# *Notes****:*** *Hanin🎀*

Object class 🡪 parent of all classes.

* String (object): String Pool, immutable (object refer to value, if you assign new value 🡪 new object has new value).
* StringBuffer (object): mutable

access modifiers:

* **public**: anywhere in the project
* **private**: variables and methods can be accessed only in the class they are declared, Private variables and methods from SuperClass are not available in SubClass(can’t be inherited).
* **protected**: over the package & (subclasses 🡪 between packages using inheritance)
* default(Package access): in the same package, when there’s no access modifier.

Final:

* final keyword: using with classes, methods & variables.
* Final variable: prevent reinitialization.
* Final methods: prevent overriding.
* Final argument: can’t be modified.
* Final Classes: prevent inheritance(can’t be extended).
* finally: block 🡪 try... catch, always executed even if handle Exception.
* finalize method: garbage collection (clean up activity is implemented in heap).

static:

**Method**: Cannot be overridden, but it can be hidden, doesn’t need object to be called,just ClassName.methodName();

**Variable**: There is only one copy of the static variable for the entire class. Instances of the class do not have a unique copy of the static variable.

**Class**: Refers to static nested classes, which cannot directly access non-static members of the enclosing class(outer class).

**Block**: Used to initialize static variables, have a higher priority than constructors during class loading. Non-static variables cannot be initialized directly inside static blocks, but they can be initialized indirectly 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.
* Validation🡪 throw Exception.
* We can call superclass constructor using “super()”method, it should be the first statement.
* Can’t be called from any method, expect another constructor.
* Super class default constructor is automatically invoked from sub class constructor.

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 the subclass constructor (METHOD).

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:
* can contain zero or more abstract methods(abstract return\_type methodName); tha
* t’s can be declared only in abstract classes.
* Can’t be instantiated.
* Allows to partially implement your class.
* interface:
  + - All methods are only abstract & public by default, but after Java-8 default & static methods with implementation.
    - All variables are always public static final (cannot be private).
    - An Interface can extend another interface.
    - Can’t extend a class, but can extend another interface.
* Hides complex implementation details and shows only essential features of an object.
* Methods without implementation.

**Abstract Class VS Interface:**

* Methods and members of an abstract class can have any visibility. All methods of an interface must be public.
* A concrete child class of an Abstract Class must define all the abstract methods. An Abstract child class can have abstract methods. An interface extending another interface need not provide default implementation for methods inherited from the parent interface.
* A child class can only extend a single class. An interface can extend multiple interfaces. A class can implement multiple interfaces.
* A child class can define abstract methods with the same or less restrictive visibility, whereas a class implementing an interface must define all interface methods as public

**Polymorphism**:- Same code giving different behavior.

* **Method Overloading:** when **multiple methods** in the **same class** share the **same name** but have **different parameters** (different numbers or types of parameters). It allows the class to provide various ways to perform a task. The method is resolved at **compile time** based on the parameter list. Like (constructors in classes).
* **Method Overriding**: happens when a **subclass** provides a **specific implementation** of a method already defined in its **parent class**. It allows a subclass to alter the behavior of the inherited method from the parent class. The method is resolved at **runtime** based on the object's type.
* 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).

**Inner Class: class declared inside another class(Outer Class) or method.**

**Static Inner Class: class declared as static inside another class.**

**Anonymous Class: class without a name.**

**Instanceof:** operator checks if an object is of a particular type

**Coupling:** measure of how much a class is dependent on other classes. There should minimal dependencies between classes. So, we should always aim for low coupling between classes.

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

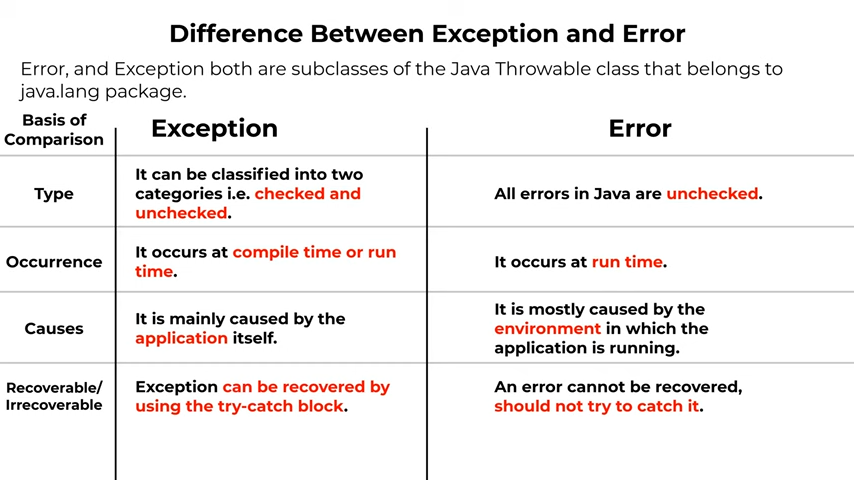
in interface, functional interfaces, optional.

Exception Handling?...

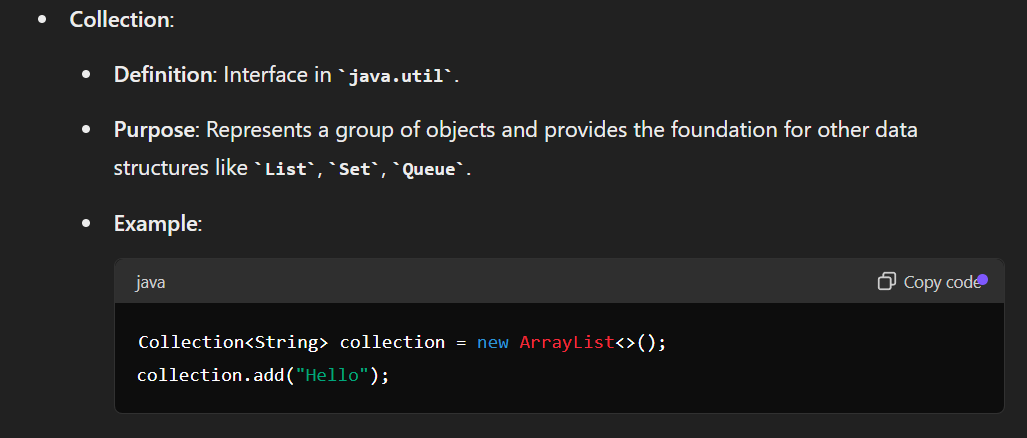
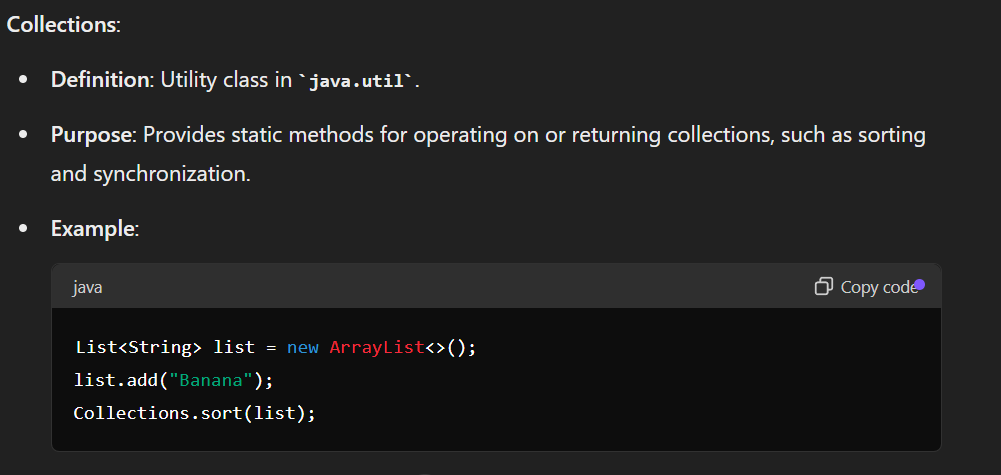
Checked exceptions (Compile-time exceptions)

* + 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 per 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.

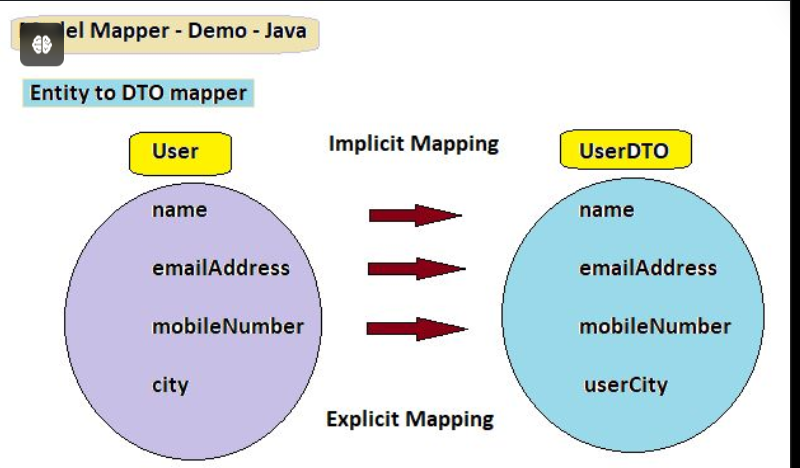
🡪using rate limiting/load balancer to solve and avoid this.

* Model Mapper: java library that makes it easy to map (convert) objects from type to another & helps in converting between DTO & entities,
  + Supports 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.**

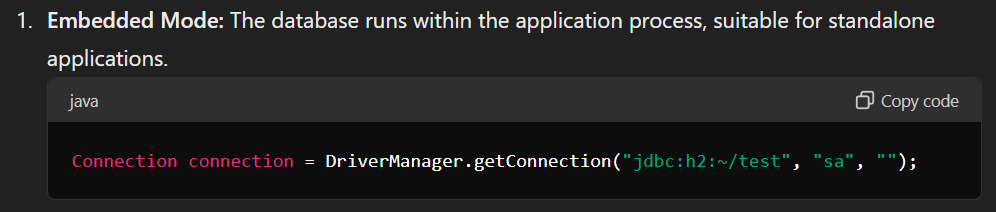
**عشان اتحكم ف الداتا اللي Model mapper بكل بساطة بستخدم ال**

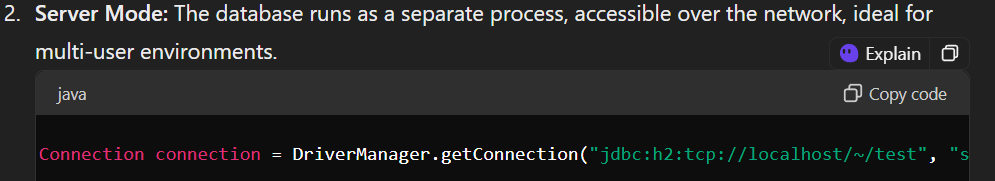
**“API” اللي هعرضها او هنقلها لل**

**ف اقدراتحكم واحدد أي اللي انقله م الداتا بيز اللي عندي واي لا زي الباسورد مثلا مش بظهره , وبتحكم كمان ف ازاي اعرض الداتا زي اني أقول اعرضلي الايميل كله كابيتل وهكذا , ف كل اللي بعمله اني بنقل الداتا اللي عايزها من كلاس لكلاس تاني باستخدام ال model mapper.**



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





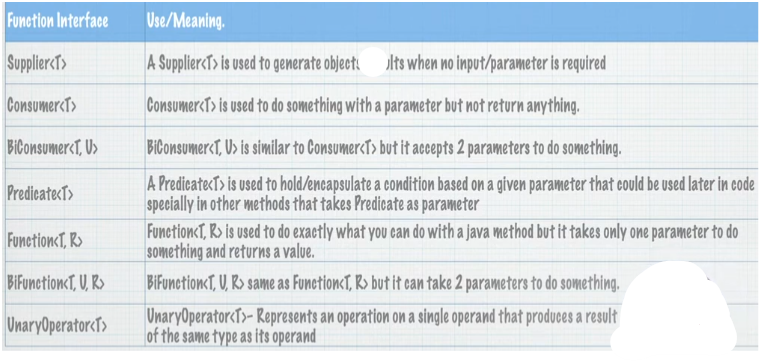
* Open source: Developers can access and modify the source code.
* High performance: Fast and efficient, consuming minimal resources.
* Small Footprint: The executable is small, suitable for small to medium applications.
* **SQL Compatibility:** Supports SQL standards and integrates well with Java frameworks.
* **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.

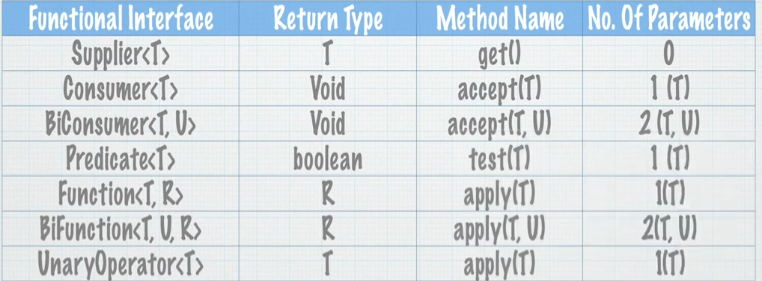
Functional Interfaces: **By java8 (java.util.function)**

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 parameter 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)

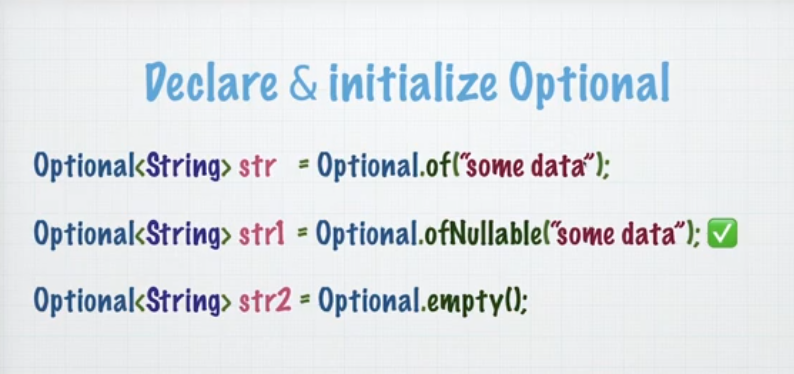
* Class :: new 🡪 constructor reference

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

Optional <T>

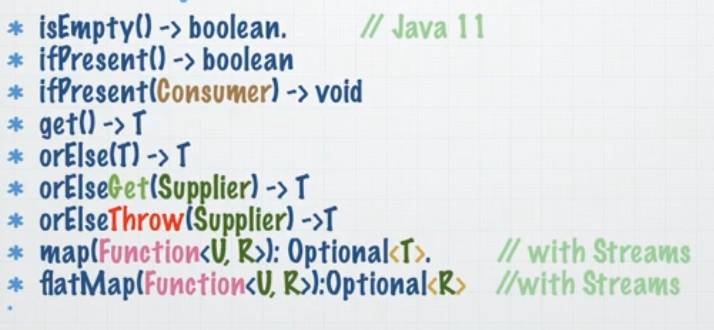
* Avoid NullPointerException @ return of method
* Increase Code Readability
* Design better APIs / Methods
* Usually used with Wrapper classes Like:[Float ,Integer ,Long…] that can be null

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 the user to check for null.
  + With Streams / Functional Programming.

Where not to use Optional? 