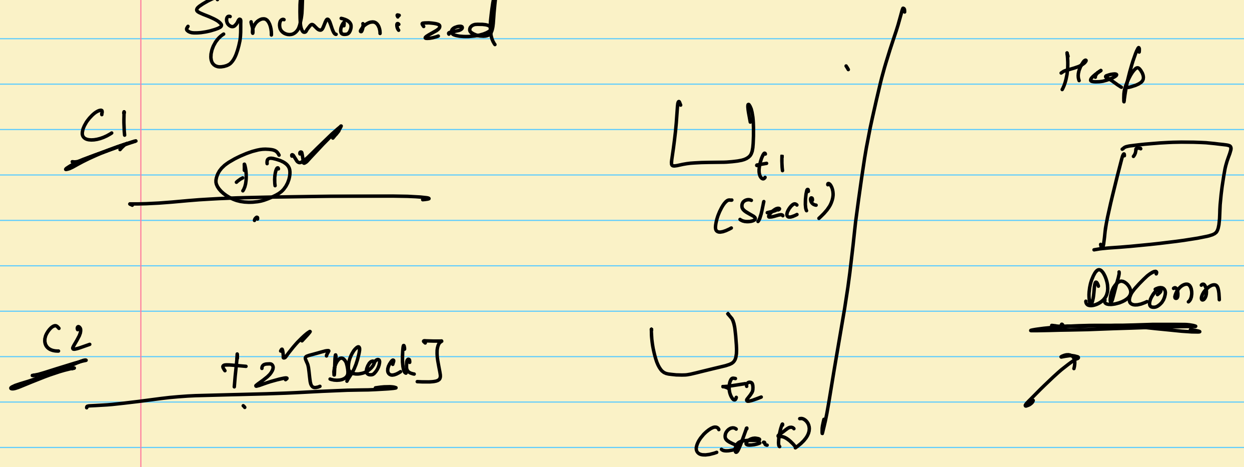
```
class DBConn {
        // params
        private DBConn() {

        }
    → ✓ private static DBConn inst = new DBConn();
       public static DBConn getInstance() {
             return inst;
}      }
```

- → Works as a singleton
- → when the class is loaded, at that time, inst is set to an instance
- → Is Also Thread-safe

| t1 | t2 |
|---|---|
| DBConn s1 = DBConn.getInst() | DBConn s2 = DBConn.getInst |

Synchronized



☆ Class is loaded once.

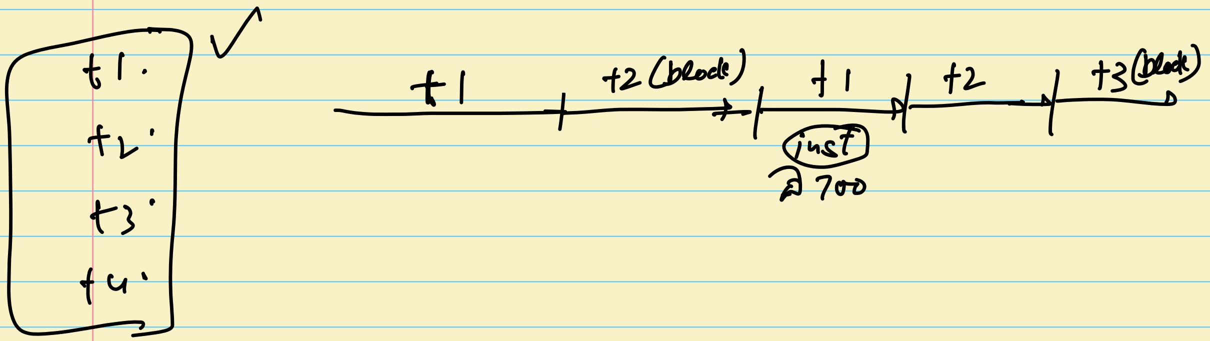
☆ Con of Eager loading Singleton.

1. Loading unnecessarily.
2. If there are parameters to be passed to initialized, we cannot use eager loading

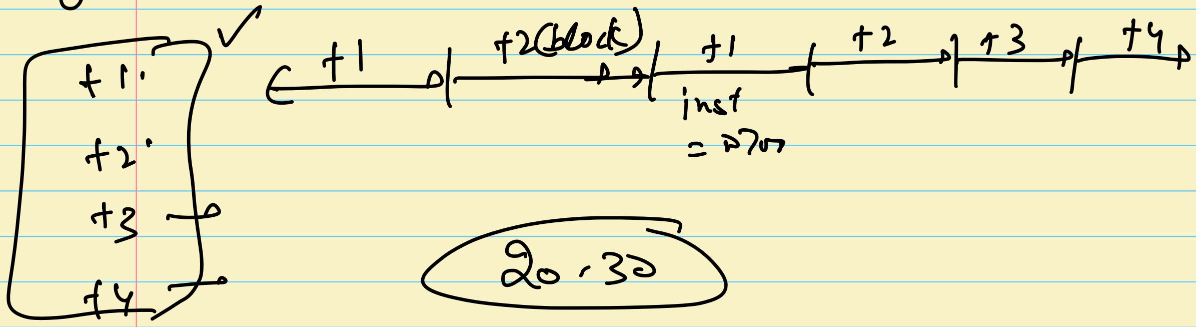
# Final Solution : Dual null check locking

Singleton1 (7.1)

↳ Entire method is synchronized



Singleton3 (7.3)



20 : 30

[7.3 = Dual null check Singleton]

Assignment

↳ 1. Reflection → Singleton violation

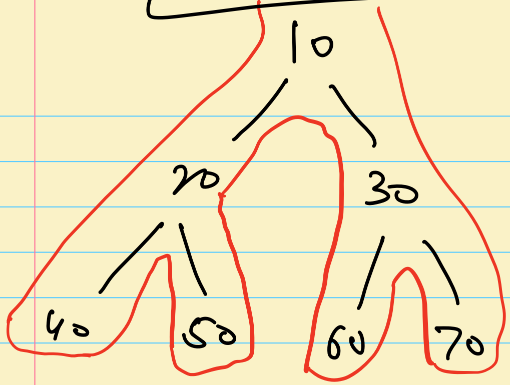
2. [Serialization & Deserialization] → Singleton

3. Clone → Singleton

✓

Dual null check locking

## 4. Sigleton implementation via Enums



String serialize() {

}

[ 40 -1 -1 50 -1 -1 20 60 -1 -1 70 -1 -1
30 10 ]

Node deserialise (String str) {

}

Break → 10:09 to 10:19

Builder Design Pattern

# Builder Design Pattern [Creational Design Pattern]

Scenario

1. A class with lot of attributes
2. Validations → [∵ creation of object]

~~V0~~

```
Student {
    name
    age            → ≥ 0 && ≤ 100 ]
    psp
    batch ☆
    id
    univName
    gradYear
    phoneNumber    → 10 digits long. ]
}
```

V1

```
class Student {
    // no ctor → default ctor

    [ Same attributes as V0 ]
}
```

Client ✓

```
Student st = new Student() ★
st.age = 10 ✓
st.name = "Sumeet" ✓
st.univName = "PEC" ✓
```

P1 → 1. No validation ✓

2. Client's can create incomplete objects ✓ { they may forget to set necessary properties }

---

V2 →

```
class Student {
    // make datamembers private ✓
    private int age;
  → public int getAge() {
        return age;
    }

  → { public Student (name, age, psp, ...
          ....) {
        if (age < 0 || age > 100) {
          → throw new InvalidArgumentException("Age can't be -ve");
        }
    }
        // validations
}
```

Client

```
Student st = new Student("Sumeet", 10, 70.0,
                         2022, . . . . . )
```

P2

1. Client code is unreadable ✓
2. Prone to error ✓
3. If I've to add a mentor or a buddy to students, ctor will change → CTE
4. Client's will pass null for parameter's they don't have a valid value for

---

V3 : Set of ctors

```
Student {
    // 10 properties

    Student (String name) {
        this(name, 0);
    }

    student ( name, psp ) {
        this (name, psp, 2000);
    }

    Student (name, psp, gradYear) {
        this.name = name
        this. psp = psp
        this.gradYear = gradYear;
    }
}
```

P3 1. $2^n$ ctors are possible

```
C
  — p1
  — p2
```

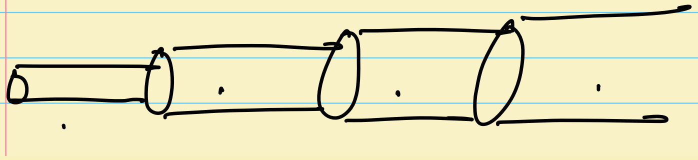
C( );

C(p1);

C(p2);

C(p1, p2);

☆ Telescoping ctors explosion



```
2.  Student (name, psp) →
             S      i
    Student (univName, psp) →
                S       i
```

Ctor overloading might not always be possible.

---

V4

```
class Student {
    // private dm
    // public getters
                         S   Object
    public Student(Map<_, _> obj) {

}   }
```

```
"name" : "Sameed"
"age"  : 10
"gradYear": 2009
```

Client

```
HashMap<String, Object> builder
                        = new HashMap<>()

[ builder.put("name", "Sumeet")
    "    .   "  ("age", 20);
    —      —

Student st = new Student(builder);
```

P4

1. Typos ✓

                                          age

```
builder.put("page", 20)
```

2. Ctor will have typecasting ✓
   & if-else conditions.

```
Student( HashMap<S, O> builder) {
    if(builder.containkey("age")) {
        age = (int) builder.get("age");
    }
}
```

V5 ✓

```
Student {
   // 10 dm → name, age, psp ... [pvt]
   // getters

   Student(Builder sb) {
      if(sb.age < 0 || sb.ages > 100)
           throw Exception
      age = sb.age
      name = sb.name
      . .
   }
}
```

```
Builder {
   int age
   S name
   — psp
   — gradYear
   [... all Student properties.]
}
```

Clients

```
Builder b = new Builder()
   b.name = "Sumeet" ✓
   b.age = 10; ✓
   b.gradYear = 2009; ✓
   ..
   .
Student st = new Student(b);
```

→ Validations ✓

→ Ctor explosion ? ✓

→ A ctor with many param → unreadable ✓

# Demo Code

PS → Student class should help you get the Builder.

V6 Add a static fn. inside Student

```
Student {



    static Builder getBuilder() {
            return new Builder()
        }
}
```

```
Builder b = Student.getBuilder();
 b. name = "ABC"
 " . age = 10
 —
 —
Student s = new Student(b);
```

P6 → According to SRP, it is builder's responsibility to create Student

builder. build()

V7

```
Student {
 // private dm
 // public getter
 p s B getB() {
     return new B()
 }
 public Student (Builder b){
     // valid & init
 }
};
```

```
Builder {
 // dm

 Student build() {
    . valid & init

    return st;
 }
}
```

Problem

↳ Client

✗
```
Student st = new Student(b)
```

✓
```
Student s2 = Student.getBuilder().
             setXYZ().
             build();
```

Soln. → Make the ctor pvt.