Name of the Experiment ..... Page No.:

Experiment No..... Date :

SOLID Design Principle

S - Single Responsibility Principle
O - Open Closed Principle.
L - Liskov Substitution Principle.
I - Interface Segregation Principle
D - Dependency Inversion Principle.

Single Responsibility Principle

There should never be more than one reason for a class to change.
* focused, single functionality
* addresses a specific concern.

Open closed principle

Software Entities (classes, Modules, Methods etc.) should be open for extension, but closed for modification.

Liskov substitution Principle.

We should be able to substitute base class objects with child class objects and this should not alter behaviour/characteristics of program

BIT

Interface Segregation Principle

Clients should not be forced to depend upon Interfaces that they don't use.

signs of Interface pollution ⟿

* Classes have empty method implementation
* Methods implementations throw UnsupportedOperationException (or similar)
* Method implementation return null / default / dummy values

Dependancy Inversion Principle

1) High level modules should not depend upon low level modules. Both should depend upon abstraction.

2) Abstraction should not depend upon details. Details should depend upon abstraction.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Design Patterns.

Creational	Structural	Behavioural.
They deal with the process of creation of objects of classes.	They deal with how classes & objects are arranged or composed.	They describe how classes & objects interact & communicate with each other.

CREATIONAL PATTERNS

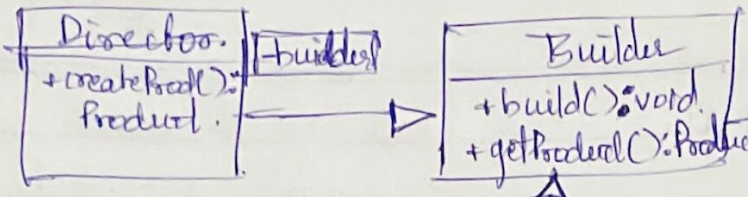
- Builder
- Simple Factory
- Factory Method
- Prototype
- Singleton
- Abstract Factory
- Object Pool.

1) Builder

- We have a complex process to construct an object involving multiple steps, then builder design pattern can help us.
- In builder we remove the logic related to object construction from "client" code & abstract it in seperate class.

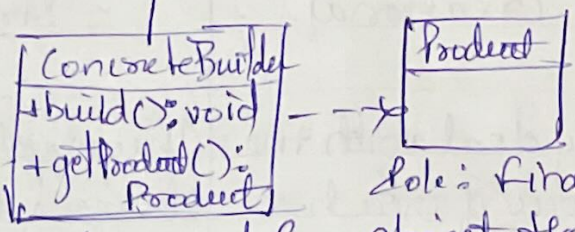
BIT

UML Diagram:



Role:
* Provides interface for creating "parts" of product.

Role:
* Uses builder to construct object
* Knows the steps & their seq to build product obj



Role: Construct parts & assembles final product
* keeps track of product it creates.
(If Interface is Created)

Role: final complex object that we want

Implementation:

* We start by creating a builder
  → Identify the "parts" of the product & provide methods to create those parts.
  → It should provide a method to "assemble" or build the product / Object.
  → It must provide a way/method to get fully build obj out. Optionally builder can keep referance to an product it has build so the same can be returned again in the future

* A director can be a seperate class (or) client can play the role of director.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Considerations

* You can easily create an immutable class by implementing builder as an inner static class. You will find this type of implementation used quite frequently even if immutability is not a main concern.

Design Considerations

* The director role is rarely implemented as a separate class, typically the consumer of the object instance or the client handles that role.
* Abstract builder is also not required if "product" itself is not part of any inheritance heirarchy. You can directly create concrete builder
* If you are running into a "too many constructor arguments" problem, then it's a good indication that builder pattern may help.

Ex:- * StringBuilder class & buffer class
* Java - builder. calander.

Drawbacks

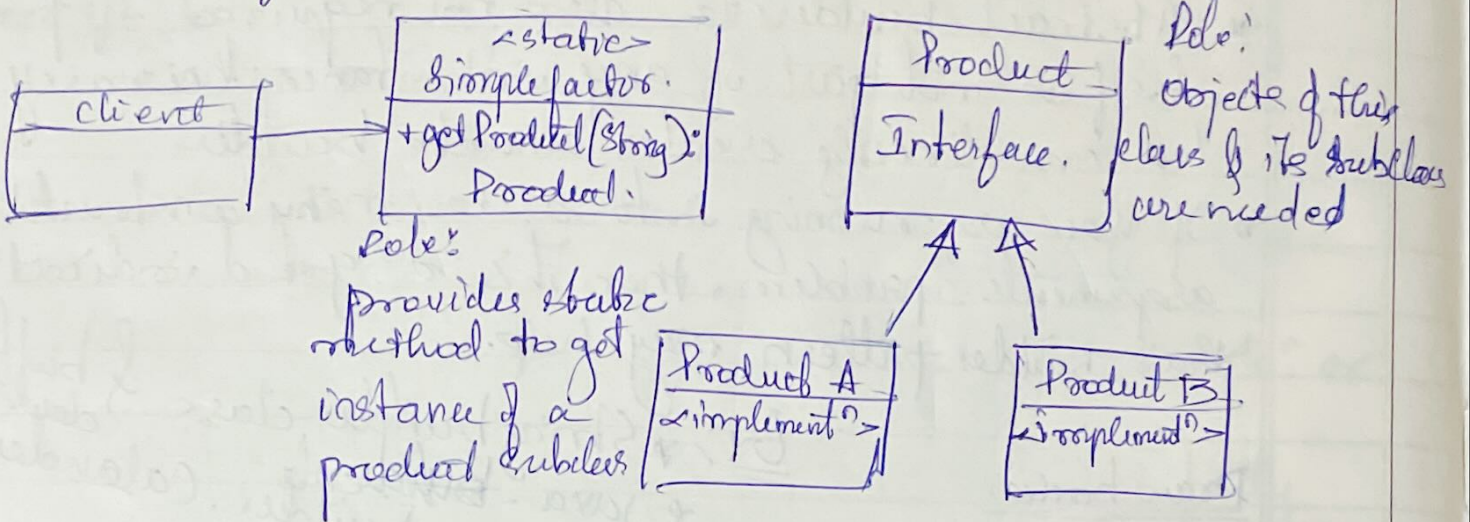
* A little bit complex for new comers mainly because of 'method chaining', where builder methods return builder object itself.
* Possibility of partially initialized obj ; User code can set only few or none of the properties using withXXX methods and call build(). If req properties are missing, build method should provide suitable defaults (or) throw exception.

BIT

2) Simple factory

* Here we simply move the instantiation logic to a seperate class and most commonly to a static method of this class.
* Some do not consider simple factory to be a design pattern, as it is simple a method that encapsulates object instantiation. Nothing complex goes on in that method.
* Typically we would to do this if we have more than one option when instantiating object and a simple logic is used to choose correct class.

UML Diagram -



Implement

* We start by creating a seperate class for our simple factory
  - → Add a method which returns desired obj instance
    - i) This method typically static & will accept some argument to decide which class to instantiate
    - ii) You can also provide additional arguments which will be used to instantiate objects.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Imp Considerations

* Simple factory can be just a method in existing class. Adding a separate class however allows other parts of your code to use simple factory more easily.
* Simple factory itself doesn't need any state tracking so its best to keep this as a static method.

Design Considerations

* Simple factory will in turn may use other design pattern like builder to construct object.
* In case if you want to specialize your simple factory in subclasses, you need factory method design pattern instead.

Ex:- java.text.NumberFormat

Drawback

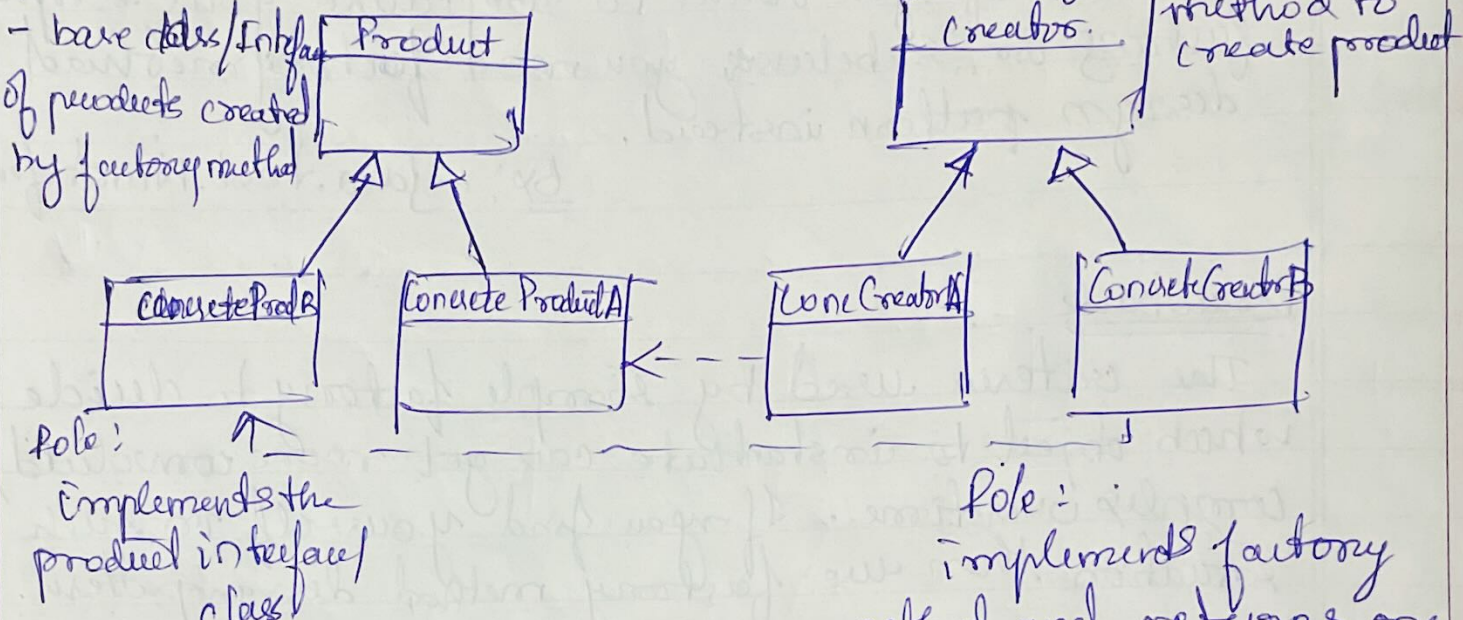
The criteria used by simple factory to decide which object to instantiate can get more convoluted / complex over time. If you find yourself in such situation then use factory method design pattern.

BIT

3) Factory Method

* We want to move the object creation logic from our code to seperate class.
* We use this pattern when we do not know in advance which class we may need to instantiate beforehand & also allow new classes to be added to system & handle their creation without affecting client code.
* We let subclasses decide which object to instantiate by overriding the factory method.

UML



Ex:- java.util.AbstractCollector → Iterator();

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Implement

* We start by creating a class for our creator
  → Creator itself can be concrete if it can provide a default object (or) it can be abstract.
  → Implementation will override the method & return an object.

Imp Considerations

* The creator can be a concrete class & provide a default implementation for the factory method. In such cases you will create some "default" object in base creator.
* You can also use the simple factory way of accepting additional arguments to choose between different object types. Subclasses can override factory method to selectively create different object of the same criteria.

Design considerations

* Creator hierarchy in factory method pattern reflects the product hierarchy. We typically end up with a concrete creator per object type.
* Template method design pattern often makes use of factory methods.
* Another creational design pattern called "abstract factory" make use of factory method pattern.

BIT

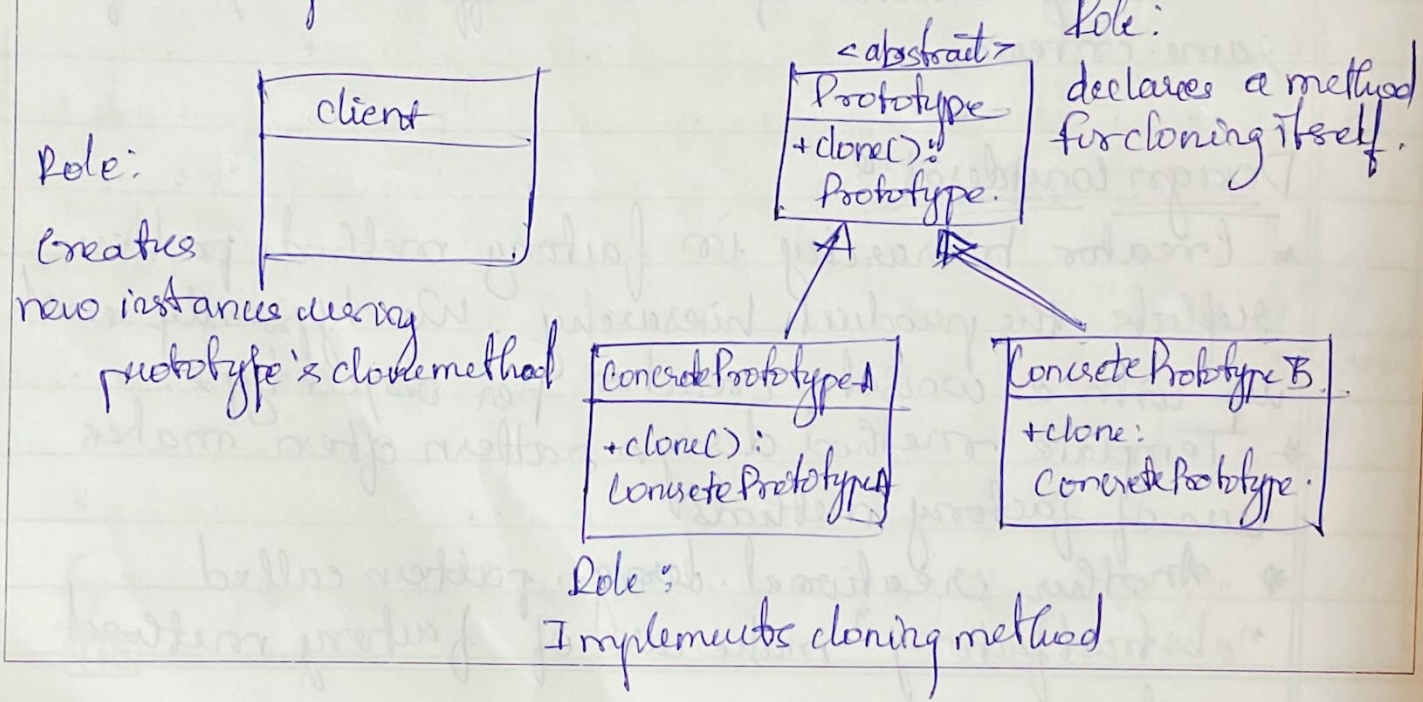
Drawback

* More complex to implement. More classes involved and need unit testing
* You have to start with Factory method design pattern from the beginning. Its not easy to refactor existing code into factory method pattern.
* Sometimes this pattern forces you to subclass just to create appropriate instance.

4) Prototype

* We have a complex object that is costly to create. To create more instances of such class, we use an existing instance as our prototype.
* Prototype will allow us to make copies of existing object & save us from having to recreate objects from scratch.

UML Diagram



Name of the Experiment ..... Page No.:

Experiment No..... Date :

Implementation

* We start by creating a class which will be a prototype
  → The class must implement Cloneable interface
  → Class should override clone method and return copy of itself
  → The method should declare CloneNotSupportedException in throws class to give subclasses chance to decide on whether to support cloning
* Clone method implementation should consider the deep & shallow copy and choose whichever is applicable.

Impl Consideration

* Pay attention to the deep copy & shallow copy of references. Immutable fields or clones save the trouble of deep copy
* Make sure to reset the mutable state of an object before returning the prototype. Its a good idea to implement this in method to allow subclasses to initialize themself.
* clone() method is protected in Object class and must be overridden to be public to be callable from outside the class.
* Cloneable is a "marker" interface, an indicator that the class supports cloning.

BIT

Design Considerations

* Prototypes are useful when you have large objects where majority of state is unchanged b/w instances and you can easily identify that state.
* A prototype registry is a class where in you can register various prototypes which other code can access to clone out instances. This solves the issue of getting access to initial instance.
* Prototypes are useful when working with composite/Decorator patterns.

Ex :- Object.clone();

→ This method is provided by Java and can clone an existing object, thus allowing any object to act as a prototype. Classes still need to be Clonable but the method does the job of cloning object

Drawback

* Usability depends upon the no. of properties in state that are immutable or can be shallow copied. An object where state is comprised of large number of mutable objects is complicated to clone().
* In java the default clone operation will only perform the shallow copy so if you need a deep copy you have to implement it.
* Subclasses may not be able to support clone and so the code becomes complicated as you have to code for situations where an implementation may not support clone.

Name of the Experiment ..... Page No.:

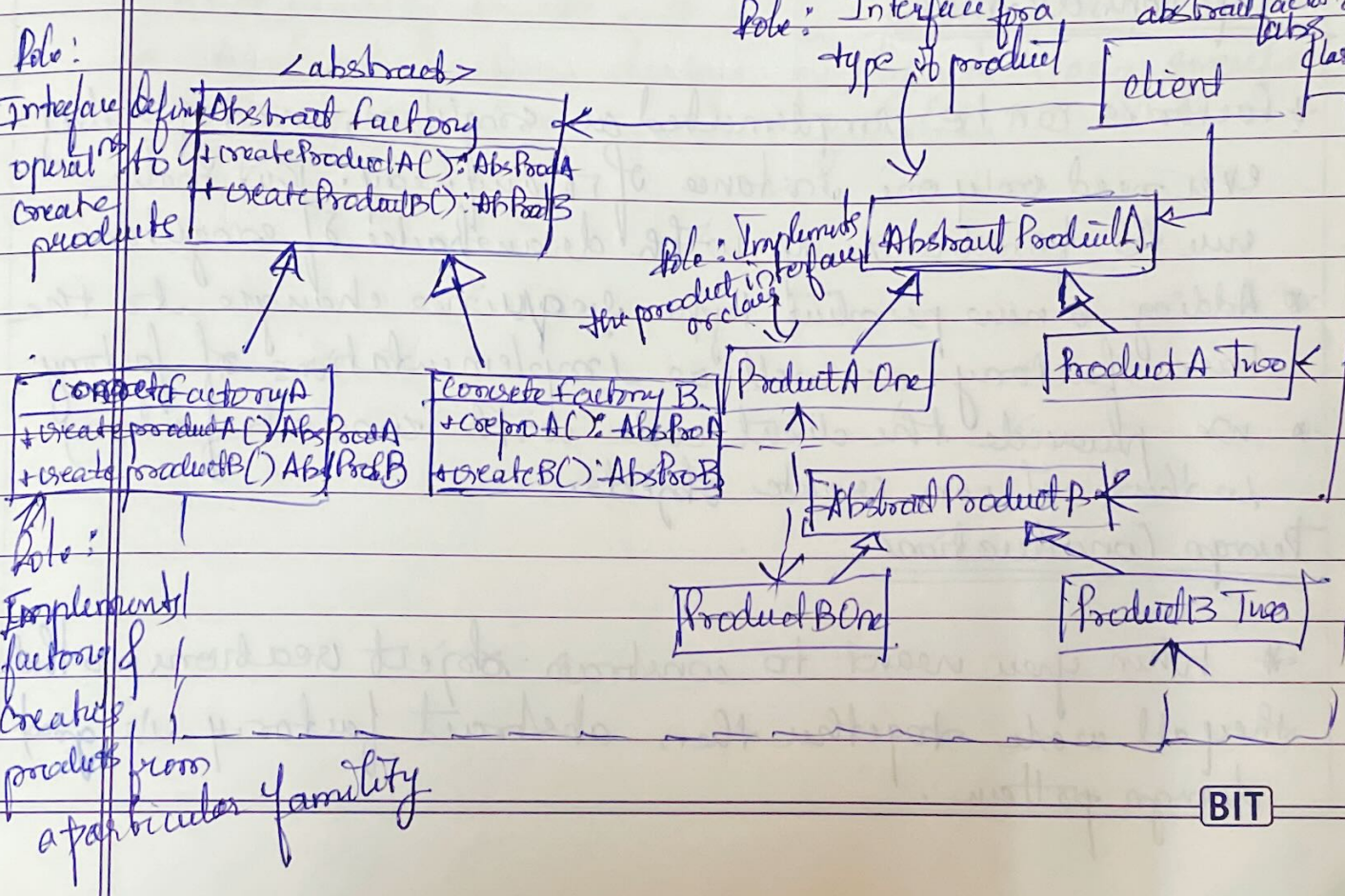
Experiment No..... Date :

5) Abstract factory.

* Abstract factory is used when we have two (or) more objects which work together forming a kit or set and there can be multiple sets. Document Builder factory libs that can be created by client code.
* So we can separate client code from concrete objects forming such a set & also from the code which creates the sets

factory method pattern to create then you can any implementations to ins to create objects.

implemented then factory method types changes to base ... of factor.



BIT

Design Considerations

* Prototypes are useful when majority of state is ... can easily identify
* A prototype registry ... registries various pr ... can access to clone or ... of getting access to it
* Prototype are useful ... Decorator patterns

⟶ This method clone an ...

Implementation

* We start by studying the product "sets".
  → Create abstract factory as an abstract class or an interface
  → Abstract factory defines abstract methods for creating products.
  → Provide concrete implementation of factory for each set of products.
* Abstract factory makes use of factory method pattern. You can think of abstract factory as an object with multiple factory method.

Imp Considerations

* Factories can be implemented as singletons, we typically ever need only one instance of it anyway. But make sure to familiarize with drawbacks of singleton.
* Adding a new product type requires changes to the base factory as well as implementations of factory
* We provide the client code with concrete factory so that it can create objects.

Design Considerations

* When you want to constrain object creations so that they all work together then abstract factory is good design pattern.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

* Abstract factory uses factory method pattern
* If objects are expensive to create then you can transparently switch factory implementations to use prototype design patterns to create objects.

Ex:- The javax.xml.parsers.DocumentBuilderFactory

Drawbacks

* A lot more complex to implement than factory method
* Adding a new product requires changes to base factory as well as all implementations of factory
* Difficult to visualize the need at start of development and usually starts out as a factory method.
* Abstract factory design pattern is very specific to the problem of "product families".

Design Patterns

Structural Design Patterns

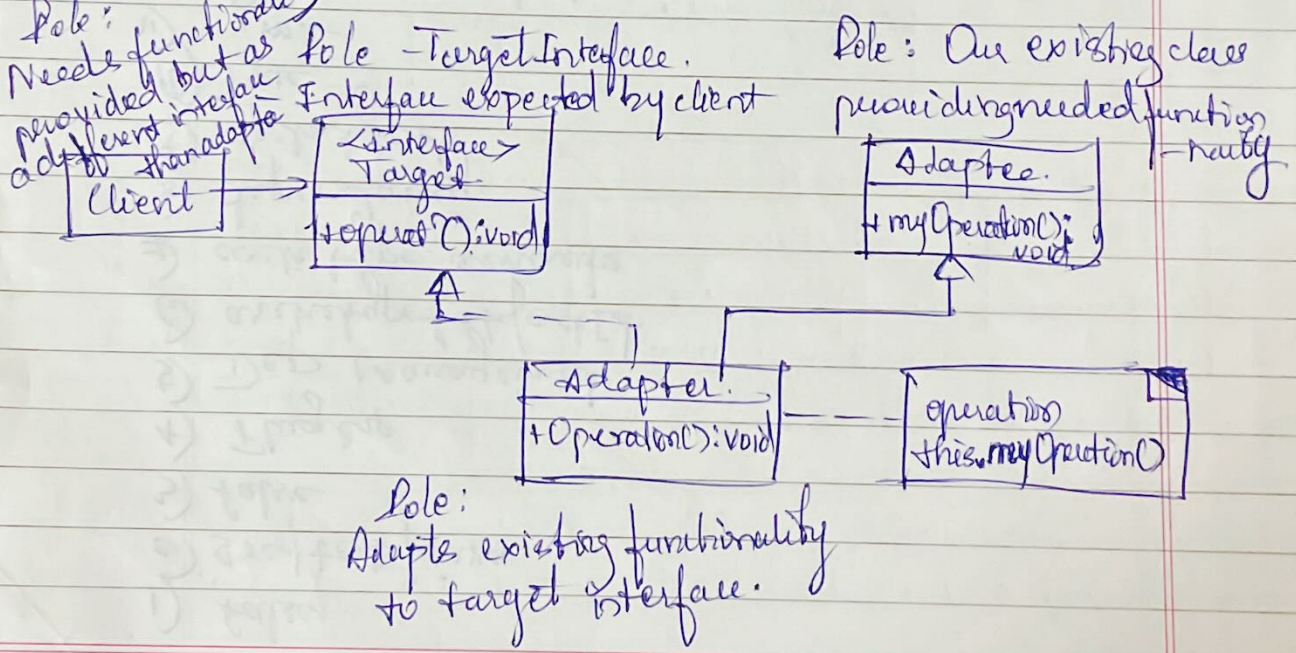
They deal with how classes & objects are arranged(or) composed.

* Adapter
* Bridge
* Decorator
* Composit
* Facade
* Flyweight
* Proxy.

1) Adapter

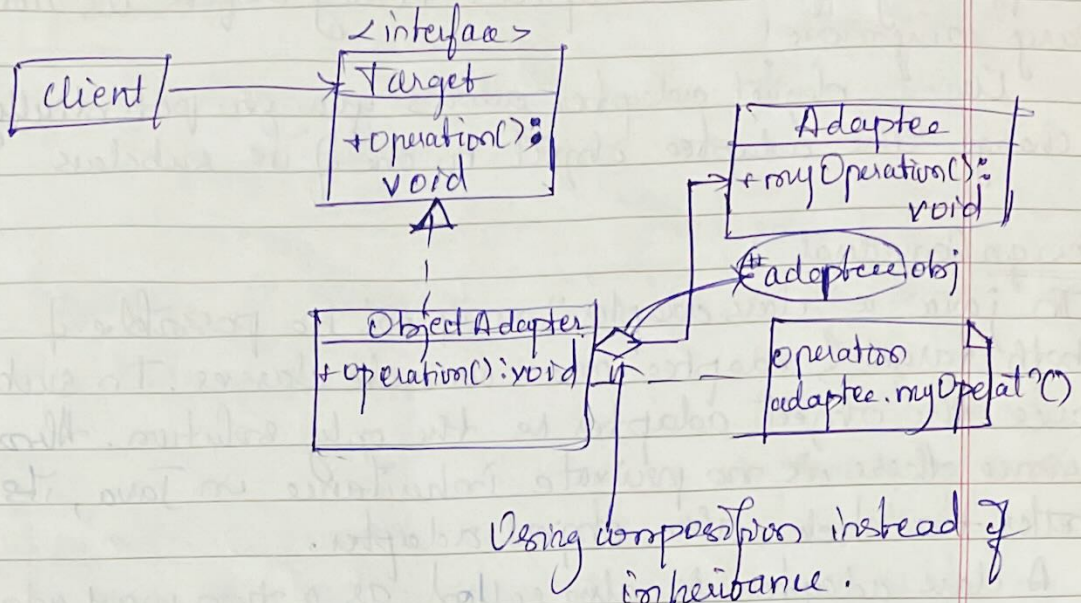
* We have an existing object which provides the functionality that client needs. But client code can't use this object because it expects an object with diffrent interface.
* Using adapter design pattern we make this existing object work with client by adapting the object to client's expected interface
* This pattern is also called as wrapper as it "wraps" existing object.

UML (class adapter/2 way adapter)



Role:
Adapts existing functionality to target interface.

(Object Adapter) UML



Implementation

* We start by creating a class for Adapter.
  → Adapter must implement the interface expected by client.
  → First we are going to try out a class Adapter by also extending from our existing class
  → In the class adapter implement, we are simply going to forward the method to another method inherited from adaptee.
  → In Object adapter - we are only going to implement target interface and accept adaptee as constructor argument in adapter i.e., make use of composition
* An object adapter should take adaptee as an argument in constructor or as a less preferred solution, you can instantiate it in the constructor thus tightly coupling with a specific adaptee.

Imp Considerations

* How much work the adapter does depends upon the differences b/w target interface and object being adapted.
  If the method arguments are same or similar, adapter has very less work to do.

* Using class adaptor "allows" you to override some of the adaptee's behaviour. But this has to be avoided as you end up with adapter that behaves differently than adaptee. Fixing defects is not easy anymore!
* Using object adapter allows you to potentially change the adaptee object to one of its subclass

Design Considerations

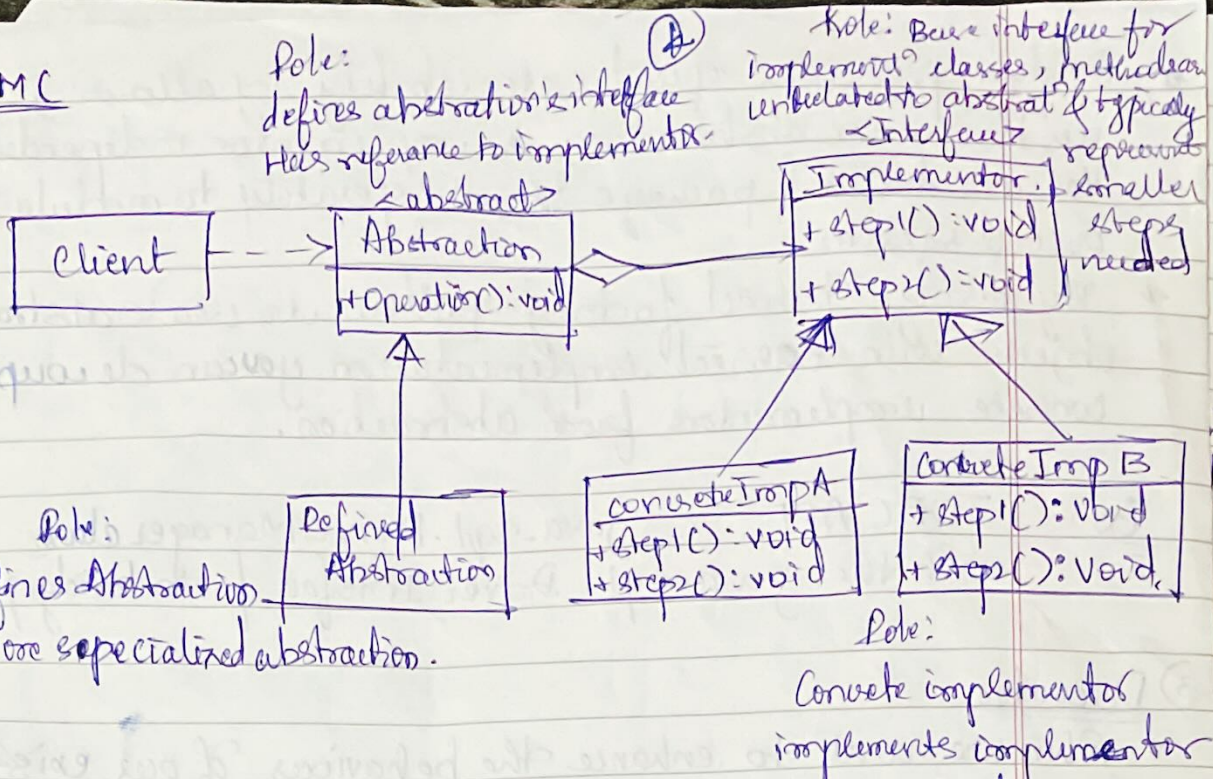
* In java a "class adapter" may not be possible if both target & adaptee are concrete classes. In such cases the object adapter is the only solution. Also since there is no private inheritance in Java, its better to stick with object adapter.
* A class adapter is also called as a two way adapter. Since it can stand in for both the target interface and for the adaptee. That is we can use object of adapter where either target interface is expected as well as where an adaptee object is expected.

Ex:- java.io.InputStreamReader and
java.io.OutputStreamWriter.
These classes adapt existing InputStream/OutputStream object to a Reader/Writer interface.

2) Bridge

* Our implementations & abstractions are generally coupled to each other in normal inheritance.
* Using bridge pattern we can decouple them so that they can both change without affecting each other.
* We achieve this feat by creating two separate inheritance hierarchies; one for implementation & another for abstraction.
* We use composition to bridge these two hierarchies.

UML



Role:
Refines Abstraction
- More specialized abstraction.

Role:
Concrete implementor implements implementor method.

Implementation

* We start by defining our abstraction as needed by client
  * → We determine common base operations & define them in abstraction.
  → We can optionally also define a refined abstraction & provide more specialized operations.
  → Then we define our implementor resp. Implementor methods do NOT have to match with abstractions. However abstraction can carry out its work by using implementor methods.
  → Then we can write one/more concrete implementor providing implementation.
* Abstractions are created by composing them with an instance of concrete implementor which is used by method in abstraction.

Impl considerations

* In case we are ever going to have a single implement° then we can skip creating abstract implementor.
* Abstrat° can decide on its own which concrete implementor to use in its constructor or we can delegate that decision to a 3rd clas.
  In last approach abstractions remains unaware of concrete implementors & provide greater decoupling

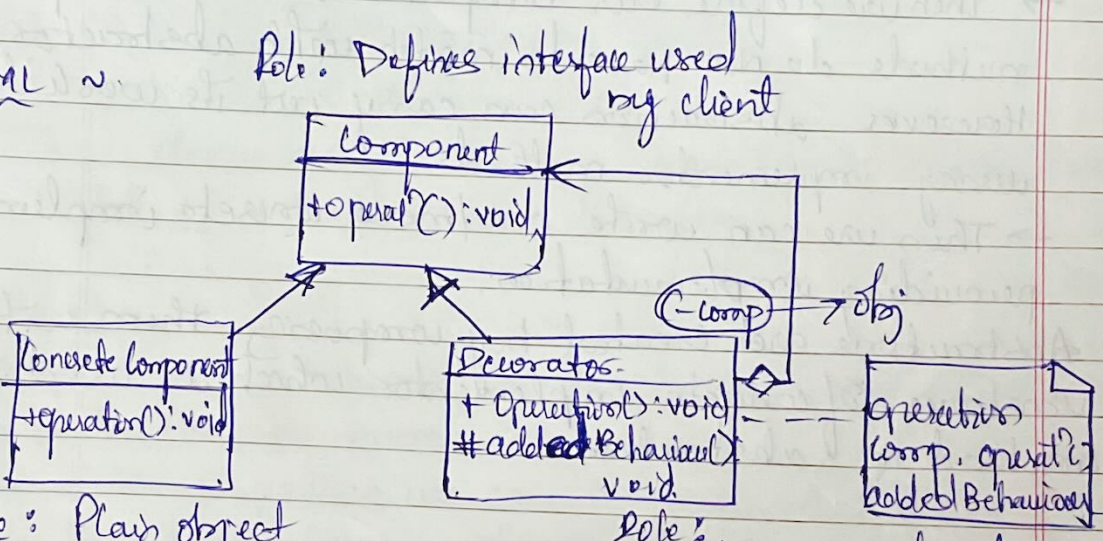
* Bridge provides great extensibility by allowing us to change abstraction & implementor independently You can build & package them seperately to modularize overall system
* By using abstract factory pattern to create abstraction objects with correct implementation. you can decouple concrete implementors from abstraction.

Ex:- JDBC API ⇒ java.sql.DriverManager class. with the java.sql.Driver Interface from bridge pattern.

3) Decorator

* When we want to enhance the behaviour of our existing object dynamically we and when required then we can use decorator design pattern.
* Decorator wraps an object within itself and provides same interface as the wrapped object. so the client of original obj doesn't need to change.
* Decorator provides alternative to subclassing for extending functionality of existing class.

UML :-



Implement

* We start with our component
* → Component defines interface needed or already used by client.
  → Concrete component implements the component
  → We define our decorator. Decorator implements component & also needs reference to concrete component
  → In decorator methods we provide additional behaviour on top that provided by concrete component instance.

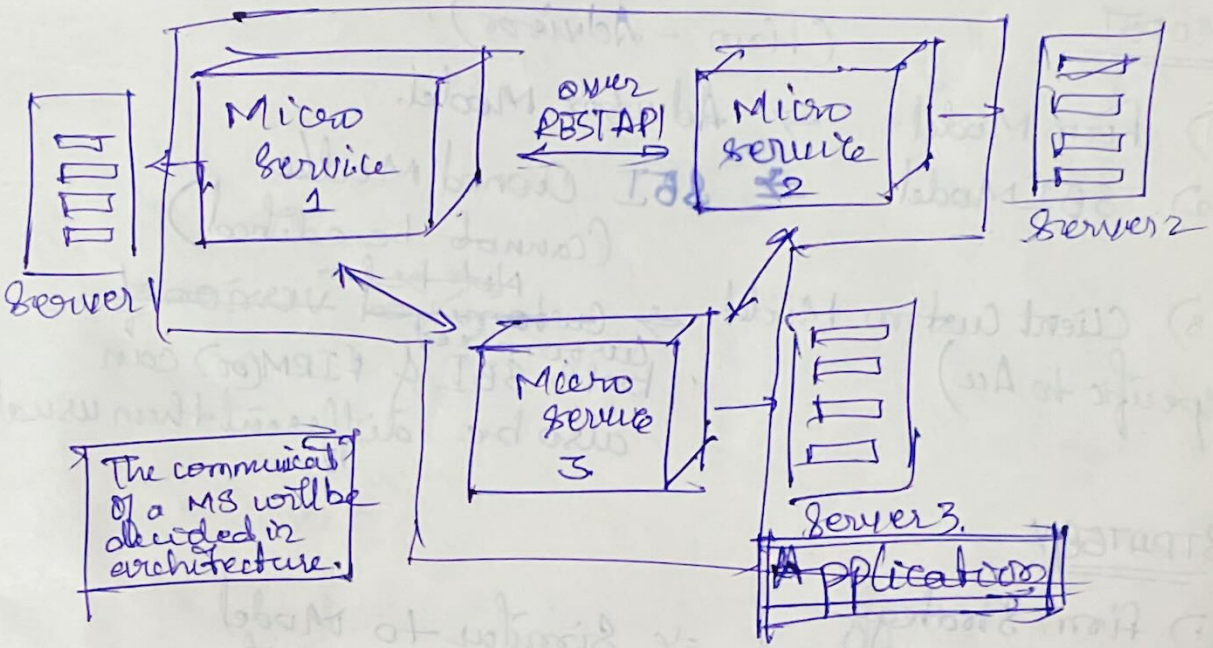
Ex:- Java I/O Package ⇒ java.io.BufferOutputStream class.

Drawbacks

* Often results in large number of classes being added to system, where each class adds a small amount of functionality. You often end up with lots of objects, one nested inside another and so on.
* Sometimes new comers will start using it as a replacement of inheritance in every scenario. Think of decorators as a thin skin over existing object.

Microservices (pranshi)

→ Microservices are a way of breaking an application or service down into standalone independent applications that can be run on different hardware or server instances where they all talk to each other over a REST API and work together to provide the functionality of an application as a whole (product).



MMT Downstream

	Integration point	Third Party
1)	Advisory world	Yes
2)	On Base	Yes
3)	Product Library	No.
4)	ADS	No.
5)	PMB	No.
6)	CAR	Yes

In MMT (16-278) 9,910 + 10,723.55
20,633.78

→ MODEL ⇒ Superior.
It can have strategies, funds, individual securities, 3rd party products

→ STRATEGY ⇒ Inferior
It will only have funds, individual securities.

MODEL (firm - Advisor)

1) Firm Model → Advisor Model.
2) SEI Model ⇒ SEI Owned Model (Cannot be edited)
3) Client Custom Model (Specific to Acc) ⇒ ~~Not be from~~ ~~Customised~~ ~~version of~~ Customised both SEI & FIRM(or) can also be different than usual

STRATEGY

1) firm Strategy
2) SEI Strategy
} ⇒ Similar to Model functionality.

MMT Dependent Teams (Work with)

1) Common Products
To give Models, Strategies, funds, Third Party Managers.
GetModels, GetManagers GetFunds.

2) PMB
Basically for authorizal (automate).
association & Rebalance

3) CAR - Client Acquision Review (3rd Party)

CAR applicat° will be used to show the demo. where MMT is a actual applicat° where creat° of Model & Strs real

A) ADS

They are search components..
Similar to Common Products but like a different DB of securities, funds etc

IAS-Common Product Lib
Mansi 29

JAVA - 8

FUNCTIONAL STYLE PROGRAMMING

All these days, we were passing primitives and object references as ~~attributes~~ arguments to the method, but now, with lambda functionality we can pass "a function" itself as a ~~attribute~~ arguments

Advantages

→ Compact and efficient code
→ parallelism.

(~~pre~~ - Java 8)

Method args ← (primitives & Obj ref)

(Java 8 :)

Method args ← primitives, Obj ref, methods)

Functional Interface

An interface which has only 1 method

Syntax :-

(parameters) → expression.

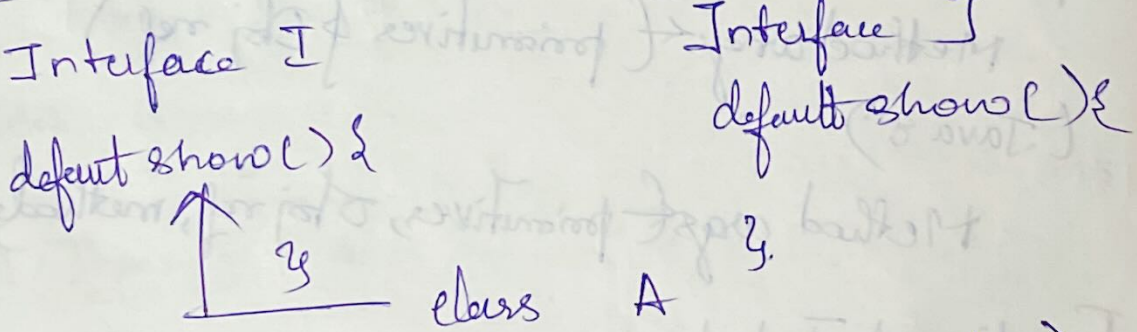
Default Methods

```
public interface name{
    public void dosomething();
    public void donot dosomething();
}

public interface name2 {
default pubvc void dvanything(){
}       SOP("Doing anything");
```

In this feature, we will be able to define a method in the interface. This will help us directly implement & use in the required classes without declaring it.

Sc-1



1) If class A implement only interface (I) then no need of any method defination
2) If class A implements both (I) & (J) Interface, then it would cause a bottleneck issues, thus, in that stage, we have to override the show method in class A (which ever the behaviour required, say from I (or) J)

⇒ Static methods

```
public interface staticmethodstrait {
    public static void show() {
        SOP ("static");
    }
}
```

Here in this scenario, without using default keyword, we can actually define a method, but instead of "default", we should use "static".

With this, we get an add on advantage, where we can actually, use the methods of the interfaces without creating any objects.

Just use Interface name.method()

ForEach & [Stream API]

```
pubwc static void main (String[] args)
{
    List<Integer> values = Arrays.asList(1,2...6);

① for (int i=0; i<6 ; i++)
  {  SOP (values.get(i));
  }

② Iterator<Integer> i = values.iterator();
  While (i.hasNext())
  {
      SOP (i.next());
  }

③ for (int i : values)
  {  SOP (i); }.
```

① Normal for loop that we use it when we want to iterate to & fro (or) from the middle.

② Iterator, where it is an interface, part of `java.util` package where we can iterate forward and even make changes / modifications.

③ Enhanced for loop, it is used only when we have to print all the data irrespective of starting & ending point in [Only 1 direction] forward direction.

④ foreach loop, It is called as "Internal Iteration", as in Java 8, they have introduced this new method, where the iteration can carry out internally which is a part of StreamAPI.

This foreach loops, needs an object of Consumer interface where consumer interface is a functional interface i.e., we can actually use lambda expressions. hence

```
values.forEach((i) -> SOP(i));
```

for iteration — functional interface

It tells that, for every value of "i", it prints the actual value of i (say from 1 to 6)

(1)(2)(3) → They are all called as external iterations, as the objects fetch the data from the for loops that are iterating externally.

Consumer Interface (java.util.function.Consumer)

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}

public static void main(String[] args)
{
    List<Integer> values = Arrays.asList(1,2,3...6);

    Consumer<Integer> c = new Consumer<Integer>()
    {
        void accept(Integer i)
        {
            SOP(i);
        }
    };

    values.forEach(c);
```

anonymous class

functional method

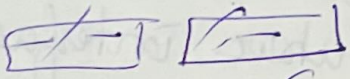
```
Consumer<Integer> c = (Integer i) -> SOP(i);
values.forEach(c);
```

```
Consumer<Integer> c = i -> SOP(i)
```

As we know, it is of Integer type.

Since we are using Object "c" only once, we can even eliminate that as well. Hence, the ~~out~~come is,

```
values.forEach(i -> SOP(i));
```



Imparitive Programming

The traditional way of programming is called as Imparitive programming. Here - we have to specify How & what we are doing.

Declarative Programming

The functional style programming is called as Declarative programming. Here we just tell what to be done. How it is done is completely managed by computer.

Lambdas

* Lambdas are basically functions, i.e., a function by itself that can be passed as a whole.
* Enables us writing more compact, faster and cleaner code.
* Before Lambdas, Anonymous classes were used to perform similar task of passing around
* λ = They are the Mathematical notation for functions.
* It was introduced by Alonzo Church in 1930s.
* Lambda - calculus functions are always anonymous i.e., they are always nameless.

* λx → x * x → square of the argument is computed
  - argument
  - body

What is a Lambda?

* Anonymous function
* Compact way to define functions
* It is useful for passing around functionality and it helps in doing so in a very compact way.
* Since it can be passed around, it can basically be termed as a expression. i.e., λ exp.
* LISP, Scala, C#, Ruby, C++, Python support λ exp.

Syntax

(Type param1, Type param2,..) → {

```
//statement1...
// statement2...
...
return something;
}.
```

Lambda exp is assigned to a variable whose type is a functional interface.

i.e., ⇓ Target Type.

functional interface variable ← λ

Lambda Example

```
Set<String> set = new TreeSet<String>(
  new Comparator<String>() {
    public int compare(String s1,
                       String s2) {
      return s1.length() -
             s2.length();
    }
  }); // Anonymous Class.
```

// Lambda Exp

```
TreeSet<String>(String s1, String s2)
  -> {return s1.length() - s2.length();}
```

```
TreeSet<String>(s1, s2) ->
  {return s1.length() - s2.length();}
```

```
TreeSet<String>(s1, s2) ->
  s1.length() - s2.length()
```

Hence the generalised syntax is

~~s1.length()~~ - ~~s2.length()~~

(parameters) -> expression.

Here comparator is a functional interface, hence this can be done, else it would throw an exception.

Hence,

"LAMBDA EXP CAN ONLY BE AN IMPLEMENTATION OF A FUNCTIONAL INTERFACE".

Other Examples:

```
() -> { }   // return void
(Book b) -> SOP(b.getTitle());
            // void but gets title.
```

```
(Book b) -> {return b.rating() >= 4.5;}
  => b -> b.rating() >= 4.5;   // boolean
  => () -> "java";   // return String.
  => () -> {return "java";}.
```

Cannot be done.

```
() -> {"java";}
  => not statement !!
```

```
() -> return "java";
  not expression.
```

based on bytecode instruction

Anonymous Class vs Lambdas

Anonymous class	Lambdas.
* Has associated obj + new()	* No associated obj (invoked during — runtime)
* verbose	* Compact.
* Instantiated on every use unless they are declared as singletons using static, final fields	* Memory is allocated only one for the method.
* Target Type (Class/Interface) can have multiple methods.	* Works with only functional interface