# **Notes:** ***Hanona🎀🍓***

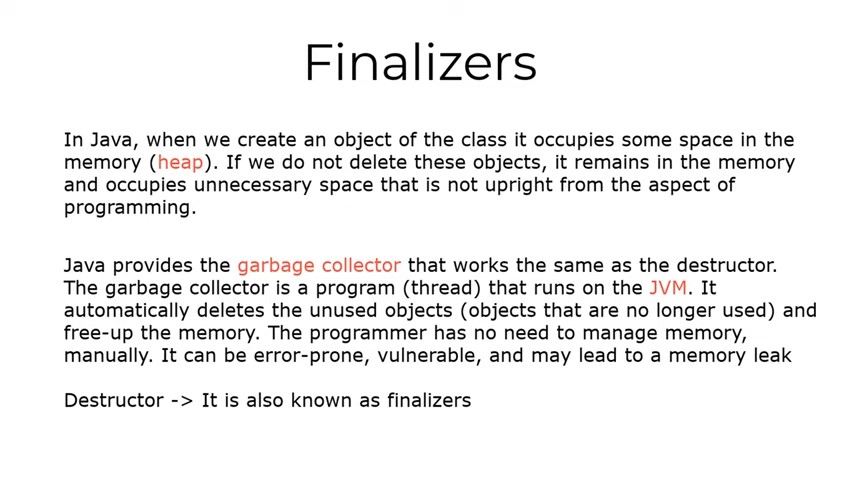
* String (object): String Pool, immutable
* String Buffer (object): mutable

access modifiers:

* public: anywhere in the project
* private: over class
* protected: over the package & (subclasses 🡪 between packages using inheritance)
* default: in the package

Final:

* final: keyword 🡪 using with classes & methods
* finally: block 🡪 try... catch, always executed even if handle Exception
* finalize: method 🡪 garbage collection ( clean up activity is implemented in heap)



static:

🡪 method: can't be overridden

🡪 variable: can't be reinitialized

🡪class: inner class ( static Nested Class )

🡪block: initialize static variable, static block has more priority

Than Constructor, inside static block, we can initialize non-

static Variables using objects.

Constructor:

* no return datatype, initialize the object state
* constructor name== class name
* can't be marked as final
* when can it be private? Singleton (one static instance can created from class), factory static method, unity class-> only contain static methods.

super, this?

* + this: Keyword 🡪 refer to the current instance of class
  + super: keyword 🡪
    - refer to the superclass of the current instance.
    - call the superclass constructor (Default) into subclass constructor (METHOD).

Stack, Heap?

* + Stack (LIFO): local & temporary variables and function call, fast
    - limited size, Managed automatically by the system
  + Heap: dynamic memory allocation using objects, DS like Arrays & LinkedList
    - manage by garbage collector 🡪 Slow, Large size

shallow copy, deep copy?

* + - * Shallow copy: copy references to nested objects
        + change changes in nested objects are reflected
        + in both the original and copied objects.
      * Deep copy: changes in nested objects do not affect each other
        + between the original and copied objects. 🡪create new object reference

throw, throws?

* + throw: throw Exception if something wrong has occurred
  + throws: the type of Exception like " FileNotFoundException "

composition, aggregation?

* + composition: relation between objects 🡪 one contain(Own) anthor (object part of anthor )
    - * containing object cannot exist without objects it contains
      * containing object is destroyed ->contained objects are also destroyed.
      * like (car & engine)
  + aggregation: relation between objects 🡪 one contain(Own) author
    - * the contained object can exist independently of the containing object
      * like(University & department)

System.out.println()?

* print in the Console 🡪
* Java.lang package 🡪 System 🡪 PrintStream 🡪 static variables like(out),methods like (println()).

\* java 8 🡪 stream, lambda, static &default (implemented) methods

in interface, functional interfaces, optional.

Concepts of OOP?

Inheritance:-

* Allows subclasses to inherit behaviors and properties from another class (superclass).
* Enables code reuse and extension without rewriting existing code.

Abstraction:-

* abstract class : at least one abstract method.
* interface: all methods are abstract by default, but after Java-8 default & static methods with implementation.
* Hides complex implementation details and shows only essential features of an object.
* Methods without implementation.

Polymorphism:-

* override 🡪 methods in subclass with a different implementation.
* Overloading 🡪 methods with the same name but different parameters type or number.
* Enables methods to be called on objects of different classes executing different behaviors based on the object type.

Encapsulation:-

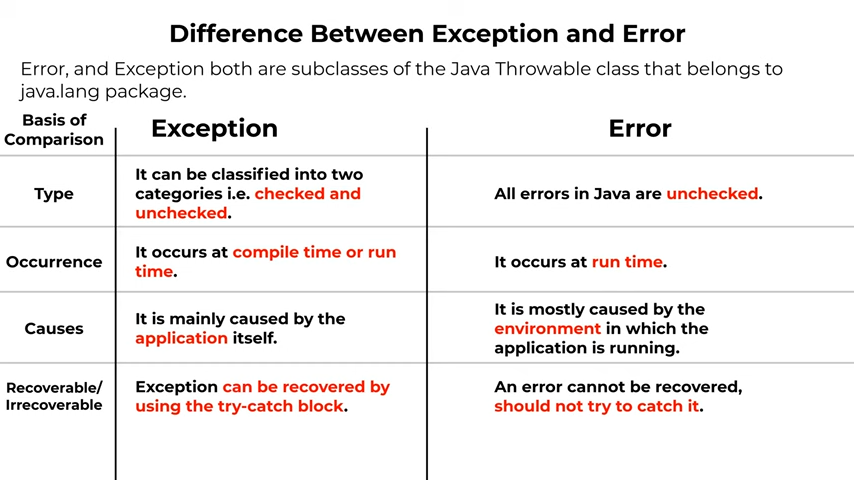
* hide data.
* direct access to some of an object's components and protects its internal state.
* private to variables, public to classes (setters & getters).

Exception Handling?...

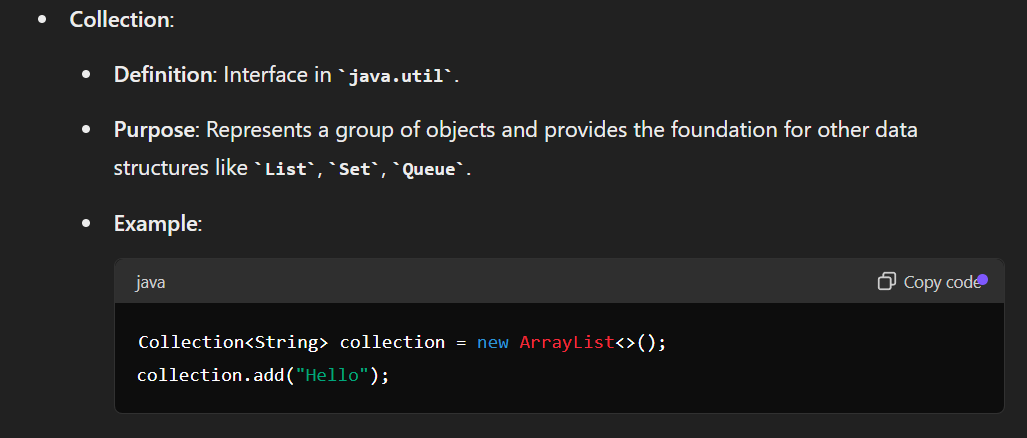
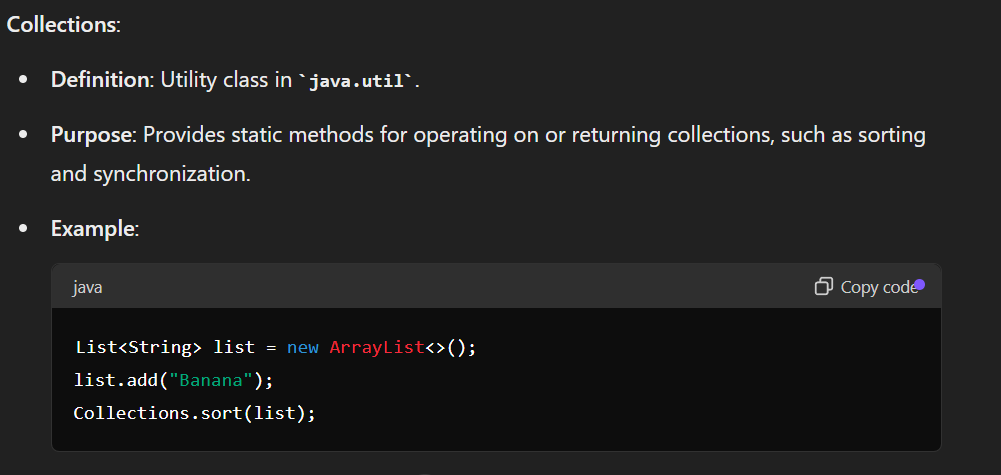
Checked exceptions (Compile-time exceptions)

* + is checked (notified) by the compiler at compilation-time,also called as compile time exceptions.
  + cannot simply be ignored, the programmer should take care of (handle) these exceptions.

Exception & error?



**collection/s** :

* Collection: 
* Collections: 

List, Map, Set

| **Feature** | **List** | **Set** | **Map** |
| --- | --- | --- | --- |
|  |  |  |  |
| **Interface** | java.util.List | java.util.Set | java.util.Map |
|  |  |  |  |
| **Allows Duplicates** | Yes | No | Keys: No, Values: Yes |
|  |  |  |  |
| **Maintains Order** | Yes | No (unless LinkedHashSet) | Yes (unless HashMap) |
|  |  |  |  |
| **Random Access** | Yes | No | Yes (for keys) |
|  |  |  |  |
| **Access Time** | O(1) for ArrayList,  O(n) for LinkedList | O(1) for HashSet, O(log n) for TreeSet | O(1) for HashMap,  O(log n) for TreeMap |
| **Null Handling** | Allows multiple null elements | Allows a single null element (for HashSet and LinkedHashSet) | Keys: Allows one null key  (for HashMap),  Values: Multiple null values |
|  |  |  |  |
| **Typical Use Cases** | Ordered lists,  frequent read access by index | Unique elements,  fast lookup and deletion | Key-value pairs,  fast lookup by key |
| **Memory Usage** | Depends on implementation, generally lower than Set | Higher memory usage due to storage of elements without duplicates | Higher memory usage due to  storage of key-value pairs |
|  |  |  |  |
| **Performance** | Fast random access, slower insertions/deletions for ArrayList; consistent time for LinkedList | Fast access, insertion, and deletion for HashSet; sorted access for TreeSet | Fast access, insertion,  and deletion for HashMap;  sorted access for TreeMap |

Streams:

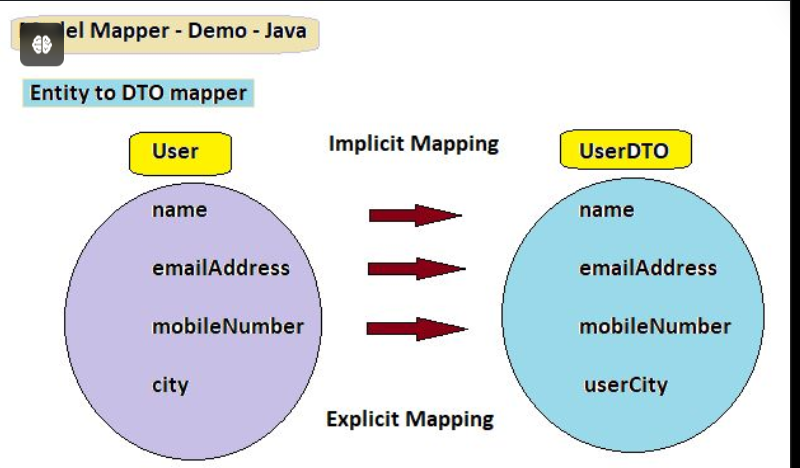
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

* wrapper for data provided to them
* not used to store -> not a data structure
* never modify underlying data given to it -> the original data don’t change.
* Stream Operations: Intermediate & Terminal operations
  + Intermediate: return a stream as a result (chained one after the other) 🡪 map, filter, sorted.
  + Terminal: make the end of chained stream calls, and return any result 🡪 collect, forEach, reduce.
* Rate limiting: Used to save server from failing , by limit the number of requests that user can make in a particular time .

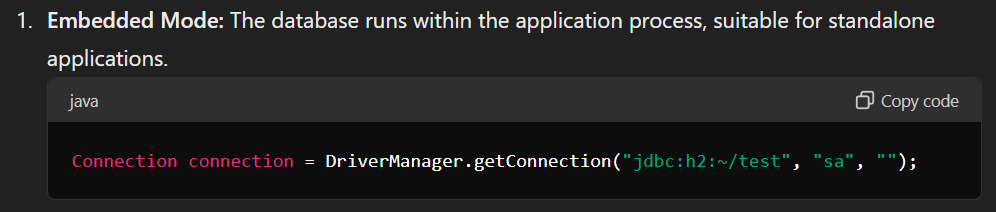
Ex: user can make 10 requests by minute . Rate limiting use

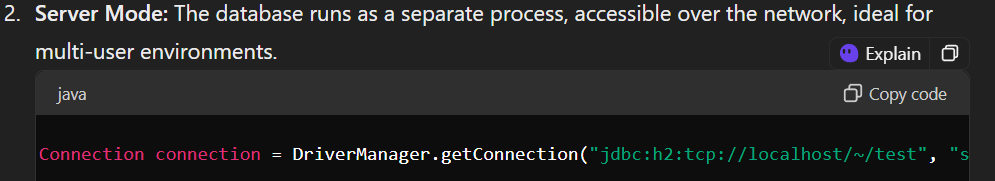
libraries like buket4j.

* SQL Injection: Attacker try to access database and reach to important data ( like password) or execute any SQL statement (update , select, delete…) by using some unconditional Statements that will always be true .
* DDOS Attack: Attackers try making the server failing by send fake requests by many attacked devices and that increase load on server, If the server goes down, too many requests will slow it down and increase costs.
* Model Mapper: java library that makes it easy to map (convert) objects from type to anthor & helps in converting between DTO & entities,
  + support inheritance and complex Structures🡪 versatile for applications dealing with complex data methods.
  + **mapping operations efficient, even when dealing with large volumes of data**
  + **integrates well with popular Java frameworks such as Spring and Hibernate, enhancing its usability in enterprise applications.**



* H2 Database: lightweight relational database management system written in java.





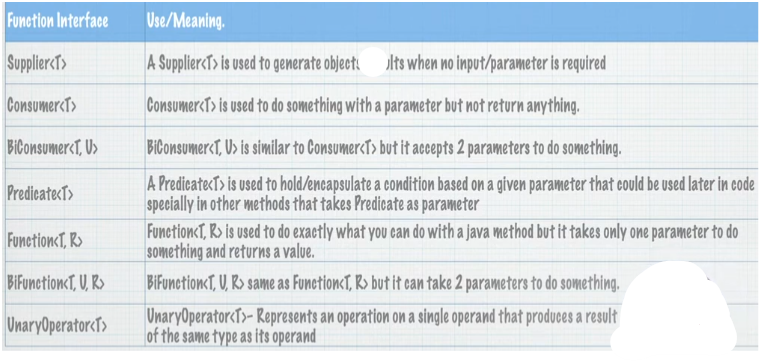
* Open source: Developers can access and modify the source code.
* High performance: Fast and efficient, consuming minimal resources.
* Small Footprint: The executable is small, suitable for small to medium applications.
* **SQL Compatibility:** Supports SQL standards and integrates well with Java frameworks.
* **Embedded & server Mode:** Can be used within **applications** (embedded) or as a standalone database **server.**
* **Management tools:** provides a web-based console for managing the database and running SQL queries.
* **Encryption:** Supports database encryption for security.
* **Backup & Restore:** Easy to backup and restore databases.

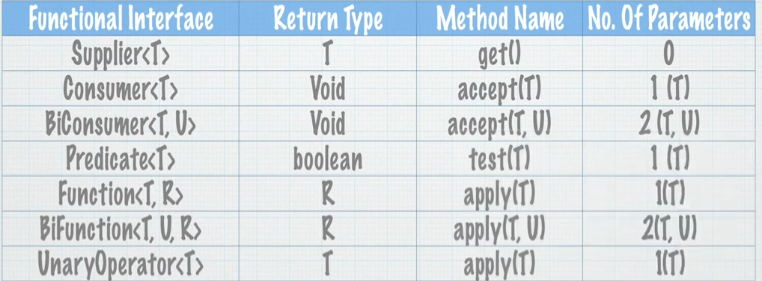
Functional Interfaces: **By java8 (java.util.function)**

Interface 🡪

* Contain only Single Abstract Method (SAM)
* Basis for lambda Expression

Built-in Functional Interface :





Method Reference

* Instance :: member method

Call method and pass parameters to it

JRE will infer to parameters types

* Class :: static method

Call static method , JRE will infer Parameter types

* Class :: member method

Call member method from first parameter

And pass the second as method argument

Ex: x.concat(y)

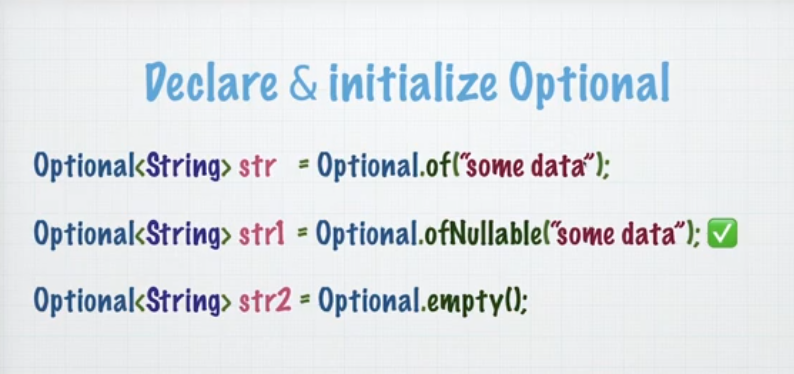
* Class :: new 🡪 constructor reference

Invoke class constructor -> Class obj=newClass();

Optional <T>

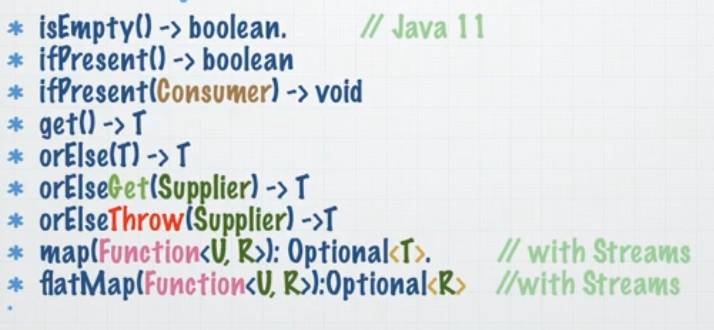
* Avoid NullPointerException @ return of method
* Increase Code Readability
* Design better APIs / Methods
* Usually used with Wrapper classes Like:[Float ,Integer ,Long…] that can be null

Ex: boolean can be True or False, Boolean can be True, False or null.



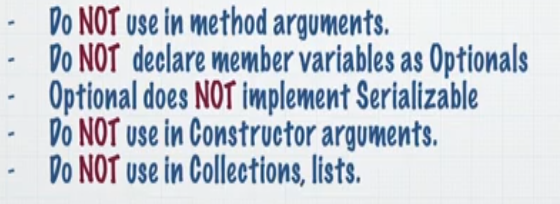
Optional.of() : doesn’t accept null values

Optional.ofNullable() : accept null values

Optional member methods : 

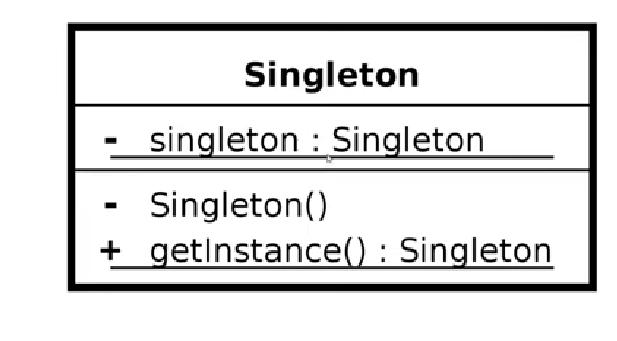
Where to use Optional ?

* + In method returns : force user to check for null .
  + With Streams / Functional Programming .

Where not to use Optional? 

Design Patterns:

* creational:
  + - Singleton:
* Ensure a class has only one instance and provide a global point of access to it
* private constructor
  + - * + static method (getInstance()) -> if there is instance return it or create an instance
        + can’t use “New” keyword
        + can use Enum -> Serialization &multithreading



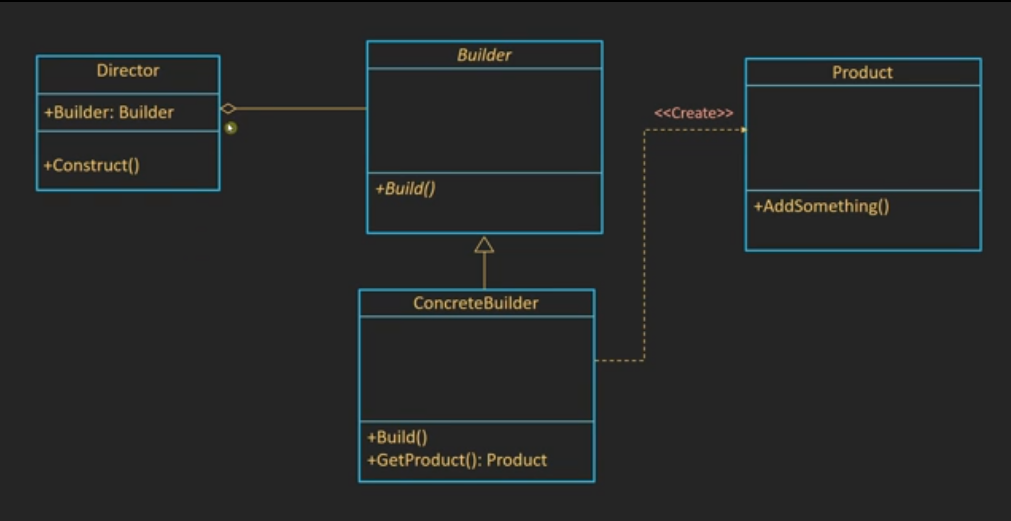
* + - Builder Pattern:

Separate the construction of a complex object from it’s

representation so that ,the same construction processes

can create different representations. (Build complex

objects step by step)…

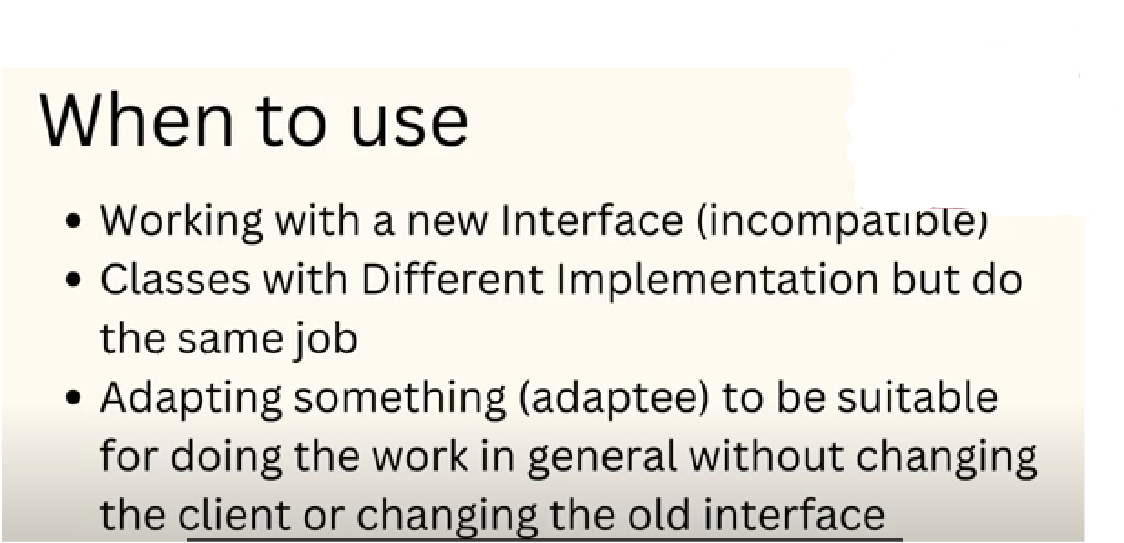


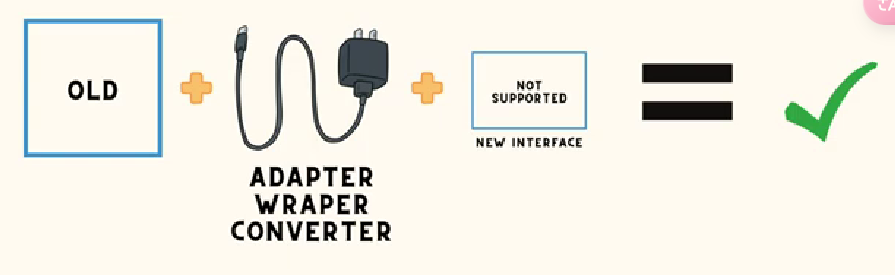
* + - factory method :

كـأني بعمل وسيط ينفذ اللي انا عايزو عشان احافظ ان كله هيتم بنفس الطريقة او الجودة ف الكلاسيز المختلفة ولما احتاج اعدل هعدل ف مكان واحد بس وهو ال )factory (

* Structural Patterns:
  + - adaptor (تكيف):

Adapt to changes, can use Inheritance or

 composition



# Proxy (wrapper):

* like a chain
* Extra behavior at run time instead of using static inheritance
* For Final Classes
* Reuse (objects/ methods/classes/… )

# Facade (Wrapper):

# Client 🡪 Facade 🡪 Complex Class (what client wants).

# The client doesn’t know anything about how the facade works.

* + Good Isolation (Changes don’t affect the client).
  + Giving simple interface for complex systems.
  + Can use more than one façade for a single system.

# 

* Behavioral: communication/interaction between classes
* Observer :
* Strategy :

# Command Pattern :

* decoupling/reducing the responsibility of the class that have to much functionality.

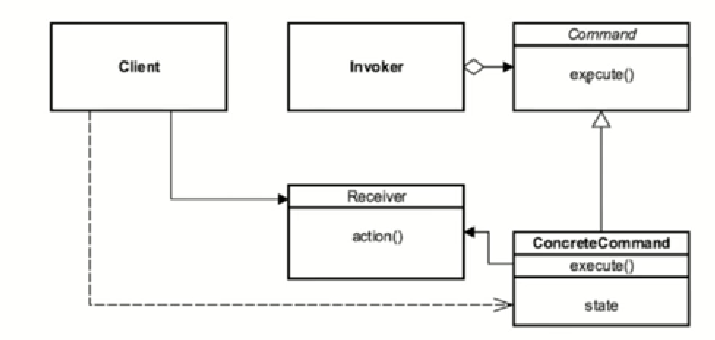
🡪 Invoker (execute some responsibilities of the class ).

عندي كلاس فيه مسؤليات (فانكشنز كتير) ومش هيقدر ينفذها كلها ف بيشوف حد يخفف من عليه شوية ويساعده ف تنفيذ بعض العمليات دي

بيبدأ يساعد الكلاس دا ويقوم بمسؤلياته Invoker

من غير ما يعرف او يفهم اي بالظبط نوع المسؤلية او ال “command” اللي هينفذه

* Encapsulating requests on object
* Undo operation.



Git

Commit :

-Git config –global core.editor “Vim”

Git commit { open editor to commit as like (subject /description)}.

-git log 🡪all commits

-git log - - oneline