Lecture 4. (Creational Design Patterns) Builder, Prototype & Singleton.

Homework:

Think of all famous interview questions for low level design and for each note the objects you ought to create for that

1. Parking Lot
2. Elevator
3. IRCTC
4. BookMyShow
5. Uber/Ola
6. Zerodha
7. Swiggy/Zomato
8. Amazon/Myntra

What of Objects You will be creating and how will You create that.

We learnt 4 creational Design Patterns so think which one You will use it for designing.

(Q/A session)

Q) Guy asks that in interview he got asked Singleton design pattern with respect to Lazy Initialisation.

A) Kaarti said she will discuss in next class.

"We will compare factory & builder in the next class"

Builder is used only when the construction process is same across all the classes.

Factory to be used when construction process is not the same. Completely different classes will be there

"Next Class we will talk about thread safety and Singleton multithreading."

(***)

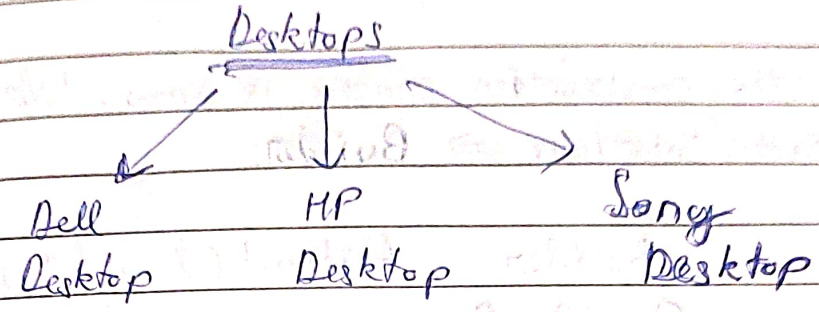
"Some body said from C++ 11, static variables are made thread safe by default".

Q) Some guy pointed application of Prototype Design Pattern.

Like generating a snapshot at a particular step. At time t=to you call the clone() method and then you called then at t=t1 again you call the clone() method So it generates two different pictures of the snapshot step. This way it can be used.

(Builder Design Pattern)

Example we took is of a Desktop building.



Now we can think of using Factory Pattern For this.

But Now our use case changes. Each Desktop will need :

RAM
Mouse
Keyboard
Speaker
Printer
Graphics Card.
Monitor.

→ If we see here, construction process is same for all

⇒ Only configuration will be different for each of the type of Desktop

Builder → Abstract class / Interface

- build RAM()
- build Processor()
- build Monitor()
- build Keyboard()

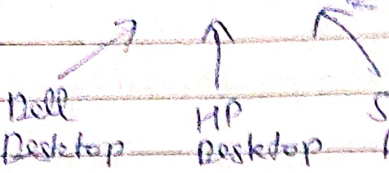
These are abstract methods

Dell Desktop Builder

HP Desktop Builder

Sony Desktop Builder

① Desktop (Product) → Final Object which needs to be built

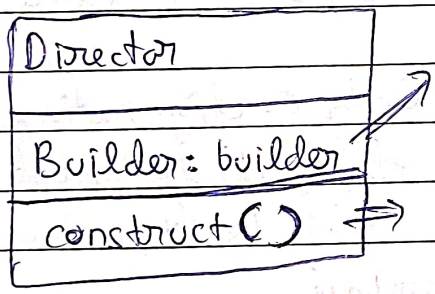


Since the construction process is same. We have built an abstract / Interface ⇒ Builder.

② Builder (Abstract Class / Interface)
↳ Refines how to build different parts of product

③ (Dell Desktop Builder, HP Desktop Builder, Sony Desktop Builder) (Concrete Builder) ⇩ Implements Steps of the Builder.
(Implementing the Abstract methods in the Builder Product)

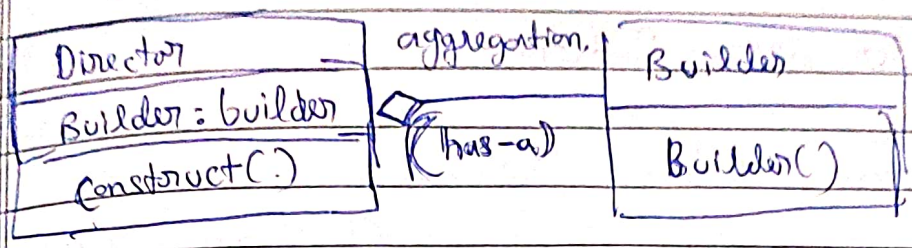
④ (Director) ⇒ (Controls the Building Process)



~~construct(Builder b)~~ ⇒ in the Constructor of Director.

```
public Director(Builder builder){
    this.builder = builder;
}
```

And You call `Director director = new DesktopBuilder();`



This use of "Director" is optional. Widely we use the chaining mechanism for Builder Design Pattern.

(***) Use case: To construct complex objects step by step. It provides a way to separate the construction of an object from its representation, making it easier to create objects with many optional parameters.

(Without Builder)

```
class User{
    private String name;
    private int age;
    private String email;
    private String address;

    public User(String name, int age){ };
    public User(String name, int age, String email){ };
    public User(String name, int age, String email, String address){};
}
```

→ Multiple constructor for multiple fields which are optional.

With Builder design Pattern.

```
public class User{

    private String name;
    private int age;
    private String email;
    private String address;

    private User (UserBuilder userBuilder){
        this.name = userBuilder.name;
        this.age = userBuilder.age;
        this.email = userBuilder.email;
        this.address = userBuilder.address;
    }

    // Static nested Builder class
    public static class UserBuilder {
        private String name;
        private int age;
        private String email;
        private String address;

        public User build (){
            return new User(this);
        }

        public UserBuilder setEmail(String email){
            this.email = email;
            return this;
        }
    }
}
```

Fields required for User ⇐

method to build final object. ⇐

Similarly for each parameter. ⇒

Usage:

```
User.UserBuilder userBuilder = new User.UserBuilder();
User u = userBuilder.setEmail("xyz@gmail").setAge(19).
         setAddress("123 & 42 Street").build();
```

Advantages of Builder Design Pattern.

1. Improves Readability
2. Flexible Object Construction
3. Immutable Objects
4. Method chaining

When to Use Builder Pattern.

1. Has many optional parameters
2. When too much constructor overloading is required.

Then I became curious and wanted to learn, how StringBuilder is implemented.

(StringBuilder Implementation)

- Uses a char[] buffer.
- Resizes dynamically → doubles in size when exceeded
- Provides faster string manipulation
- Not thread-safe

Got to learn about StringBuilder a lot.

① public final class StringBuilder extends AbstractStringBuilder
implements java.io.Serializable, CharSequence {

public StringBuilder(int capacity) {

super(capacity);

}

:

}.

② AbstractStringBuilder class manages the character buffer

abstract class AbstractStringBuilder {
char[] value; → Internal character Array.
int count;

3. Append operation

public AbstractStringBuilder append(String str) {

int len = str.length();
→ ensureCapacity(count+len);
→ do the actual appending
count += len; → Increase the size of StringBuilder String
return this;
}

This increases the size of the char Array if required.

Internally it creates a new Array of double the size and copies the existing characters into new Array.

4. public AbstractStringBuilder insert(int offset, String str)

Insert operation

⇒ shift characters
⇒ insert into the offset

return this;

}.

5. toString() method.

Converting StringBuilder to String

public String toString(){

return new String(value, 0, count);

}

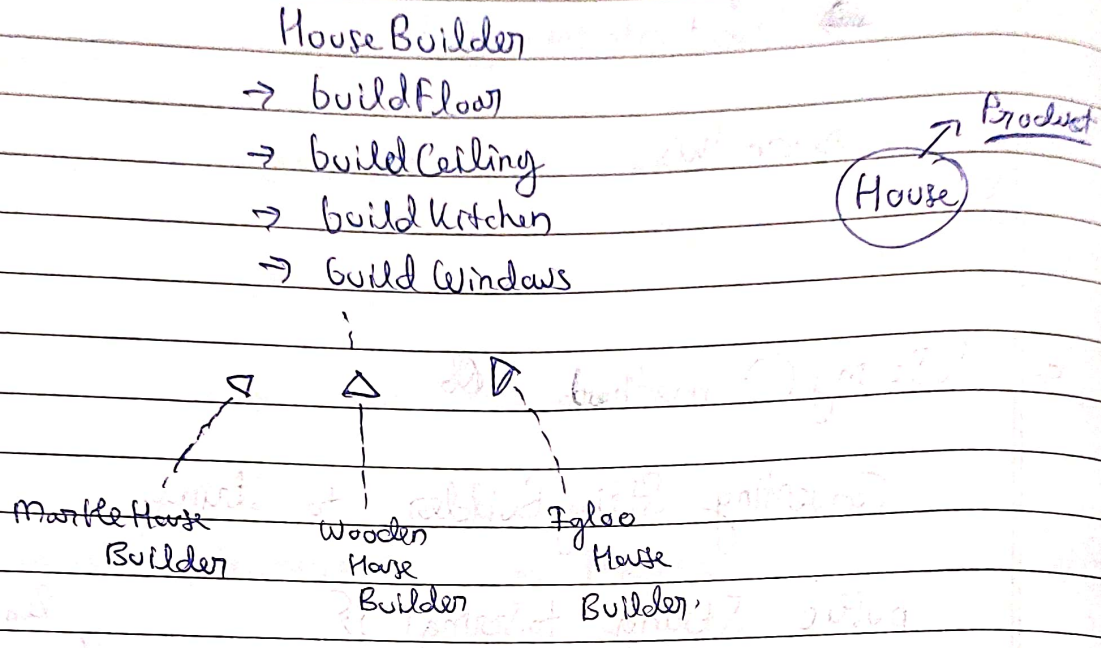
value → character Array
count → length of the String

StringBuilder V/S String

1) High memory usage - Each time new Object created	1) Low, modifies existing object
2) Performance slow (new objects more GC time)	2) High performance.
3) Thread Safety → Yes.	3) No Thread safety
4) For immutable	4) For modifyable

(Few more Examples of Builder Design Pattern)

House Example.



CivilEngineer (~~HouseBuilder~~ houseBuilder){

Director.

}

buildHouse ( ){

Doing the thing ⇒ {
houseBuilder.buildFloor()
houseBuilder.buildCeiling()
:
houseBuilder.build();
}

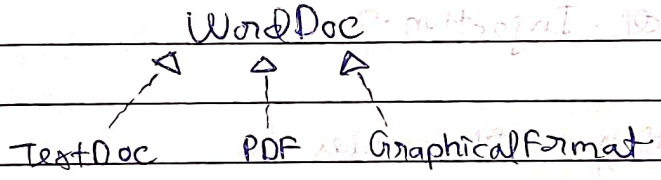
Example given in the book ("Gangs of Four")

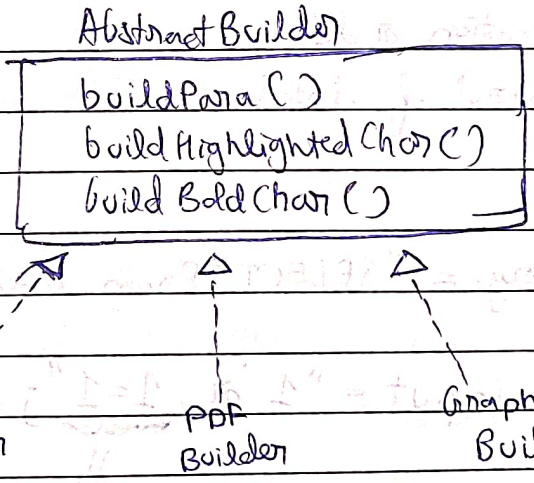
You are given a wordDoc and you want to export it into (TextDoc, PDF or Graphical Format), any of the formats.

wordDoc

In a word Document, we traverse through character by character, and for each type of character we have to do some formating

1. build Para → New Paragraph is starting
2. build Highlighted Character → New character is highlighted
3. build Bold Character → To handle Bold Character.





Each Builder having its own implementation to handle builder and then building it to the required exportType.

Builder Design Pattern practical Applications.

1) SQL Builder → Used to build queries.

Traditional Approach:

```
String query = "SELECT * FROM users WHERE age > " + minAge;
if (city != null) {
    query += " AND city = '" + city + "'";
}
```

This is on the fly query modification.

This approach is prone to SQL injection.

Q) What is SQL Injection?

A) Manipulating SQL queries to
1) Unauthorized data access
2) Data modification or deletion
3) Taking control of database.

Example.

```
String query = "SELECT * from users where id = " + input;
```

now if input = "1 or 1=1";

this will result into TRUE.

And All users will be fetched.

1) Sensitive info might get leaked.
2) Slowing down DB.

To avoid this, we use SQLBuilder.

```
class SQLQueryBuilder{

    private StringBuilder query;

    public SqlQueryBuilder(){
        this.query = new StringBuilder();
    }

    public SqlQueryBuilder select(String columns){
        query.append("SELECT ").append(columns).append(" ");
        return this;
    }

    public SqlQueryBuilder from(String table){
        query.append("FROM ").append(table).append(" ");
        return this;
    }

    public SQLQueryBuilder where(String condition){
        query.append("WHERE ").append(condition).append(" ");
        return this;
    }

    public SQLQueryBuilder and(String condition){
        query.append("AND ").append(condition).append(" ");
        return this;
    }

    public String build(){ → to remove trailing spaces.
        return query.toString().trim();
    }
}
```

Usage of SQL Query Builder.

```
String query1 = new SqlQueryBuilder()
                .select("*")
                .from("users")
                .where("age > 25")
                .orderBy("name", "ASC")
                .build();

String query2 = new SQLQueryBuilder()
                .select("id, name, email")
                .from("users")
                .where("active = Y").build();
```

(JSon Builder using Builder Design Pattern)

```
class JsonBuilder{

Map<String, Object> jsonMap = new HashMap<>();

public JsonBuilder add(String key, Object value){

        jsonMap.put(key, value);
        return this;
}

public JsonBuilder addObject(String key, JsonBuilder nestedObject)

{

        jsonMap.put(key, nestedObject.build());
        return this;
}
```

```
public JsonBuilder addArray(String key, Object... values){
    jsonMap.put(key, Arrays.asList(values));
    return this;
}

public String build(){
    return jsonMap.toString();
}
}
```

Usage:

```
psvm(){
    JsonBuilder jBuilder = new JsonBuilder();
    String json1 = jBuilder.add("name", "John")
                           .add("age", 30)
                           .addArray("hobbies", "Reading", "Dancing")
                           .addObject("address", new JsonBuilder()
                                   .add("street", "123 Bonting Street")
                                   .add("city", "Kolkata")).build();
```

Why did we use Json Builder, Builder Design Pattern

String json = "{ \"name\" : \"John\", \"age\" : 30, }";

- Hard to modify
- Nesting is error prone
- Not dynamic

Prototype Design Pattern

Use Case: we want to create the exact same clone of an ~~class~~ object that we have created.

obj1 is there and we want to create exactly another obj2. Clone it from (obj1 ⇒ obj2)

class ~~Pro~~ {

private

a
b
c
d
} ⇒ private members are there

Now lets say, we want to create a copy instance of another object.

We would have to call getter / setter of each of the attributes again and again to finally set the values and create a clone.

We use copy constructor / ~~clone~~ constructor.

In prototype design Pattern, we

```
interface Prototype{
    public void clone();
    public void ----
}
```

Concrete class implements the (Prototype)

↳ this has a clone() method.

(Example)

```
                                    ↗ Prototype
abstract class NetworkDevice{
    public abstract NetworkDevice clone();
    public abstract void display();
    public abstract void update(String newName);
}
// Product1
class Router extends NetworkDevice{
    private String name;
    private String ip;
    private String securityPolicy;

    public Router(String name, String ip, String securityPolicy){
        this.name=name;
        this.ip= ip;
        this.securityPolicy=securityPolicy;
    }
```

```
@Override
public NetworkDevice clone(){
    return new Router(name, ip, securityPolicy);
}

@Override
public void display(){
    sout("display");
}

@Override
public void update(){
    sout("update command");
}

}.

// Product 2.
class Switch extends NetworkDevice{
    String name;
    String protocol;

    public Switch(String name, String protocol){
        this.name = name;
        this.protocol = protocol;
    }
    @Override
    public NetworkDevice clone(){
        return new Switch(name, protocol);
    }
    ---
    ---

}.
```

Usage in main

```
psvm(){
    NetworkDevice routerPrototype = new Router("xyz", ...);
    NetworkDevice switchPrototype = new Switch("abc", "http");

    NetworkDevice routerClone = routerPrototype.clone();
    NetworkDevice switchClone = switchPrototype.clone();
}
```

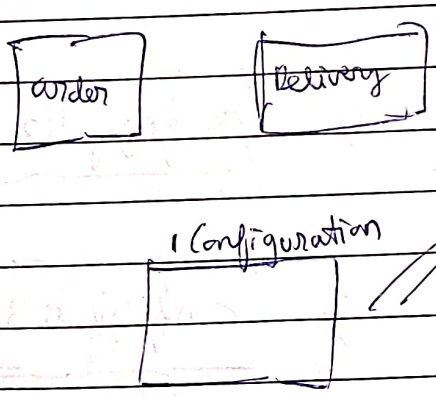
~~Singleton Design Pattern~~

Main points of (Prototype Design Pattern.)

* Delegate the process of cloning or creating a clone to the class. "I dont want to do it You do it".

(Singleton Design Pattern)

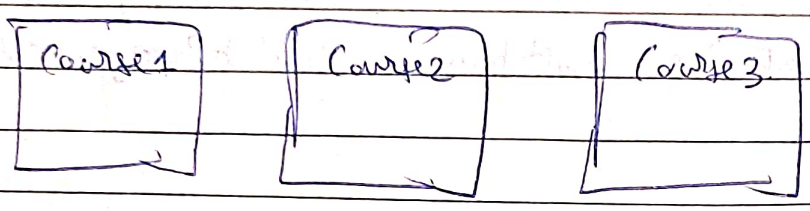
UseCase: Let say there are lot of modules in the class.



Another example.

Let's say an Ed/Tech platform. which has configurations.

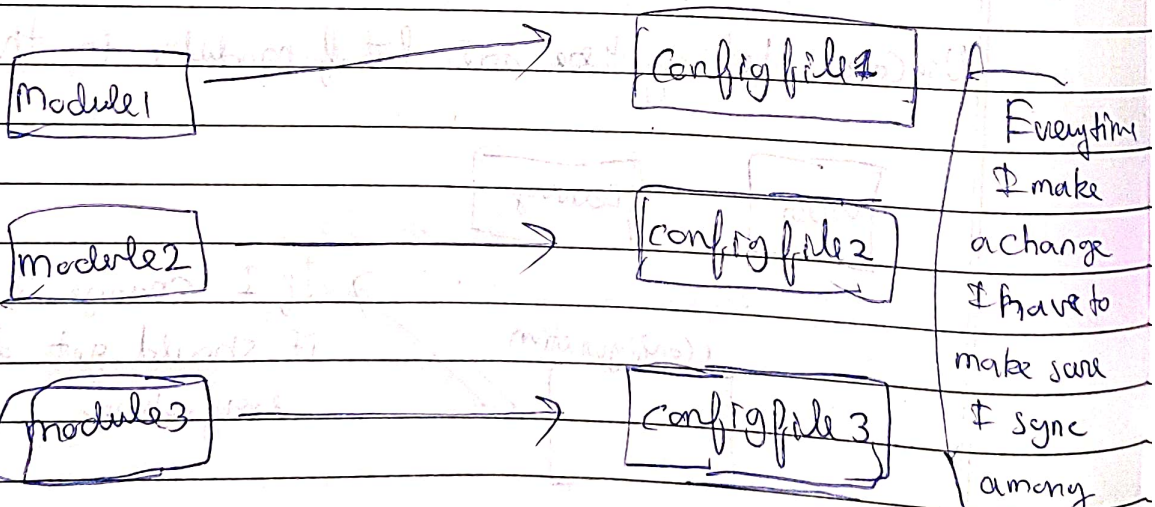
Configuration → (Light Theme / Dark Theme)



If we change configuration, it should get displayed everywhere, that is the main concern of the design pattern.

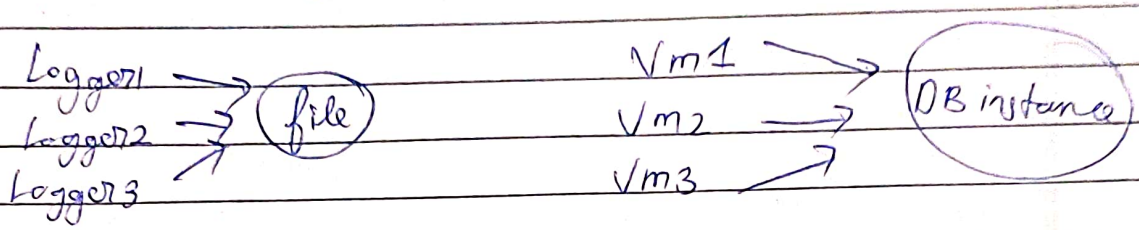
It should not be like, if we switch from course 2 from course 1 then our theme goes to default.

If I have 3 config files.



Instead of this, Have only 1 config object and no other instance of this class can be created.

Examples → Logger, DB instance.



→ Single Object of the class.

1st this to make sure, no one else can create any more Objects of the class

↳ To handle this we make our Constructor private

2nd. static instance, only want 1 instance of the class.

3rd how to access the instance, for that have a static getter

```
class Singleton {

    private static instance;

    private Singleton() {

    }

    public static getInstance() {

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

        return instance;
    }
}
```

Multithreading, & other scenarios in next Lecture.