

Phase1

Phase 1 : Java + Servlet API

Phase 2: Spring Framework

Phase 3: DevOps

Java - 8

IoT

Java-8 : Functional Programming

Functional Interface
default method
static method
Lambdas
Streams
Method references
Optional
Concurrent Support in Collection API
DateTime API
Nashorn Engine (JS engine)

Imperative style of programming

- # Classical style/Traditional style
- # pure OOPs
- # Focus how to perform operation
- # Object mutability : bugs

Declarative Style

- # Focus on result you want
- # Analogous SQL
- # Object Immutability
- # Functional Programming

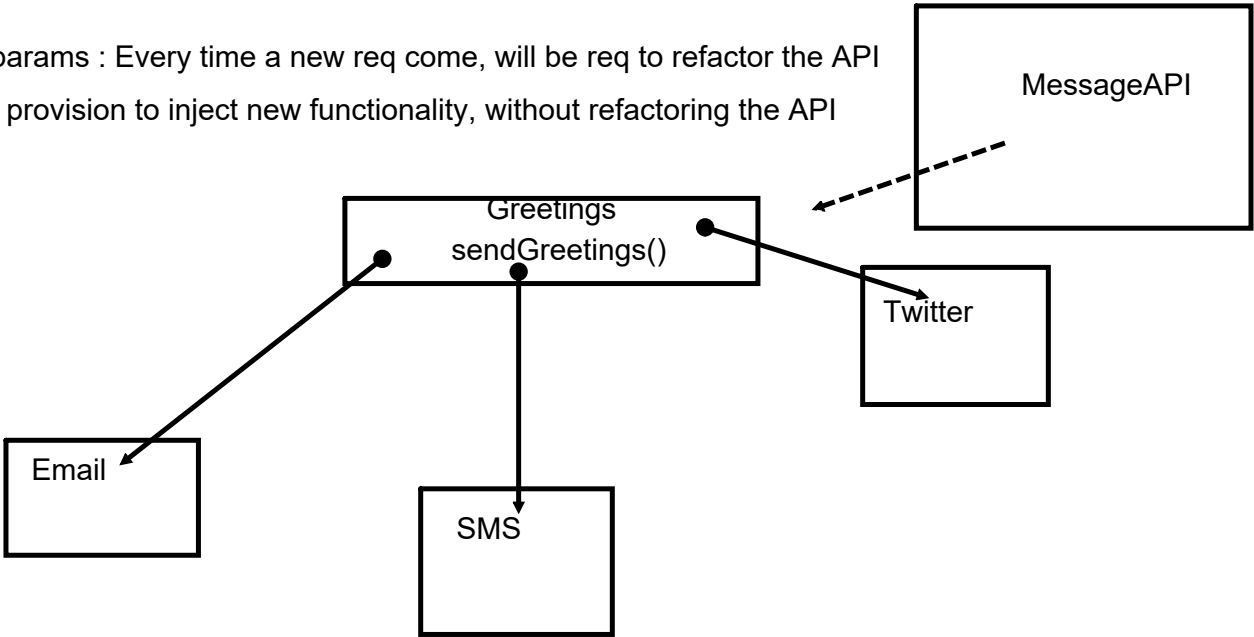
List of numbers

fetch unique number

Reverse domain naming convention

Phase1

params : Every time a new req come, will be req to refactor the API
: provision to inject new functionality, without refactoring the API



Functional Programming : Functions(pure) are first class citizens

No Object Overheads

variable/instance/reference : object
reference = function

New datatype would not have been
backward compatible

Expect a special datatype from JAVA : Function
Function twitter = ()

Extended the behavior of existing feature : interface

Syntax : Lambda

1. no access modifier : (not the part of any class)
2. no name (anonymous function)
3. no return type (can return values)
4. params : no param type
5. <param> -> {<definition>}

```
void fun(){  
}  
()  
-> {  
}
```

```
void fun(String str1,String str2){  
}  
(str1,str2)->{  
}
```

```
void fun (String str){  
}  
str -> {  
}
```

```
void fun(String str){  
    <single inst>  
}  
str-> <single inst>
```

```
void fun(String str){  
    -----  
    -----  
}  
  
str -> {  
    -----  
    -----  
}
```

```
void add(int a, int b){  
    return a+b;  
}  
(a,b)-> a+b; // return is by default associated  
(a,b) -> {  
    return a+b;  
}
```

Functional Interface

Contains only 1 abstract method, any number of default and static

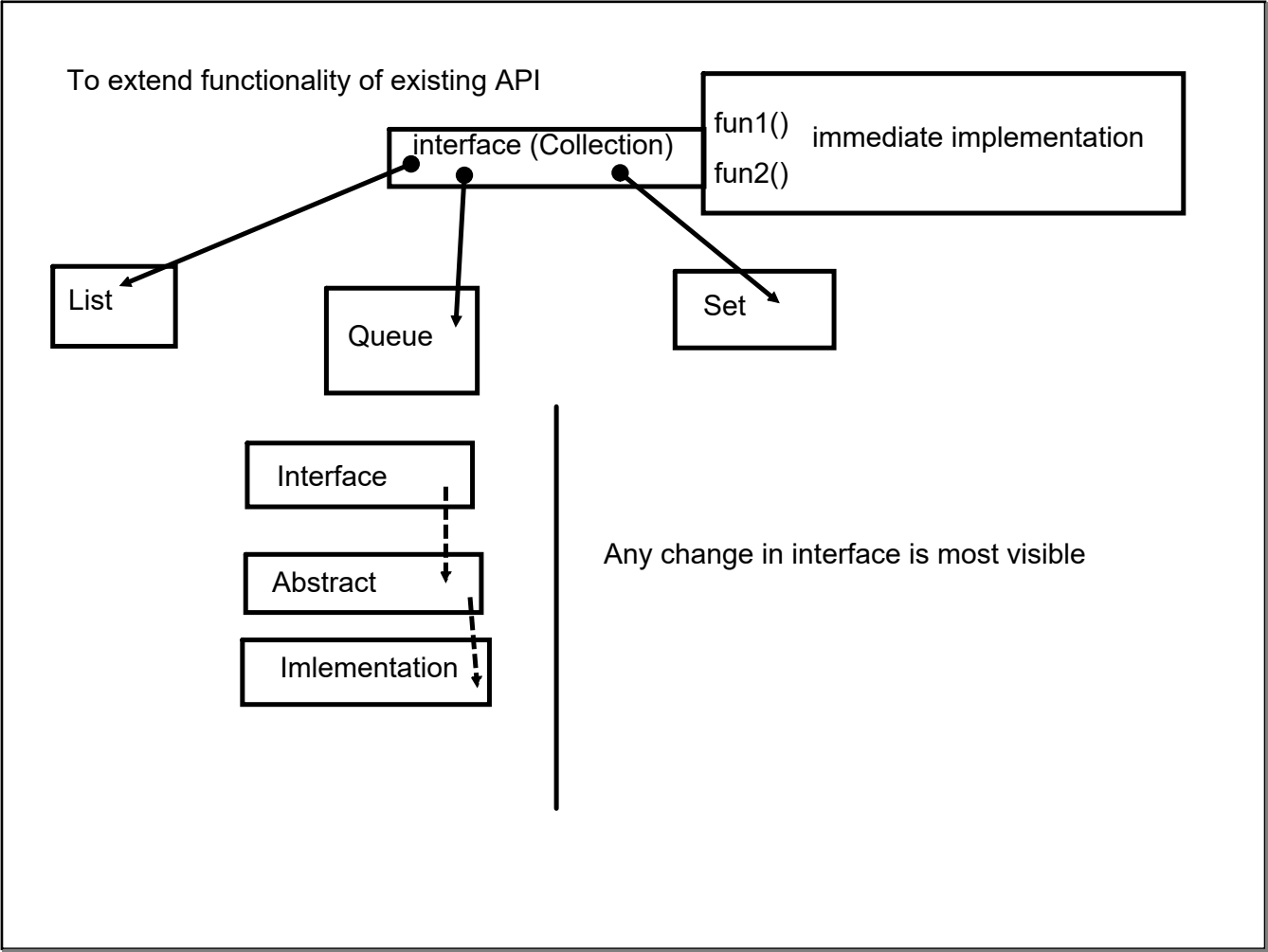
Only Functional Interfaces can refer to lambda expression

Signature of Lambda expression must match with the only abstract method of FI

Interface :

Define function inside an interface.

Interface can have functions with definitions as well



Existing feature :

#Functional Interface

Comparable

Comparator

Runnable

=> Specialized Libraries of Functional Interface

=> Streams

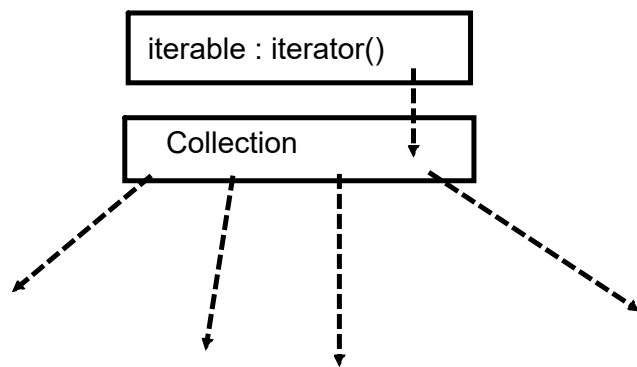
Lambdas with Local Variable

1. Lambdas have an access over local variables and instance of enclosing scope
2. Effectively final
3. Not Allowed to use the same local variable name as param or redeclaration inside body

No restriction on instance variable

=> Easier to perform the concurrent operation : immutability

Collection of string : traverse through collection and display all strings in collection



Functional API : Bunch of functional interface:

few prototypes have been identified with common usage

java.util.function

Consumer: BiConsumer

void accept(<>) : Consume the data

Predicate : BiPredicate

boolean test(<>) : Add some condition and revert back

Function : BiFunction, UnaryOperation, BinaryOperator

<> apply(<>) : Transformation

Supplier

<> get() :

Streams : Pure Functional

Perform operations on collections or I/O resource :

Safe

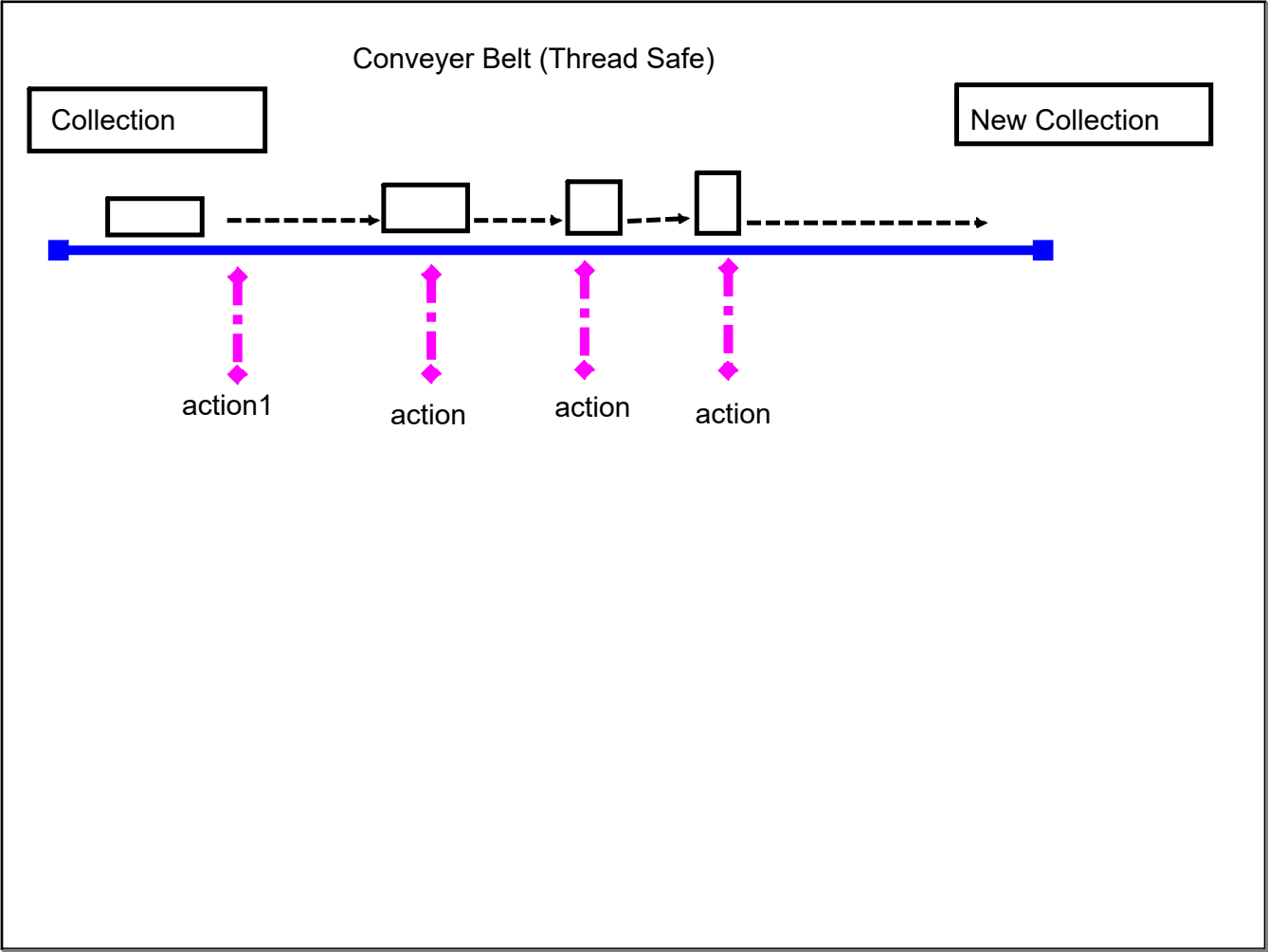
Immutability : Thread safe

Efficient Way

Not a data structure: not going to store any data

Lazy processing model

Parallel Stream : Parallel operation easily without spawning the thread



SBA1 : use-case

SBA2 : use-case

SBA3 :

End-to-End

1. continuous process
2. Milestone
3. walk-through (Friday)
4. Group based : group evaluation + individual eval

Every Stream must have a terminal activity
Else : Stream will not initiate

Every Stream

1. Initiate the stream
2. Intermediate activity (optional)
3. Terminal activity

Parallel Stream

Constraints in parallel streams

1. Order of records matter
2. where using a mutable service/data : Not a thread safe
3. activities, inherently complex, degrade performance

1,2,3,4,5,6,7.....

4,2,6,3,1

result = 0

forEach(Consumer)

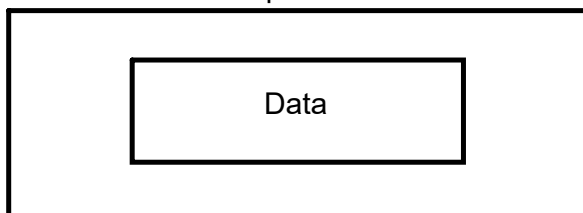
4 --> thread1
= 0 + 4
result = result + 4

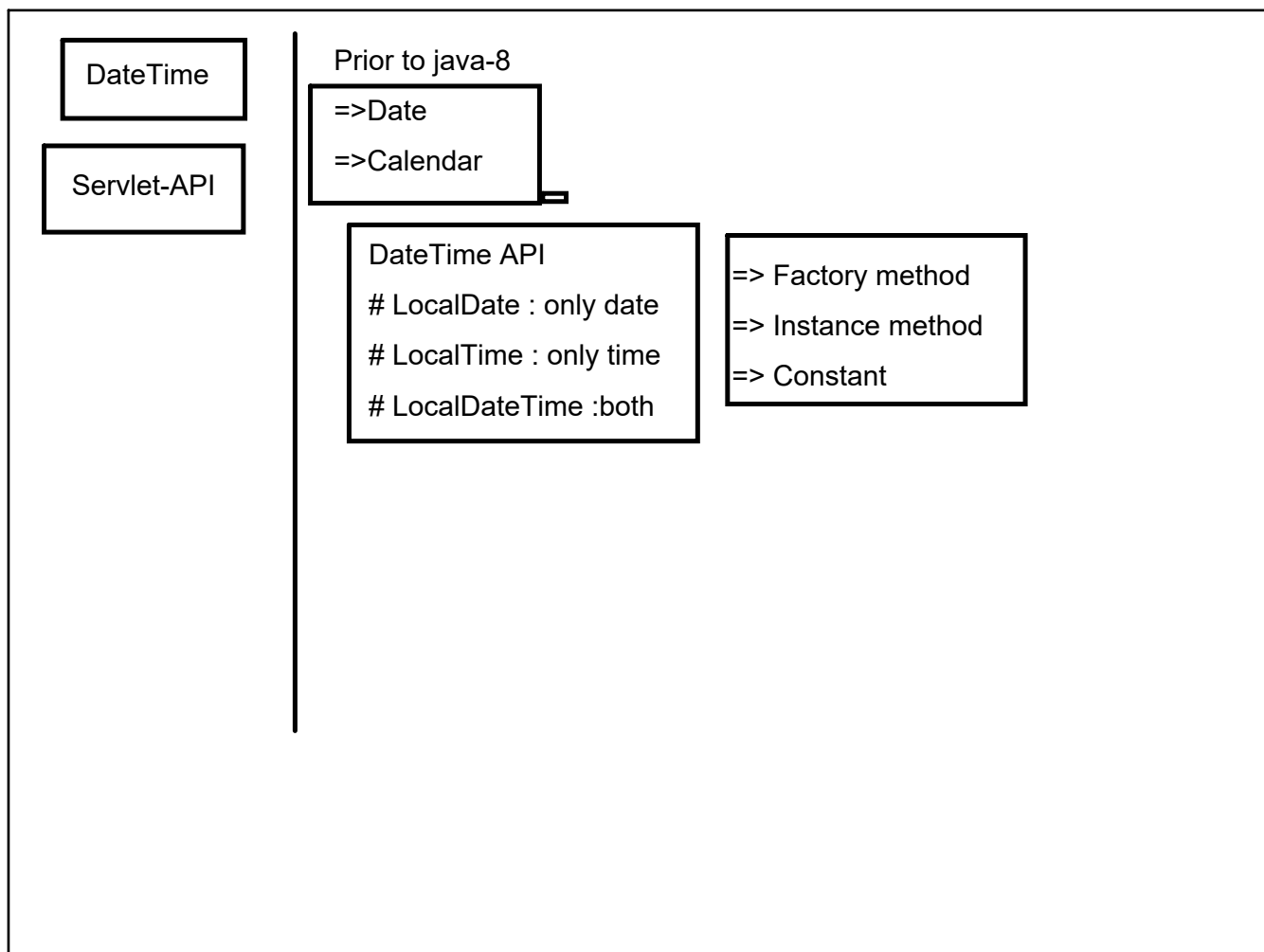
2 --> thread2
= 0 + 2
result = result + 2

Optional : to avoid null reference exception
if any object being initialized by some external logic
need to check if it is

Suggested best Practise : never return raw data

Optional





Servlet-API:

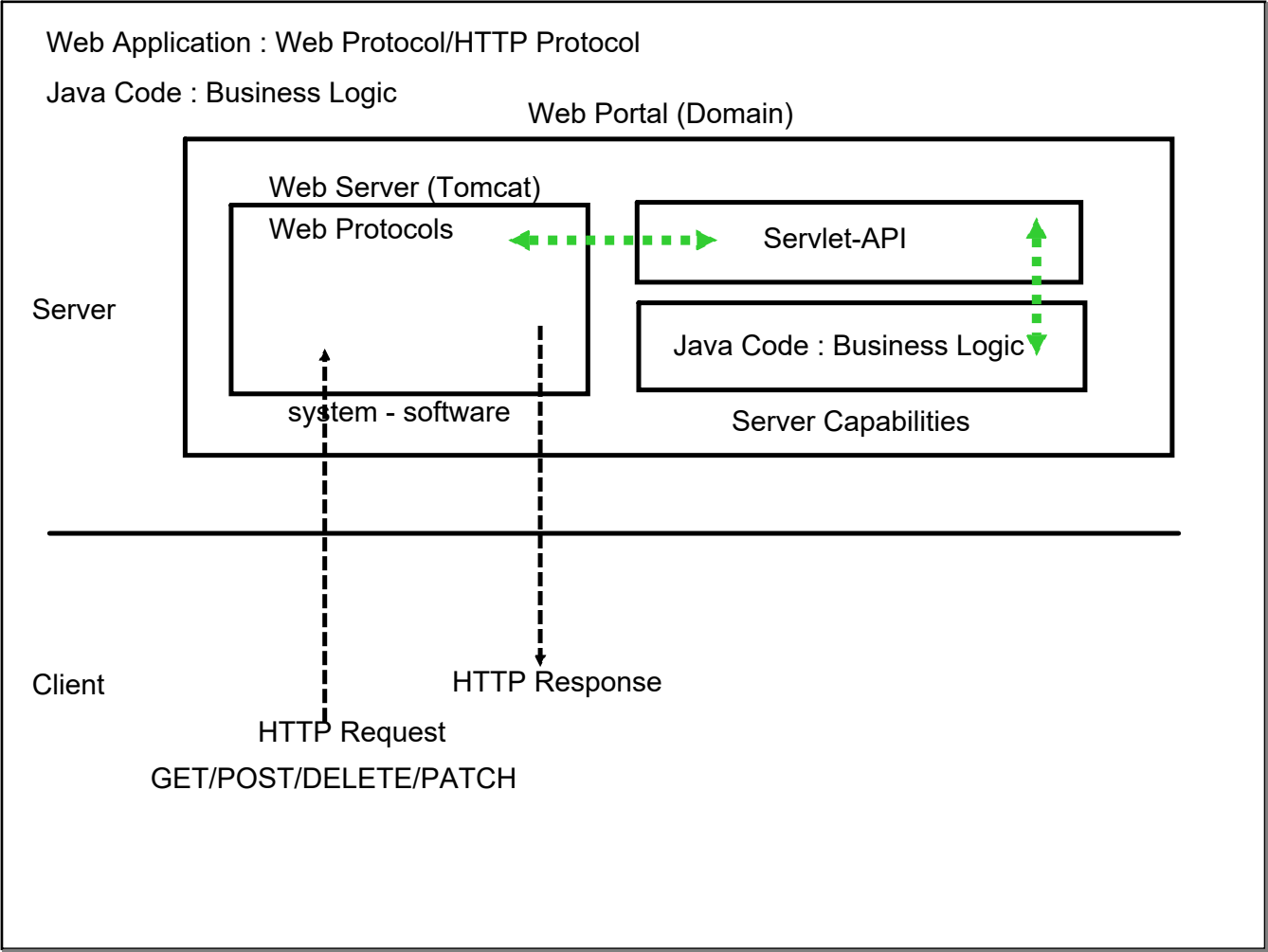
Popular API to create web application using Java

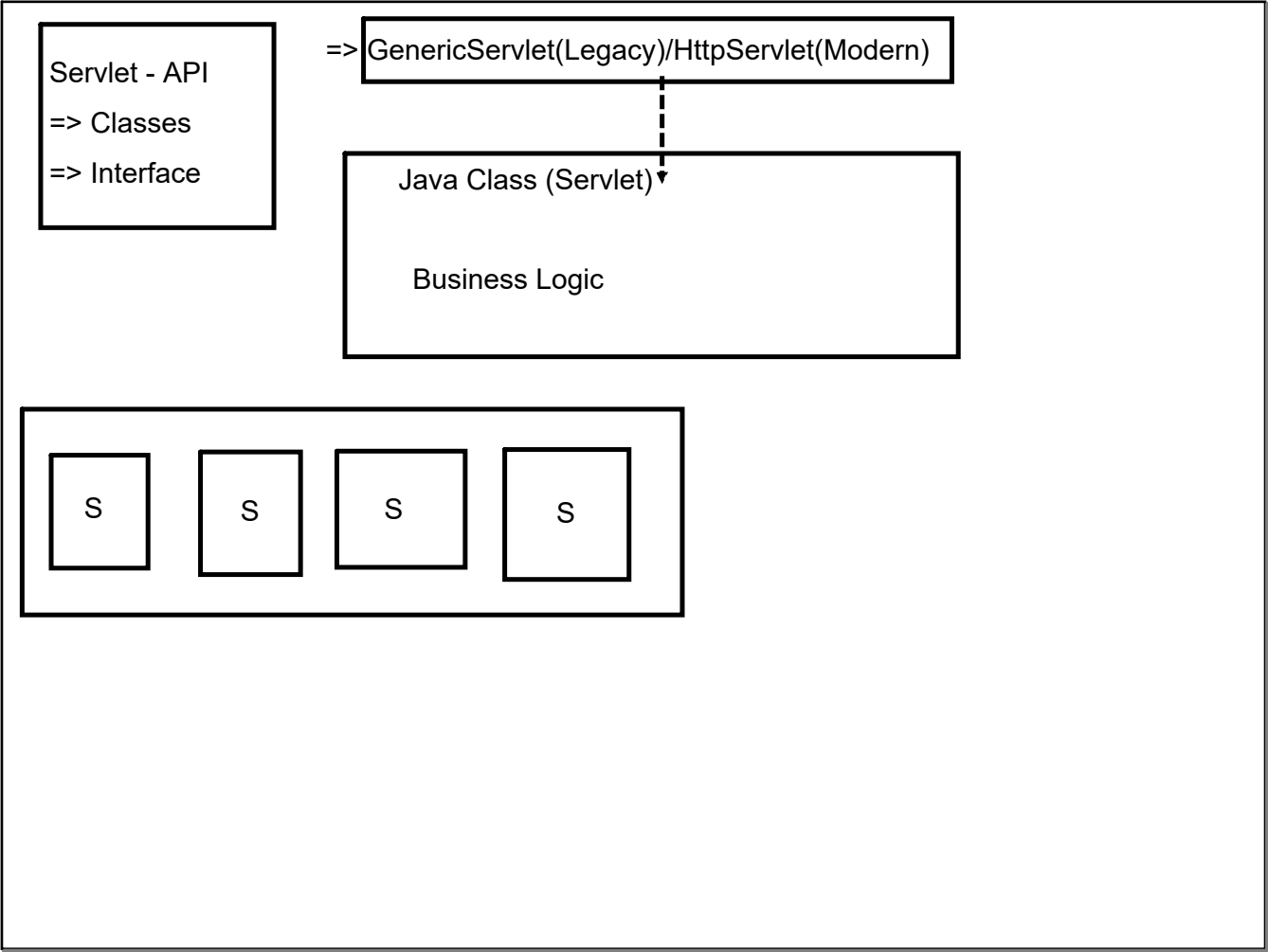
=> Core Java + Servlet-API

=> Complex Framework

Major Framework of JAVa

JavaEE, Spring, Struts, EJB....





GenericServlet : cannot differentiate among HTTP Verbs :

HttpServlet : can differentiate

```
class MyServ extends HttpServlet{  
}
```

Servlet

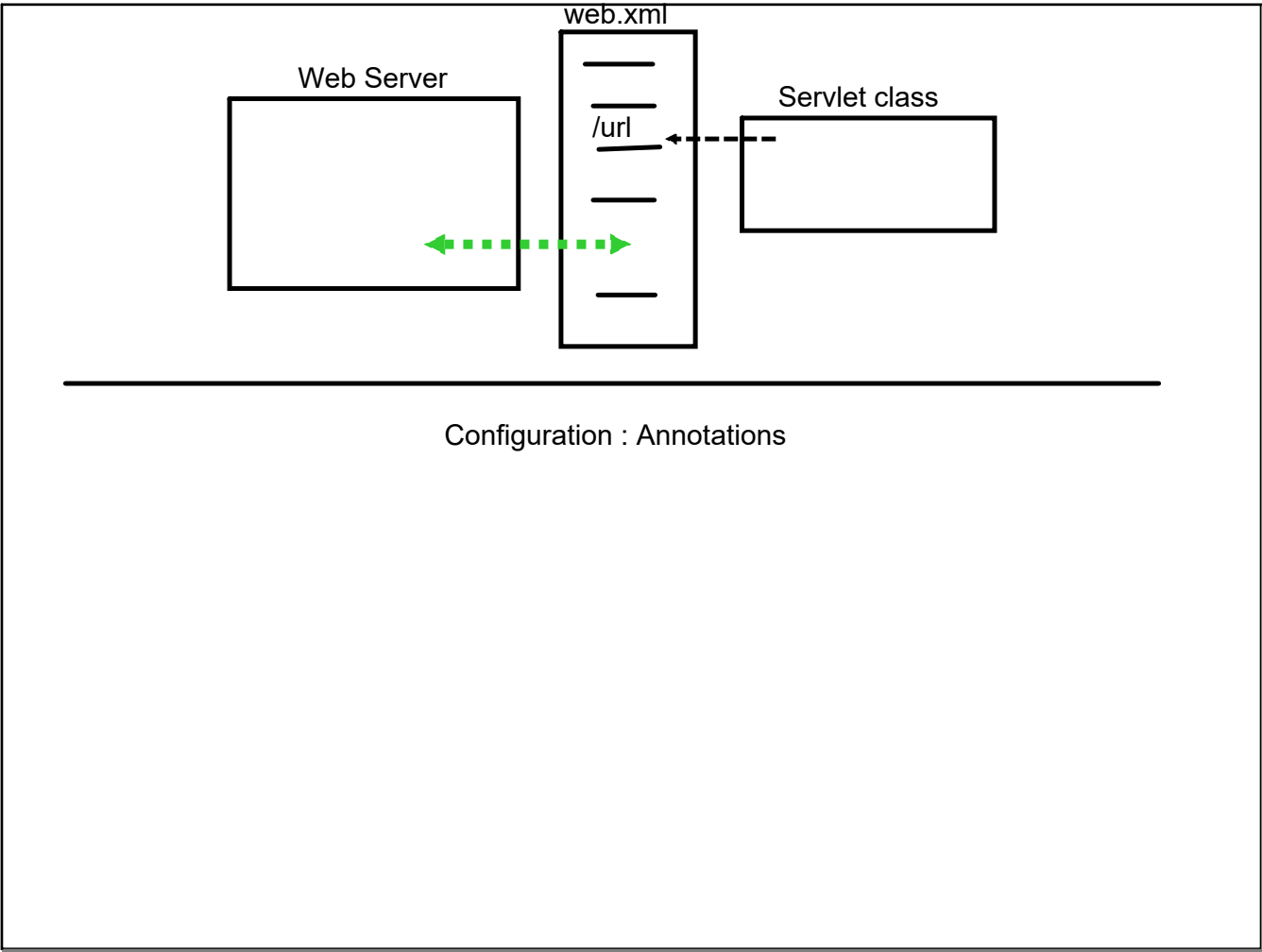
- # Every servlet needs to be registered with WebServer

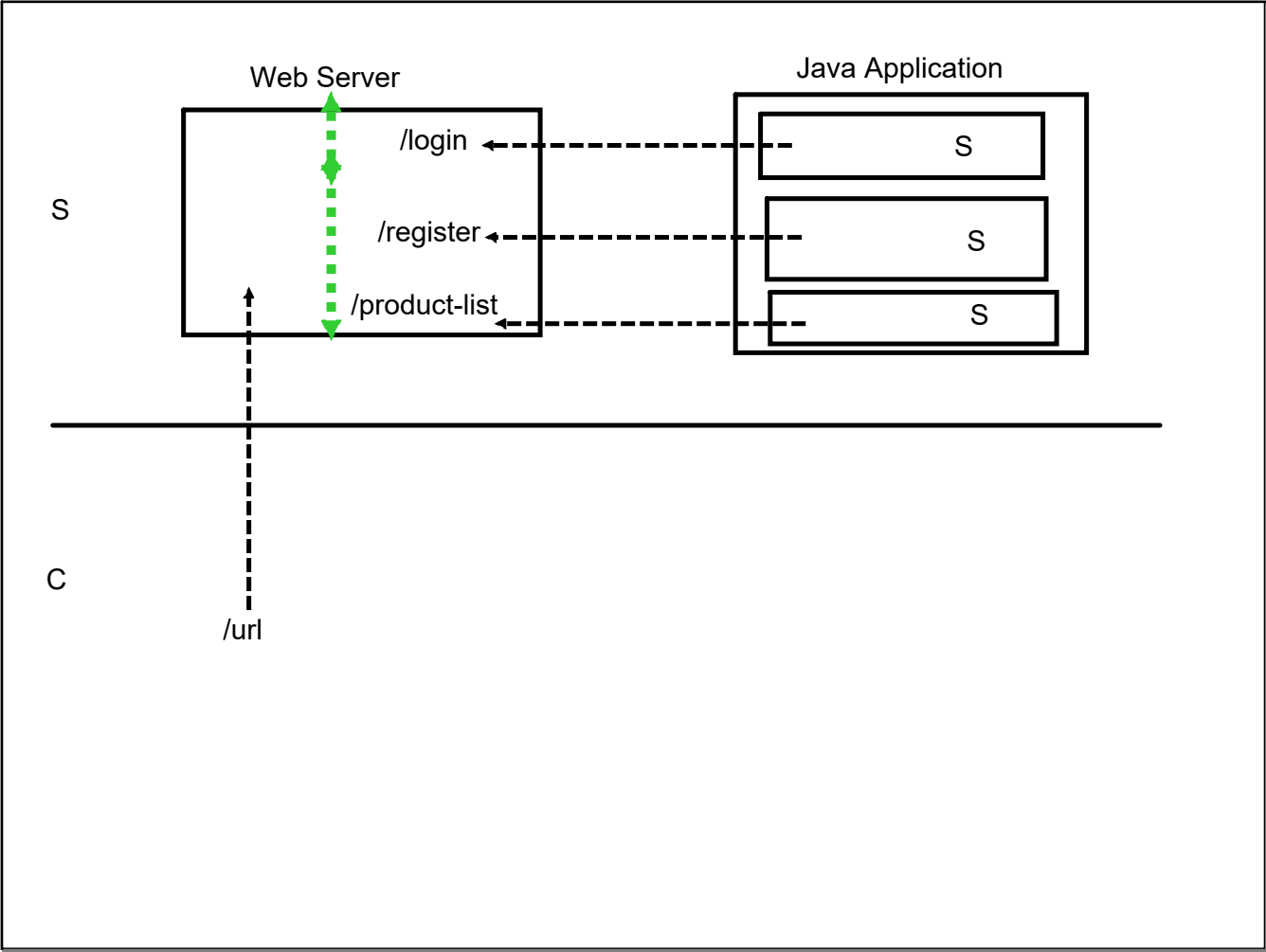
- # Registration will be based on URL

- # Registration will be done through a special manifest file

 - Deployment Descriptor : web.xml (de-facto std)

- # Once a servlet is registered , Web Server starts managing the lifecycle of Servlet





MANaging the life cycle:

1. Create an object of that class
2. launches life-cycle of servlet class
 - a. init() : phase : prepare for request processing
 - b. service() : phase : access over request, processing, respond back
 - c. destroy() : phase : release the resources
3. Make the servlet object available for garbage collection

First Request : 1->2(a)->2(b)----> cache the object

Next Request(s) : 2(b) ----> cache the object

Whenever :

Servlet is not requested for long time/capacity :

2(c)-->3

Sequential : 2(b)

Parallel : n user requesting same servlet

Object cached :

=> n new object of servlet

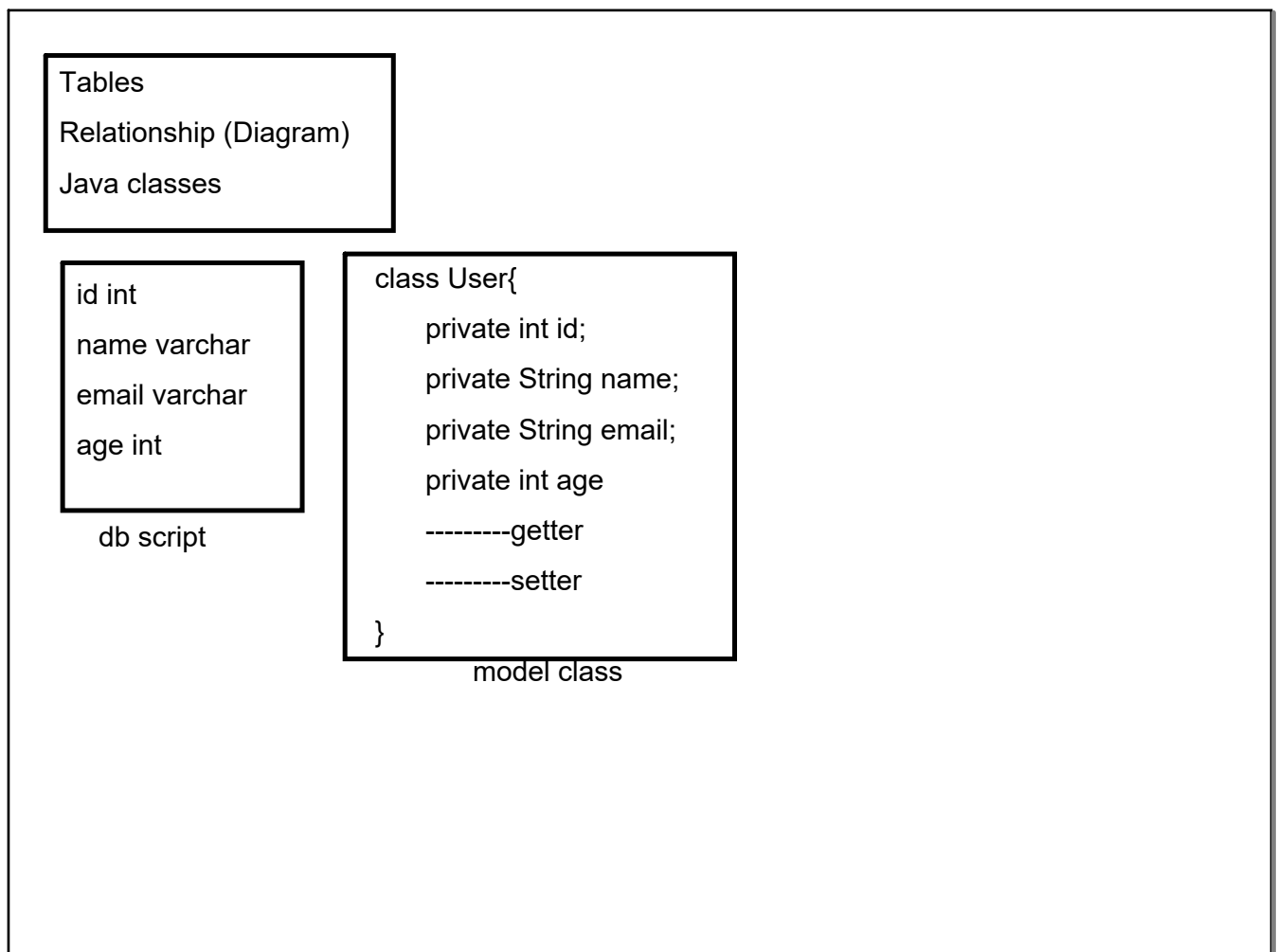
=> queue of request

=> multithreading : threads of service (phase(2(b)))

Servlet must use Thread safe service

Names1 :

Name 2:

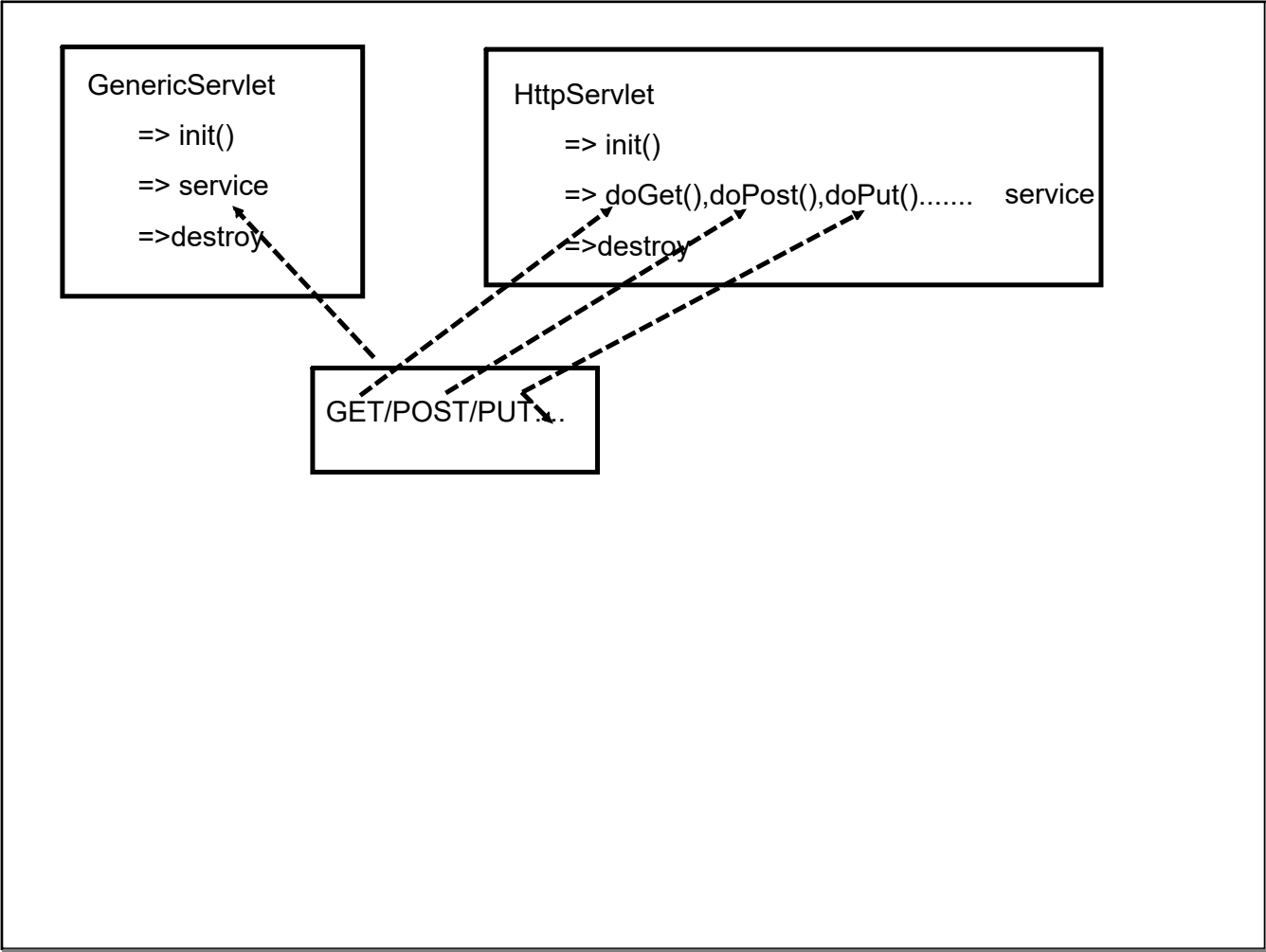


Java Application + Servlet-API
Template : Dynamic Web Project

Eclipse IDE : plugin the Tomcat
auto process

Tomcat Server

1. build and package (war) the project
2. deploy/copy to working dir TOMCAT
3. Launch the server

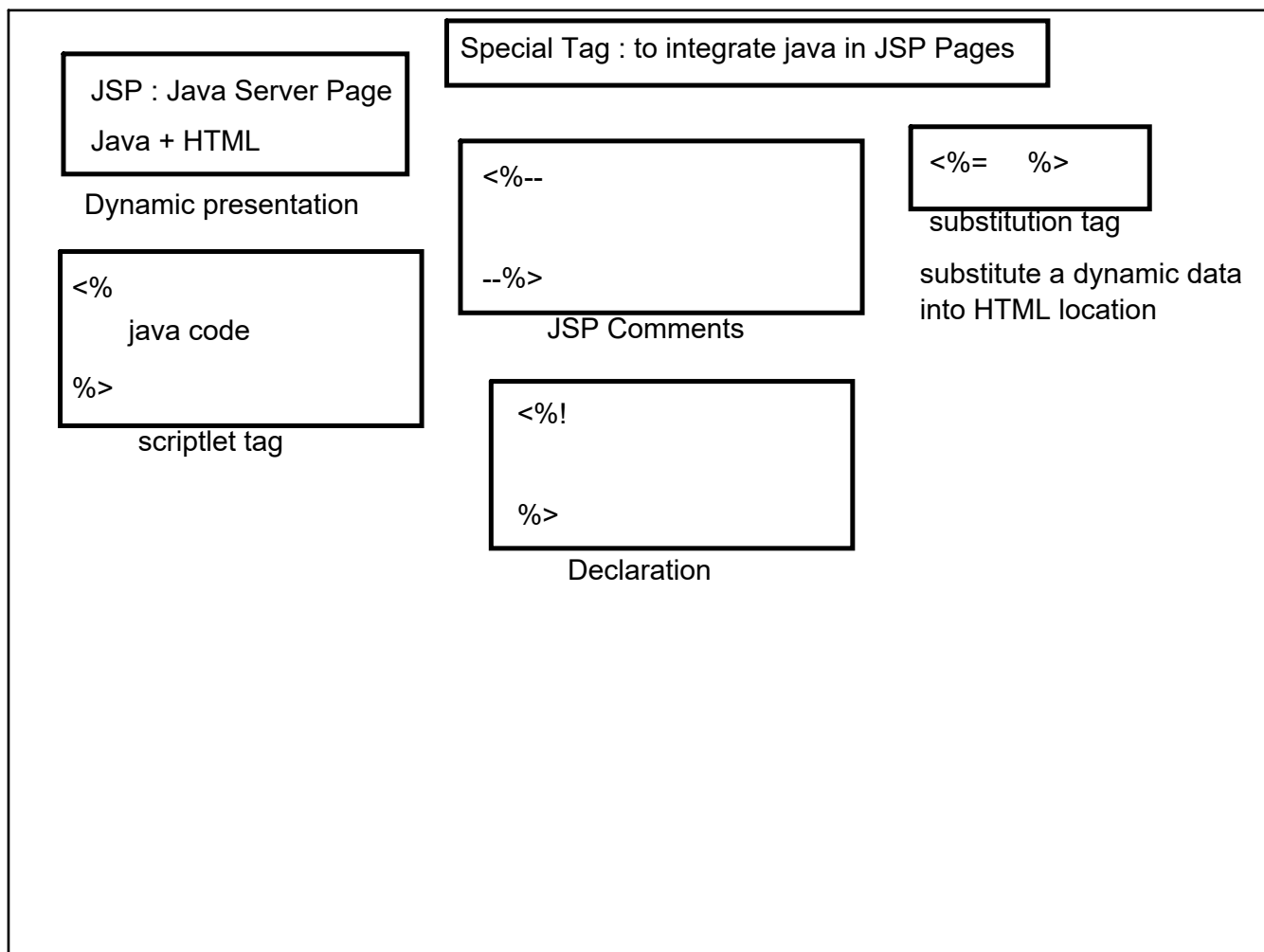


```
HttpServletRequest request => all info/functionality about request  
HttpServletResponse response => all info/functionality about response
```

Response : HTML

JAVA CODE RESPOND BACK

1. Login Form (home-page)
2. Info as request -----> Servlet
3. Fetch those info
4. business logic for credential
5. Welcome // Invalid



JSP : Concept

=> Does not exists at runtime

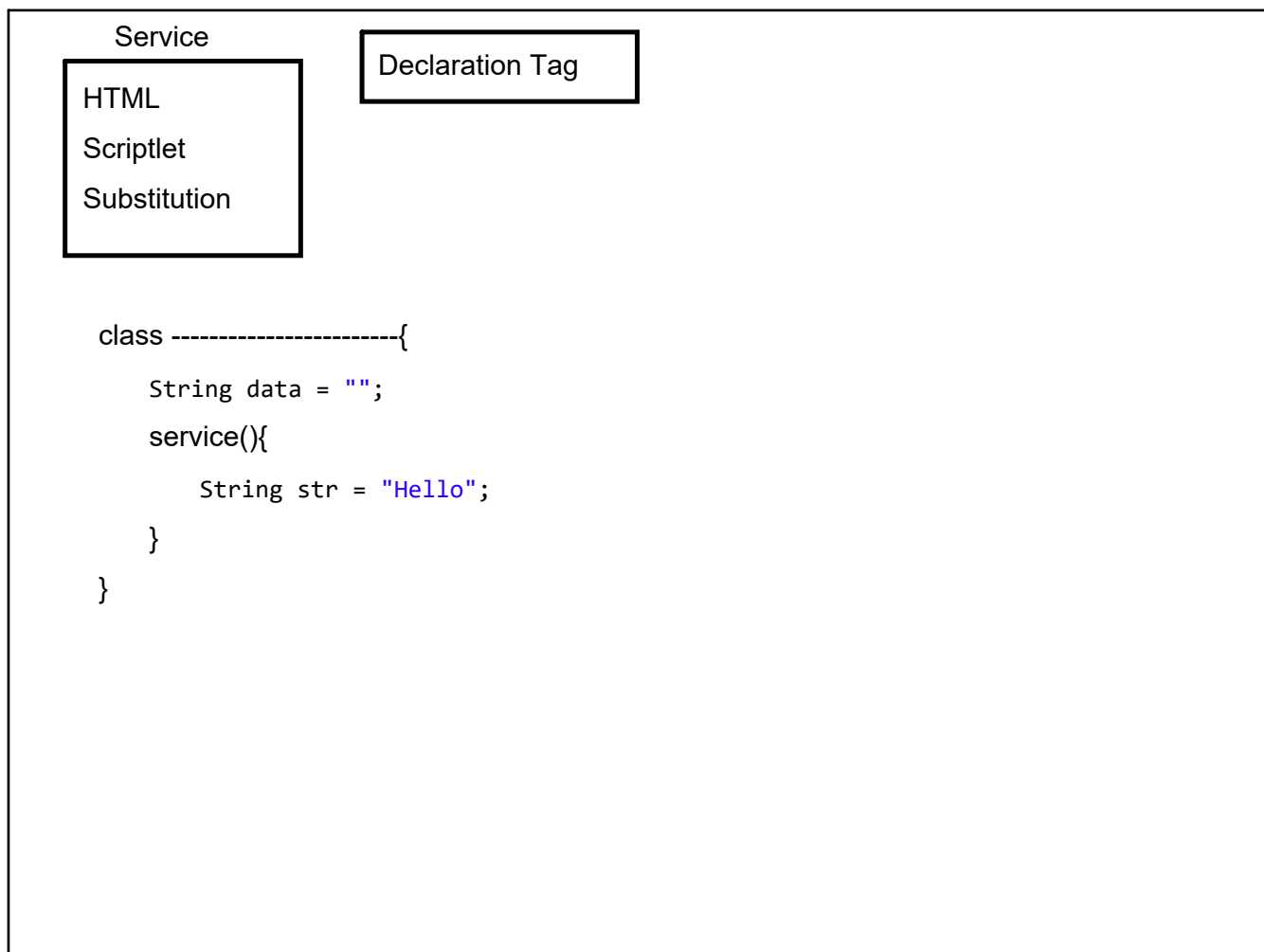
=> Another way of writing the servlet

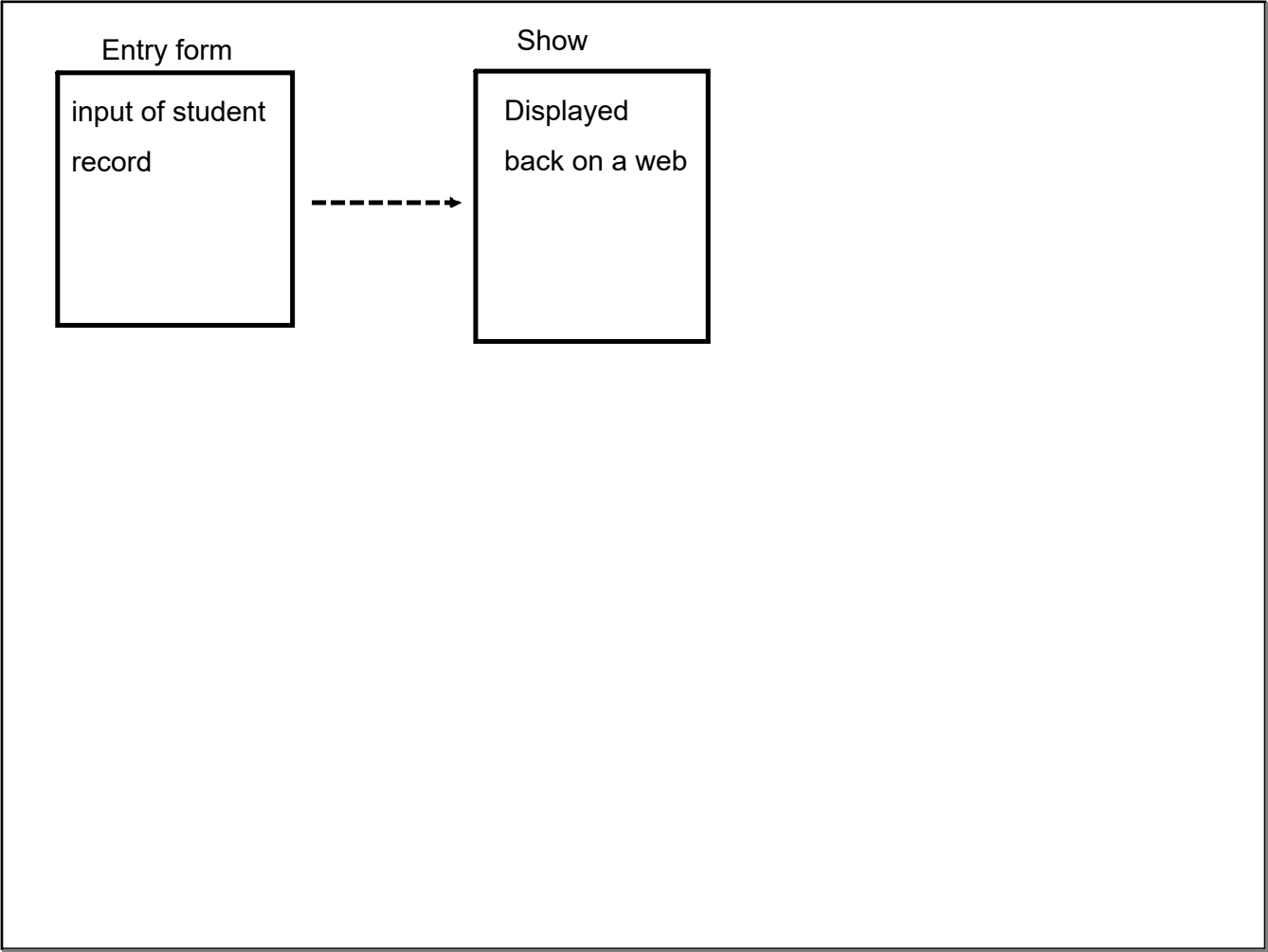
=> JSP-----Servlet

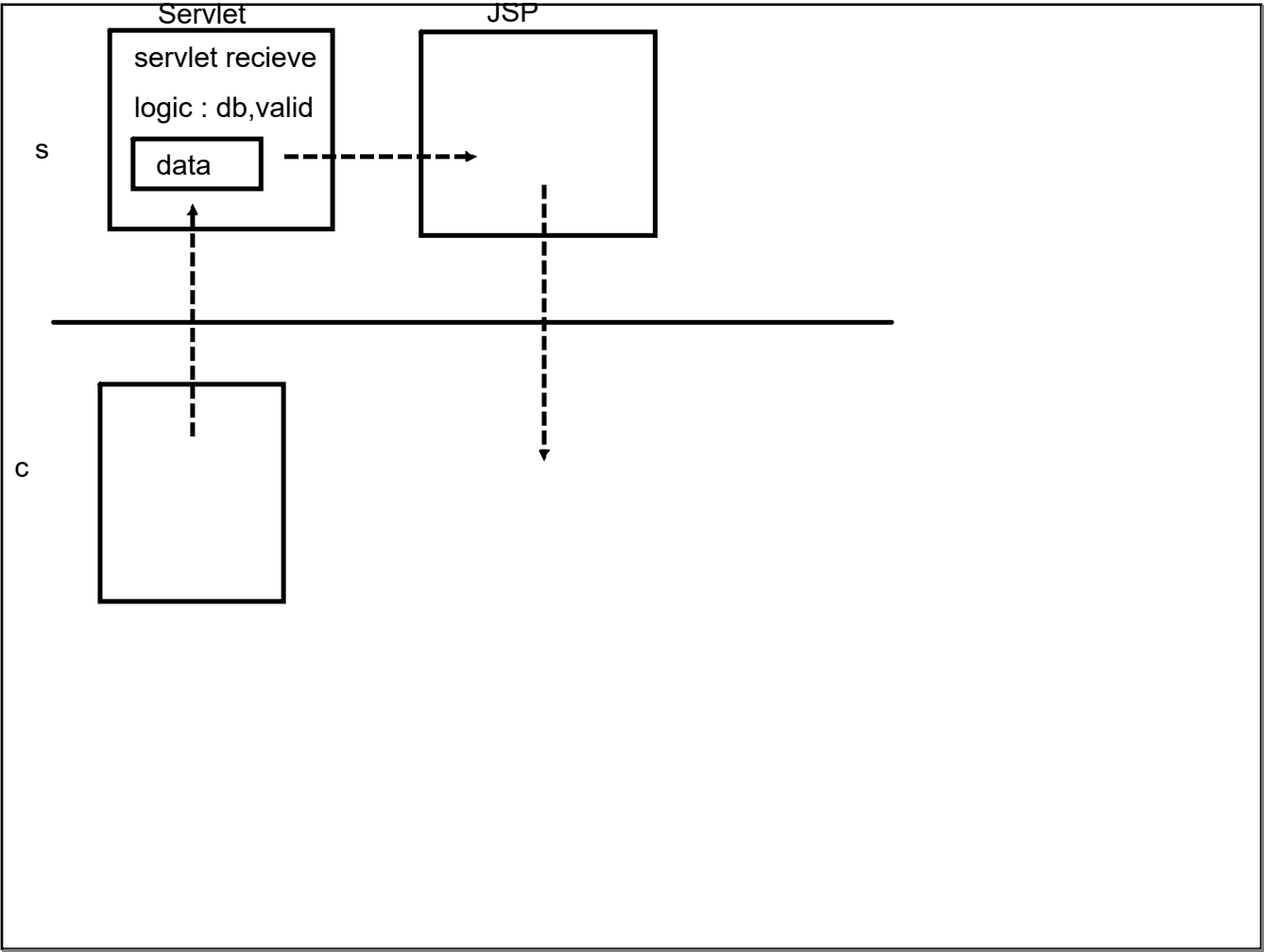
Dynamic web Create :

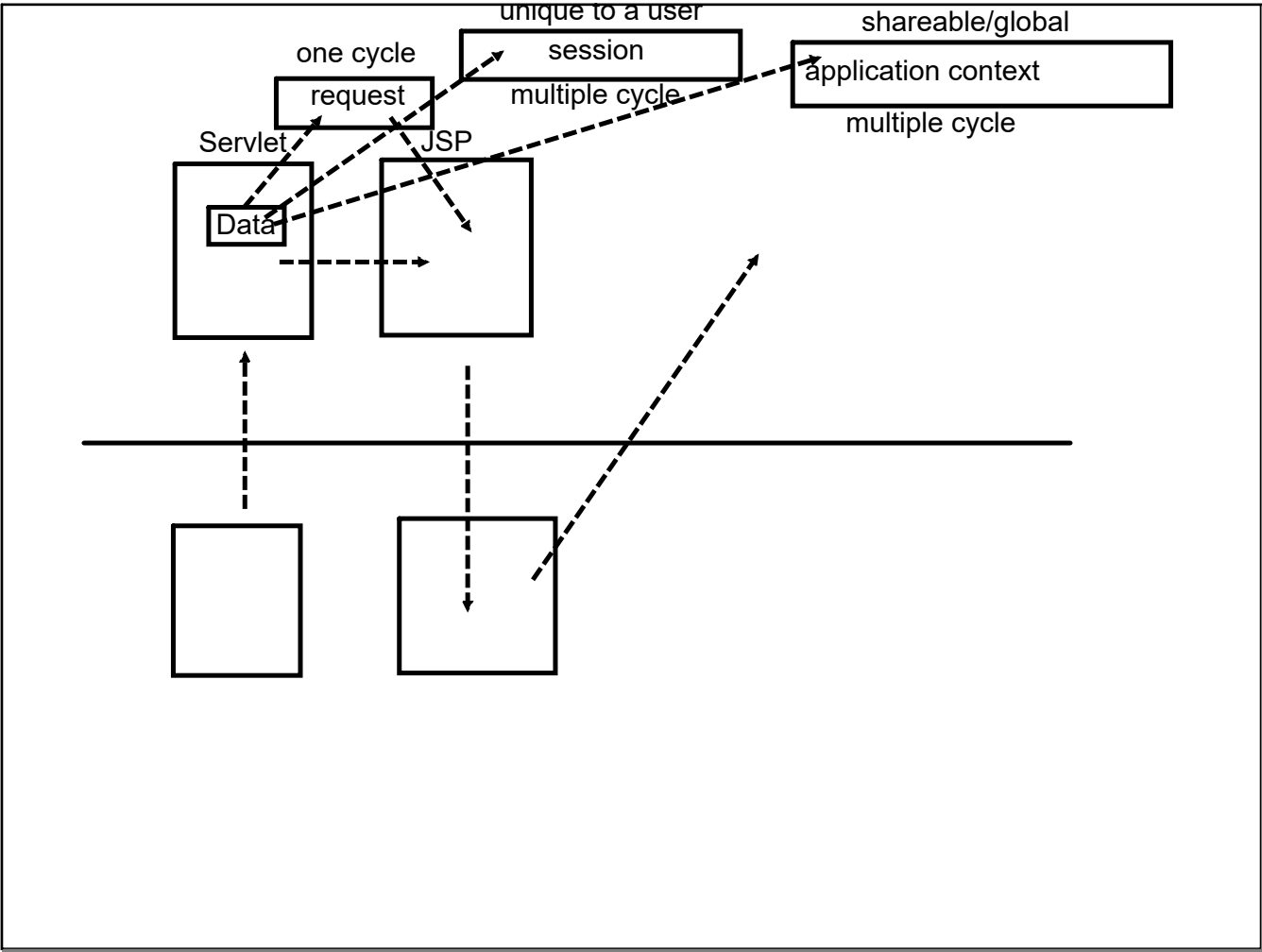
Servlet : Java business logic (class instance)

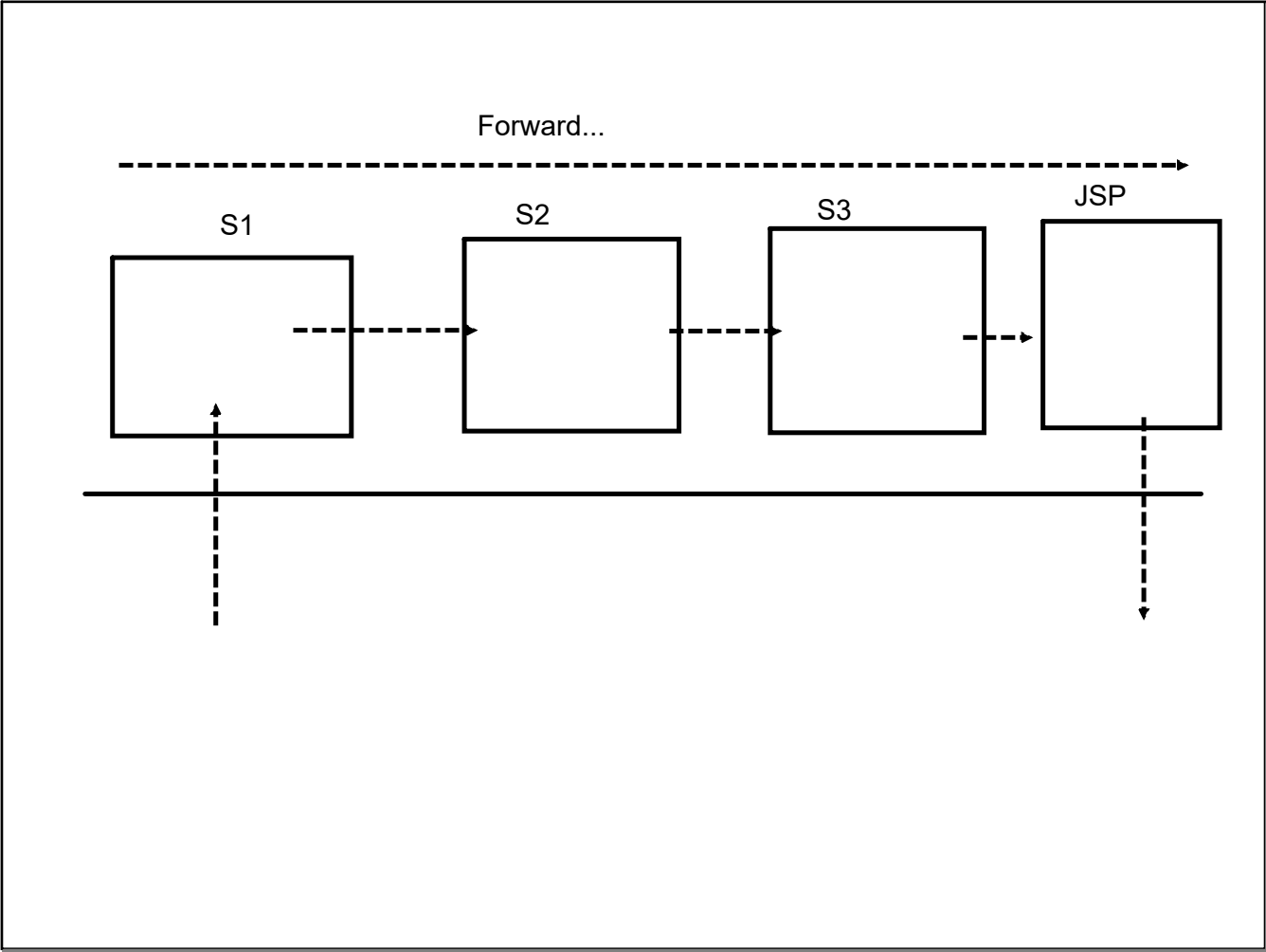
Servlet : Heavy presentation (jsp instance)

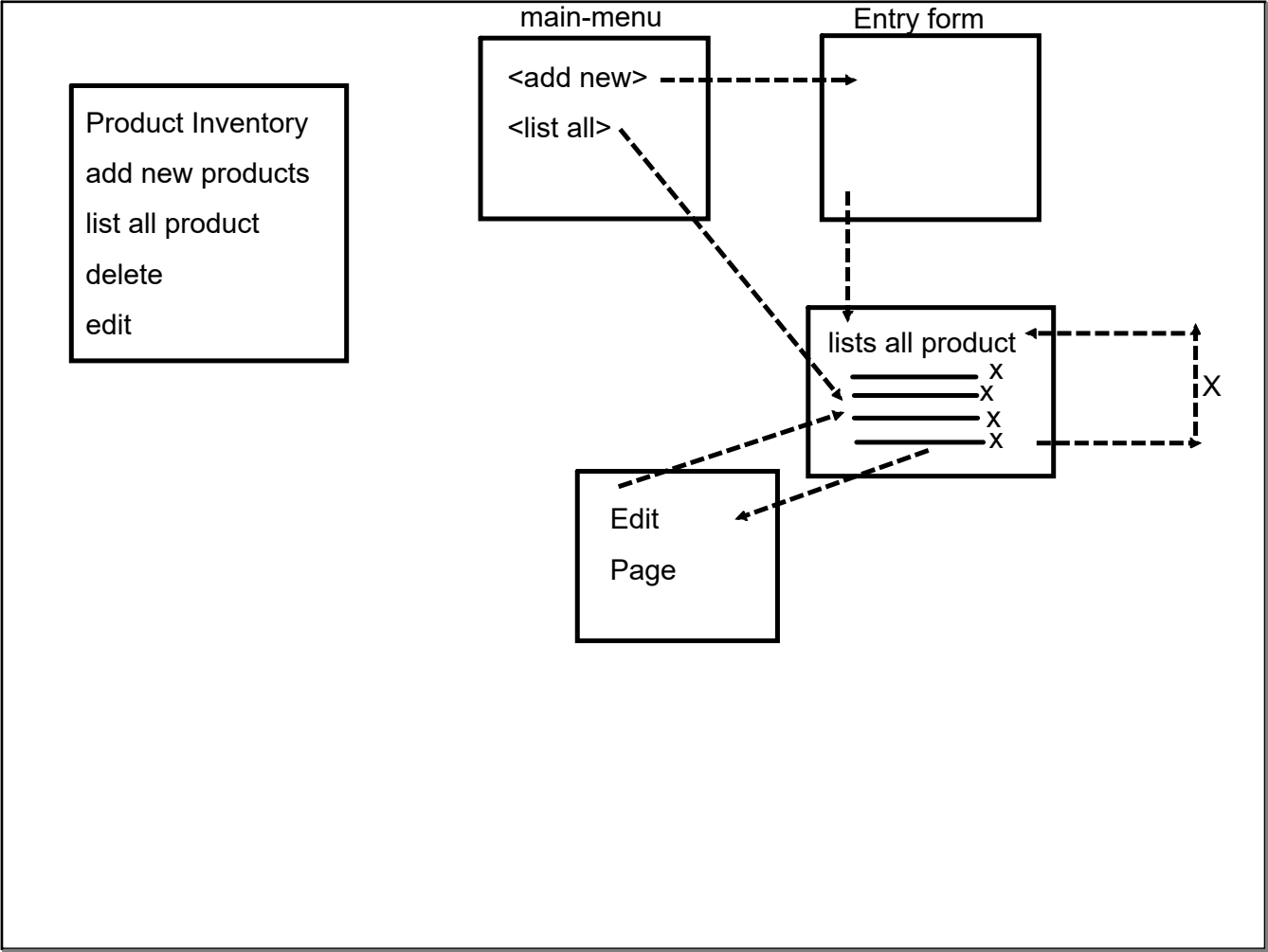


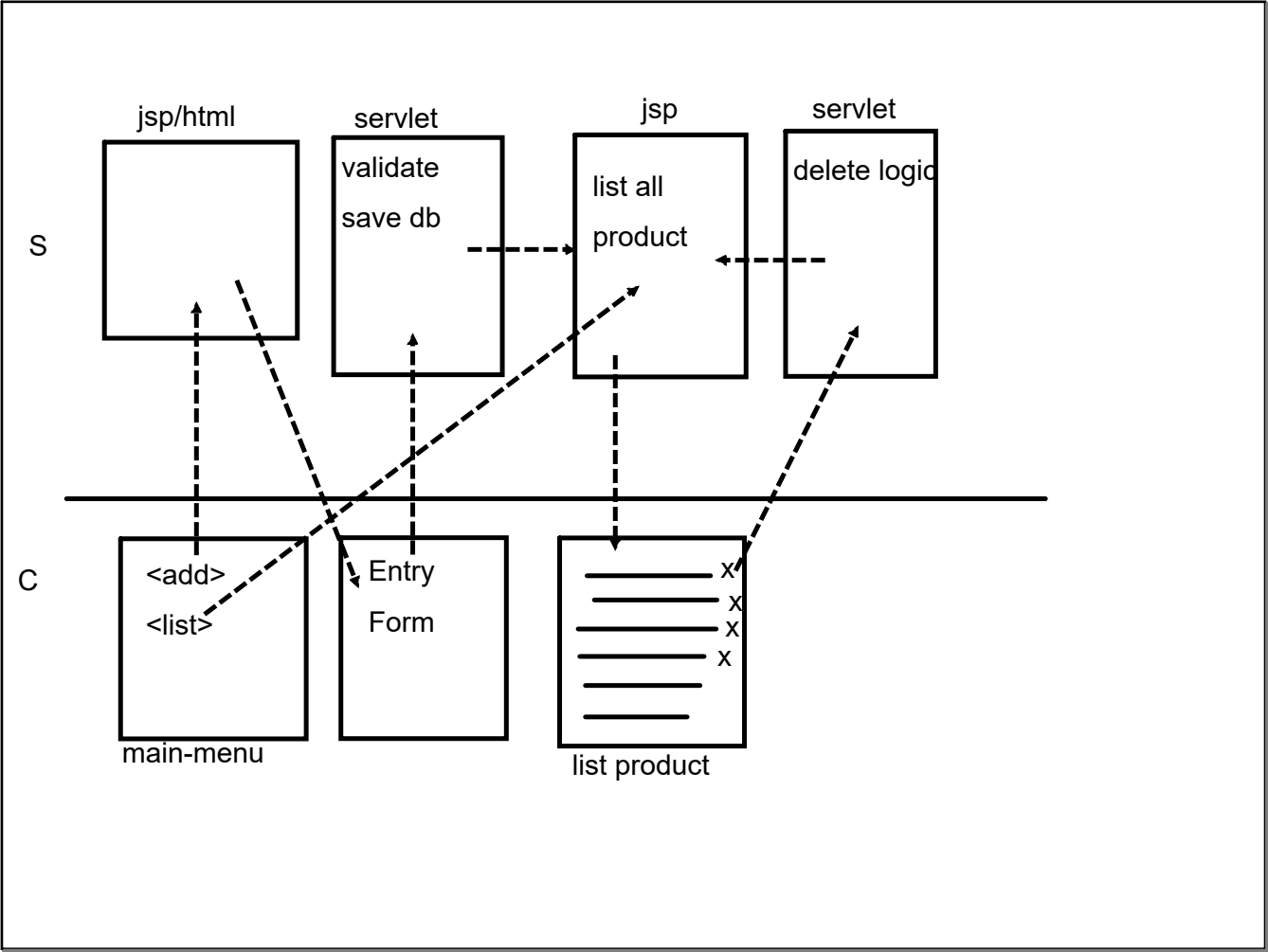










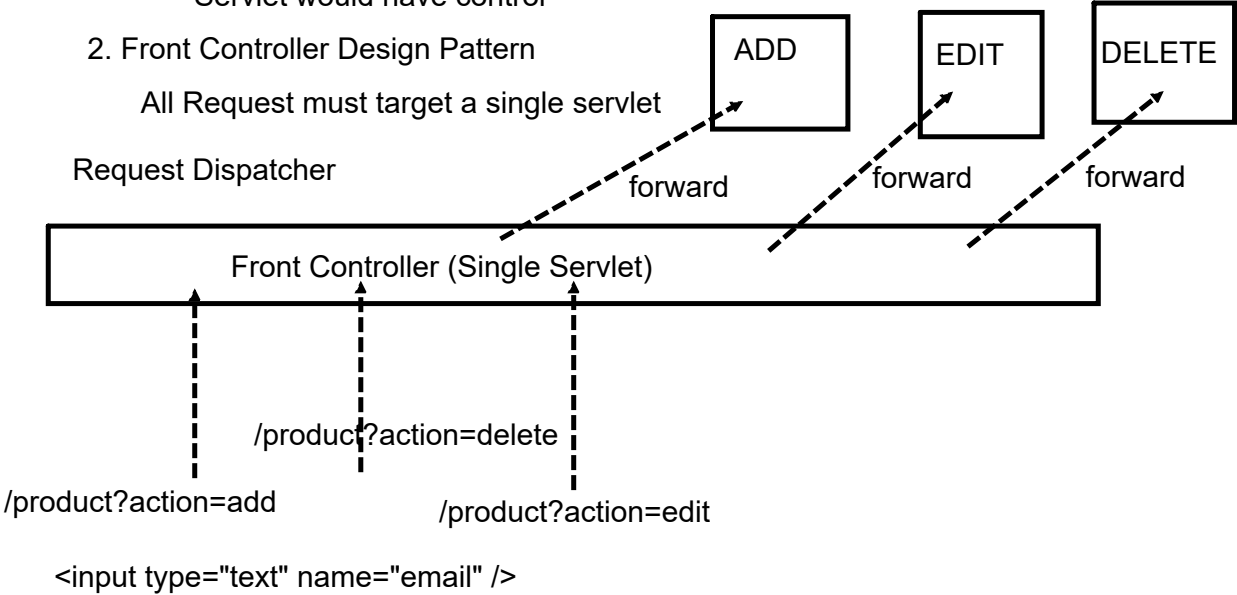


MVC Architecture

Best Practice

- 1. Never be calling a html/JSP
All calls to presentation resource, should be through servlet only
Servlet would have control

- 2. Front Controller Design Pattern
All Request must target a single servlet



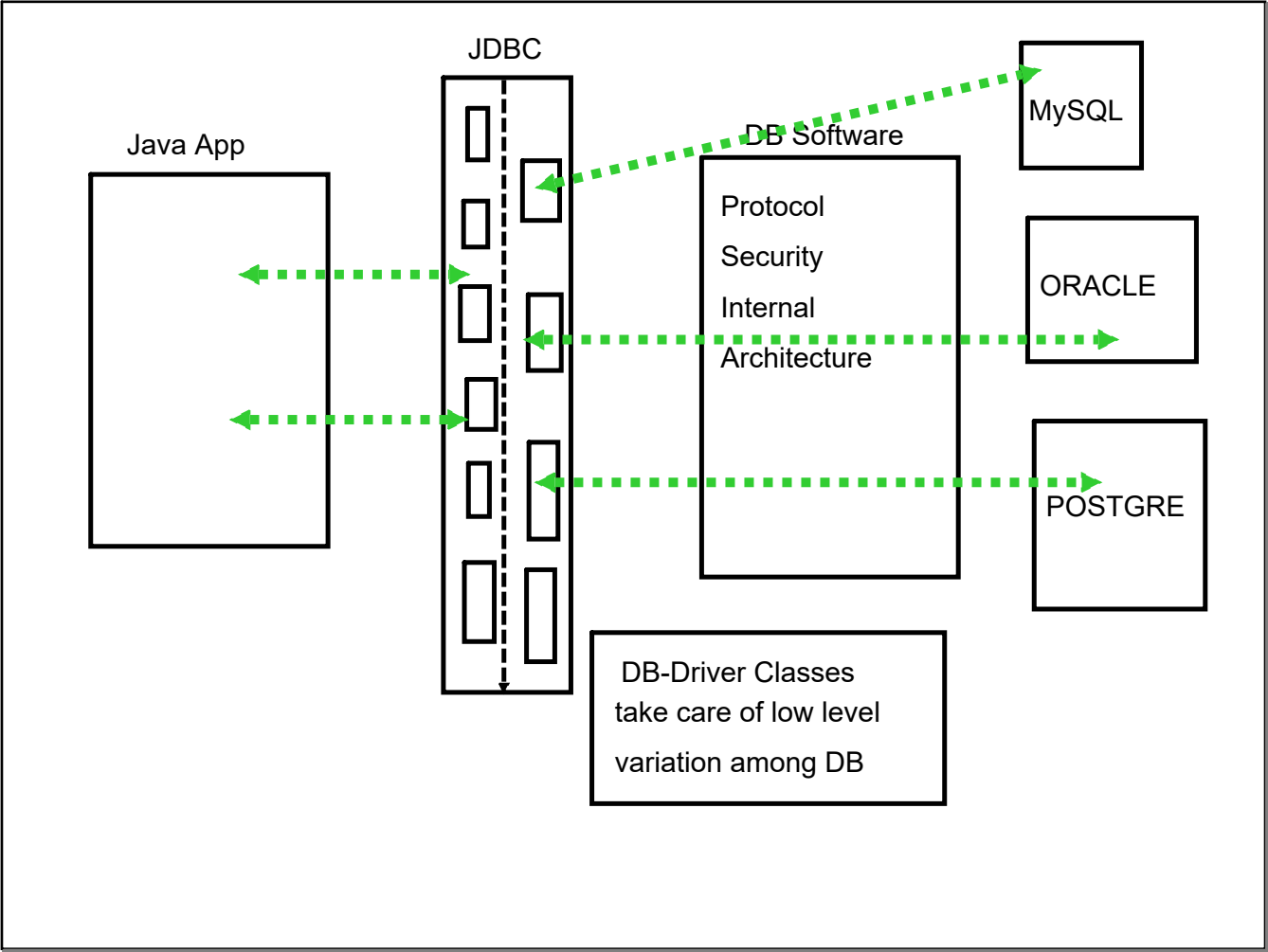
M : Model : request/session/context

V : View : JSP/HTML

C : Controller : Servlet

Separation of concern

JDBC : Java Data Base Connectivity

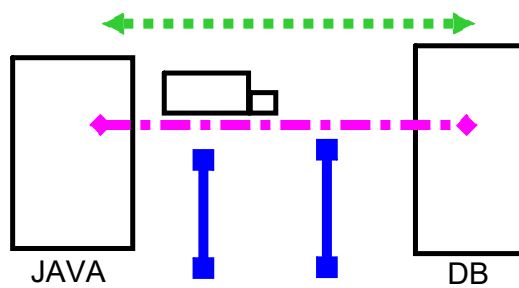


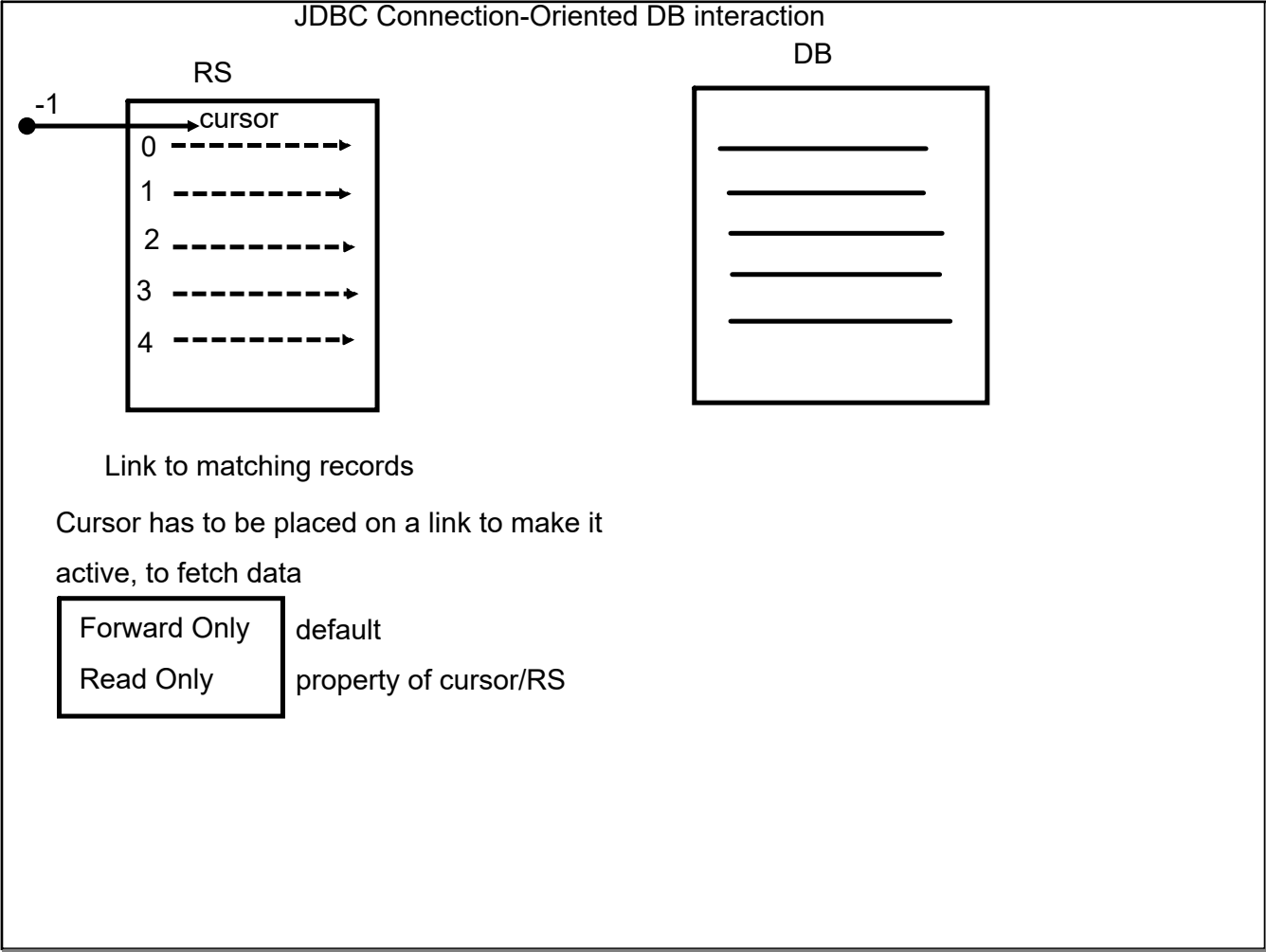
Steps :

1. Load the driver class of that type
2. Create a connection
3. Create a statement
4. Fire the query

MySQL : mysql-connector-java.jar

Load a Java Class in memory





Batch Processing

Allows to create a batch of queries

Allow to run them at one go

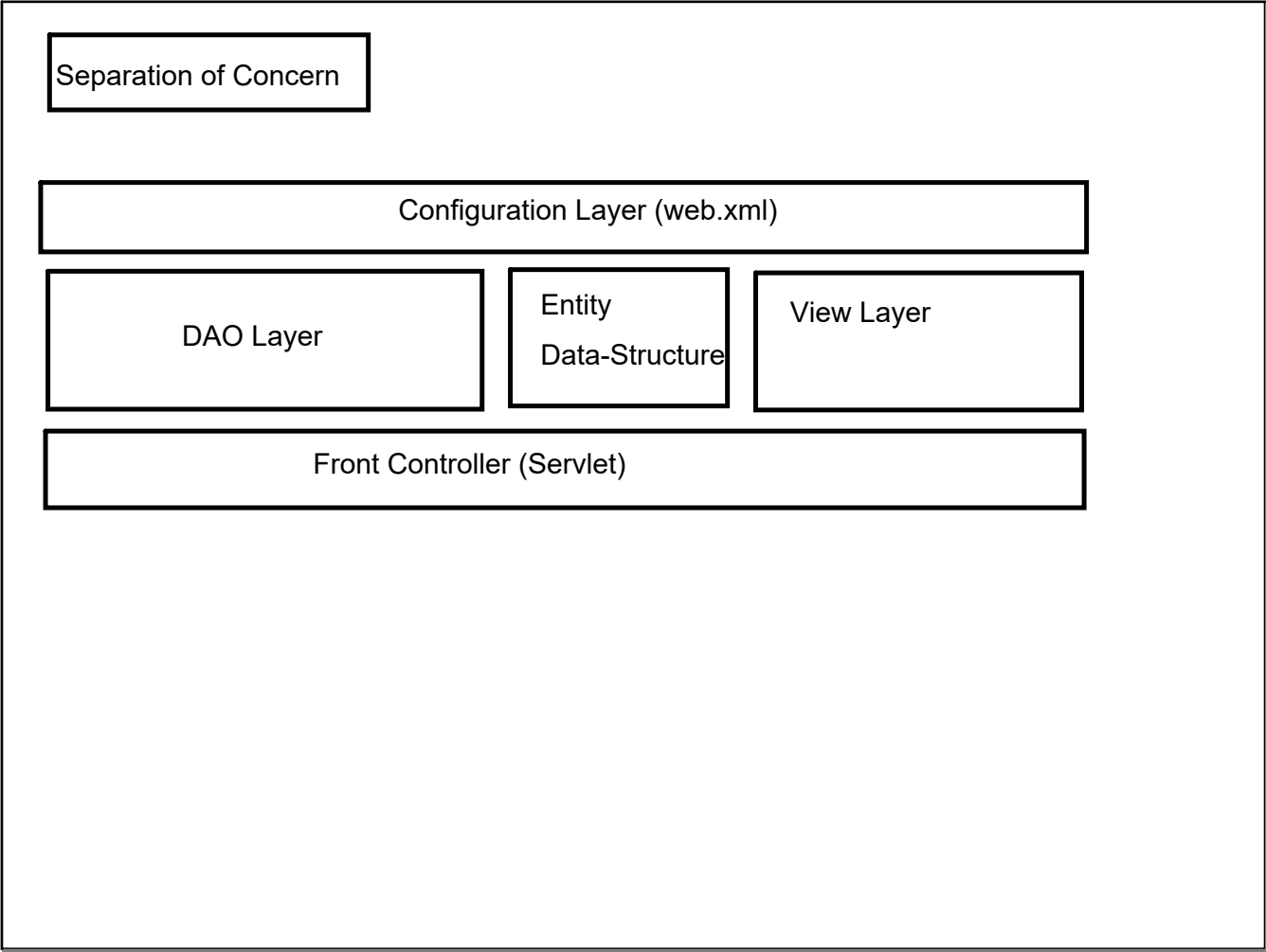
Transactional Mode

Group of query :

Either all of them shall execute or none of them, with the provision of rollback

A

B



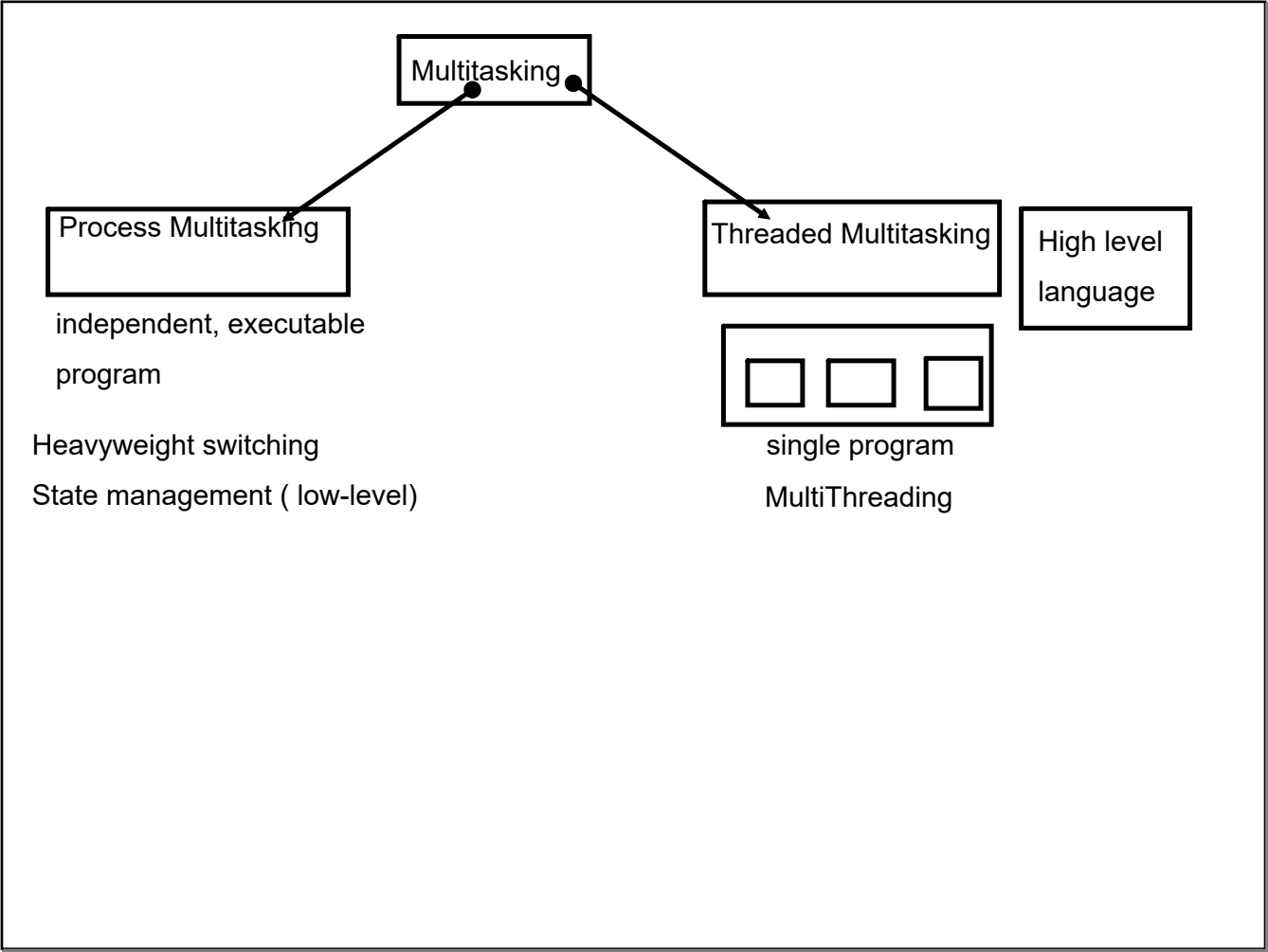
Multithreading in Java:

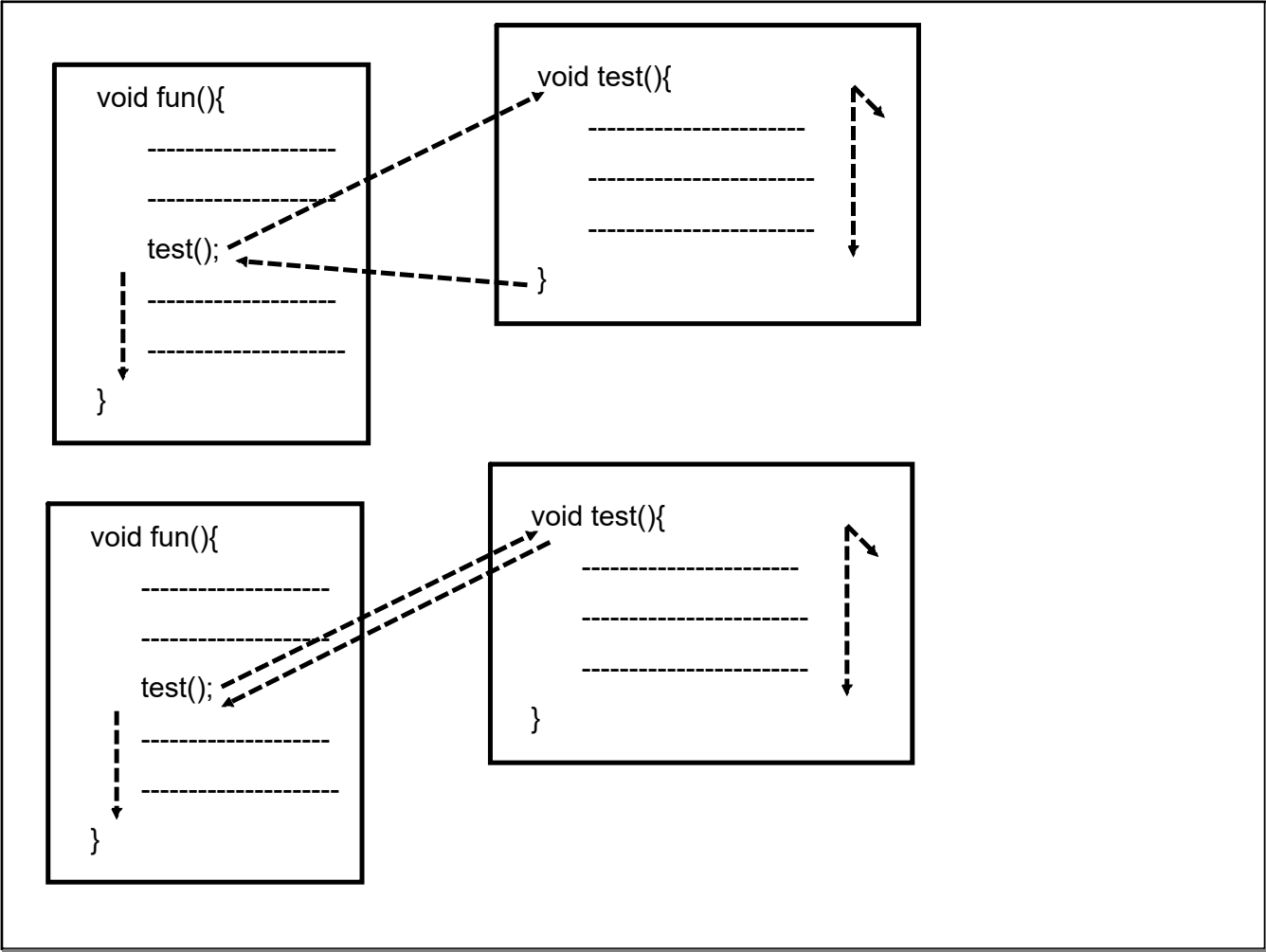
Running more than 1 functions simultaneously in async manner

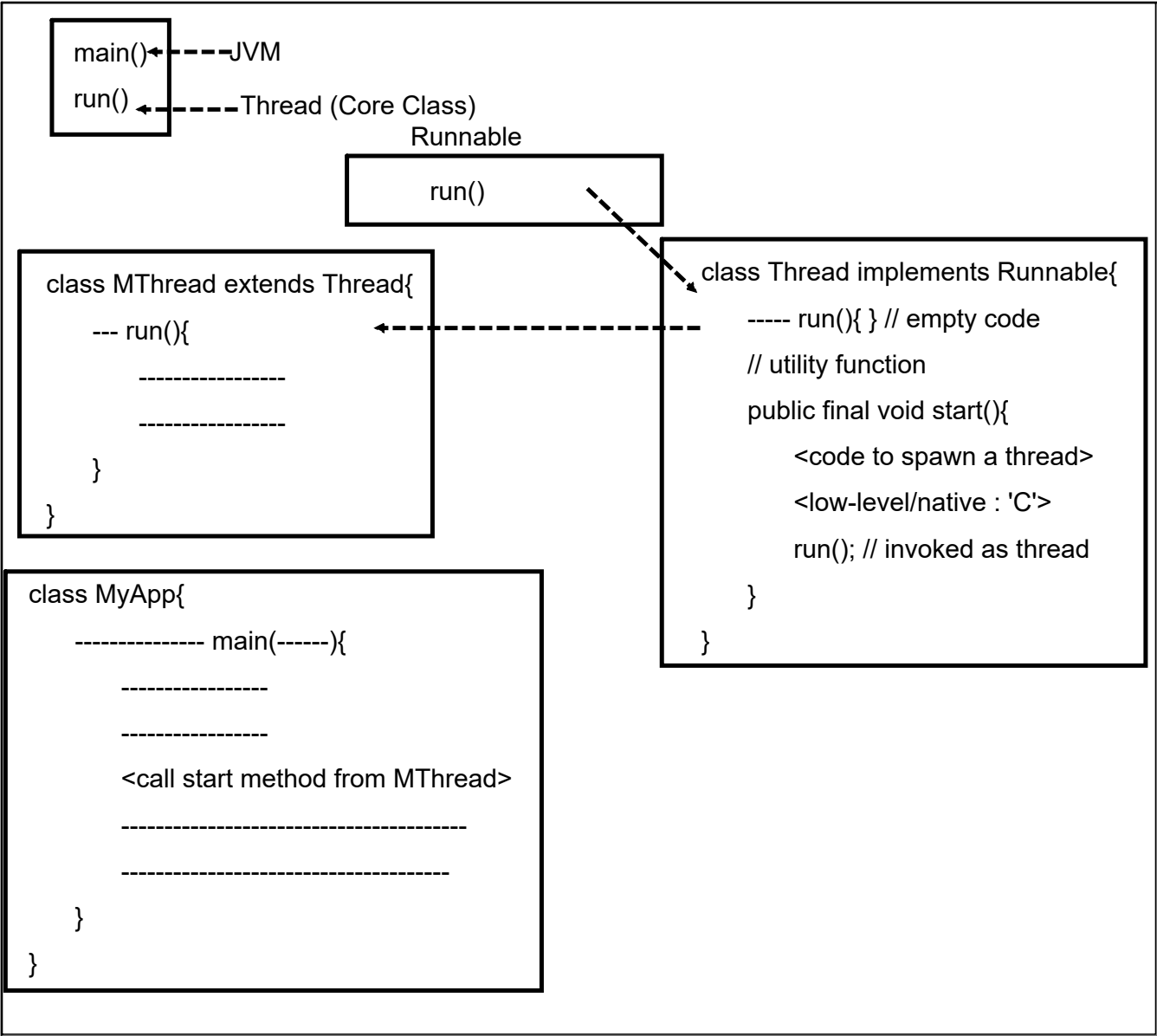
Multiprogramming :

Multitasking : in interleaved fashion (time-sharing) : not parallel

MultiProcessing : parallel / batch processing

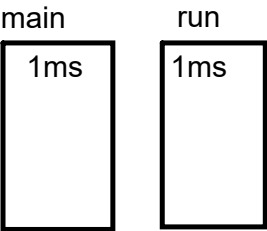




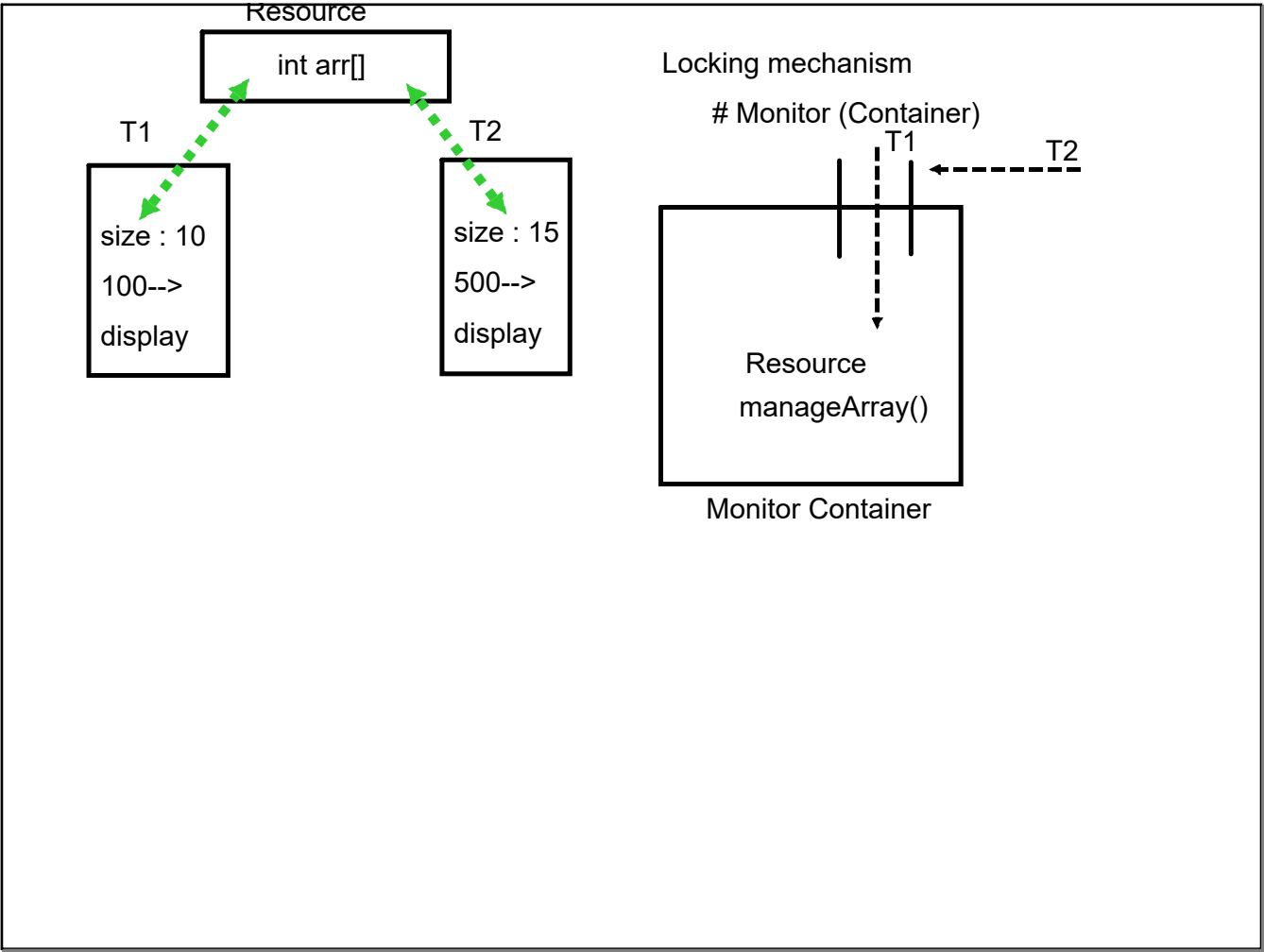


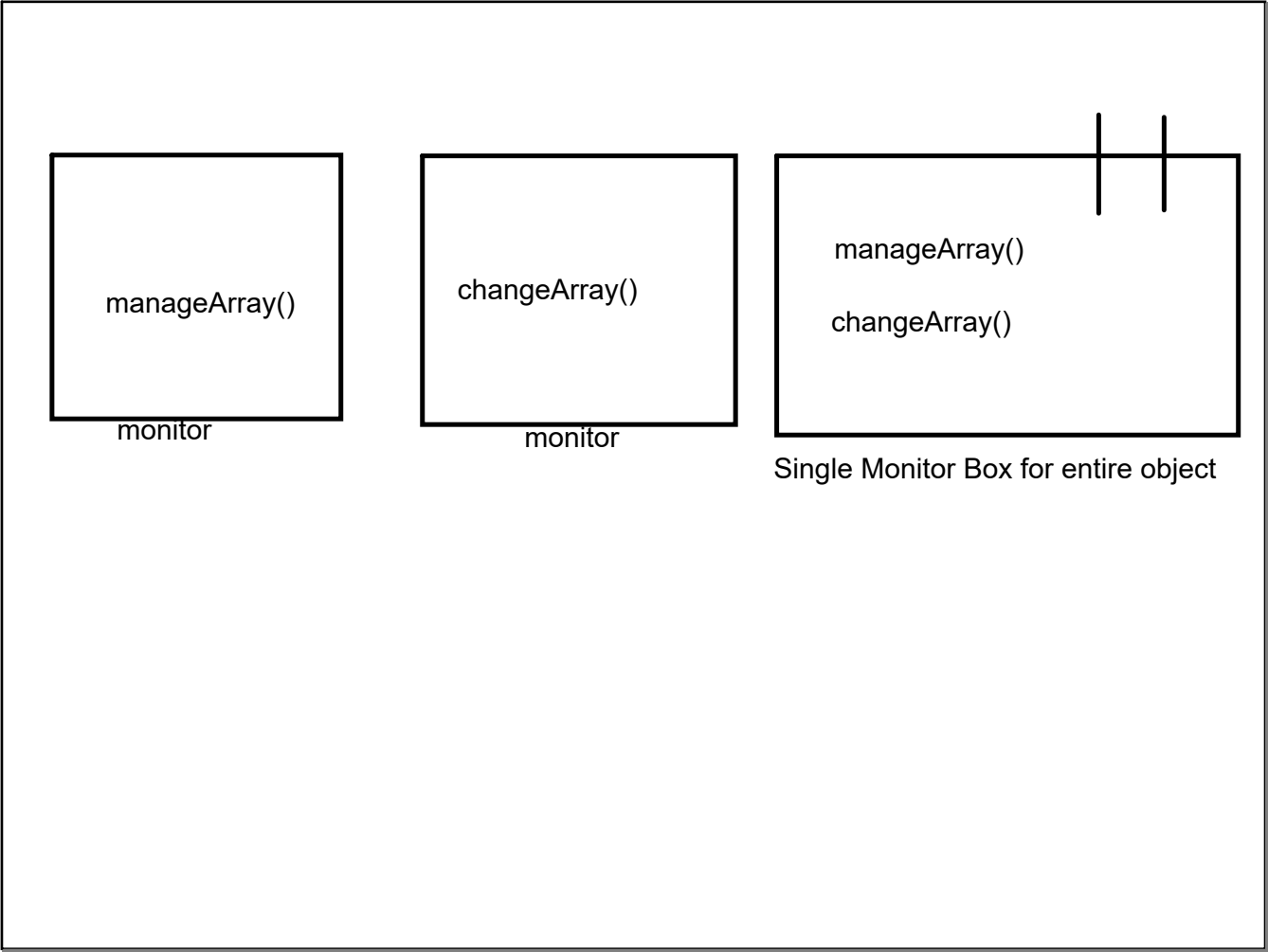
Time Slice and the task allotted to a thread

Output device are slow
buffer memory
mixing in buffer memory



Sharing the common resources
Data in-consistency





Functional Locking

- # Resource does the locking , not the client/thread

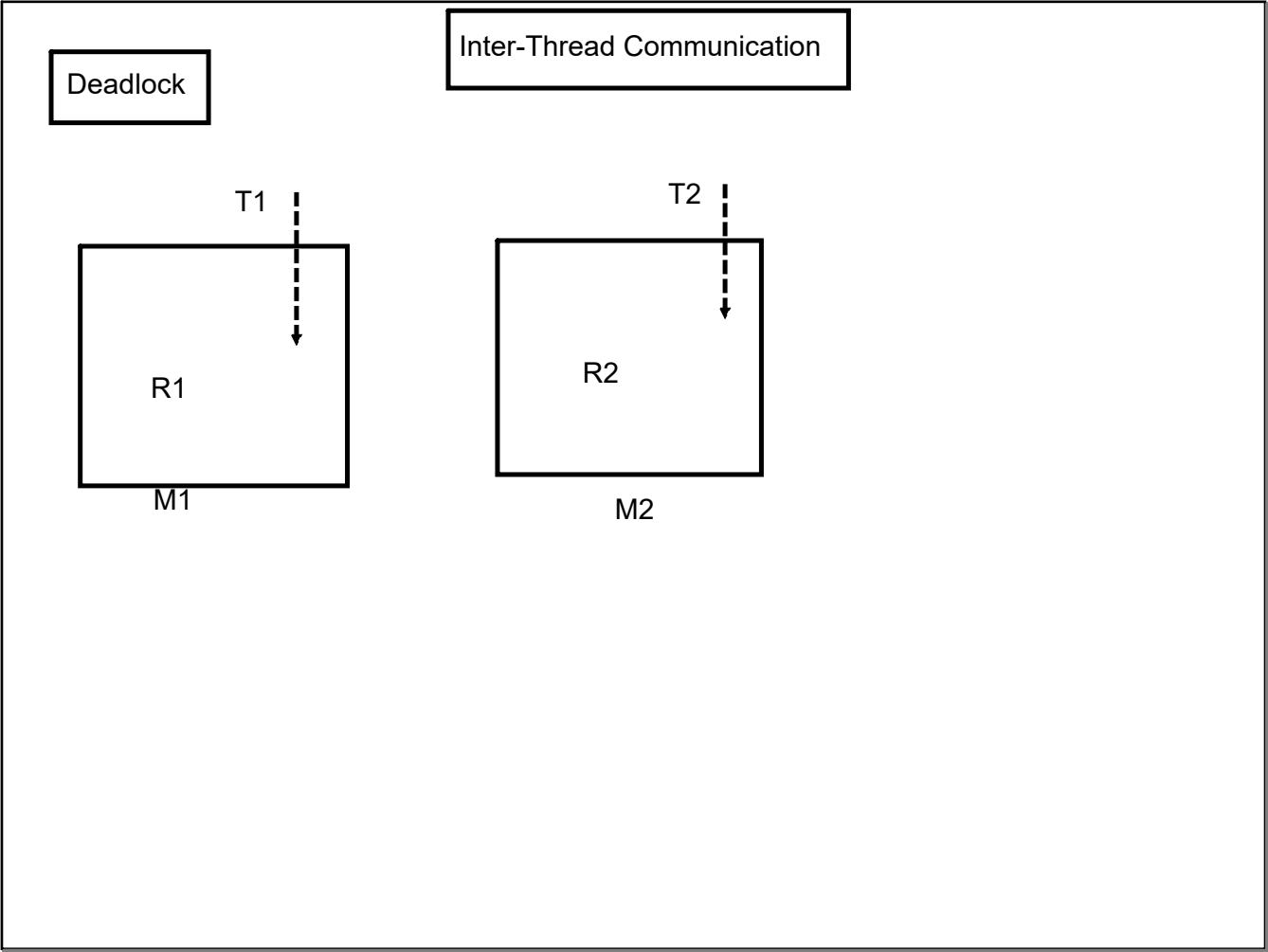
- # Permanent

Data Locking:

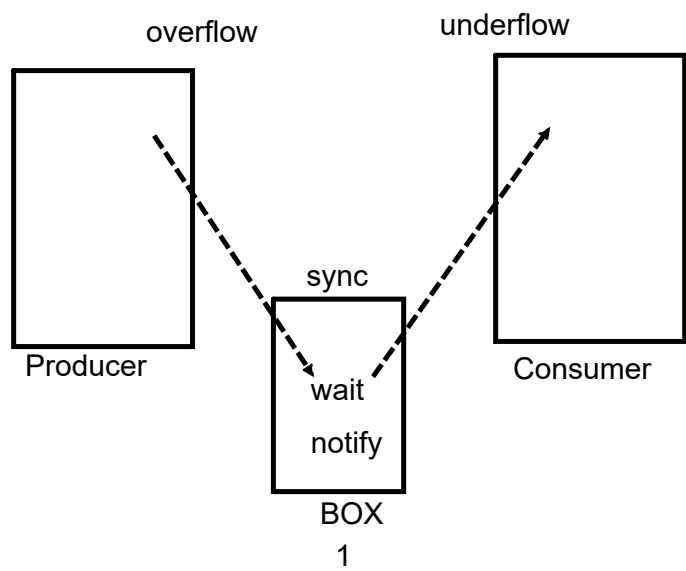
- # Cannot lock permanently, time frame based (block based)

- # Client/Thread can lock data

- # Lock Entire Object



Producer-Consumer : Classical Multitasking Use-Case



- 1. Different Speed
- 2. Box is of finite capacity

Inter-Thread Communication

