If you have any questions or suggestions regarding the notes, please feel free to reach out to me on WhatsApp: **Sanjay Yadav Phone: 8310206130**

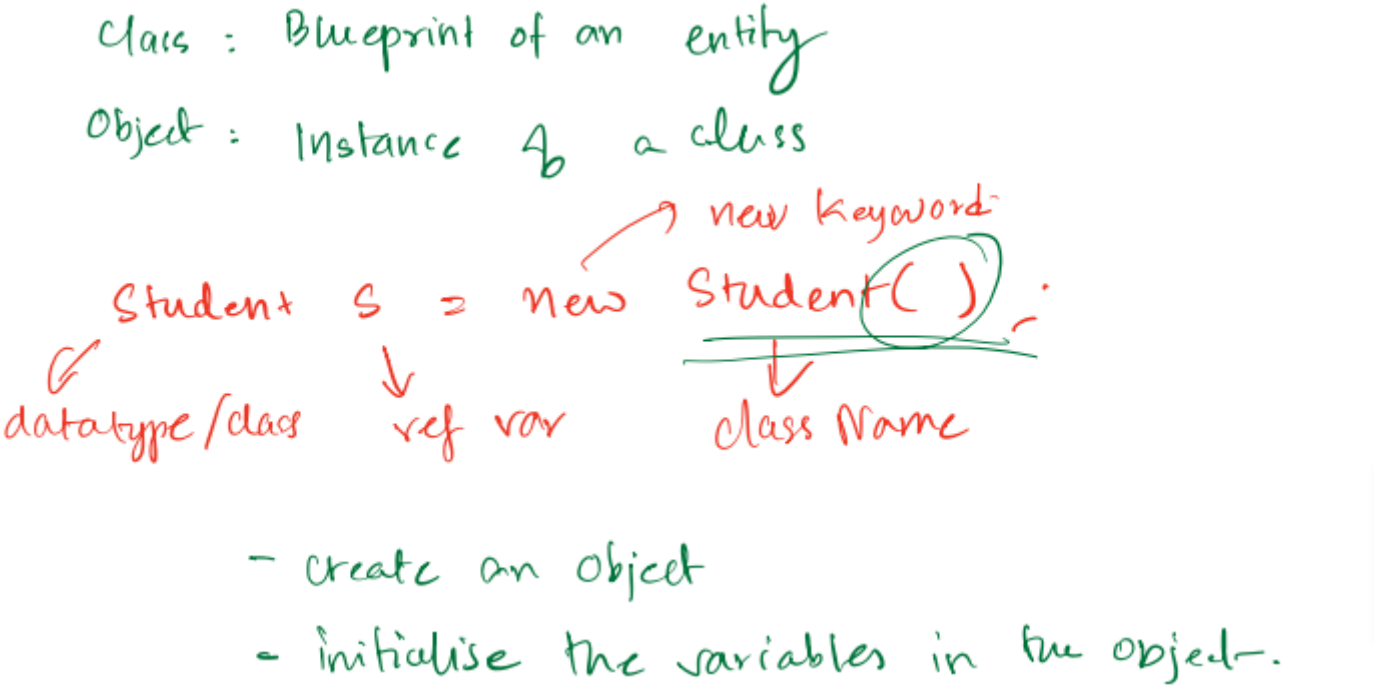
Install java in VS Code
https://dev.to/hadyrashwan/setup-java-and-vs-code-on-ubuntu-4ahl

Install InteliJ
https://linuxhint.com/install-intellij-idea-on-ubuntu-20-04/

Backend LLD: OOP-1: Intro to OOP

- **Introduction ToOOPS**
- **Principles Pillars of OOPS**
  - **Abstraction**
  - **Encapsulation**
  - **Polymorphism**
  - **Inheritance**

Packages -> Packages are essentially folders referred to as "Packages" within a directory structure. When creating a class manually, it's necessary to declare the package at the top of the file using the package PackageName syntax. If you're using an Integrated Development Environment (IDE) to create the class, the IDE will typically insert this package declaration automatically.



Introduction to OOPS -> The word object-oriented is a combination of two terms, object and oriented.
The dictionary meaning of an object is "an entity that exists in the real world", and oriented means "interested in a particular kind of thing or entity".

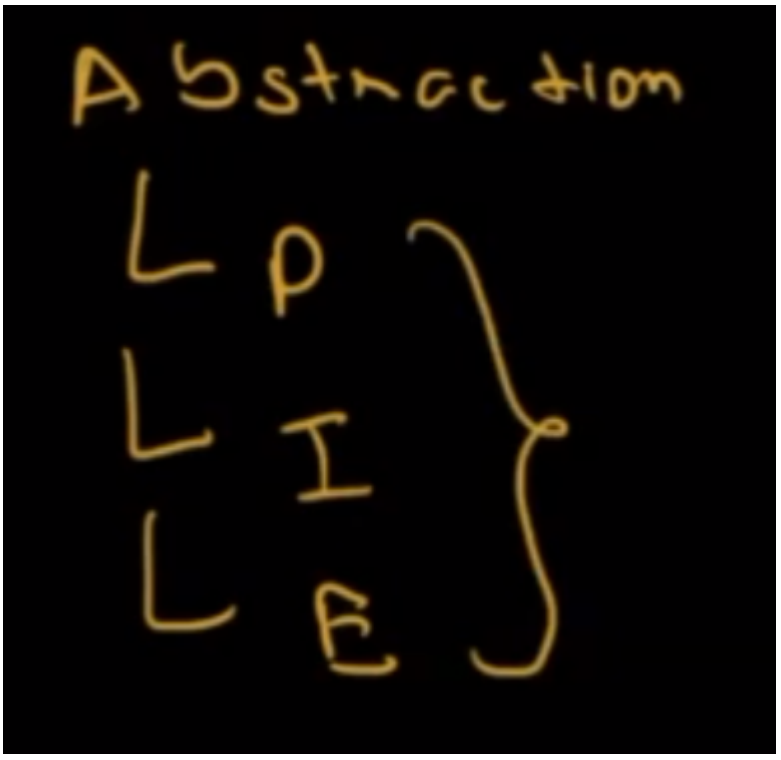
In basic terms, OOP is a programming pattern that is built around objects or entities, so it's called object-oriented programming.

A paradigm (type) of programming

There are 4 type paradigm (fundamental style) of programming

- Procedural
- OOPS
- Functional
- Reactive

- **Procedural** -> A set of instructions. procedural programming relies on a linear top-down approach. It executes a series of instructions in a step-by-step manner, manipulating data using procedures or functions.

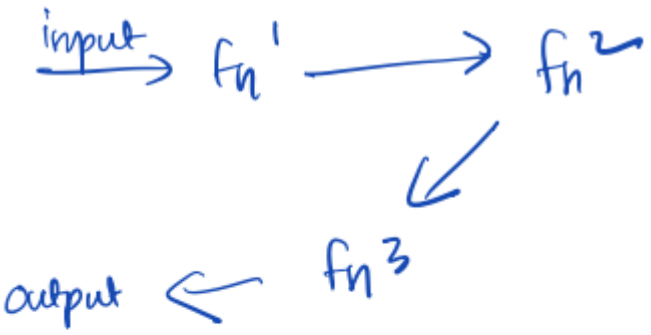


Procedural

Procedure — A set of instructions

↳ functions / methods

```
function (input data) {
    return output
}
```




```
a() {
  - -
  - -
}

b() {
  - - -
  a();
  - - -
}

Main() {
  a();
```

Akash is teaching

India is winning the WC

Everyone is thinking of a line

Somebody is sleeping

subject + verb

Relatable

Procedural code

verb

subject

```
printStudent(name, id, age, batch, psp)
{
    .  .  .
    .  .
    .  .
}
```

```
struct Student {
    name;
    age;
}
```

OOPs

OOPs

```
printStudent(Student)
```

Something being done on someone.

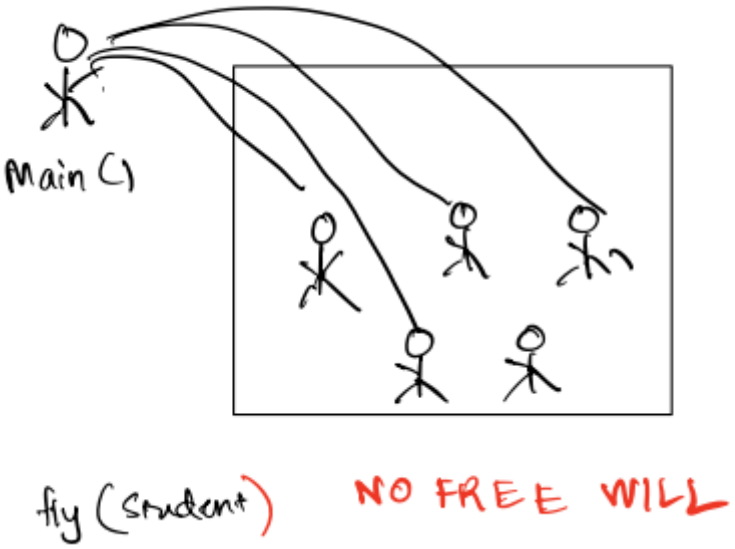
```
attendExam(Student)
```

```
Student.printStudent();
```

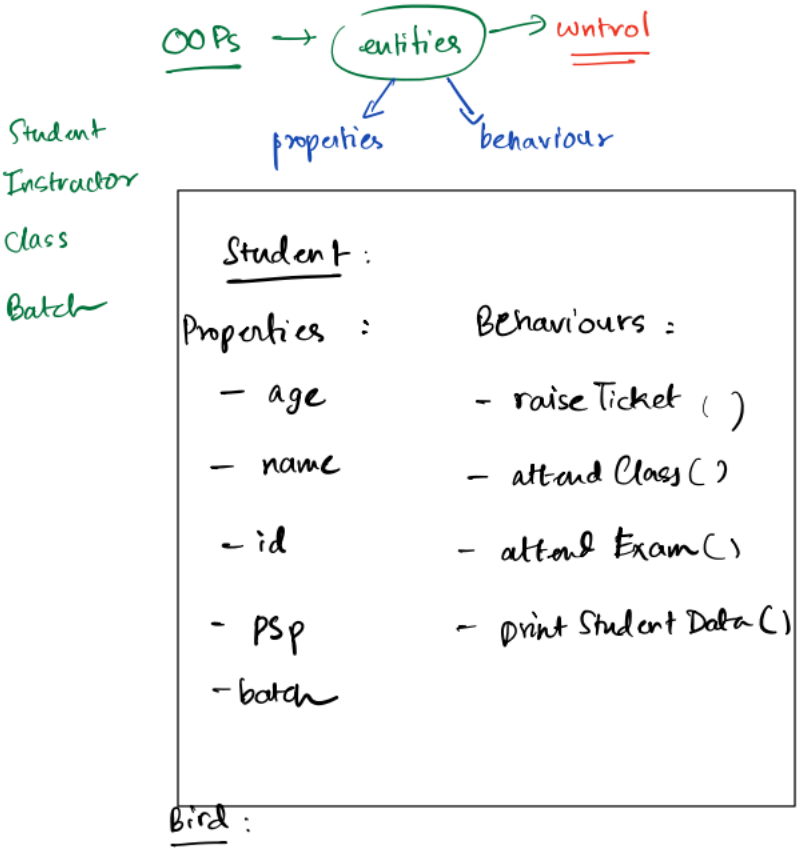
Somebody is doing something.

```
Student.attendExam();
```

In the image below, we can observe a situation where one person is trying to manage all puppets in procedural programming. Everything is controlled by the main function, and as the complexity grows, it becomes difficult to maintain and turns into a messy situation. That's why object-oriented programming (OOP) is preferred over procedural programming to address these challenges and provide a more organized structure.



OOPS -> OOPS, or Object-Oriented Programming, is centered around entities with a primary focus on specific objects or concepts. An entity in this context can be anything—a name, place, living or non-living thing, relationship, idea, etc.—each possessing distinct properties and behavior.



4 pillars of oops

- **Abstraction**
- **Encapsulation**
- **Inheritence**
- **Polymorphisim**

Abstraction -> Abstraction means present complex concept in simpler and understandable manner. Basically, we use abstraction to handle complexity by allowing the user to only see relevant and useful information.

A good example to explain this is driving an automatic car. When you have an automatic car and want to get from point A to point B, all you need to do is give it the destination and start the car. Then it'll drive you to your destination.

What you don't need to know is how the car is made, how it correctly takes and follows instructions, how the car filters out different options to find the best route, and so on.

The same concept is applied when constructing OOP applications. You do this by hiding details that aren't necessary for the user to see. Abstraction makes it easier and enables you to handle your projects in small, managable parts.

Due to abstraction:

- Increased readability
- Enhanced comprehensibility
- Improved manageability
- Greater extensibility

Encapsulation -> Encapsulation is like a protective capsule that bundles different medicines together, shielding them from the external environment. Similarly, in Java, encapsulation consolidates and secures all properties and behaviors, preventing unauthorized access to these attributes and functions in a class. This is the concept that binds data together. Functions manipulate the info and keep it safe. No direct access is granted to the info in case it's hidden. If you wish to gain access to the info, you need to interact with the modifiers.

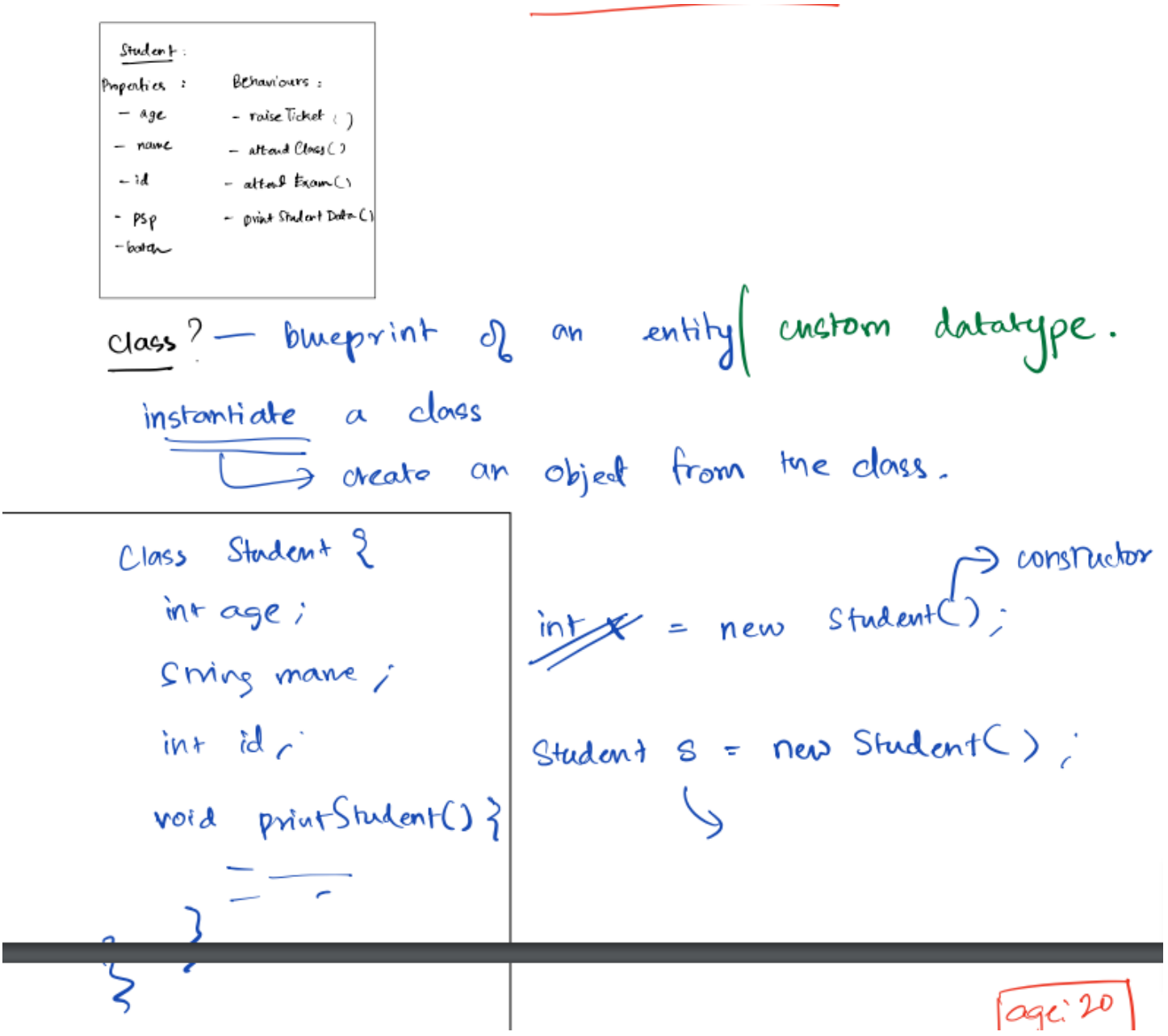
Class ->these are user-defined data types that act as the blueprint for objects, attributes, and methods. **It defines a data structure and behavior** that objects created from the class will have. The structure of a class includes fields (variables) and methods (functions) that describe the properties and actions associated with the objects.

Fields/Attributes: These are variables that hold data or state for an object. For example, in a Car class, you might have fields like color, model, and year.

Methods/Functions: These are procedures or behaviors that the objects created from the class can perform. For example, a Car class might have methods like start(), drive(), and stop().

Object: An instance of a class. When you create an object from a class, you are creating a specific instance with its own set of data and behavior, based on the blueprint provided by the class.

So, in summary, a class in OOP serves as a blueprint for creating objects, defining their structure and behavior.



Polymorphisim -> Polymorphism refers to the ability of a function or method to take on different forms or behaviors based on the context in which it is used. It allows objects of different types to be treated as objects of a common type, providing a way to create more flexible and reusable code. There are two types of polymorphism in Java: compile-time (static) polymorphism and runtime (dynamic) polymorphism.

Compile-Time (Static) Polymorphism: Compile-time polymorphism is achieved through method overloading.

Method Overloading:
In method overloading, multiple methods have the same name but differ in the type or number of their parameters.

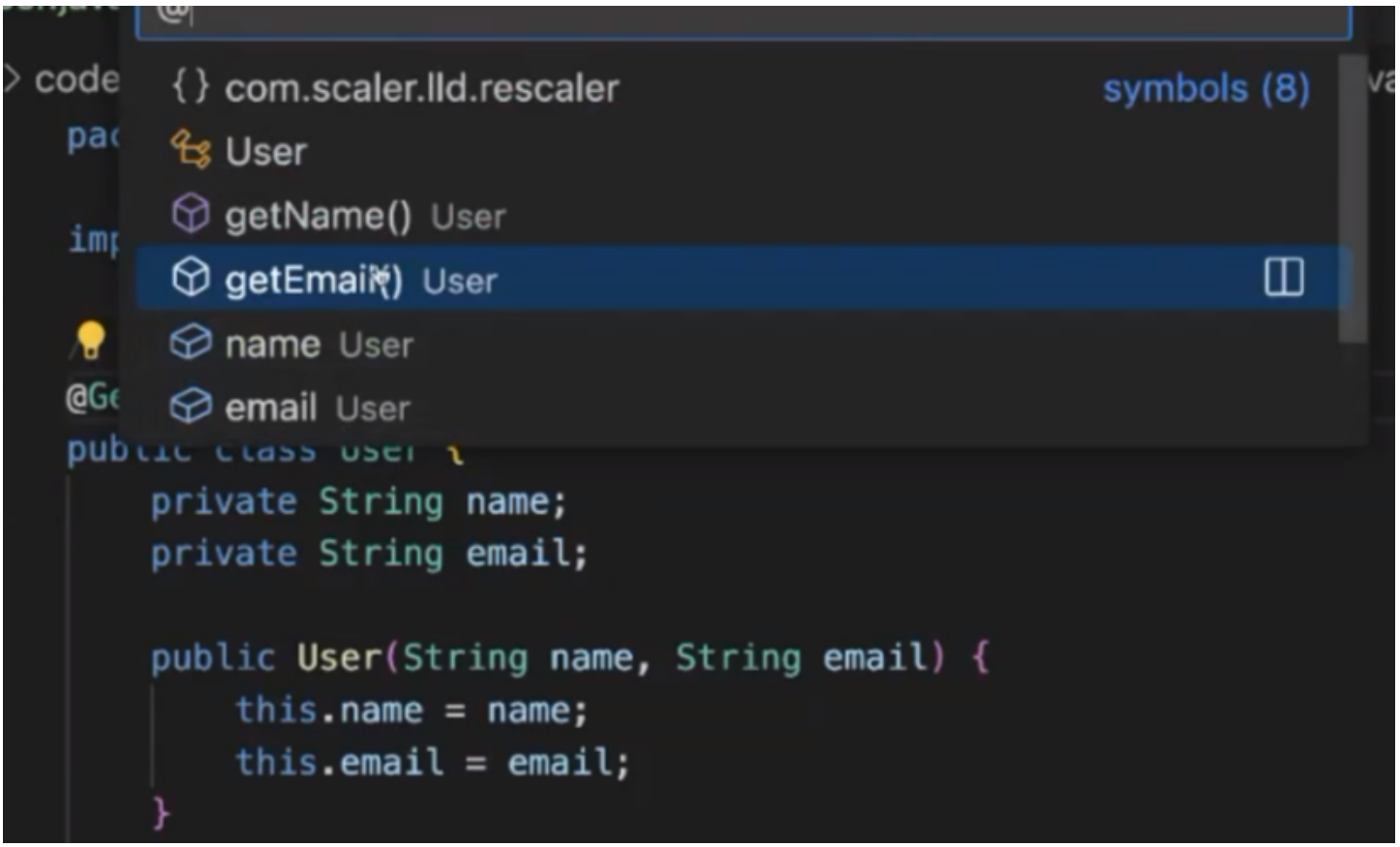
```
public class Example {
    void display(int num) {
        System.out.println("Integer: " + num);
    }

    void display(String text) {
        System.out.println("String: " + text);
    }
}
```

Note:
1. In method overloading, only the datatype and parameter number are considered in determining the method signature. If the return type is different, it will not be counted as a change in the method signature. Additionally, if the datatype of the parameters is the same and only the names of the parameters are different, it is also not considered a difference in the method signature.

2. Lombok is a library that simplifies Java code by providing annotations, such as @Getter, which automatically generates getters and setters for class properties.

```java
import lombok.Getter;

Parent class
@Getter
public class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }
```

Below we can see it created getName and getEmail fro filed of name and email it is the cleanest way to create gaters and setters.



Runtime (Dynamic or subtype) Polymorphism: Runtime polymorphism is achieved through method overriding and interfaces.

SubType: Subtype polymorphism, also known as runtime polymorphism or dynamic polymorphism, is a specific form of polymorphism in object-oriented programming (OOP). It occurs when a class or type can represent instances of its own type as well as instances of one or more of its subtypes (derived classes). This concept is closely associated with method overriding.

```
Animal myDog = new Dog();
myDog.sound(); // Calls Dog's sound method at runtime
```

Method Overriding: In method overriding, a subclass provides a specific implementation for a method that is already defined in its superclass.

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    } }

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
Animal myDog = new Dog();
myDog.sound(); // Calls Dog's sound method at runtime
```

Interfaces: Interfaces in Java allow for achieving polymorphism through interface implementation.

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }
}

Shape myCircle = new Circle();
myCircle.draw(); // Calls Circle's draw method at runtime
```

Polymorphism in Java enhances code flexibility, maintainability, and extensibility by allowing objects to be treated more generically, promoting code reuse and abstraction.

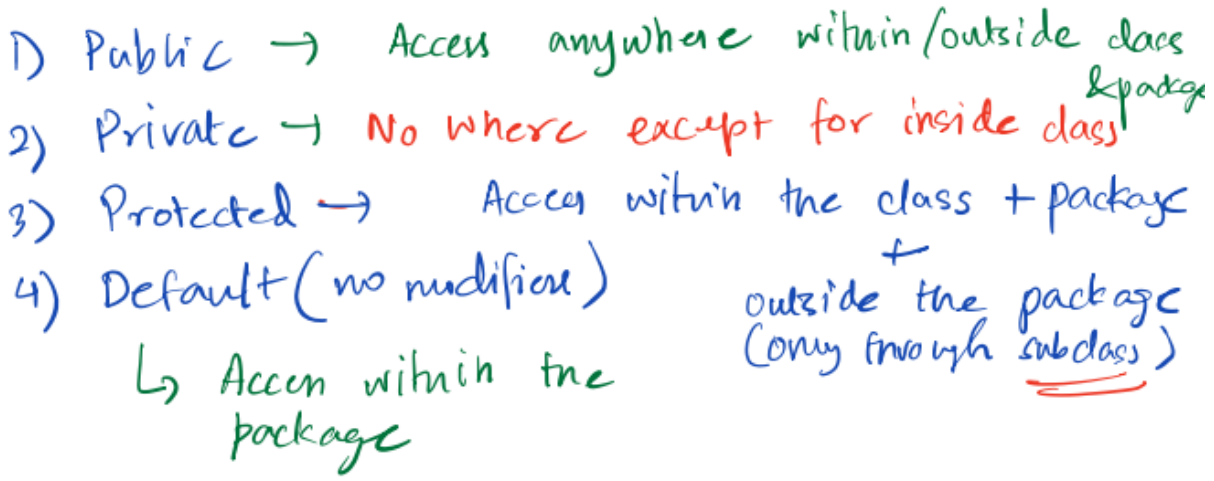
OOP-2: Access Modifiers and Constructors - 16-Nov-23

- **Access Modifiers**
- **Introduction to Constructors**
- **Default constructor**
- **Manual Constructor**
- **Copy Constructor**
- **Deep and Shallow copy**

Access Modifiers - Access modifiers are keywords in object-oriented programming languages that determine the visibility and accessibility of classes, methods, and other members within a program. Different programming languages may have different access modifiers, but some common ones include:

- **Public** -> Access anywhere within/outside class and package (Folder)
- **Private** -> No Where except for inside class
- **Protected** -> Within the class and package. It will access outside the package also but only through Sub Class like **Public Class StudentChild extends Student**
- **Default** (No modifiers) -> Within the class and package

Note -> If we have a regular method in a class, we can use the keyword this to refer to it without creating an object. But, if the method has certain modifiers like "public static," we need to create an object first before using it, even if it's in the same class. These rules apply when working inside the class, either through an object or within the method.



	Within Same Class	Within same package	Outside the package- (Subclass)	Outside the package- (Global)
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes (only to derrived class)	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

```java
package AccessModifiers;

public class Student {
    public String name;
    private int psp;
    protected String batch;
    int age;

    static int count;

    public static void main(String[] args) {
        Student s = new Student();
        s.count++;

        Student x = new Student();

        System.out.println(x.count);

        s.name = "Akash";


    }

    void doSomething(){
        this.name = "Akash";
        this.psp = 10;
        this.batch = "Feb23Morning";
        this.age = 20;
    }

}
```

If there are classes with the same name in different packages, you can instantiate an object by specifying the package name followed by the class name or import the class and than create without instantiate

For instance:
PackageName.ClassName objectName = new PackageName.ClassName();

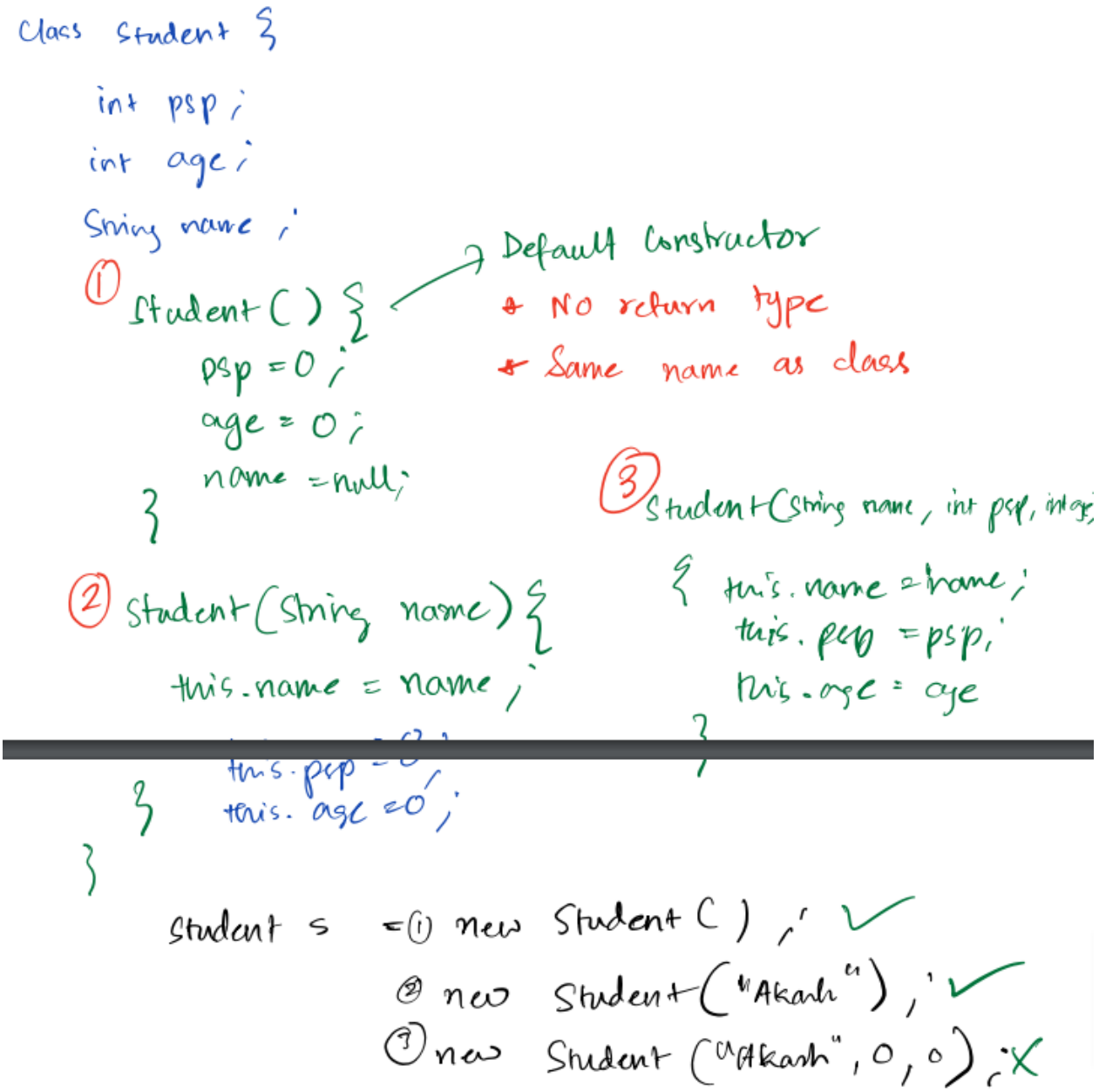
For example, if you have a class named Student in the StudentFolder package, you would create an object like this:
IntroductionOopsStudent.Student s = new IntroductionOopsStudent.Student();

For import:
import IntroToOOPs.Student
Student s = new Student();

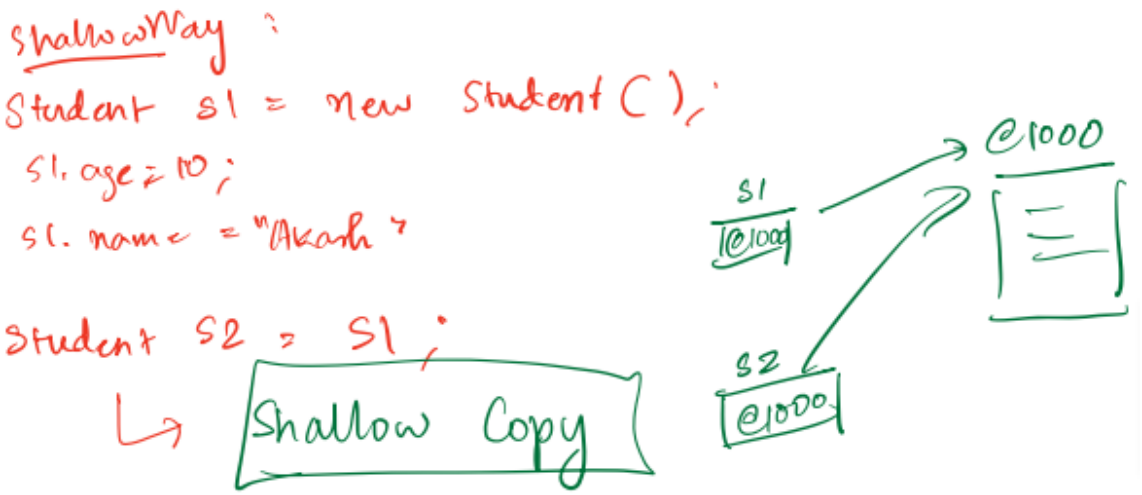
Introduction to Constructors -> Constructors are responsible for creating objects and initializing the variables within those objects. The default constructor is automatically generated when a class is created, and it assigns default values to all variables. For example, for int, it is 0, for float it is 0.0, for string it is null, and for boolean, it is false.

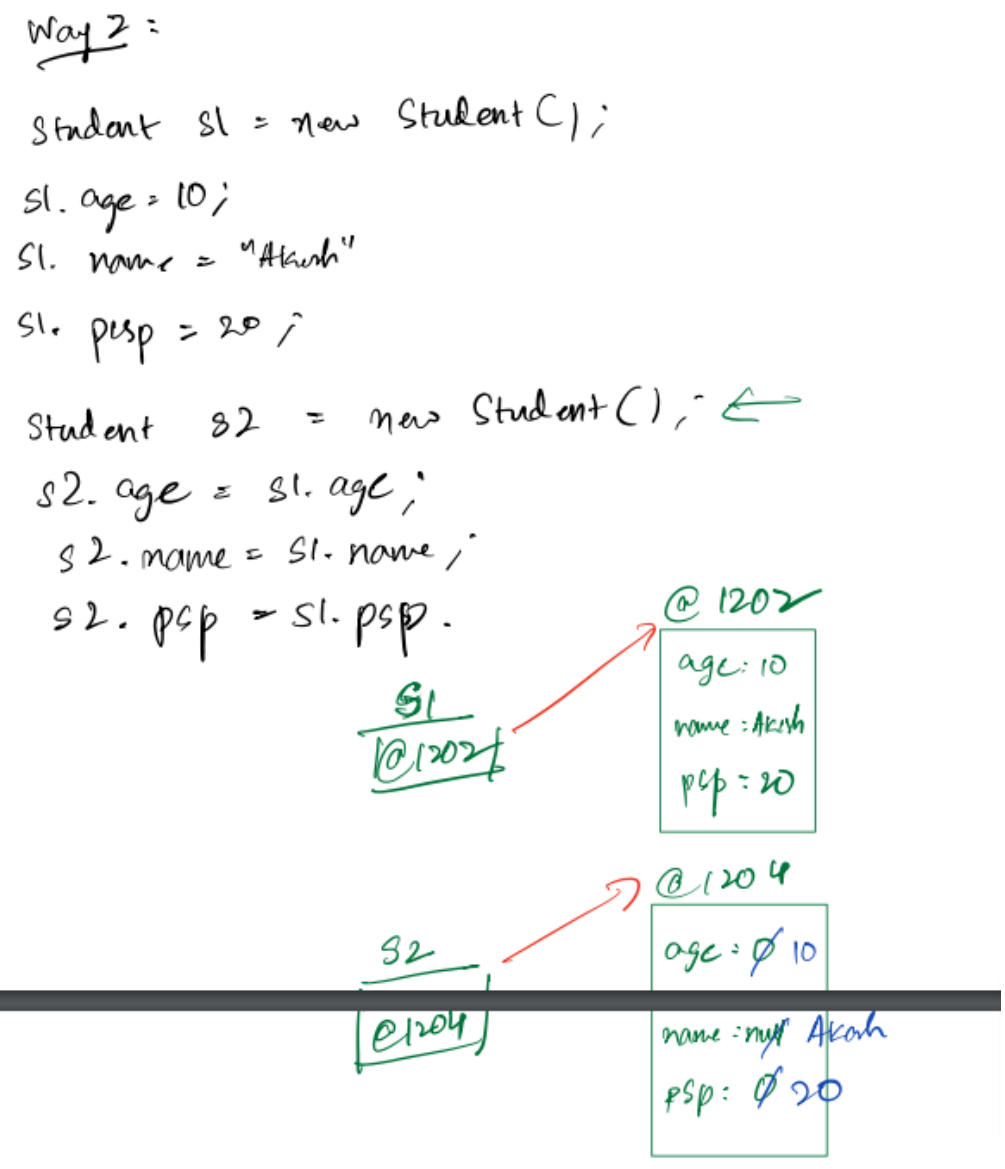
- Constructors don't have a return type.
- They share the same name as the class.
- Multiple constructors can be created within a class.
- Different constructors can have different parameters.
- Constructors with the same variable length cannot be created.

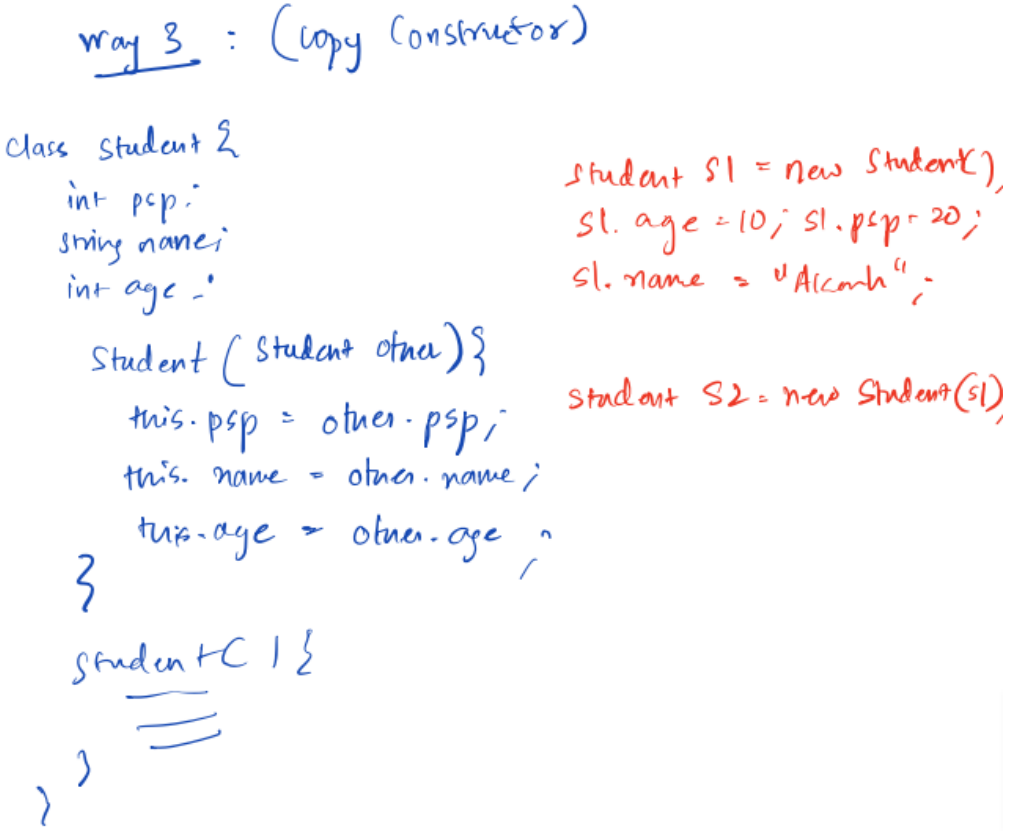
Manual Constructor -> We can write custom constructor paramtrize or without parmater



Copy Constructor -> A copy constructor in object-oriented programming is a special type of constructor that creates a new object by copying the values of an existing object of the same class. Its purpose is to provide a way to create a new instance with the same state as an existing instance.



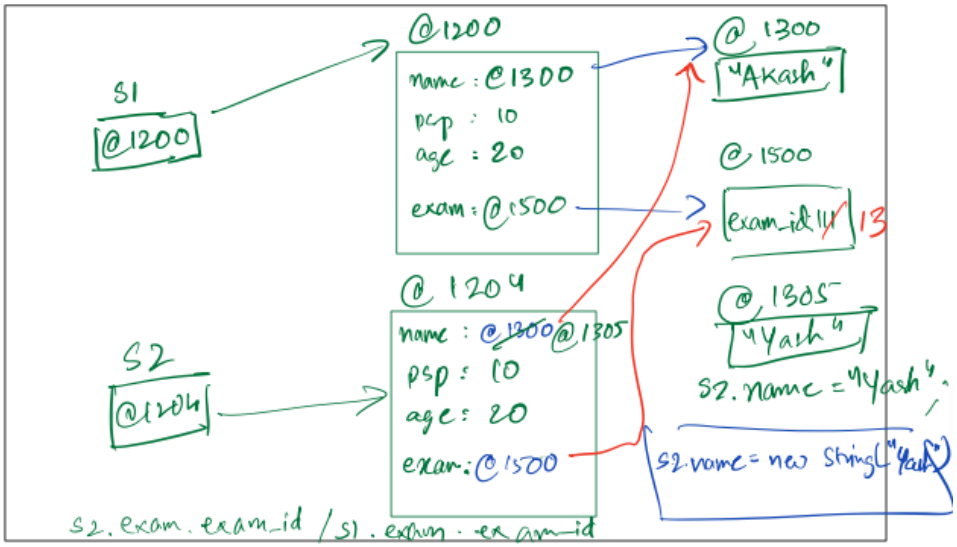




Note:

1. When we change the content of a string by assigning it a new value, the reference-type variable nature of strings means the change reflects across different variables. However, when we use a constructor to assign values, each object gets its own copy, and modifying one doesn't affect others. Despite being of reference type, the immutability of strings ensures that assigning a new value creates a new object with that value.

2. Copying a constructor involves ensuring that the copy doesn't include objects from other classes. If it does, and those objects contain references, modifying one instance may impact others. For a successful copy, structure the constructor to exclude references to objects from other classes, ensuring each instance remains independent.



OOP-3: Inheritance and Polymorphism - 20th Nov

- **Inheritance and Constructor Chaining**
- **Polymorphism**
- **Method overloading**
- **Method Overriding**

Inheritance -> Inheritance is a mechanism that allows a new class (subclass or derived class) to inherit properties and behaviors from an existing class (superclass or base class). The subclass can reuse the fields and methods of the superclass, promoting code reuse and creating a hierarchical relationship between classes.

Constructor Chaining -> Constructor chaining is the process of calling one constructor from another within the same class or between base and derived classes.

In Java, the super() keyword is used to invoke the constructor of the superclass, and this() is used to call another constructor within the same class.

Constructor chaining ensures that common initialization code in one constructor is reused by other constructors.

```java
class Animal {
    private String name;

    // Constructor for Animal class
    public Animal(String name) {
        this.name = name;
        System.out.println("Animal constructor called");
    }

    public void eat() {
        System.out.println(name + " is eating");
    }
}

class Dog extends Animal {
    private String breed;

    // Constructor for Dog class
    public Dog(String name, String breed) {
        super(name); // Call the constructor of the superclass (Animal)
        this.breed = breed;
        System.out.println("Dog constructor called");
    }

    public void bark() {
        System.out.println(name + " is barking");
    }
}
```

Polymorphism -> It a concept that allows objects of different types to be treated as objects of a common type. It enables a single interface to represent various types and allows objects to be processed in a generic way or **we can say same object has different behaviour like user object can be admin, customer, manager**

There are two types of polymorphism in Java:

1. **compile-time polymorphism** (also known as method overloading)
2. **runtime polymorphism** (achieved through method overriding).

Compile-Time Polymorphism (Method Overloading):
Method overloading allows a class to have multiple methods having the same name, but with different parameters (either a different number of parameters or different types of parameters).

```
public class Example {

 // Method with two int parameters
    public int add(int a, int b) {
        return a + b;
    }


 // Method with three int parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with two double parameters
    public double add(double a, double b) {
        return a + b;
    }
}
```

The compiler determines which method to call based on the number and types of arguments during compile-time.

Note: Return type can we anything but signature should be different like
void add(),
int add (par1)

Runtime Polymorphism (Method Overriding):
Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is achieved by using the @Override annotation.

```
public class Animal {
    public void makeSound() {
        System.out.println("Some generic sound");
    }}

public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}

public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}
```

At runtime, the JVM determines the actual object type and calls the overridden method of that object type.

```
Animal dog = new Dog();
Animal cat = new Cat();

dog.makeSound(); // Calls Dog's makeSound
cat.makeSound(); // Calls Cat's makeSound
```

This is often achieved through interfaces and abstract classes, allowing for more flexible and extensible code.

Polymorphism helps in writing more generic and reusable code by treating objects of different classes in a uniform way. It contributes to the flexibility and extensibility of code, allowing new classes to be added with minimal modifications to existing code.

OOP-4: Interfaces, Abstract Classes, Composition, Association - 21 Nov

- **Interfaces**
- **Abstract Class**
- **Static Keyword**

Interfaces -> an interface is a contract that is similar to a class. It is a collection of abstract methods. When a class implements an interface, it inherits the abstract methods declared in the interface and must provide concrete implementations for all of them. In addition to abstract methods, interfaces can also contain constants and default methods.

Here are key features and rules related to interfaces:

Declaration:
You declare an interface using the interface keyword. An interface can contain abstract methods, constants (implicitly public, static, and final), and default methods.

```
interface Drawable {
    void draw(); // Abstract method
    int DEFAULT_COLOR = 0xFF0000; // Constant
    default void describe() {
        System.out.println("This is a drawable object."); // Default method
    }
}
```

Implementing Interfaces:
A class implements an interface using the implements keyword. The class must provide concrete implementations for all the abstract methods declared in the interface.

```
class Circle implements Drawable {
    @Override
    public void draw() {
        // Provide implementation for the draw method specific to Circle
    }
}
```

Multiple Inheritance:
Unlike classes, a class can implement multiple interfaces.
This allows a class to inherit the abstract methods and constants from multiple sources.

```
class Square implements Drawable, Shape {
    @Override
    public void draw() {
        // Provide implementation for the draw method specific to Square
    }
```

```
    @Override
    public double area() {
        // Provide implementation for the area method specific to Square
    }
}
```

Default Methods:

Java 8 introduced the concept of default methods in interfaces.
Default methods provide a default implementation for a method in the interface.
Classes that implement the interface can use the default implementation or override it.

```
interface Loggable {
    default void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

```
class Logger implements Loggable { // No need to override the default log method
}
```

Static Methods:

Java 8 also introduced static methods in interfaces.
Static methods are associated with the interface and can be called using the interface name.

```
interface MathOperations {
    static int add(int a, int b) {
        return a + b;
    }
}
```

```
class Calculator implements MathOperations {
    // Use the static add method
    int result = MathOperations.add(3, 5);
}
```

Note:

- Interface methods are by default public and abstract like if we will write void run(); if we will not defined it will default public and abstract
- Interface variables are by default public+static+final like int a; is default pbulic + static+final without adding this keyword
- Interface all method must be overriden inside the implemnting classes else that subclass also made abstract class
- Interface nothing but deals (contract) between client and developer

In summary, interfaces provide a way to achieve abstraction and support multiple inheritance. They are used to define a contract that classes must adhere to, and they are widely used in Java programming, especially in scenarios where multiple classes need to share a common set of methods. Interfaces are a fundamental part of Java's support for object-oriented programming and design.

Abstract Classes: an abstract class is a class that cannot be instantiated (we can create object) on its own and may contain abstract methods. Abstract classes are typically used as base classes in inheritance hierarchies. They allow you to define a common interface for a group of related classes while leaving some of the implementation details to be provided by the concrete subclasses.

Here are some key characteristics and rules related to abstract classes:

Declaration: You declare an abstract class using the **abstract** keyword. Abstract classes can have both abstract methods (methods without a body) and concrete methods (methods with an implementation).

Abstract classes can have instance variables (fields), constructors, and other features of a regular class.

```
abstract class Animal {
    String name;
    abstract void makeSound(); // Abstract method
}
```

Abstract Methods: Abstract methods are methods without a body (no implementation). Any class that extends an abstract class with abstract methods must provide concrete implementations for those methods.

```
class Dog extends Animal {
    void makeSound() {
        System.out.println("Woof! Woof!");
    }
}
```

Instantiation:

You cannot create an instance of an abstract class using the new keyword.
Abstract classes can be used as references, and you can create objects of concrete subclasses.

```
Animal myDog = new Dog(); // OK
// Animal myAnimal = new Animal(); // Not allowed (abstract class cannot be instantiated)
```

Inheritance: Abstract classes support the concept of inheritance. Subclasses of an abstract class inherit both the abstract and concrete methods of the superclass. It is required to implement all method ob abstract methods in subclass if we will partially implemented then we have to add abstract keyword to subclass

```
class Cat extends Animal {
    void makeSound() {
        System.out.println("Meow!");
    }
}
```

Constructors: Abstract classes can have constructors, and they are called when an instance of a concrete subclass is created.

```
abstract class Shape {
    int sides;
    Shape(int sides) {
        this.sides = sides;
    }

    abstract void draw();
}

class Circle extends Shape {
    Circle() {
        super(0); // Call the constructor of the abstract class
    }
    void draw() {
        // Provide implementation for the draw method specific to Circle
    }
}
```

In summary, abstract classes provide a way to define common behavior for a group of related classes while allowing for variations in the implementation details. Abstract classes can have a mix of abstract and concrete methods, and they are a key feature in building class hierarchies with shared functionality.

OOPs Backlog - Static and Final - 23 Nov

- **Static keyword**
- **Final keyword**
- **IDE Setup in mac**

Static keyword -> In Java, the static keyword is used to define members (fields, methods, and nested classes) that belong to the class itself rather than to instances of the class. Here are some common uses of the static keyword in Java:

Static Variables (Class Variables):
A static variable, also known as a **class variable**, is a variable that belongs to the class rather than to instances of the class. It is shared among all instances of the class.

```
public class Example {
    // Static variable
    public static int staticVariable = 0;
}
```

Accessing the static variable: Example.staticVariable

Static Methods:
A static method is a method that belongs to the class rather than to instances of the class. It can be called on the class itself, without creating an instance of the class.

```
public class Example {
    // Static method
    public static void staticMethod() {
        // Code for the static method
    }
}
```

Calling the static method: Example.staticMethod()

Static Block:
A static block is a block of code enclosed in curly braces {} and preceded by the static keyword. It is executed when the class is loaded into the Java Virtual Machine (JVM).

```
public class Example {
    // Static block
    static {
        // Code in the static block
    }
}
```

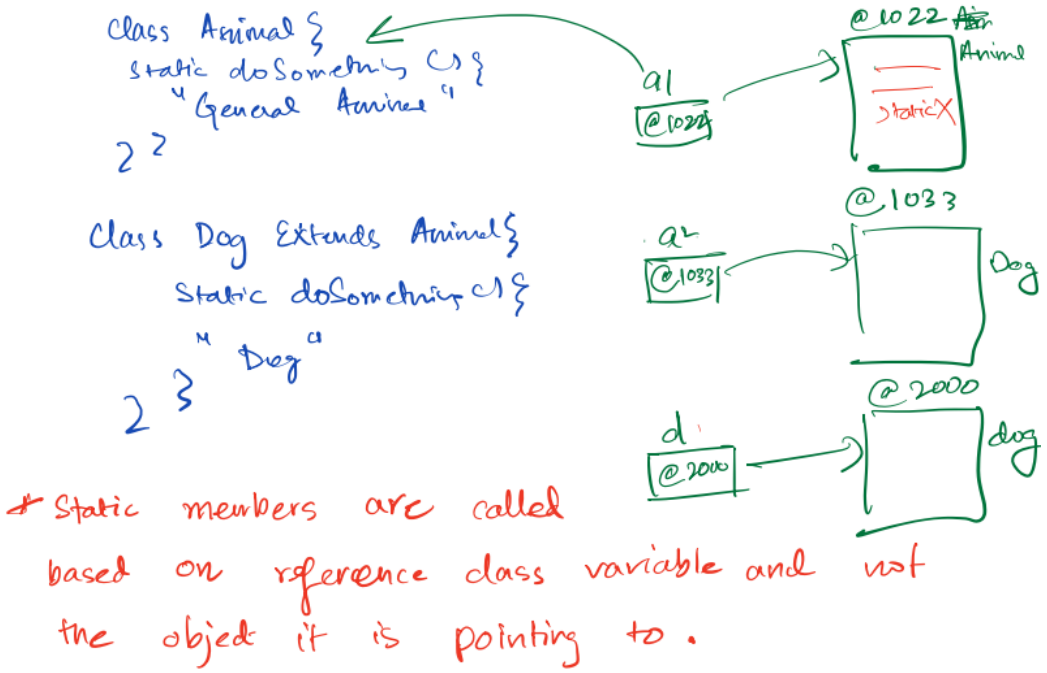
The static block is useful for initializing static variables or performing one-time initialization tasks.

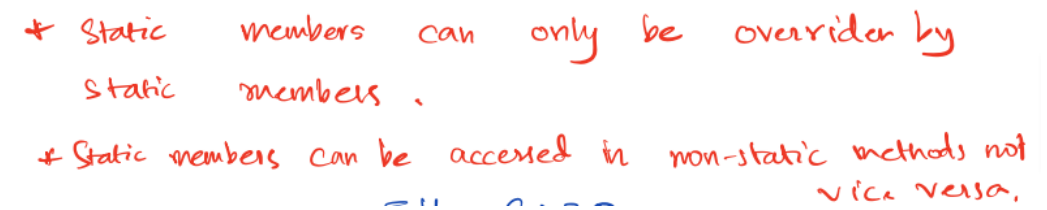
Static Nested Classes:

A static nested class is a nested class that is marked as static. It is associated with the outer class and can be instantiated without creating an instance of the outer class.

```
public class Outer {
    // Static nested class
    static class Nested {
        // Code for the static nested class
    }
}
```

Creating an instance of the static nested class:
Outer.Nested nestedInstance = new Outer.Nested();





Note ->

- It's important to note that static members are associated with the class itself and not with instances of the class. They are loaded into memory when the class is loaded, and you can access them without creating an instance of the class. It is recommended to do not access static variable using instance of object. It will create confusion between static and nonstatic members.
- Main class has to be static because by adding static it will be runnable without create a obj of class.
- Static members called based on reference not by instance like below

Final -> final keyword is used to indicate that a variable, method, or class cannot be changed or overridden.

Final Variables:
When applied to a variable, the final keyword means that the variable's value cannot be changed once it has been assigned.

```
public class Example {
    // Final variable
    final int constantValue = 10;
}
```
Once constantValue is assigned a value (e.g., 10), it cannot be reassigned.

Final Methods:
When applied to a method, the final keyword indicates that the method cannot be overridden by subclasses. Subclasses can still inherit the method, but they cannot provide a different implementation.

```
public class Parent {
    // Final method
    public final void finalMethod() {
        // Code for the final method
    }
}

public class Child extends Parent {
    // Attempting to override the final method will result in a compilation error
}
```

Final Classes:
When applied to a class, the final keyword indicates that the class cannot be subclassed. In other words, no other class can extend a final class.

```
public final class FinalClass {
    // Code for the final class
}

// This will result in a compilation error
public class Subclass extends FinalClass {
    // Code for the subclass
}
```
The idea behind making a class final is to prevent any further extension, ensuring that the class's behavior is not altered.

Final Arguments:

When applied to method parameters, the final keyword indicates that the parameter cannot be modified within the method.

```
public class Example {
    public void exampleMethod(final int parameter) {
        // The 'parameter' variable is effectively final and cannot be modified
    }
}
```

This is more of a documentation feature, signaling to other developers that the method won't modify the value of the parameter.

Using the final keyword can provide benefits in terms of code safety, maintainability, and performance optimizations. It helps in expressing your intent clearly and can sometimes enable the compiler to perform certain optimizations.

Concurrency-1 and 2: Intro Processes and Threads - Dec9 & Dec 12

- **Processes**
- **How program gets executed on your computer ?**
- **How data related to a process is stored Process Control Block (PCB)**
- **Problems with multiple processes**
- **Threads; Introduction**
- **Breaking some myths; CPU doesn't execute a process :(**
- **The new structure of PCB after threads**
- **Single core vs Multicore**
- **Consurrency vs Parallelism**
- **Execution of a thread**

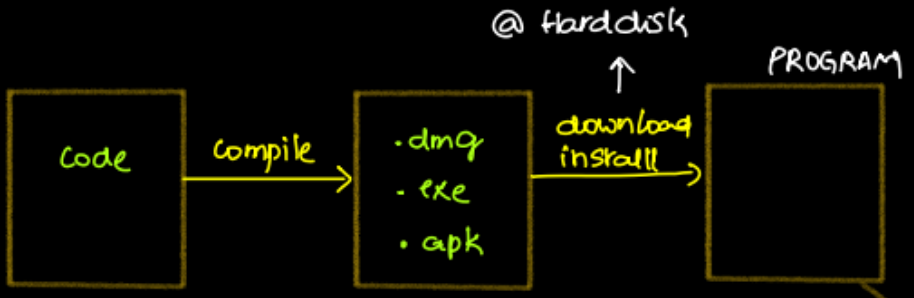
Processes:

- A process is an independent program in execution. It has its own memory space, resources, and a set of data structures to manage its execution.
- Processes are isolated from each other, meaning one process cannot directly access the memory or resources of another process. This isolation provides security and stability to the system.
- Processes communicate with each other through inter-process communication (IPC) mechanisms, such as message passing or shared memory.
- Creating a new process typically involves duplicating the existing process (parent process) to create a new, independent process (child process).

Note: When ever code in installed and keep it is called programme and when it loads in ram it called process.

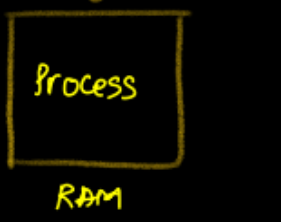
How a program executes on Computer

How an appⁿ can come into your device? (Laptop/mobile).



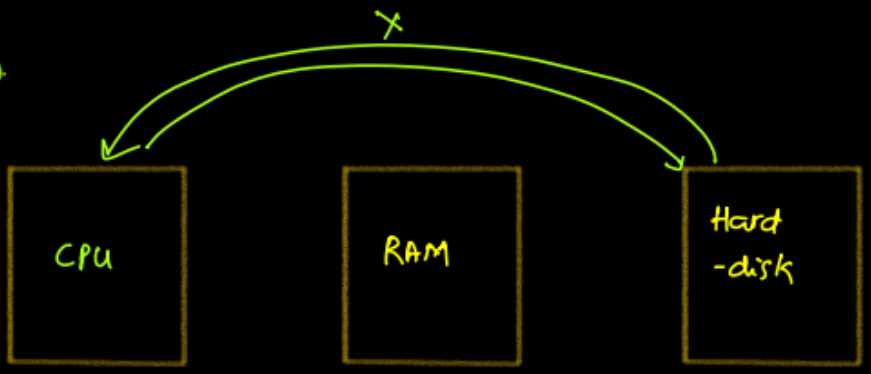
Program: An applⁿ sitting on my computer waiting to be run.
[Hard-disk].

Process: A program that is currently running
[RAM]



Why don't we run a program from hard-disk, why load into RAM?

Comp. arch.

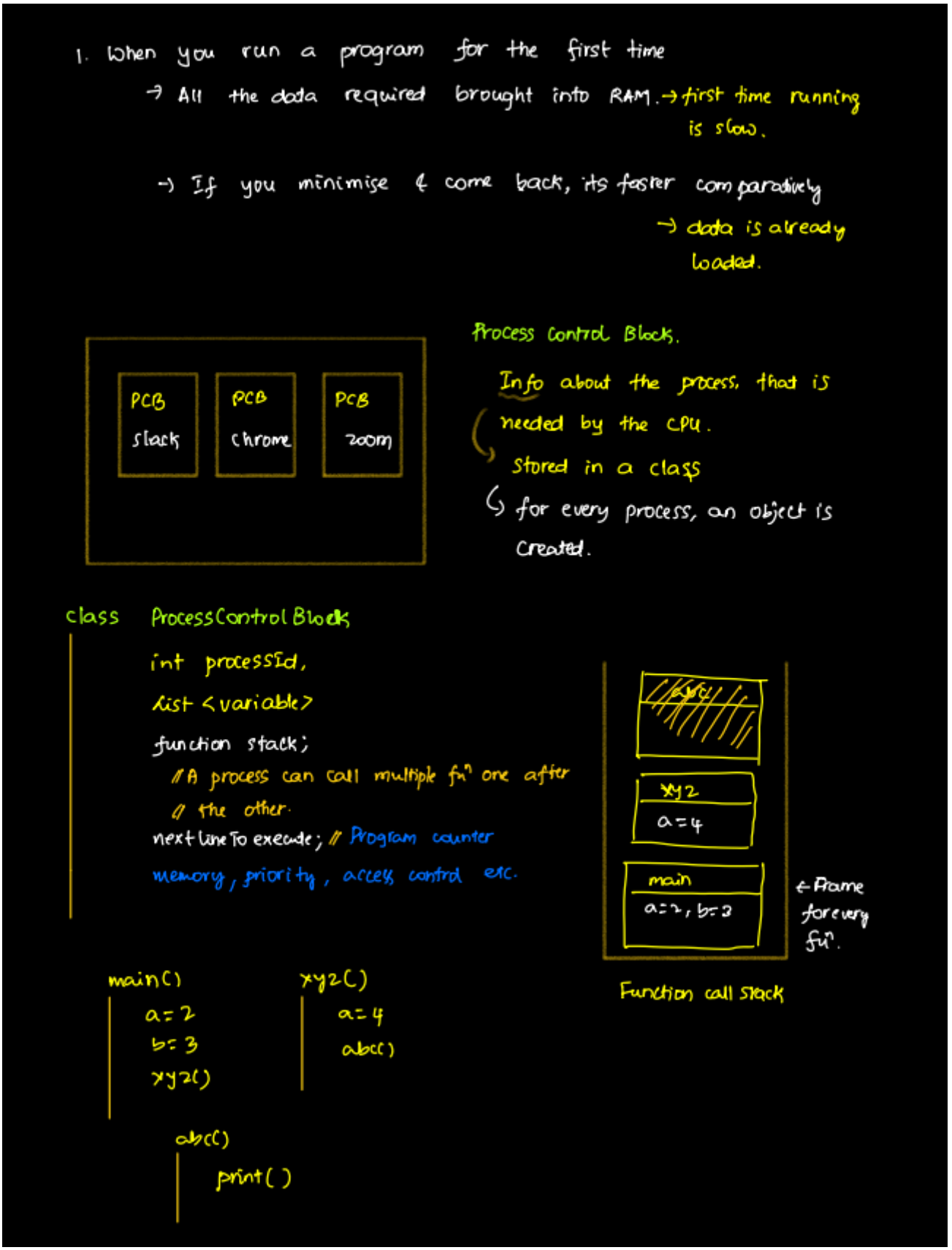


To read 1MB sequentially from RAM ⇒ 250 μs.

To read 1MB sequentially from Hard-disk ⇒ 20,000 μs

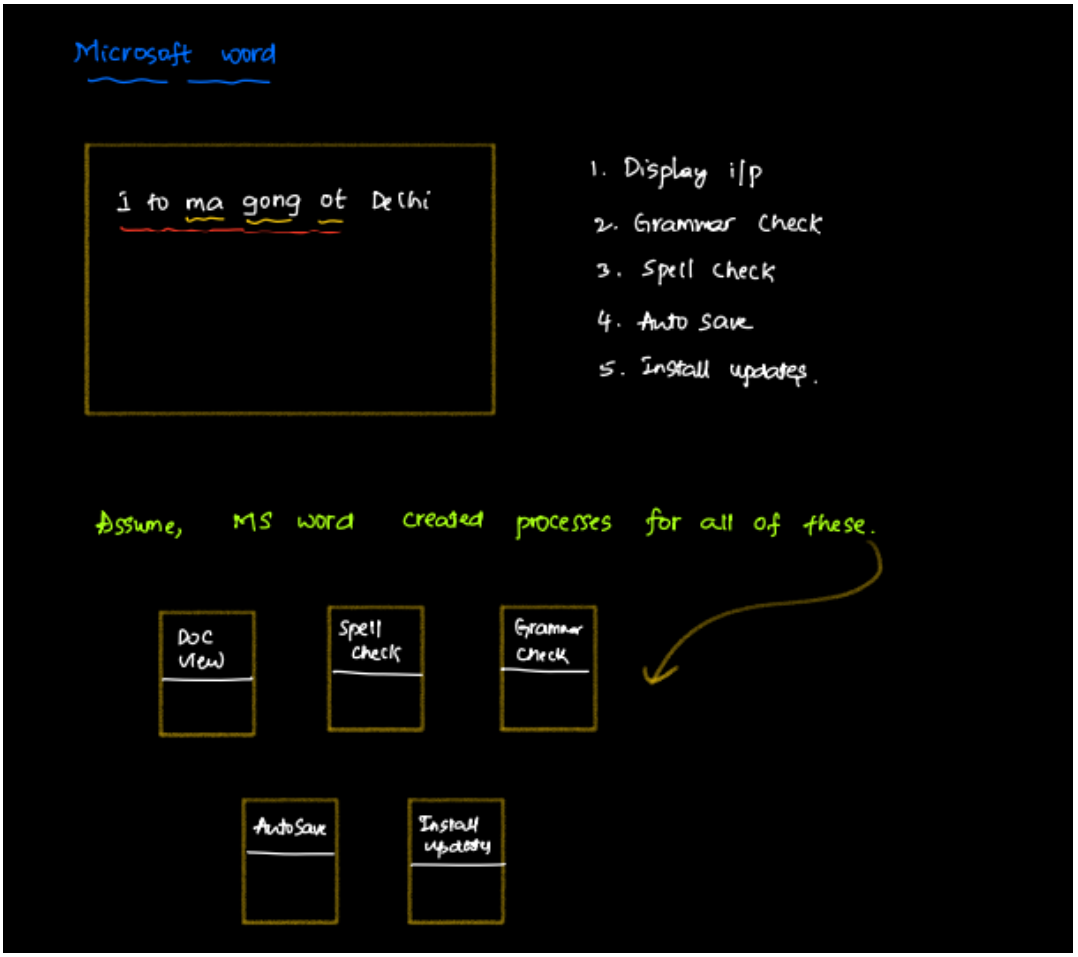
2.24 GHz.
$= 2.24 \times 10^9$ opⁿ/sec.

How data related to a process is stored Process Control Block (PCB)

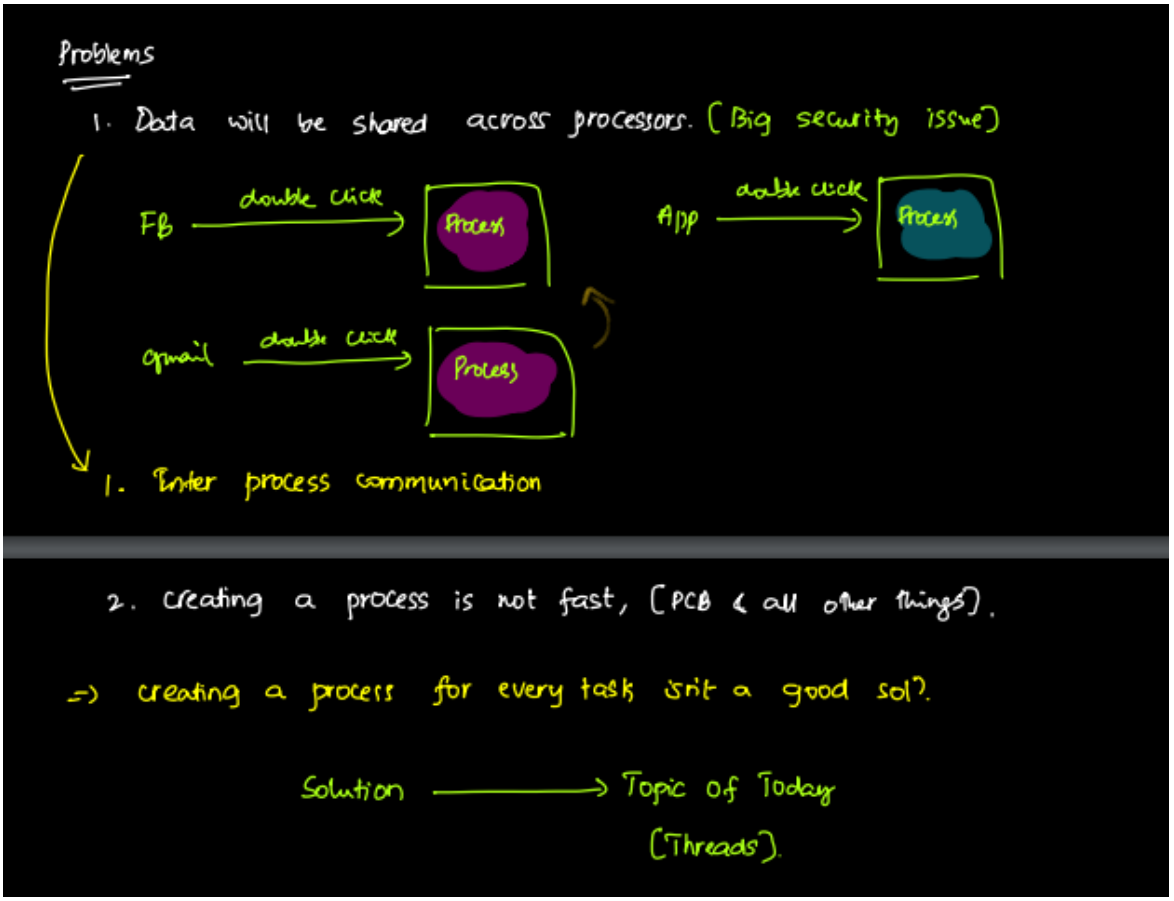


Microsoft word, what happens behind the scenes without thread with process only ?

Microsoft Word performs multiple tasks simultaneously. The diagram below illustrates what happens behind the scenes. If we opt for a process instead of a thread, it will reveal the challenges that may arise when dealing with multiprocessing rather than multithreading.

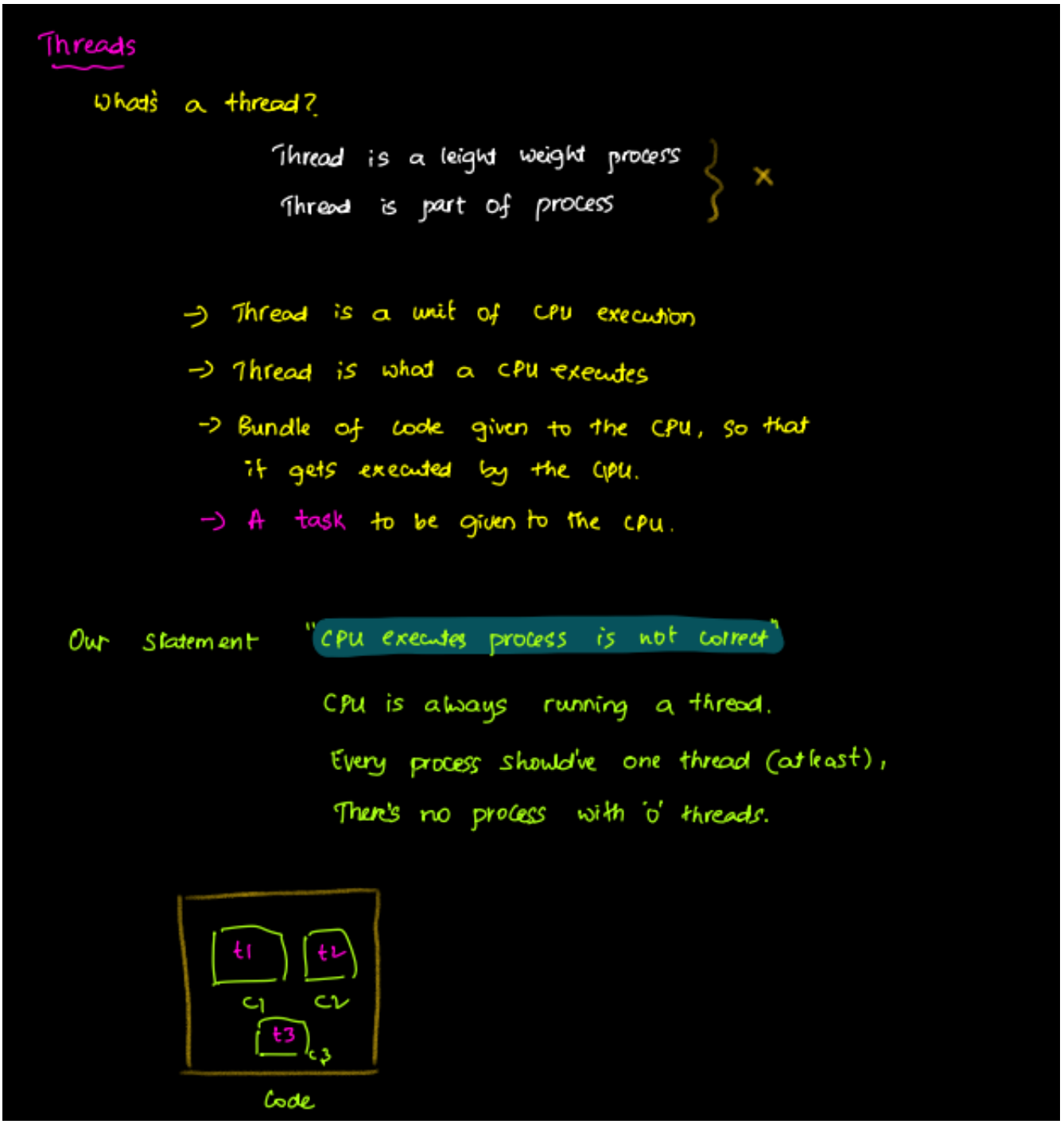


Problems with multiple processes

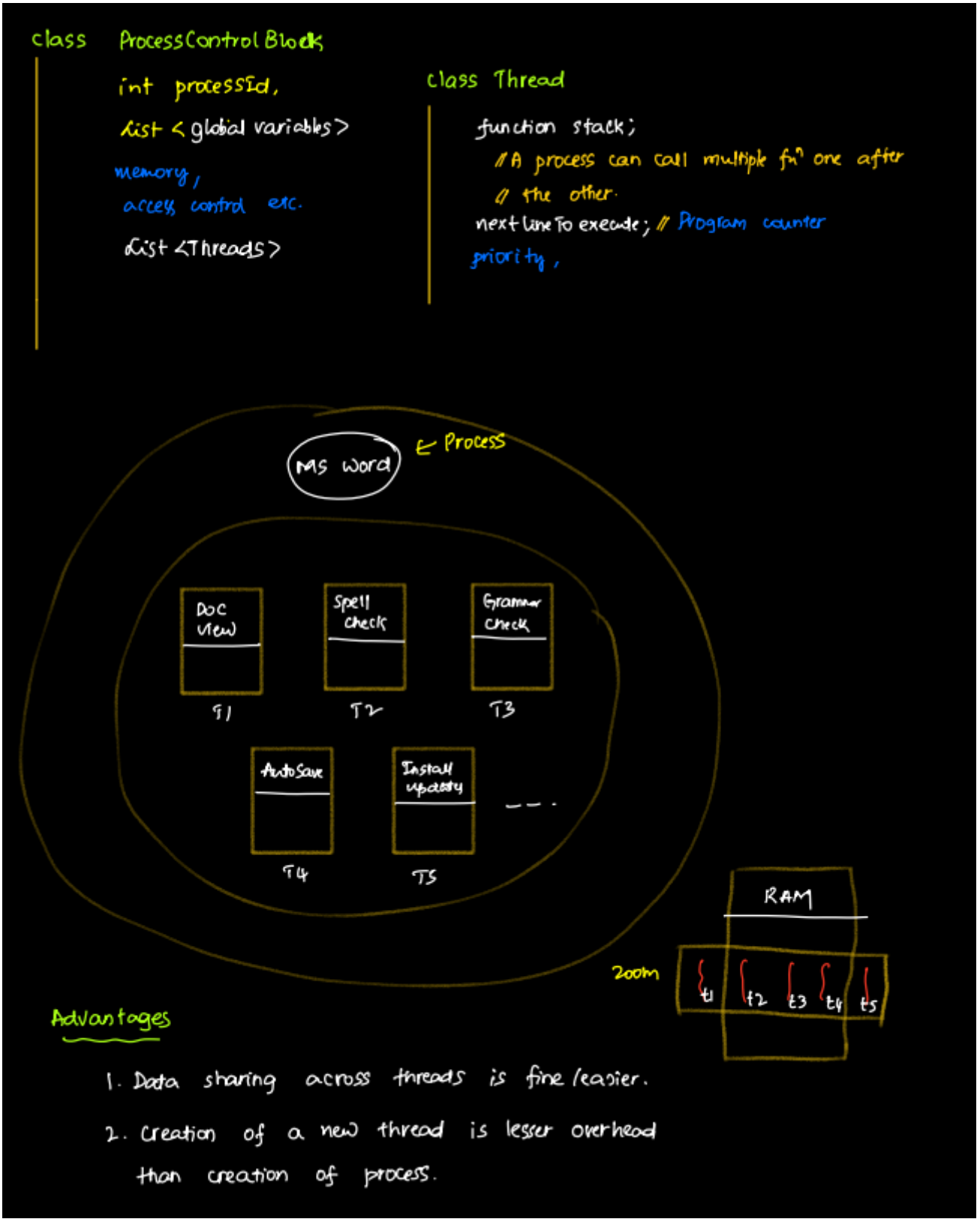


Threads:

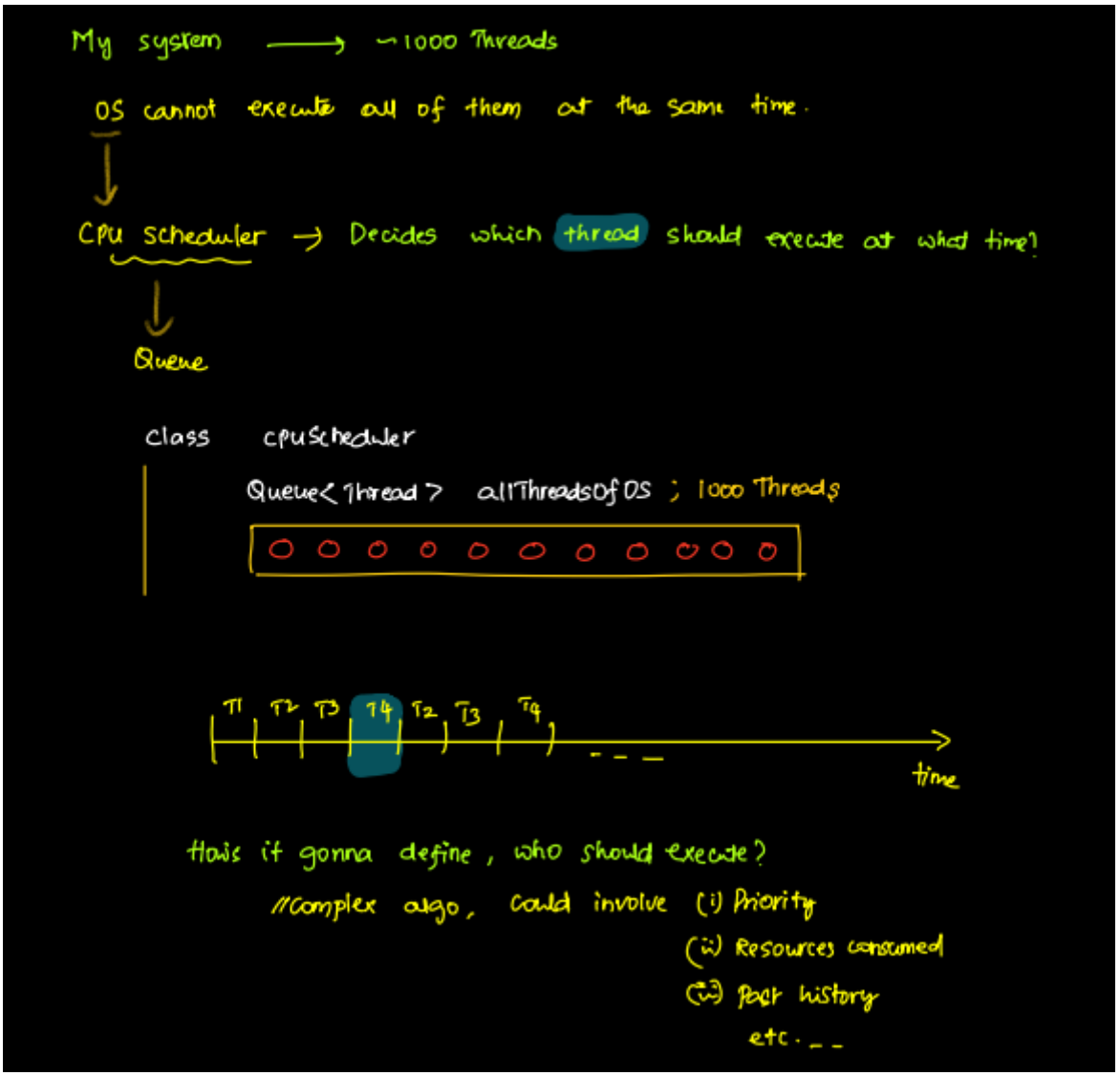
- A thread is the smallest unit of execution within a process. Multiple threads within a process share the same memory space and resources.
- Threads within the same process can communicate with each other more easily than processes since they share the same memory space.
- Threads can be considered as lightweight processes because they require fewer resources to create and manage compared to processes.
- Threads within the same process can run concurrently, allowing for parallel execution of tasks.



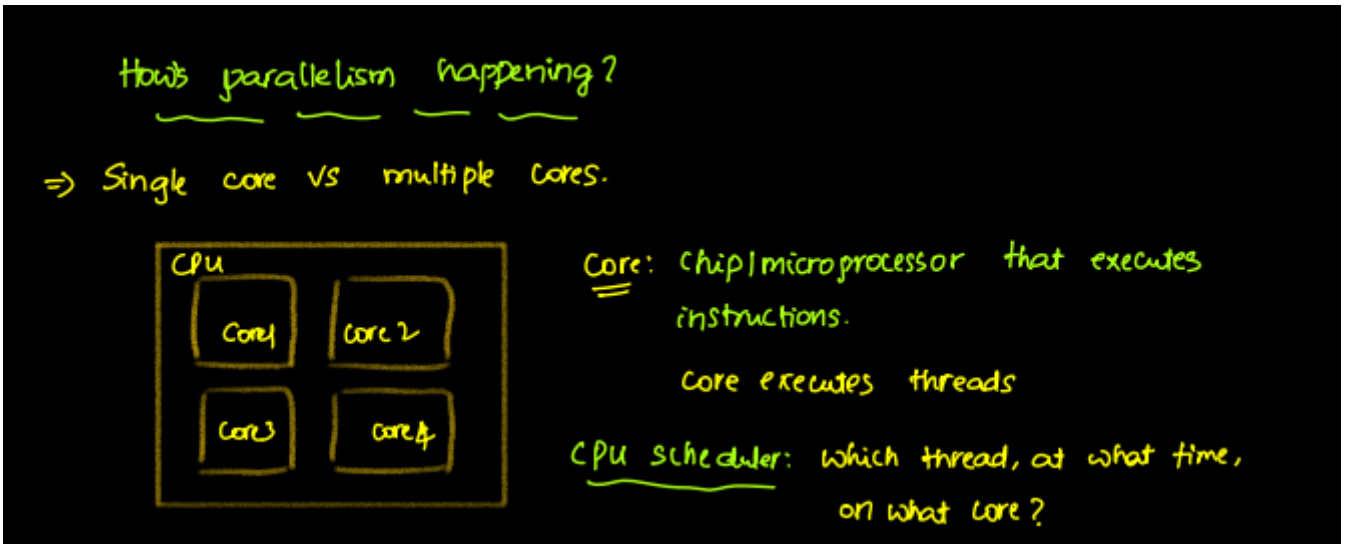
The new structure of PCB after threads & Advantages of multiple threads over multiple process



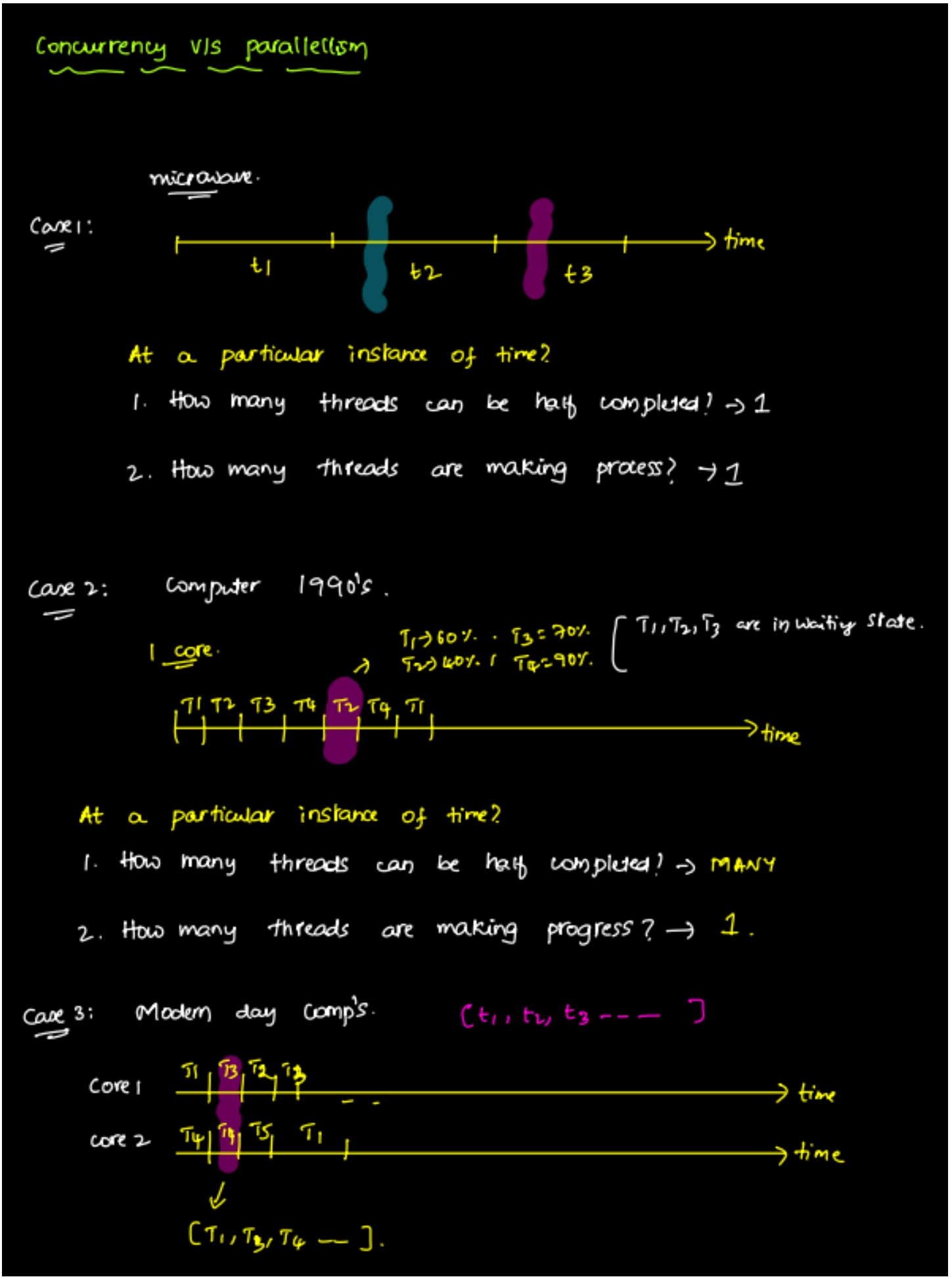
CPU Scheduler



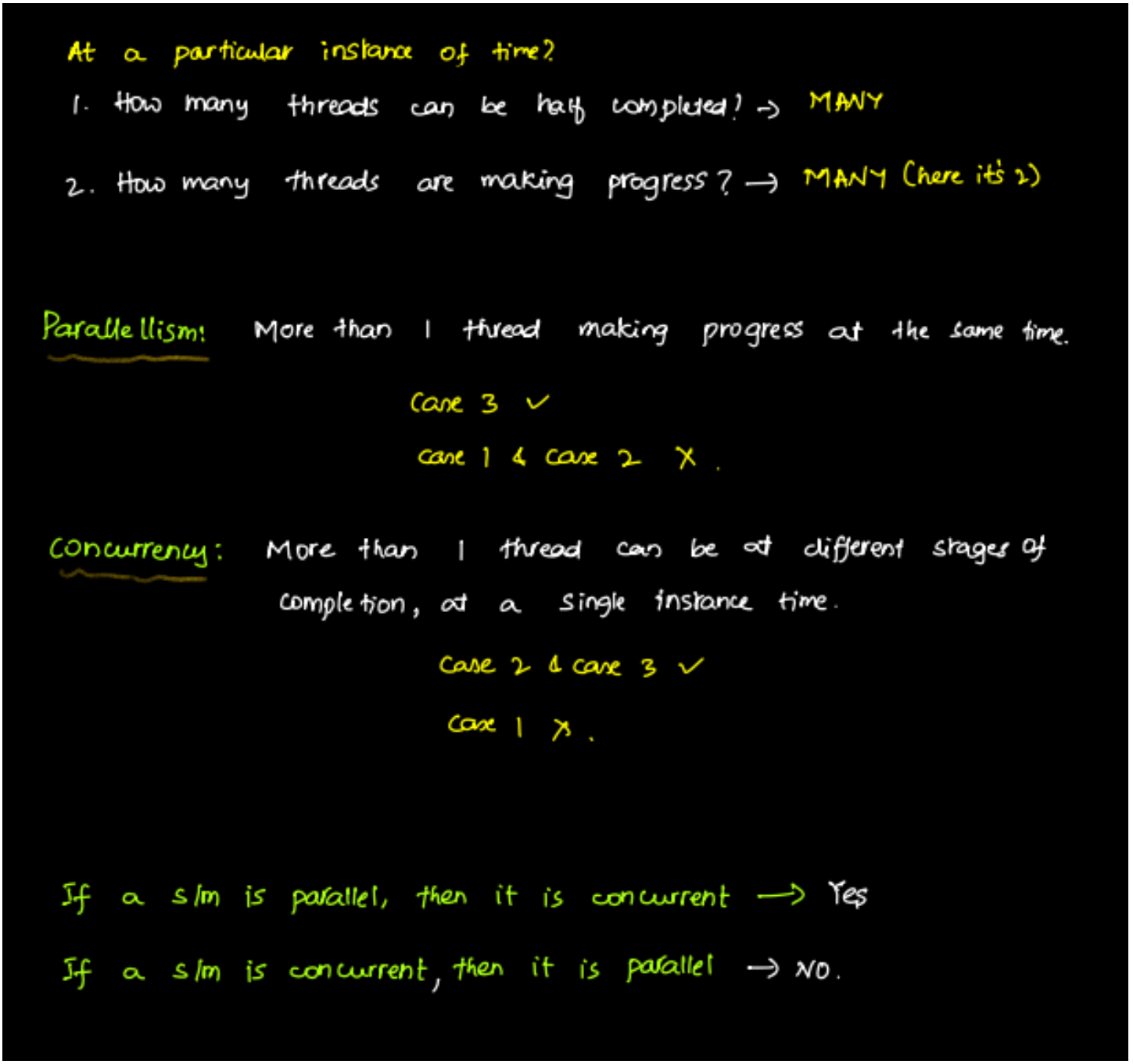
Single vs multiple core



Concurrency vs Parallelism in different types of CPUs

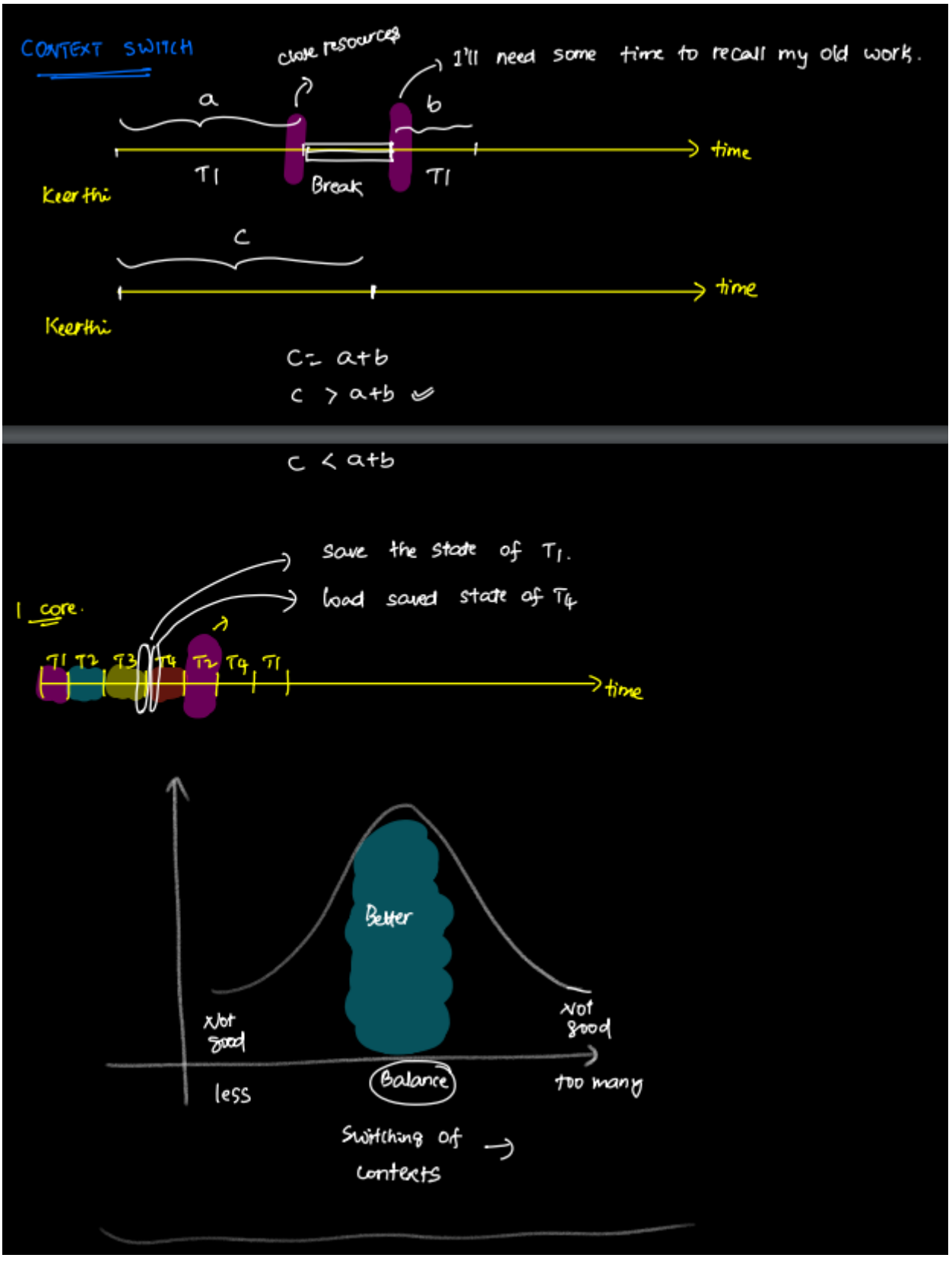


Concurrency, Parallelism explain with examples and difference



Context Switching - "context switching" refers to the process where the execution context of a currently running thread is temporarily suspended to allow another thread to run. This switch occurs at the operating system level and is managed by the Java Virtual Machine (JVM) and the underlying operating system.

Context switching is an essential aspect of multithreading as it allows the system to efficiently utilize resources and provide the illusion of concurrent execution. However, excessive context switching can incur overhead, impacting performance.



How to think in terms of "TASKS"

1. **How to write multithreaded code, reasons for all the syntax.**
2. **Reason why we need "implement Runnable"**
3. **What run(), start() method does ?**

How to design multithreaded program

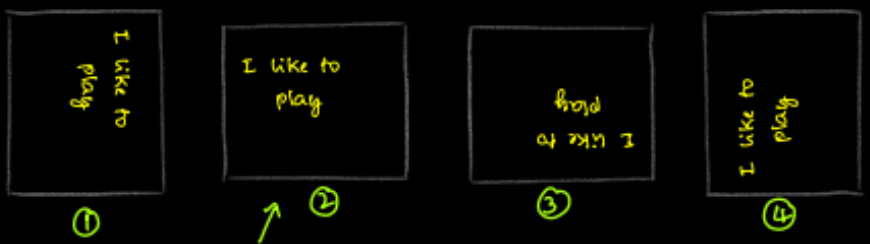
Don't think in terms of what threads to create.

Think of the tasks that you can potentially do in parallel.

Identify what parts of an app^{n}. can be done in parallel.

Problem statement

1. You're given an image which can be rotated by 90°, 180°, 270° or not rotated at all (0°), You've a function that can convert the image to text if the text is oriented properly.



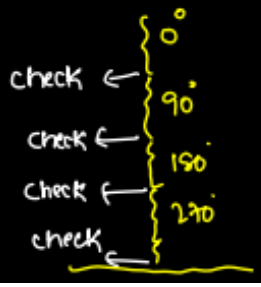
① ② ③ ④

Model can only read this. But the images can look like ①, ③ & ④ as well

Soln: Run the algorithm for 0°, 90°, 180°, 270°, If you get the o/p return it.

Do you see any tasks that can be done in parallel?

↳ Rotated image -> Check



instead

Step1 : Define the task.

For each of the task that is created [that can be potentially done in parallel], create a class for that.

HelloWorld Printer

Print "Hello, World" from a separate thread.

ParallelTask

=> Create a class

Whatever tasks needs to be achieved should be inside some method.

What should be the name?

```
class HelloWorldPrinter
    void print()
        print("Hello, World")
    void doSomething()
```

For the parallel task, create a thread.

```
HelloWorldPrinter hp = new HelloWorldPrinter();
Thread t = new Thread(hp)
t.start()
```

Pass an instance of the task, which you intend to do in parallel.

// there, the are two methods, which method should be executed in parallel? : Ambiguity for the compiler.

```
class HelloWorldPrinter implements Runnable
    void run()
        print("Hello, World")
    void doSomething()
```

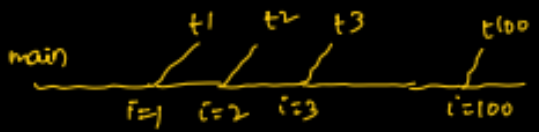
```
interface Runnable
    void run();
```

✗ `t.run()` → will only call run() method as a normal fn call.
There's no thread being created here.

✓ `t.start()`
- It'll create the thread t
- It'll provide code inside `run() method` to CPU, asking for it to be `executed by thread 't'`.
- once executed, it'll also delete the thread.

Q: Print `1-100`, each number should be printed from different thread.


```
class PrintNumber implements Runnable
    int no;
    PrintNumber(int num)
        no = num
    Void run()
        print(no + "Printed by" + Thread.currentThread.getName());
```




```
main()
    for(i=1; i≤100; i++)
        PrintNumber printer : new PrintNumber(i)
        Thread t = new Thread(printer)
        t.start()
```

Concurrency-3: Executors and Callables - 14 Dec

- **Executors and Thread Pools**
- **Callables**

Executorss and Thread pools -> In Java, associating one thread with one operating system task can be expensive. To address this, a more cost-effective solution is to implement a thread pool. A thread pool is a collection of threads, where, for instance, 10 threads are created in the pool, and then 1000 tasks are assigned to this pool.

To establish a fixed thread pool, the Executors class provides a static method: Executors.newFixedThreadPool(10). Instead of initiating threads with the start method, tasks are executed within a loop using the execute method of the ExecutorService.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class NumberPrinter {
        int number;
        NumberPrinter void(num){
                number = num;
        }

        void printNumber(){
                System.out.println(number);
        }
}

public class ThreadPoolExample {
   public static void main(String[] args) {
      // Create a fixed thread pool with 10 threads
      ExecutorService service = Executors.newFixedThreadPool(10);

      // Submit tasks for execution
      for (int i = 0; i < 100; i++) {
        NumberPrinter n = new NumberPrinter(i);
          service.execute(n);
      }

      // Output the current thread's name
      System.out.println("Thread Name: " + Thread.currentThread().getName());

      // Shut down the thread pool
      service.shutdown();
   }}
```

Here, tasks are submitted to the service, and internally, the thread pool executor utilizes a blocking queue to store and manage tasks. The ten threads concurrently fetch and execute tasks from the queue. This ensures thread safety for concurrent operations.

For system-intensive tasks where utilizing all available cores is desirable, the core count can be determined using Runtime.getRuntime().availableProcessors() to dynamically set the thread pool size.

```
int coreCount = Runtime.getRuntime().availableProcessors();
ExecutorService service = Executors.newFixedThreadPool(coreCount);
```

However, for IO-intensive tasks such as database operations or network calls, a fixed pool size based on the number of cores may not be optimal. In such cases, a larger thread pool size may be necessary for efficient execution.

Task Type	Ideal pool size	Considerations
CPU intensive	CPU Core count	How many other applications (or other executors/threads) are running on the same CPU.
IO intensive	High	Exact number will depend on rate of task submissions and average task wait time. Too many threads will increase memory consumption too.

Thread Pool ->
Java provides four types of thread pools:

FixedThreadPool:
In this pool, a fixed number of threads are specified by the user. Tasks are submitted to the pool, and each thread in the pool is responsible for executing a task. It is suitable when a predetermined number of threads can efficiently handle the workload.

To create a fixed thread pool, use:

```
ExecutorService service = Executors.newFixedThreadPool(n);
```

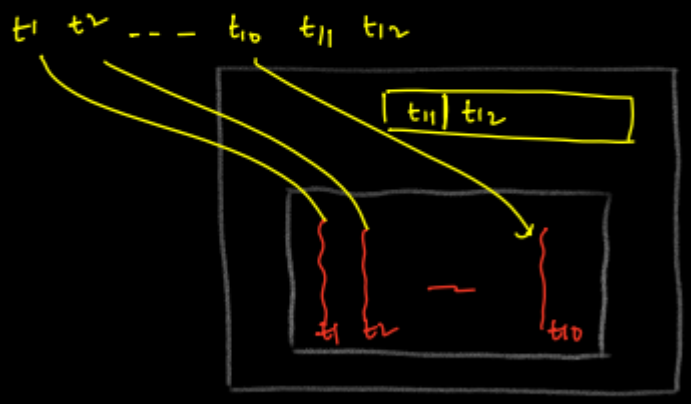
CachedThreadPool:
Unlike FixedThreadPool, there is no fixed number of threads in this pool. It employs a synchronous queue, replacing the need for a task queue. When a task is submitted, the pool looks for an existing free thread to execute the task. If no free thread is available, a new thread is created. Threads that remain idle for 60 seconds are terminated.
Create a CachedThreadPool using:

```
ExecutorService service = Executors.newCachedThreadPool();
```

Types of thread pools

1. Fixed Thread Pool.

```
ExecutorService es = Executors.newFixedThreadPool(10);
```



Note: Fixed Thread Pools are having of limit on max. no. of threads that can be created.
However, They'll reach the max no. only when it is required.

Ex: for 2 tasks, at the max 2 threads only are created.

2. Cached Thread Pool

Whenever a new task comes --

1. If there's an empty thread, Re-use that thread. (i.e assigns the tasks to the empty thread).
2. Else, It'll create a new thread and assign the new task came in.

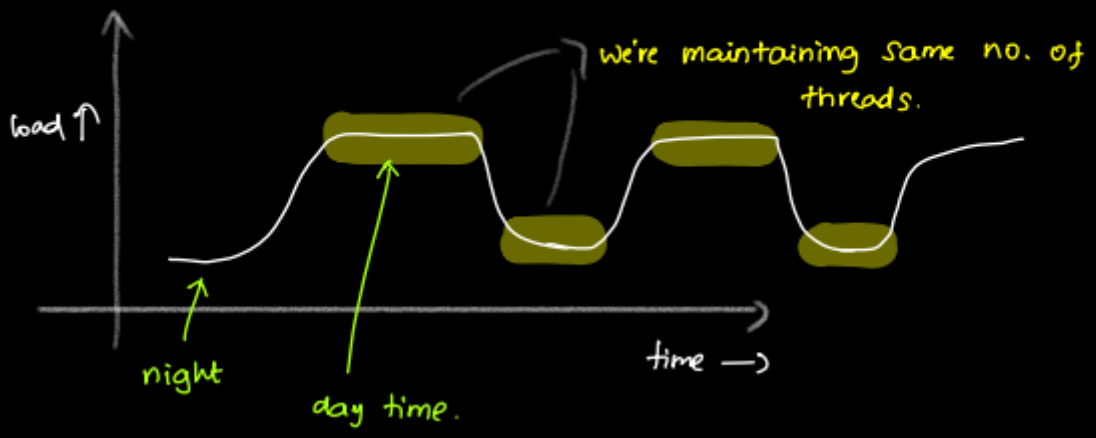
Note: There's no need for queue of tasks here.
CachedThreadPool never deletes a thread that it has created.

Some advantages

1. It'll re-use the threads
2. CPU scheduler [task queue] is free.
3. No overhead of thread deletion.

Some disadvantages

1. For a higher load, you've lot of threads => CPU utilisation is more.
2. Even if load is decreased, we still have many threads. which we created for higher load.



ScheduledThreadPool:
This pool is designed for scheduling tasks with delays. It allows tasks to be executed periodically, with a specified delay between each execution. Ideal for scenarios like performing security checks or login validations at regular intervals.
Use ScheduledExecutorService and its schedule method to create and schedule tasks.
there are 3 type of method

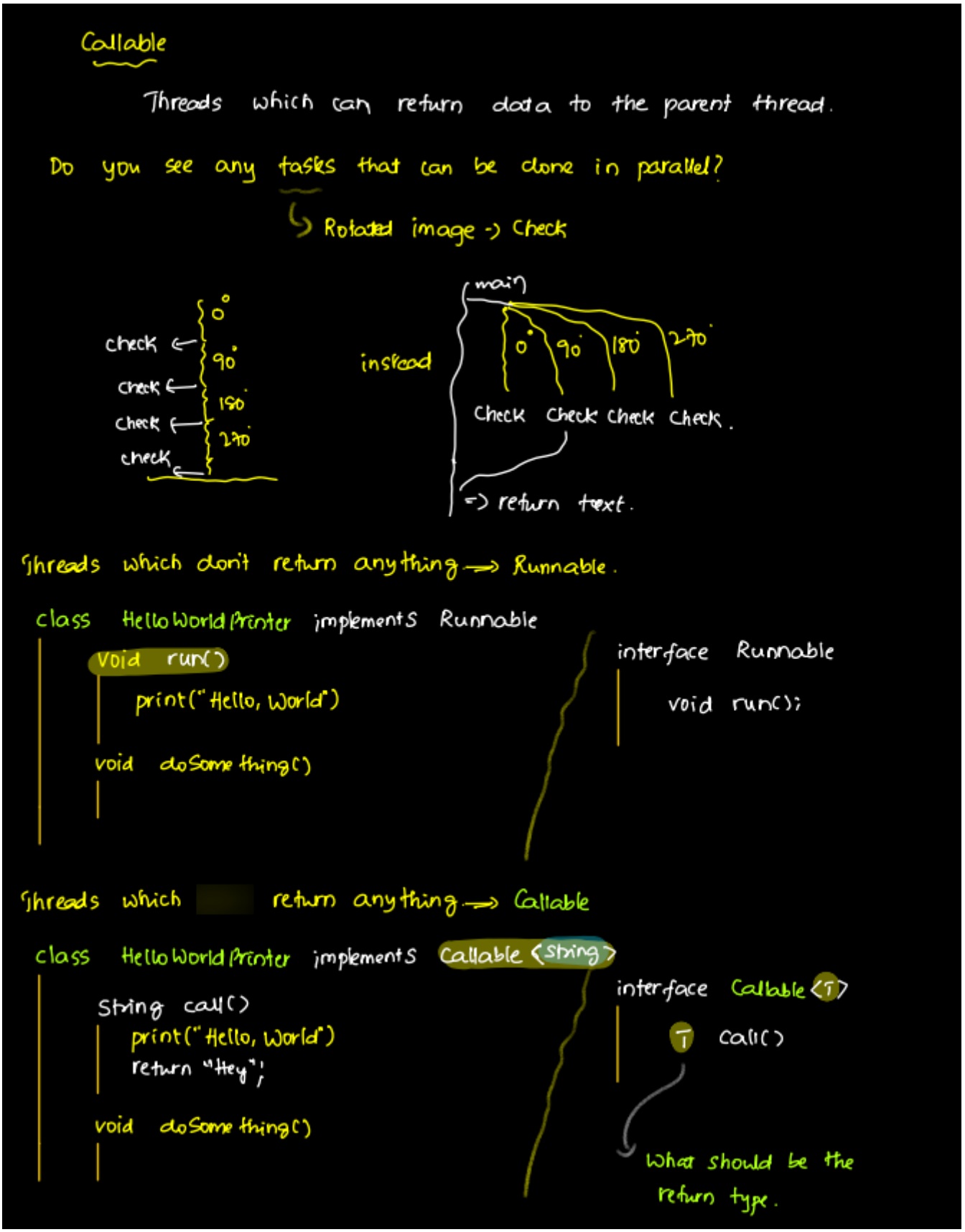
- **service.schedule** - Delayed Execution: In this scenario, we schedule a task to execute after a specified delay.
- **service.scheduleAtFixedRate** - Fixed Interval Execution:This scheduling approach involves executing our task at fixed intervals, with a consistent delay between each execution.
- **service.scheduleWithFixedDelay** - Delayed Execution with Fixed Interval: In this case, we schedule a task to commence after completing its previous execution, introducing a fixed delay between consecutive executions.

```
blic static void main(String[] args) {

  // for scheduling of tasks
  ScheduledExecutorService service = Executors.newScheduledThreadPool( corePoolSize: 10);

  // task to run after 10 second delay
  service.schedule(new Task(),  delay: 10, SECONDS);

  // task to run repeatedly every 10 seconds
  service.scheduleAtFixedRate(new Task(),  initialDelay: 15,  period: 10, SECONDS);

  // task to run repeatedly 10 seconds after previous task completes
  service.scheduleWithFixedDelay(new Task(),  initialDelay: 15,  delay: 10, TimeUnit.SECONDS)


atic class Task implements Runnable {
  public void run() {
      // task that needs to run
      // based on schedule
  }
```

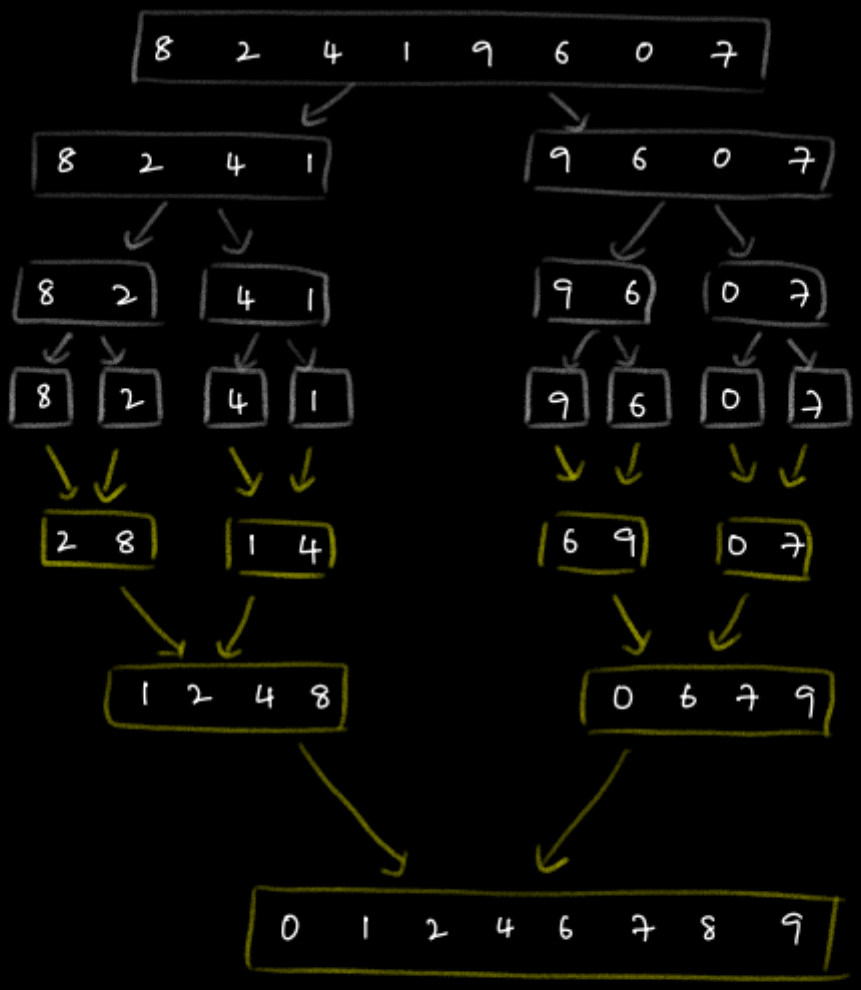
SingleThreadedExecutor:
In this pool, only one thread is used for executing tasks. It ensures that tasks are processed sequentially, one after the other. Useful in scenarios where you want to maintain a specific order of execution or ensure thread safety.

To create a single-threaded executor: ExecutorService service = Executors.newSingleThreadExecutor();

Callable -> Callable is an interface in the java.util.concurrent package that represents a task that can be executed asynchronously and returns a result. It is similar to the Runnable interface, but it can throw checked exceptions and returns a result.



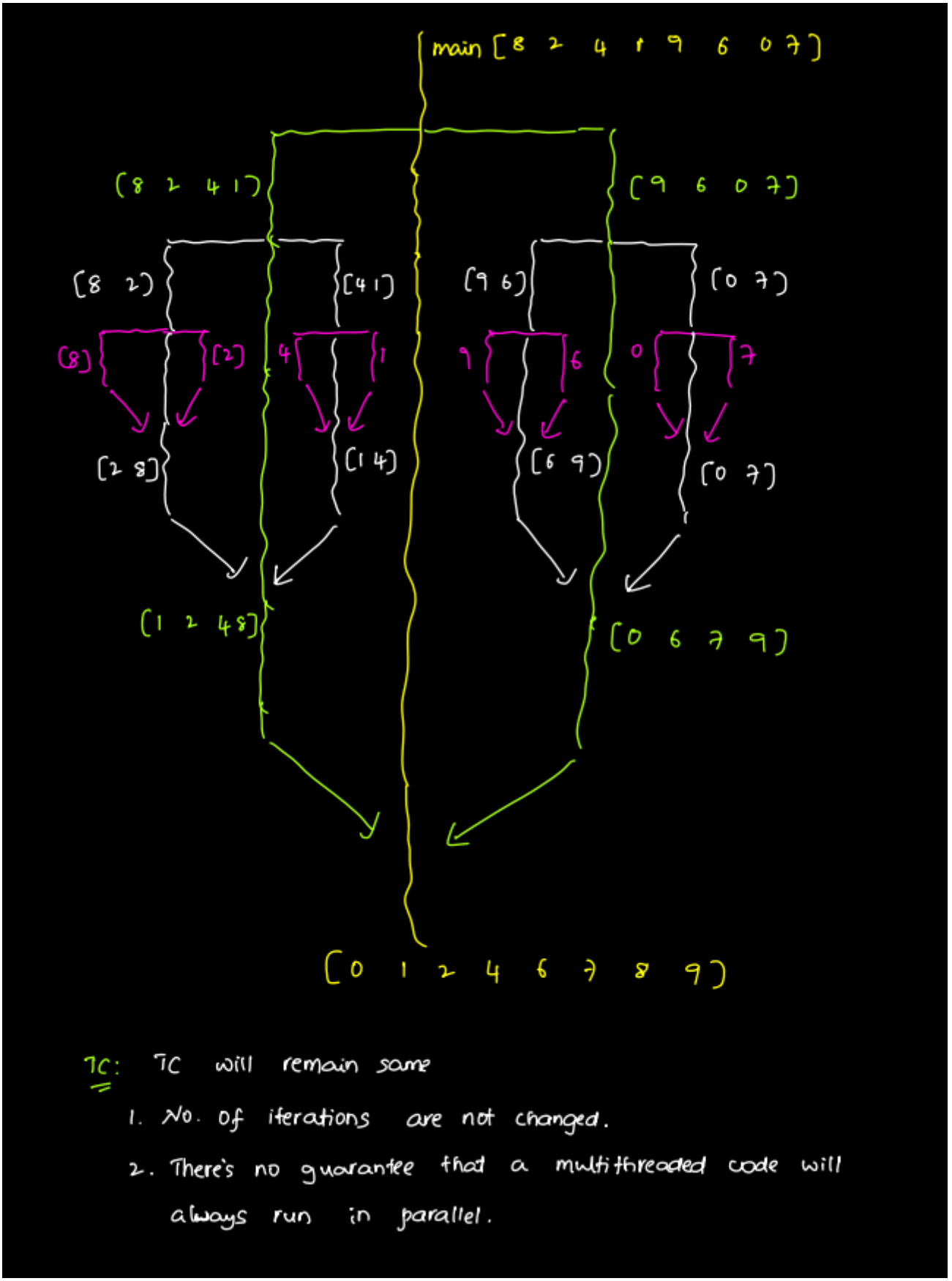
Merge sort



Code

```
Sort (array)
    lefthalf = [ ], right = [ ]
    Sort (left half)
    Sort (right half)
    merge (lefthalf, righthalf)
```

Sorting left & right are not dependent on each other.

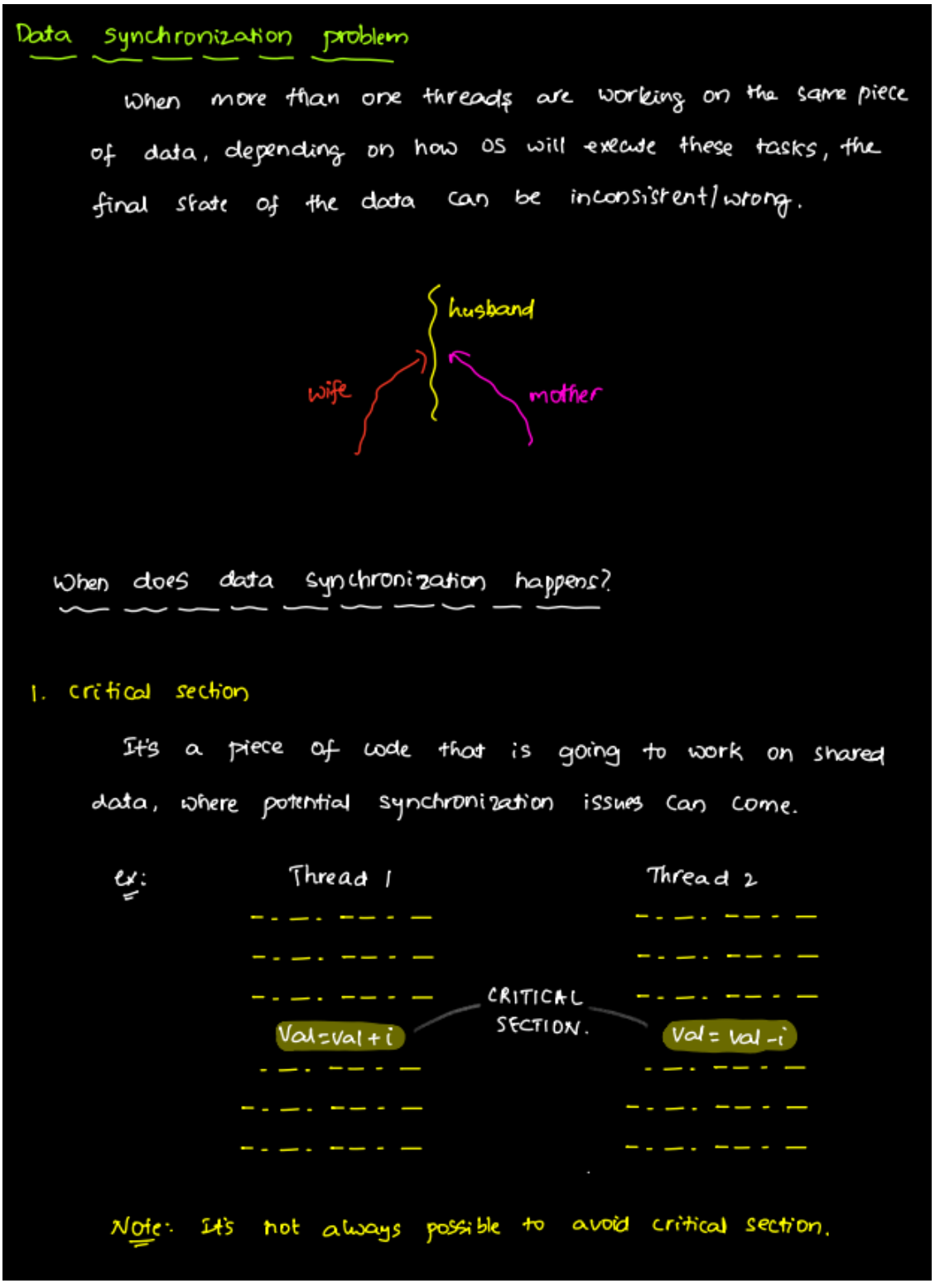


https://github.com/kkumarsg/lld/blob/main/src/main/java/threads/mergesort/Sorter.java

Concurrency-4: Intro to Synchronization - Dec 16

- **When does the synchronization problem happen**
- **What is the Ideal solution for the synchronization problem**
- **Mutex**

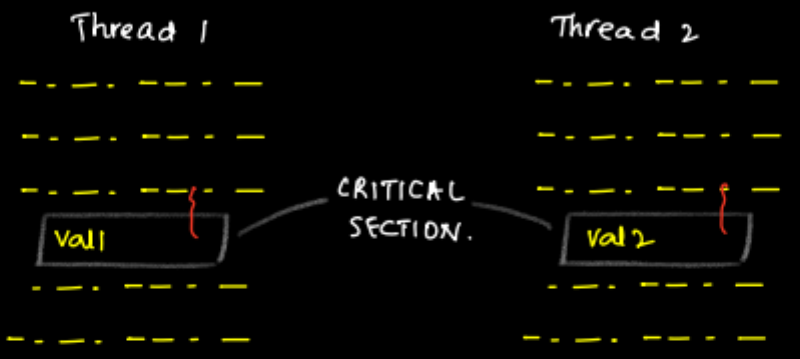
When does the synchronization problem happen -> synchronization problems occur in multi-threaded environments when two or more threads access shared resources concurrently, and at least one of them modifies the shared resource. Without proper synchronization mechanisms, such as the use of the synchronized keyword or other concurrency utilities



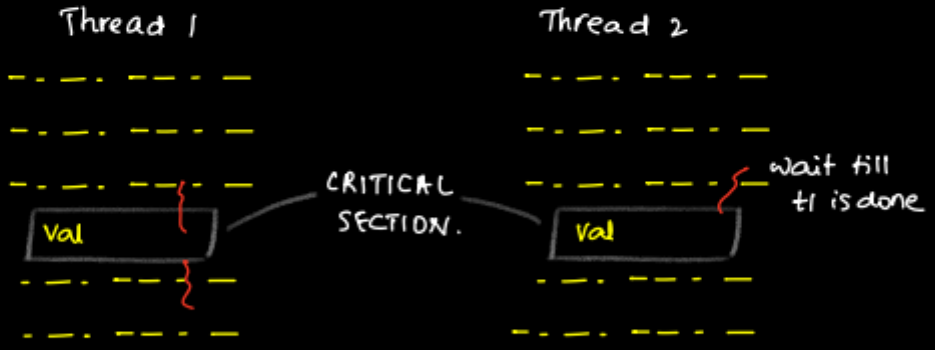
2. Race condition

When more than one threads are entering the critical section of same variable at the same time.

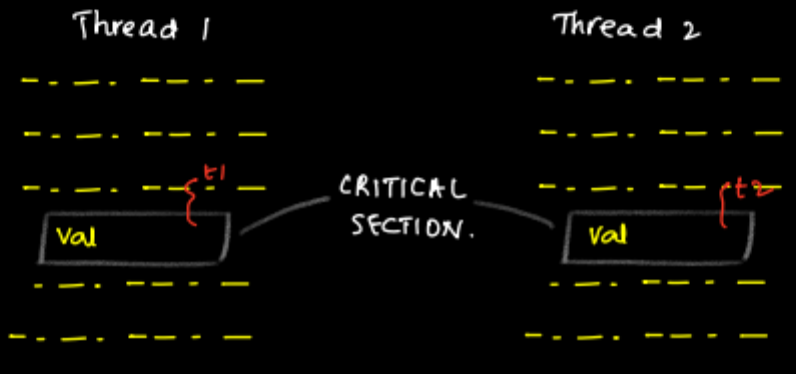
No problem in this.

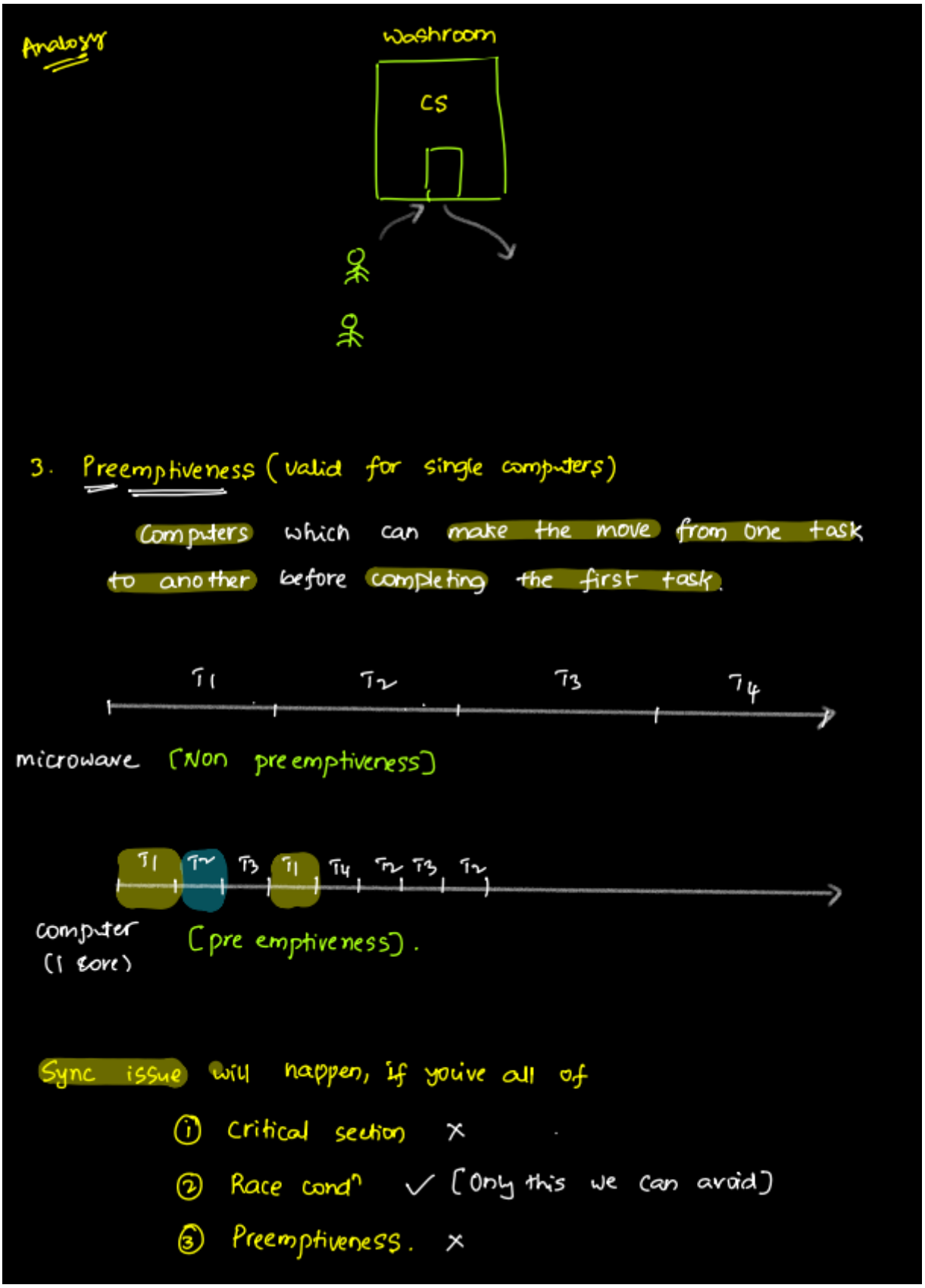


No problem in this.



This is race condition





As above we can not avoid Critical section and Preemptiness but we can avoid Race condition to avoid that condition we use suncroniztion or lock.

What is the Ideal solution for the synchronization problem -> The ideal solution for synchronization problems depends on the specific requirements of your application. However, a widely accepted approach is to use a combination of proper synchronization mechanisms provided by Java, such as the synchronized keyword, ReentrantLock, and other concurrency utilities.

Properties of good soln to synchronization problem.

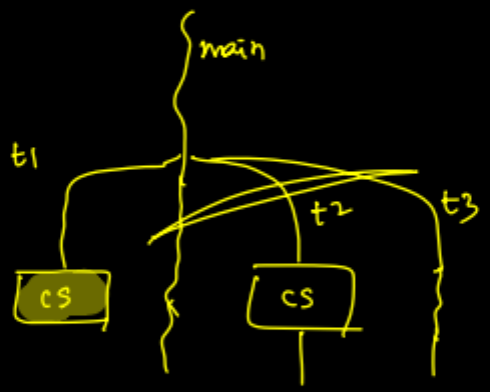
1. Mutual exclusion [MUST HAVE]

If one thread is already inside critical section of a variable, then no other thread should allowed inside CS for the same variable.

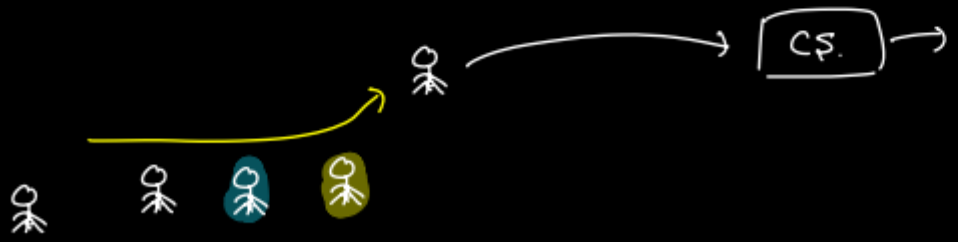
=) No race cond^n.

2. Progress

-) The overall app^n must be making progress

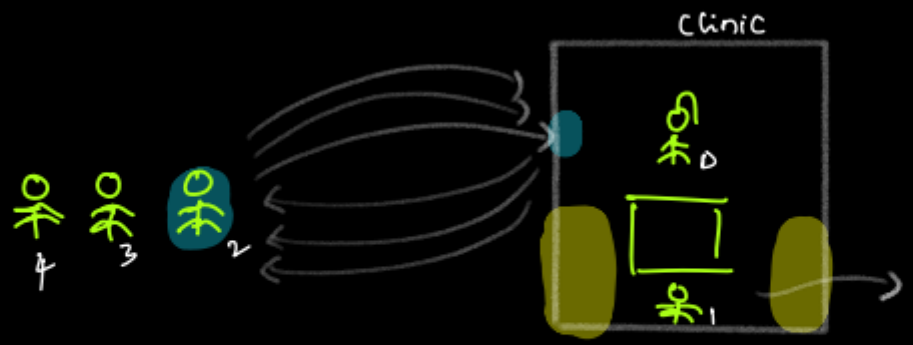


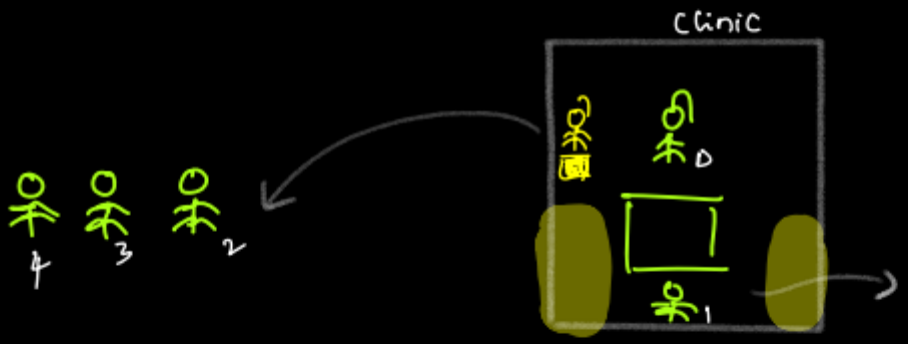
3. Bounded wait



1. No task should waif for indefinite times
2. There must be a bound on after how many tasks, will a particular task be allowed to enter critical section.
3. There must be some sort of FCFS basis.

4. No busy waiting.



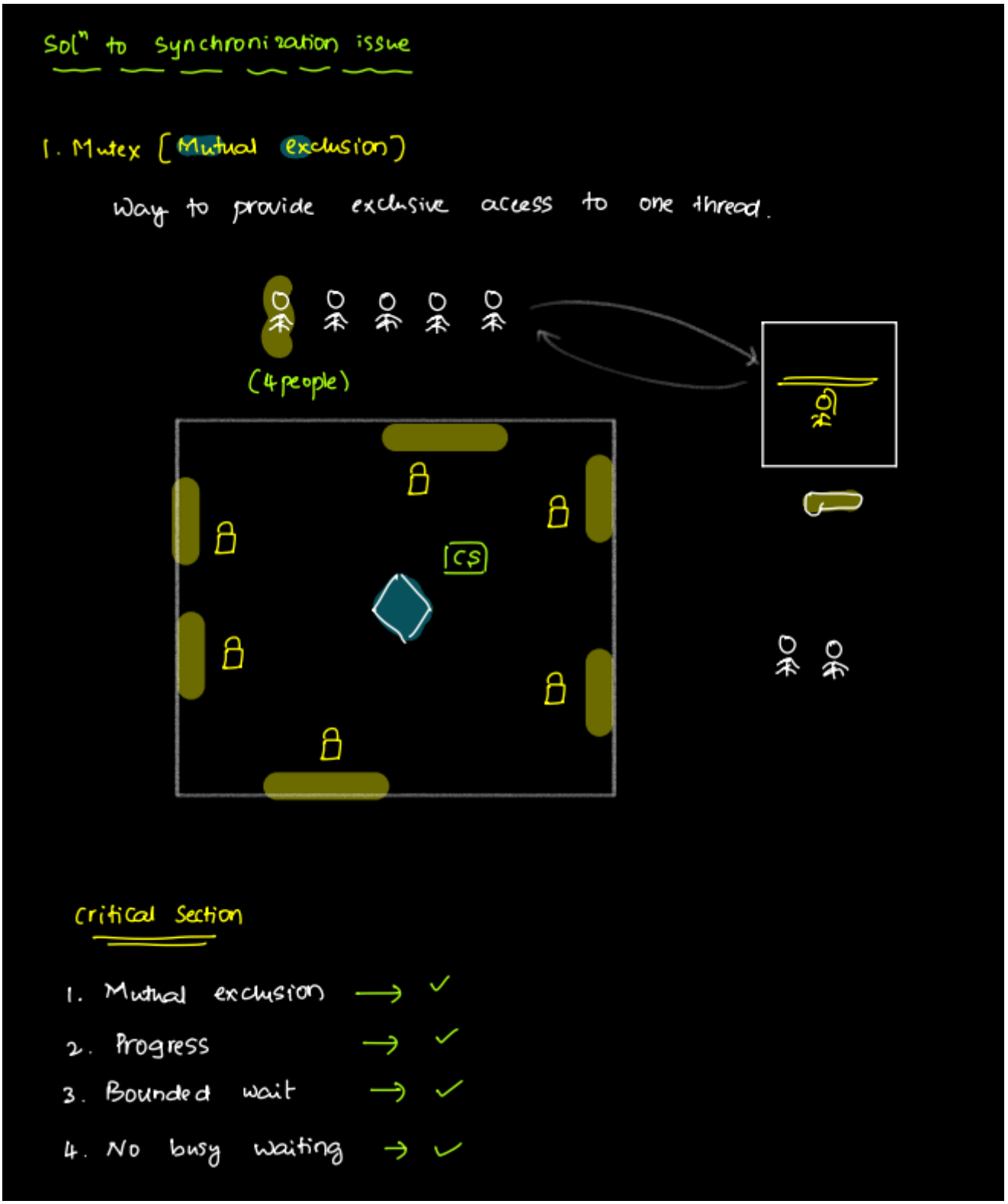


Bad

```
while (true)
    if(! available)
        Thread.sleep(1m)
    else
        break;
```

Not effective utilization of CPU.

Mutex -> a mutual exclusion (mutex) is a program object that prevents multiple threads from accessing the same shared resource simultaneously. A shared resource in this context is a code element with a critical section, the part of the code that should not be executed by more than one thread at a time.



Concurrency-5: Synchronization - Dec 19

- **Synchronized**
- **Producer consumer problem**
- **Atomic data types**

1. Synchronized -> synchronization is a mechanism to control the access of multiple threads to shared resources to prevent data corruption and ensure consistency in concurrent programming. The synchronized keyword and java.util.concurrent package provide tools for synchronization.

Synchronized Methods: Use the synchronized keyword to declare a method synchronized.

```
public synchronized void synchronizedMethod() {
    // Synchronized method body
}
```

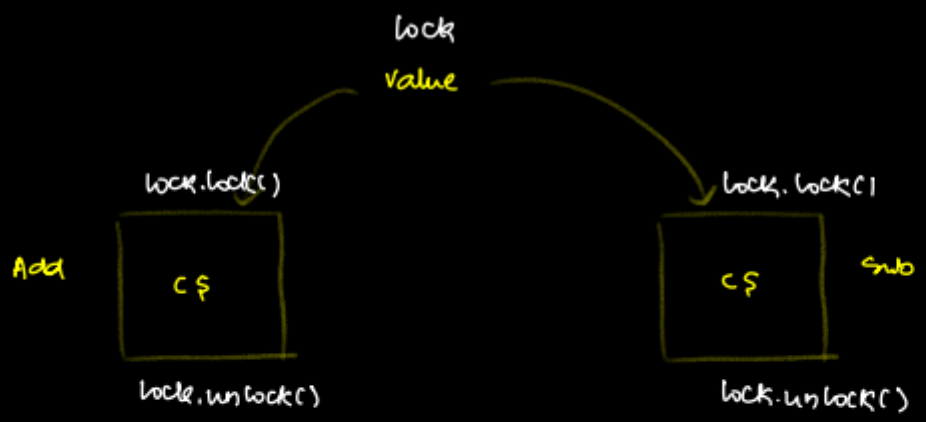
Synchronized Blocks: Synchronize a specific block of code rather than an entire method.

```
public void someMethod() {
    // Non-critical section
    synchronized (pass object which we have to apply lock) {
        // Critical section
    }
    // Non-critical section
}
```

Static Synchronization: Apply synchronization at the class level using the static keyword.

Example:

```
public static synchronized void staticSynchronizedMethod() {
    // Synchronized static method body
}
```



Target:

Can we somehow prevent passing the lock object and get away with sync issues just by using 'value' object.

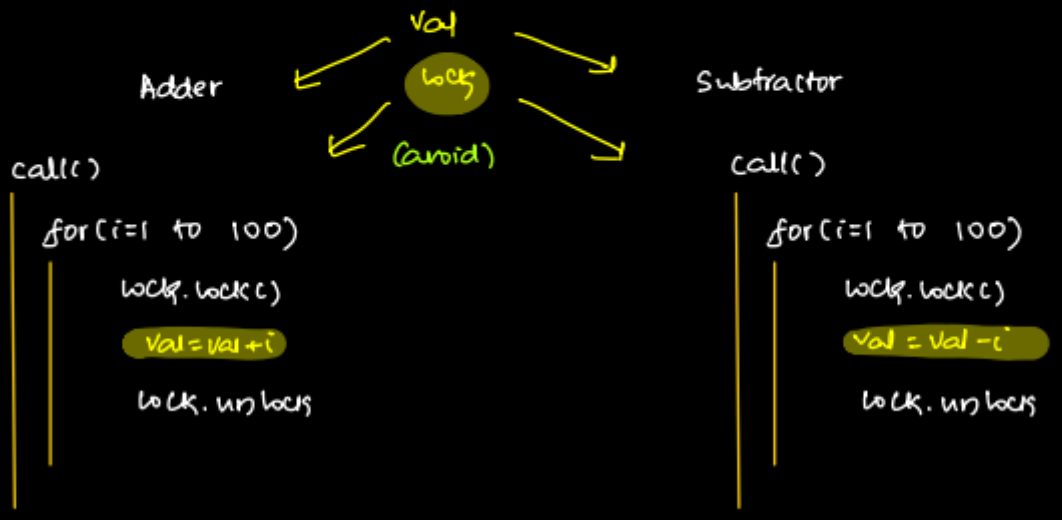
Sol":

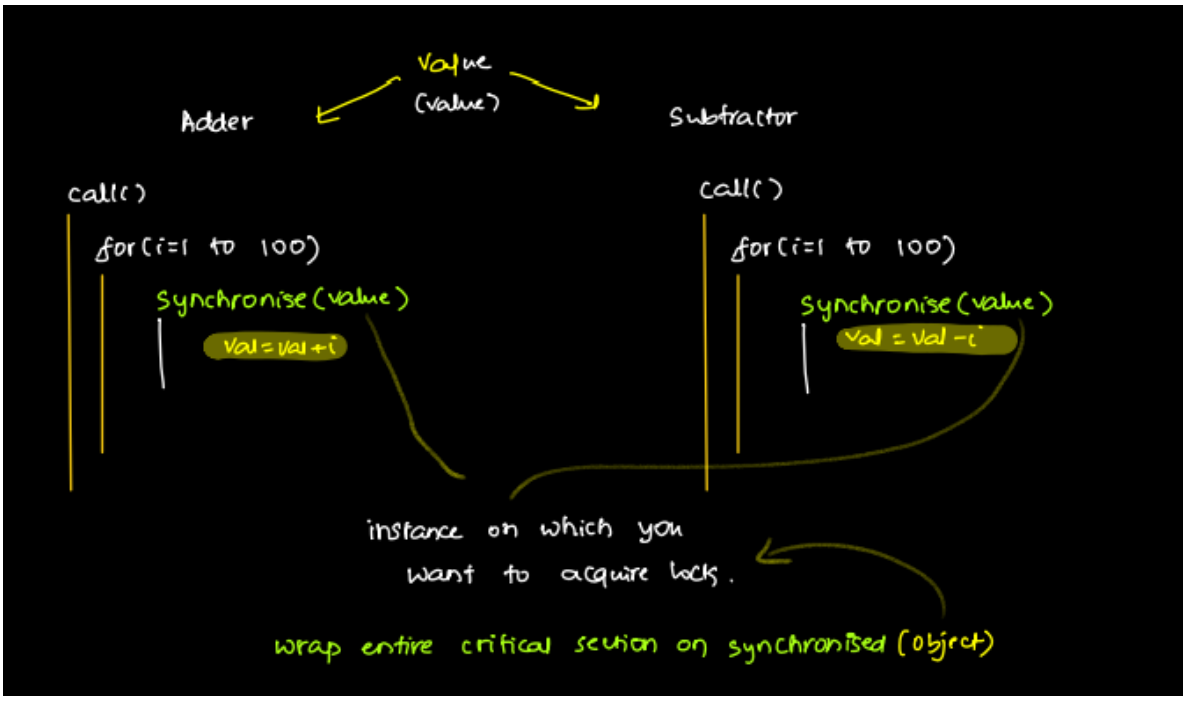
Yes! in Java, every object you create will have an implicit lock attached it.

You cannot see this, but you trigger this functionality.

↓

Synchronise.





Mutex

- Passing a lock
- synchronised

When synchronization fails?..

When you want to acquire lock on multiple objects, synchronised keyword fails to provide sol?

Ex:

book(1,2,4,5)

1 lock
2 lock
3 lock
4 lock
1 unlock
2 unlock
1
1

book(4,5,6,7)

Sync(4)
Sync(5)
Sync(6)
Sync(7)

```
book(List<Seat> seats)
    for(Seat s: seats)
        sync(s)
            // book s
```

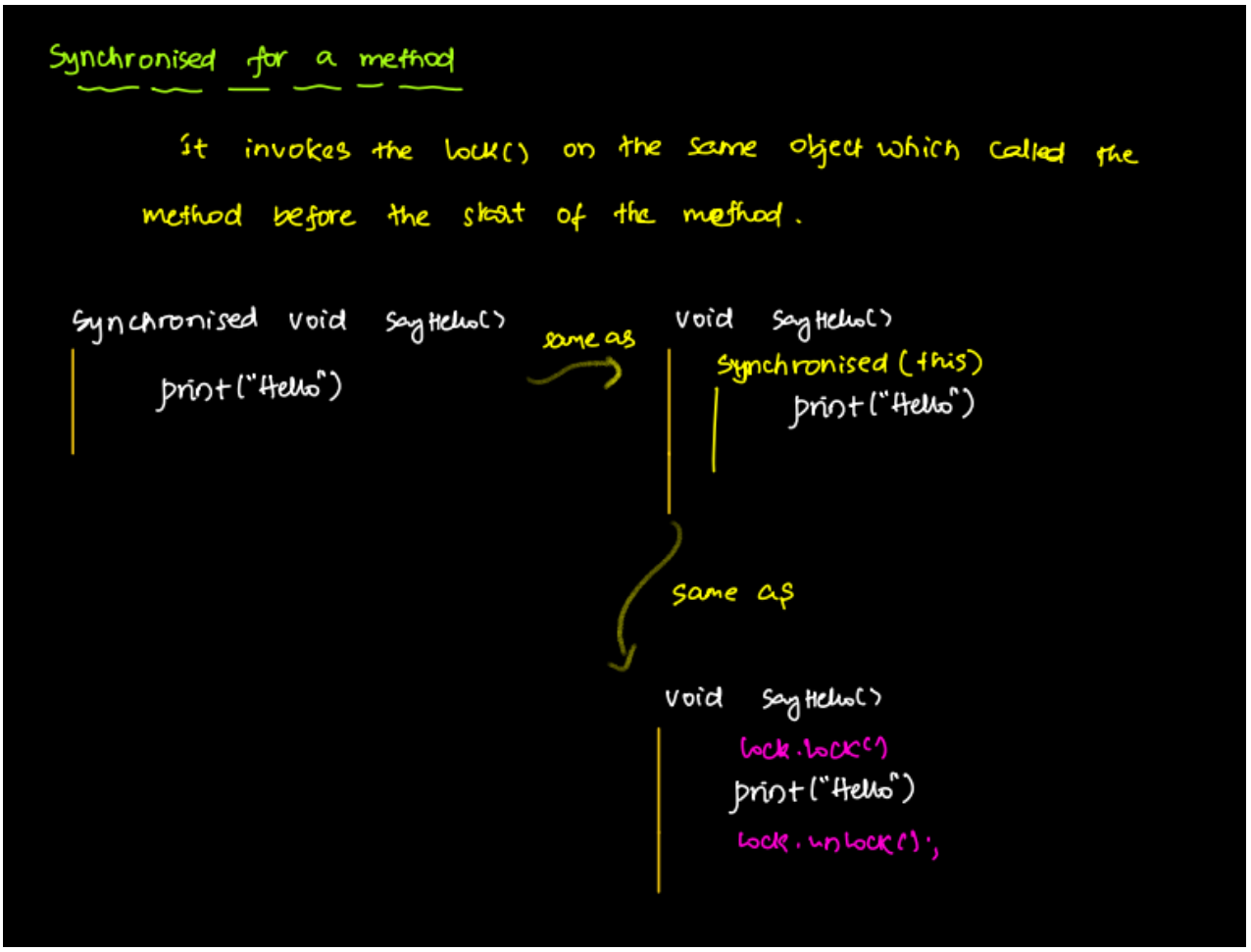
Doesn't work.

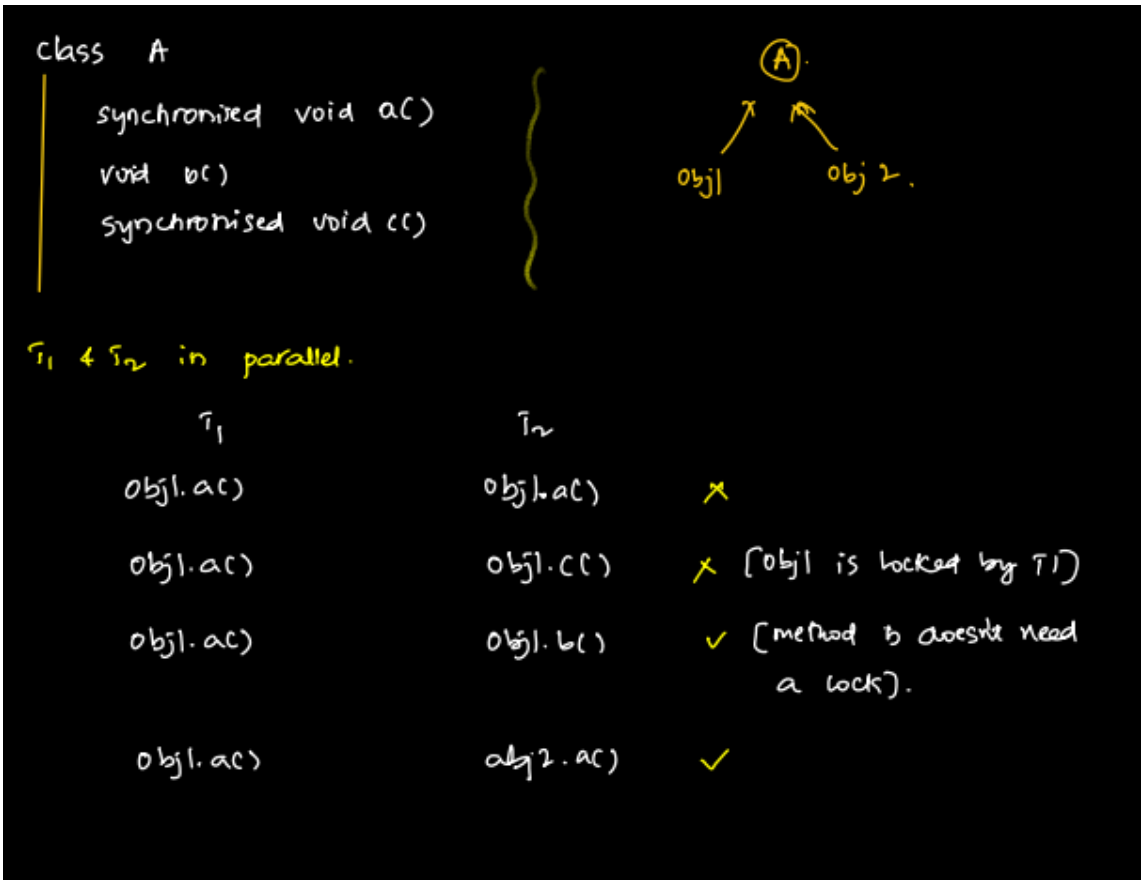
①, ②, ③ -- sequential.

If you can use 'synchronise' use that only, when you can't use Mutex maybe.

Mutex will do two task

1. **passing a lock**
2. **Apply synchromized using the lock object**





Reentrant Synchronization:

Java supports reentrant synchronization, meaning a thread can acquire the same lock multiple times without deadlock.

Example:

```
synchronized void outerMethod() {
    innerMethod();
}

synchronized void innerMethod() {
    // Critical section
}
```

Locks from java.util.concurrent.locks Package:

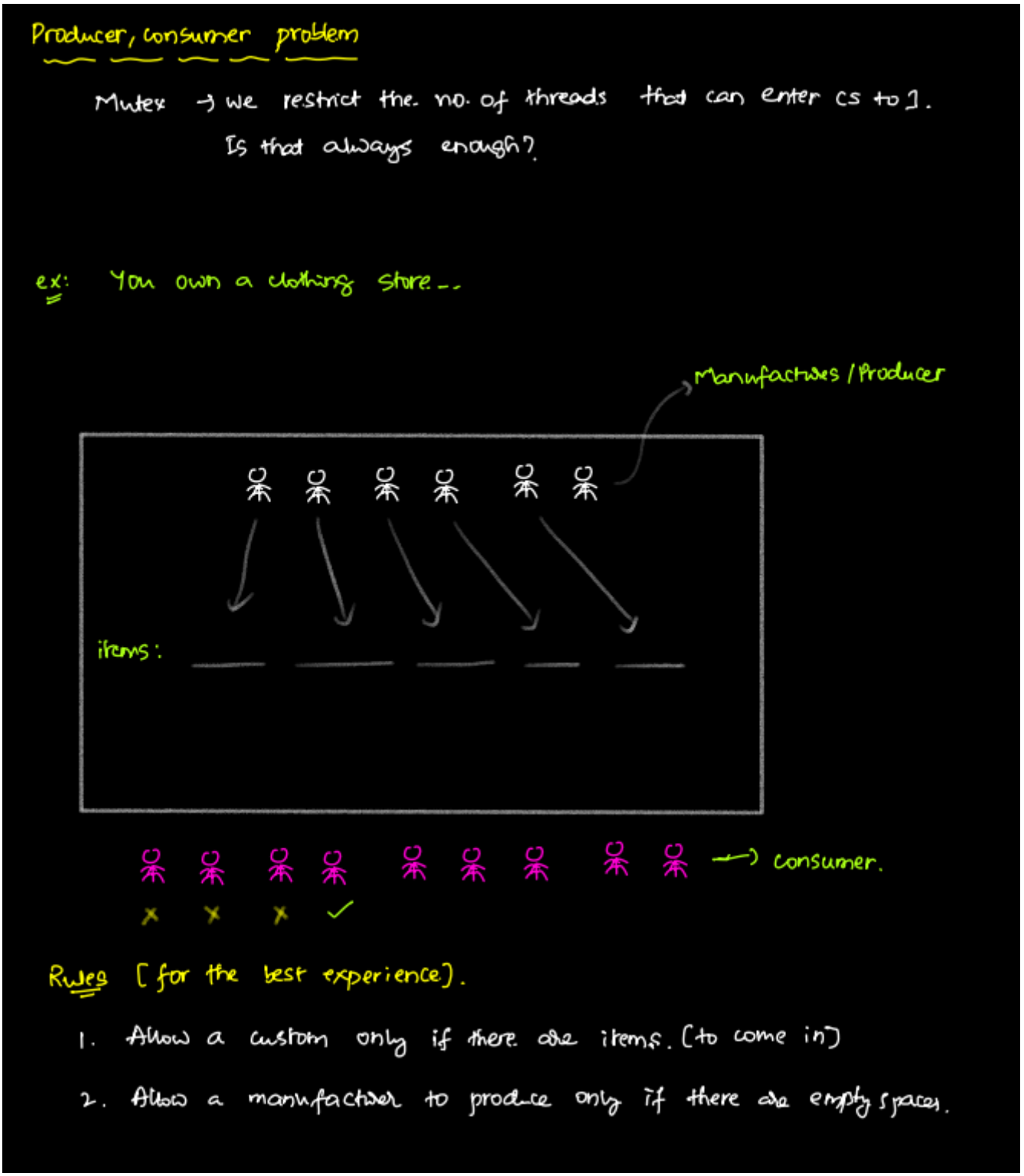
Provides more fine-grained control over synchronization using ReentrantLock and ReadWriteLock.

Example:

```
import java.util.concurrent.locks.ReentrantLock;

ReentrantLock lock = new ReentrantLock();

public void someMethod() {
    lock.lock();
    try {
        // Critical section
    } finally {
        lock.unlock();
    }
}
```

Synchronization is essential for ensuring thread safety and preventing race conditions in multithreaded Java applications. Careful use of synchronization mechanisms is crucial to maintaining correctness and performance in concurrent programs.

Producer consumer problem



```
class Store
    List<Object> items
    int maxSize
```

```
class Producer
    Store store
    produce()
        while (true)
            if( store.items.size()
              <store.maxSize)
                store.items.add()
```

```
class Consumer
    Store store
    consume()
        while (true)
            if( store.items.size()>0)
                store.items.remove()
```

Tasks are produce & consume. => Threads.

T_2 (Produce) T_1 (Produce) T_2 (consume) T_3 (consume)

size() < max
add()

size()>0
remove()

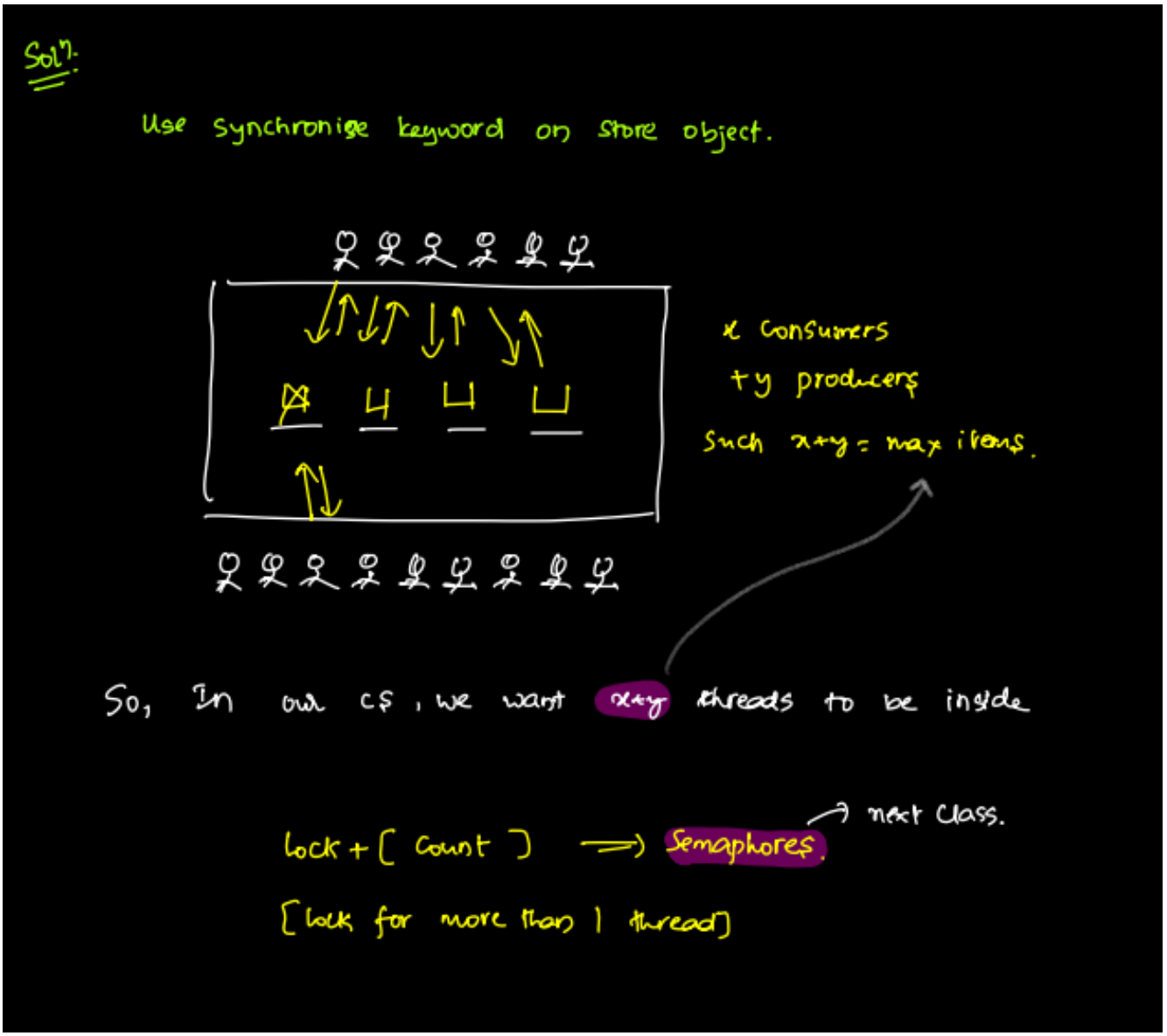
size()>0
remove()

Size → ✗ 0.

T_3, T_2 > size(1).

T_3 ~~~ Size → 0.

T_2 ~~~ (size → -1) => Error.



Atomic data types -> In Java, the java.util.concurrent.atomic package provides classes that support atomic operations on underlying variables. These classes are part of the Java Concurrency Framework and are designed to facilitate concurrent programming by ensuring that certain operations are atomic, meaning they are executed as a single, indivisible unit without the possibility of interruption.

Here are some commonly used atomic data types/classes in Java:

AtomicInteger: Represents an int value that may be updated atomically.

```
import java.util.concurrent.atomic.AtomicInteger;

AtomicInteger atomicInt = new AtomicInteger(0);
atomicInt.incrementAndGet();
```

AtomicLong: Represents a long value that may be updated atomically.

```
import java.util.concurrent.atomic.AtomicLong;

AtomicLong atomicLong = new AtomicLong(0);
atomicLong.incrementAndGet();
```

AtomicBoolean: Represents a boolean value that may be updated atomically.

```
import java.util.concurrent.atomic.AtomicBoolean;

AtomicBoolean atomicBoolean = new AtomicBoolean(true);
atomicBoolean.compareAndSet(true, false);
```

AtomicReference: Provides an object reference that may be updated atomically.

```
import java.util.concurrent.atomic.AtomicReference;

AtomicReference<String> atomicReference = new AtomicReference<>("initialValue");
atomicReference.compareAndSet("initialValue", "newValue");
```

AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray:

These classes provide atomic operations on arrays of integers, longs, and object references respectively.

```
import java.util.concurrent.atomic.AtomicIntegerArray;

AtomicIntegerArray atomicIntArray = new AtomicIntegerArray(new int[]{1, 2, 3});
atomicIntArray.getAndIncrement(0);
```

These atomic classes are useful in scenarios where multiple threads may be updating shared variables concurrently, and you want to ensure that these updates are done atomically without the need for explicit synchronization using locks. They are part of the broader Java Concurrency API, which helps in writing thread-safe and scalable concurrent programs.

Atomic data types → Something that cannot be divided.

Adder

Val = Val + i

Subtractor

Val = Val - i

They're not atomic.

T1

Get value → V. [100]

Do calculation → V + 2 [102]

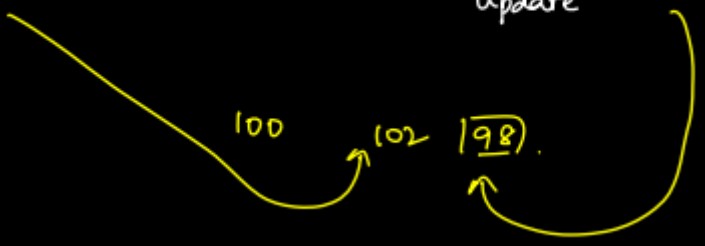
Update

T2

Get value → V. [100]

Do calculation → V - 2 [98]

Update



Atomic data types

They provide pre-implemented atomic behaviour for some common methods.

Concurrency-6: Synchronization with Semaphores - Dec 21

- **Semaphores**
- **Deadlocks and how to avoid them**
- **Concurrent Datastructure**
- **How to decide no of threads**
- **Some problems on leetcode**

Shemaphors -> a semaphore is a synchronization primitive that controls access to a shared resource by using a set of permits. Semaphores are often used to control access to a pool of resources or limit the number of threads that can access a particular section of code simultaneously. The java.util.concurrent package provides the Semaphore class for this purpose.

Here's a brief explanation of how semaphores work in Java:

Semaphore Initialization: You create a Semaphore object by specifying the number of permits it should have initially. This determines how many threads can access the shared resource simultaneously.

```
import java.util.concurrent.Semaphore;
Semaphore semaphore = new Semaphore(3); // Allow three threads to access the shared resource concurrently
```

Acquiring Permits: Threads can acquire permits from the semaphore using the acquire() method. If a permit is available, the thread acquires it; otherwise, it blocks until a permit is released.

```
try {
    semaphore.acquire(); // Acquire a permit
    // Code that accesses the shared resource
} catch (InterruptedException e) {
    // Handle interruption
} finally {
    semaphore.release(); // Release the permit
}
```

Releasing Permits: After a thread finishes using the shared resource, it should release the permit using the release() method. This allows other waiting threads to acquire permits.

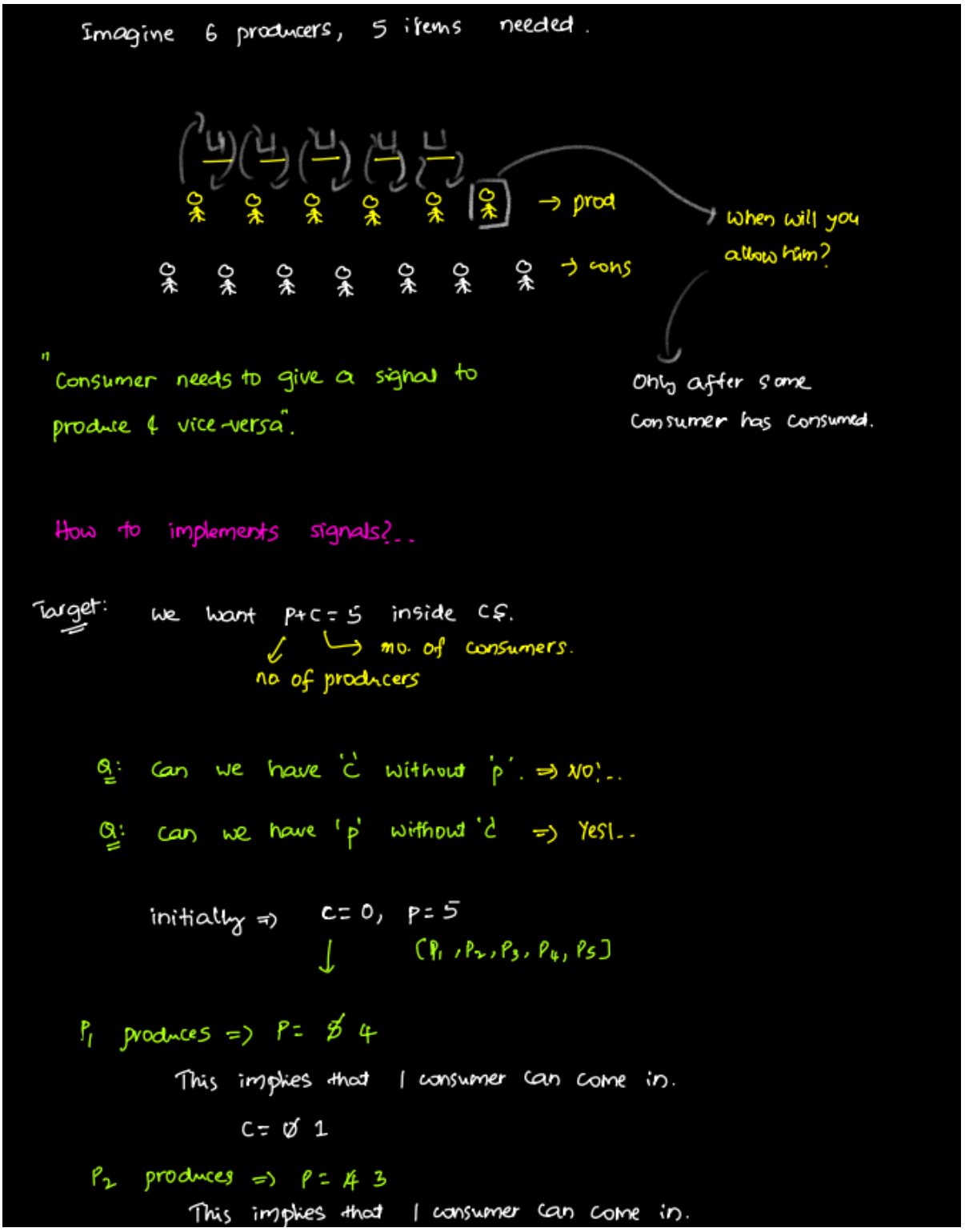
```
try {
    // Code that accesses the shared resource
} finally {
    semaphore.release(); // Release the permit
}
```

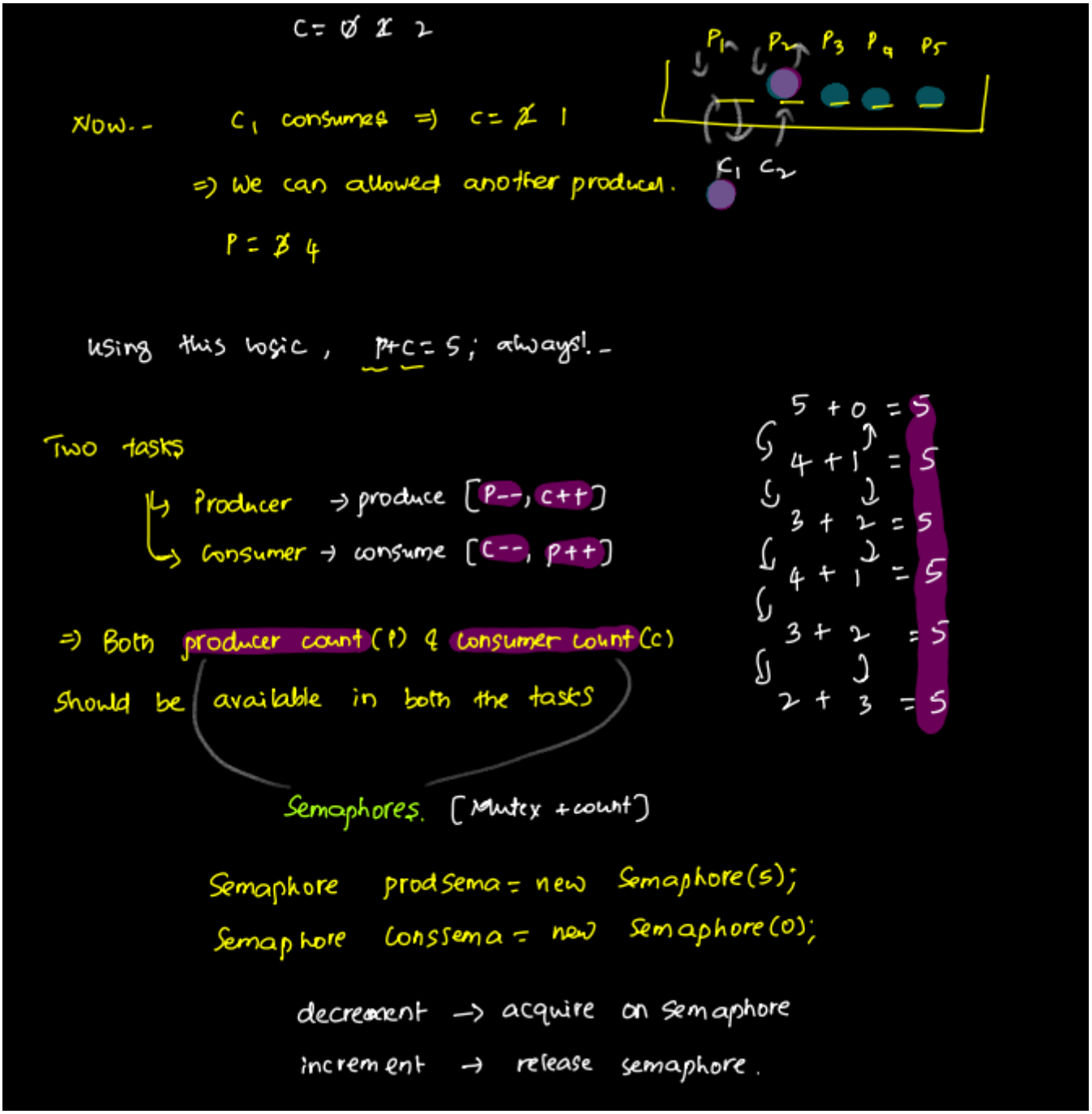
Available Permits: You can check the number of available permits using the availablePermits() method.

int availablePermits = semaphore.availablePermits();

Semaphores are a powerful tool for coordinating access to shared resources in concurrent programming. They help control the number of threads that can concurrently access a particular section of code or a shared resource.

Producer consumer problem using semaphore



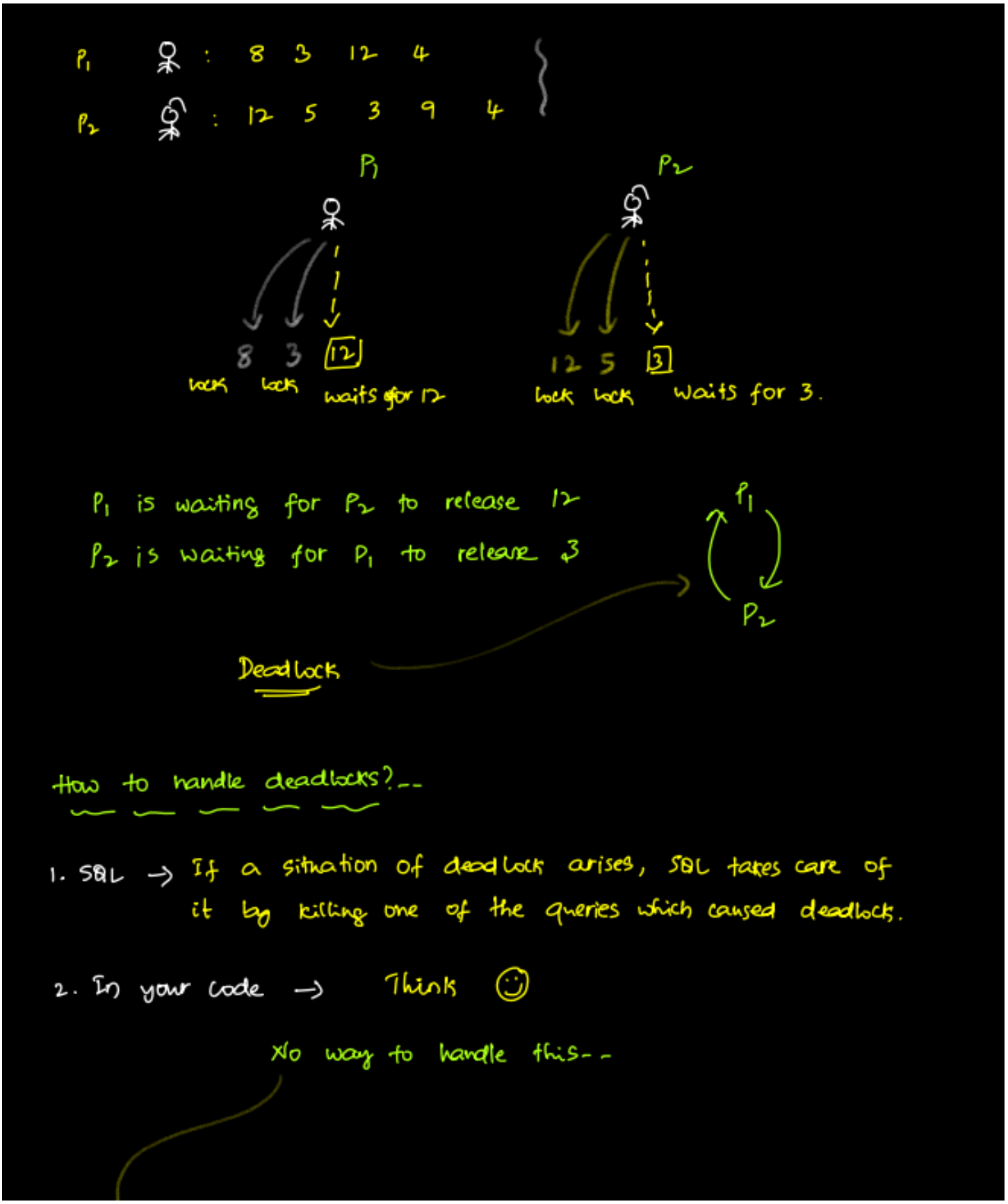


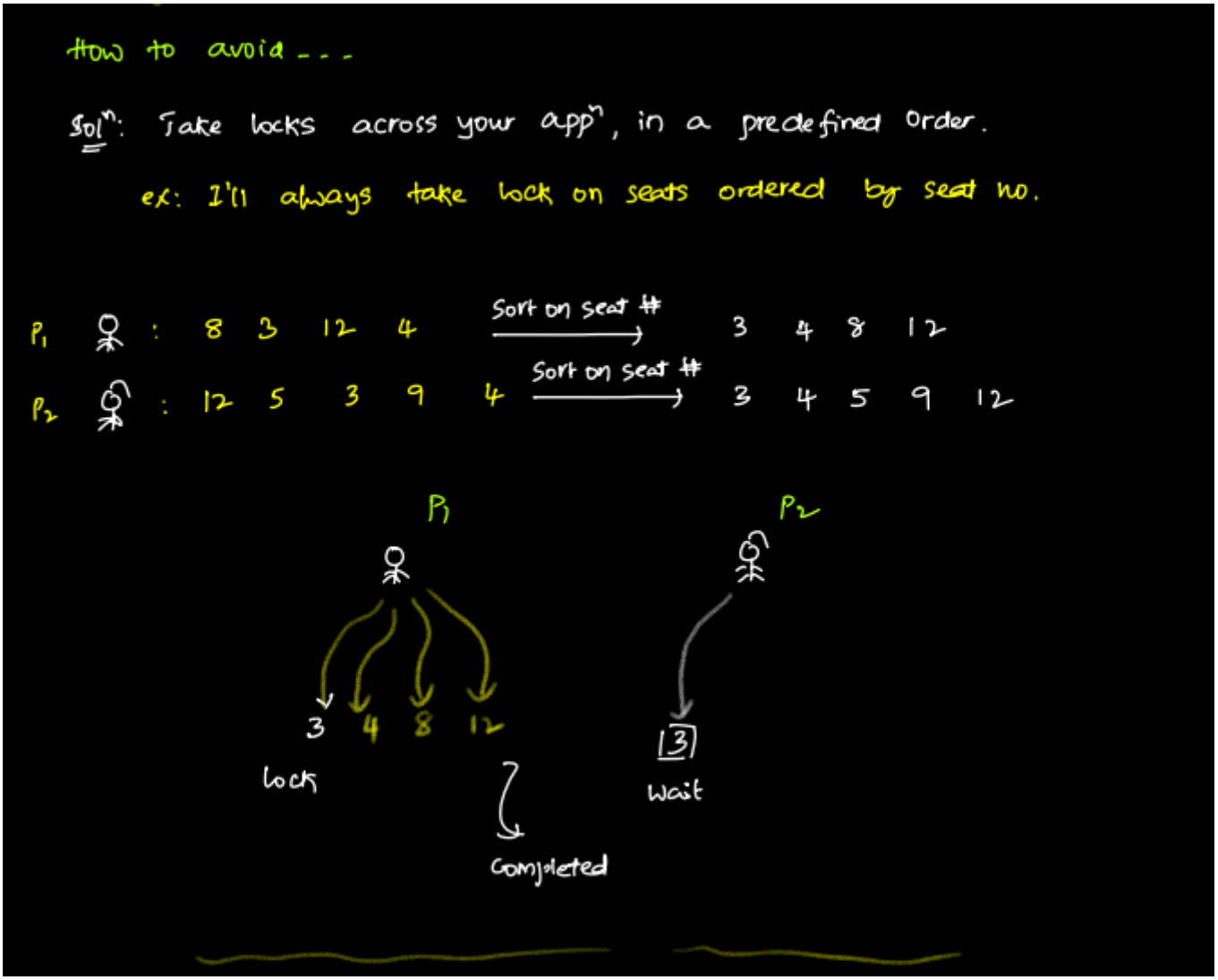
In this problem, we want to efficiently manage the use of shared resources by multiple threads. Initially, we consider a basic approach using the synchronized keyword, which allows only one producer or consumer to access the resource at a time, causing others to wait. However, this can lead to inefficiencies.

To enhance our strategy, we opt for employing a "semaphore," allowing multiple threads (limited to 5 in our scenario) simultaneous access to the resource, eliminating potential bottlenecks. However, relying solely on a semaphore may not be sufficient. There's a risk that multiple threads could concurrently modify the data of the same collection, resulting in data corruption or inconsistency issues, such as ArrayIndexOutOfBoundsException.

To solve this, we suggest using concurrent data structures along with the semaphore. This combination ensures that multiple threads can safely access the critical section without risking data structure synchronization problems, offering a more robust solution.

Deadlock -> a situation where two or more processes are unable to proceed because each is waiting for the other to release a resource. In other words, the processes are caught in a circular waiting pattern, and none of them can make progress. Deadlocks can occur in multi-threaded or multi-process environments where different components or entities contend for shared resources.





Concurrent Datastructure -> A concurrent data structure is a type of data structure designed to be safely accessed and modified by multiple threads or processes concurrently. In a multi-threaded or multi-process environment, where different components may be working simultaneously, it's essential to ensure that data structures can handle concurrent operations without leading to race conditions, data corruption, or inconsistencies.

The term "concurrent data structure" encompasses various data structures that are specifically crafted to support concurrent access. Some common examples include:

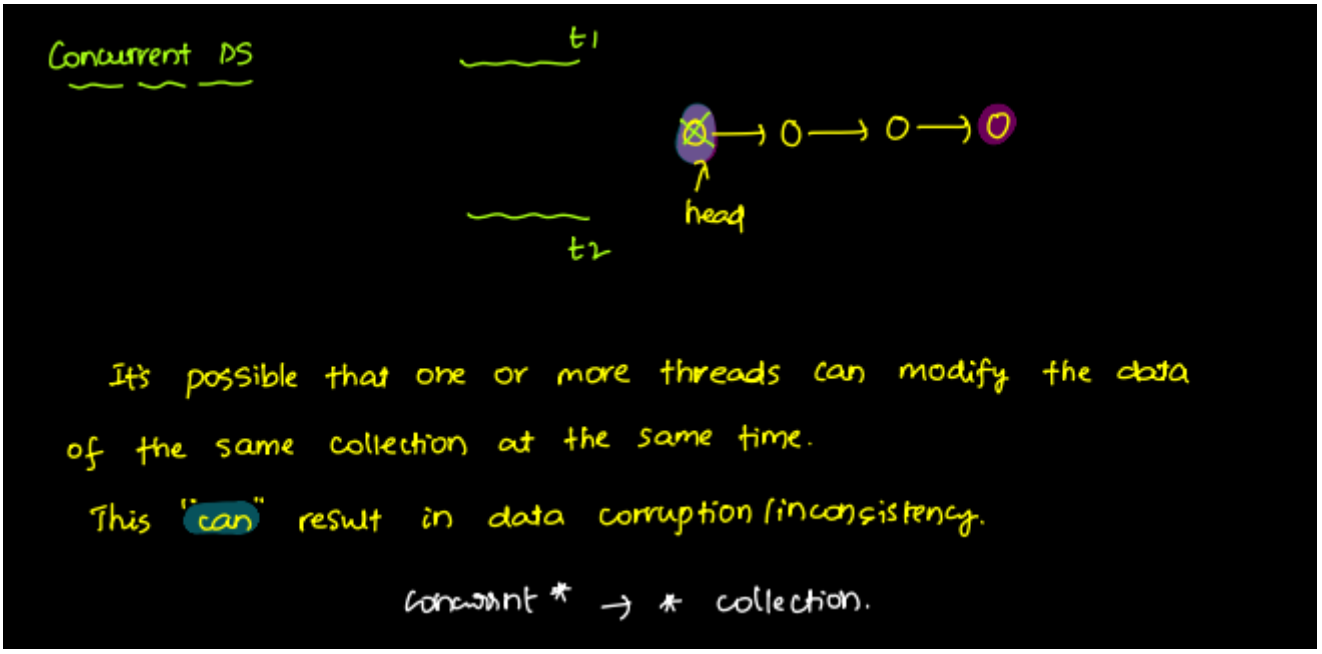
Concurrent Collections: Specialized collections like ConcurrentHashMap and ConcurrentLinkedQueue in Java that are designed to be thread-safe and support concurrent read and write operations.

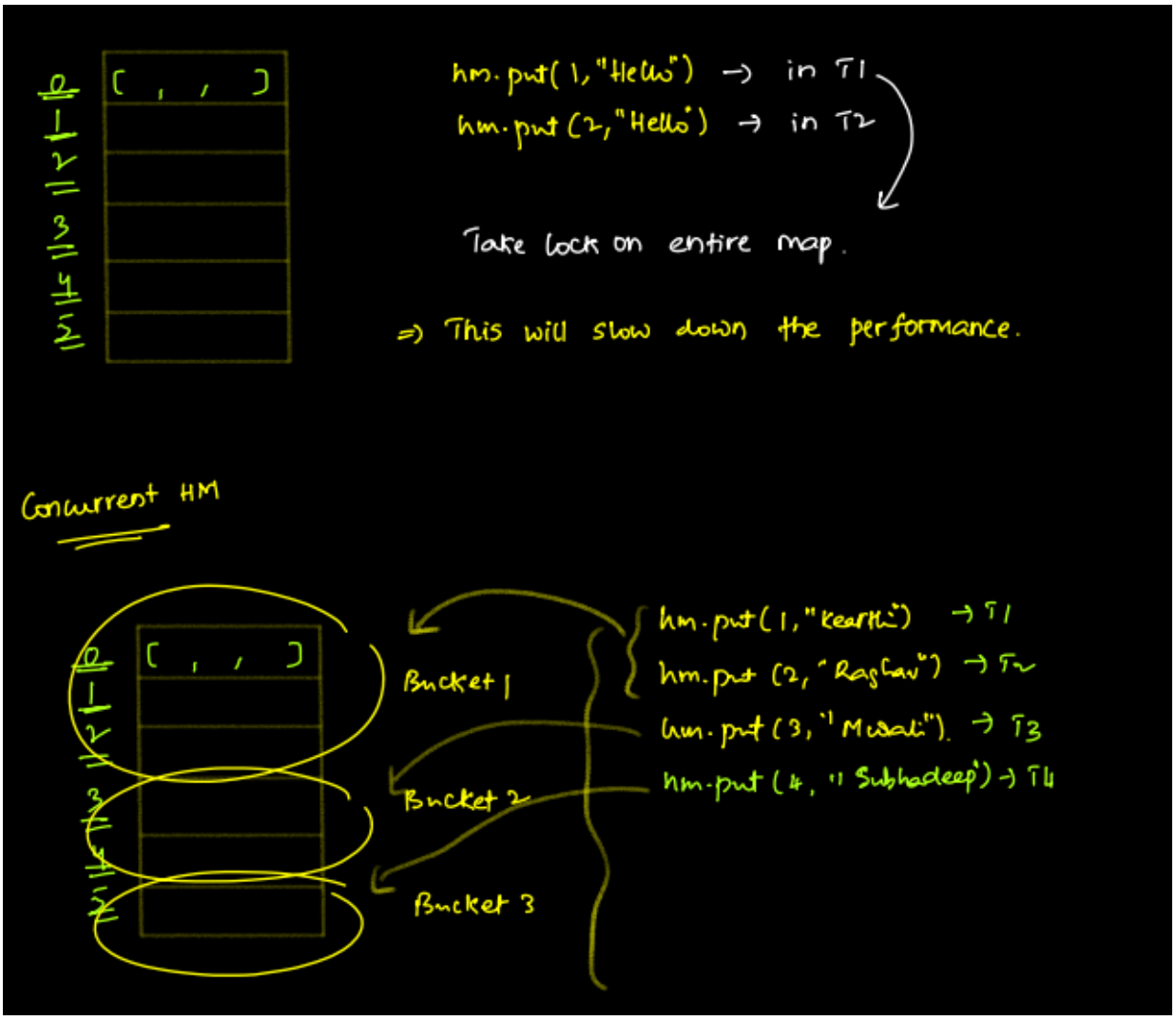
Concurrent Queues: Queues that can be accessed concurrently, allowing multiple threads to enqueue and dequeue elements without conflicts. Examples include lock-free queues.

Concurrent Lists: Lists that support concurrent access, enabling multiple threads to insert, remove, or traverse elements without interfering with each other.

Concurrent Trees: Tree-based data structures, such as concurrent versions of binary search trees, that allow for concurrent insertion, deletion, and search operations.

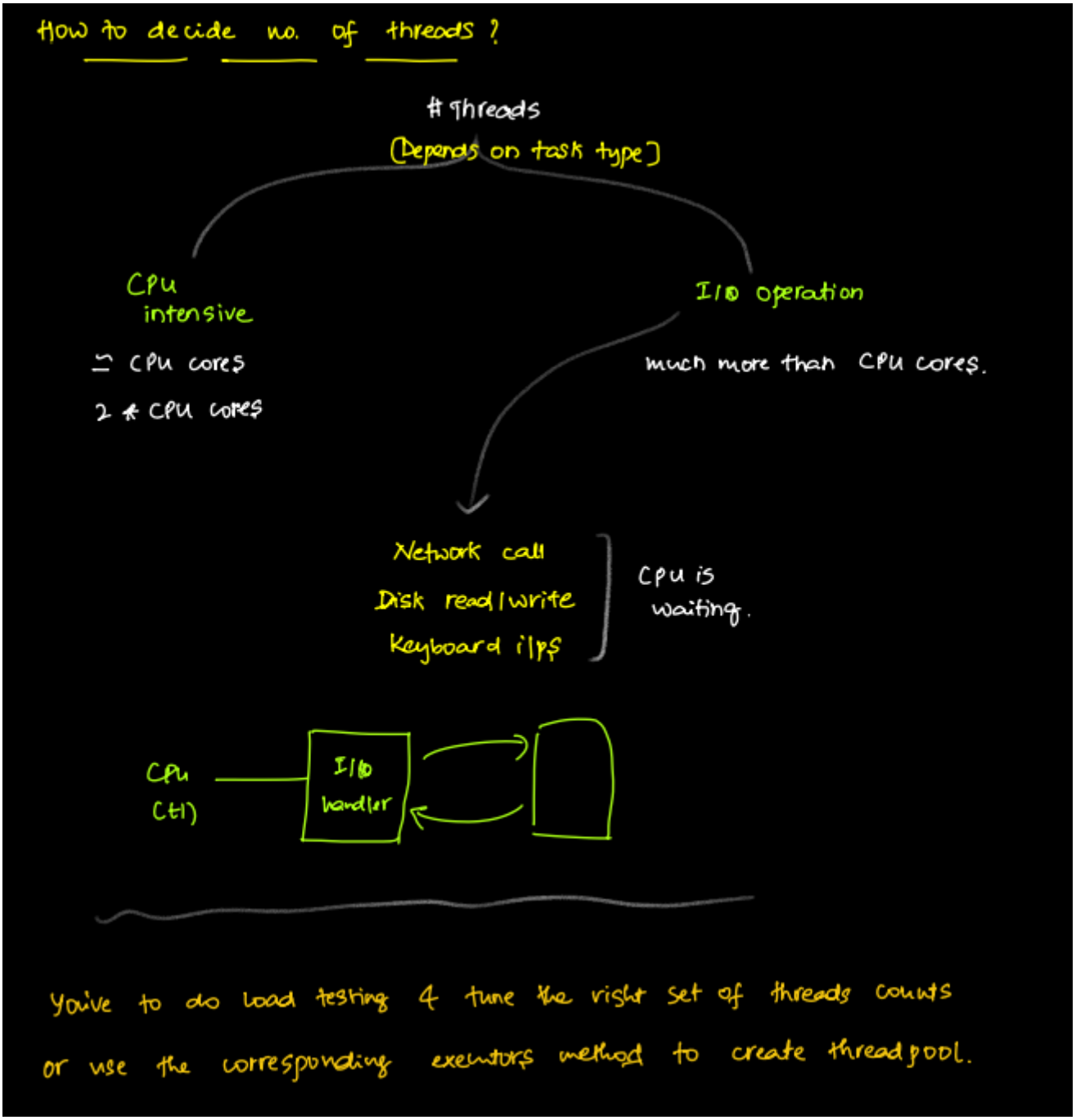
Lock-Free Data Structures: Data structures designed to operate without the need for traditional locks, using atomic operations to ensure thread safety.





The primary goal of concurrent data structures is to provide a level of synchronization or coordination that allows multiple threads to work with the data structure concurrently without causing conflicts or compromising data integrity. These structures often use techniques like locking, atomic operations, or other synchronization mechanisms to achieve thread safety.

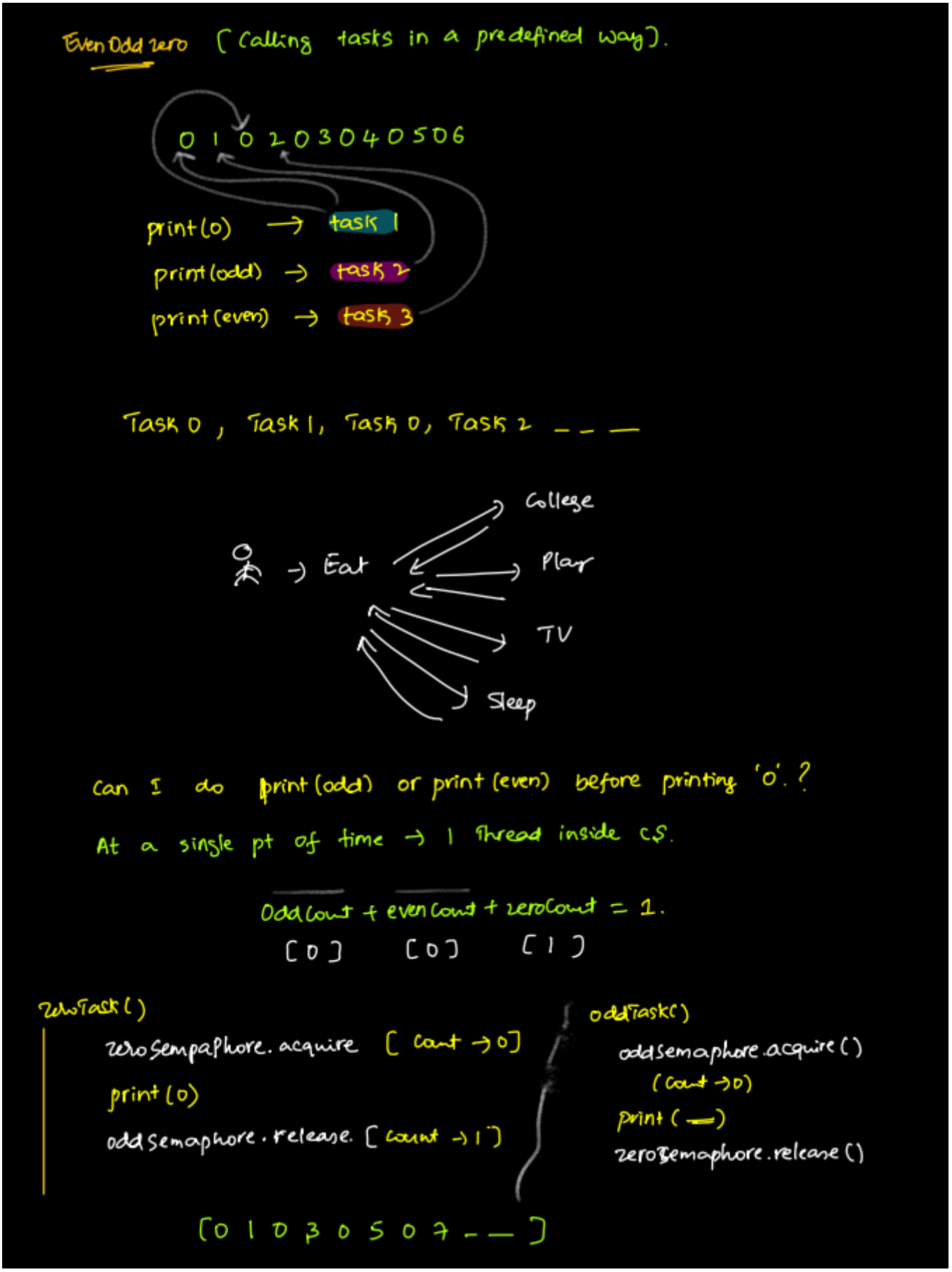
How to decide no of threads



Some problems on leetcode

https://leetcode.com/tag/concurrency/

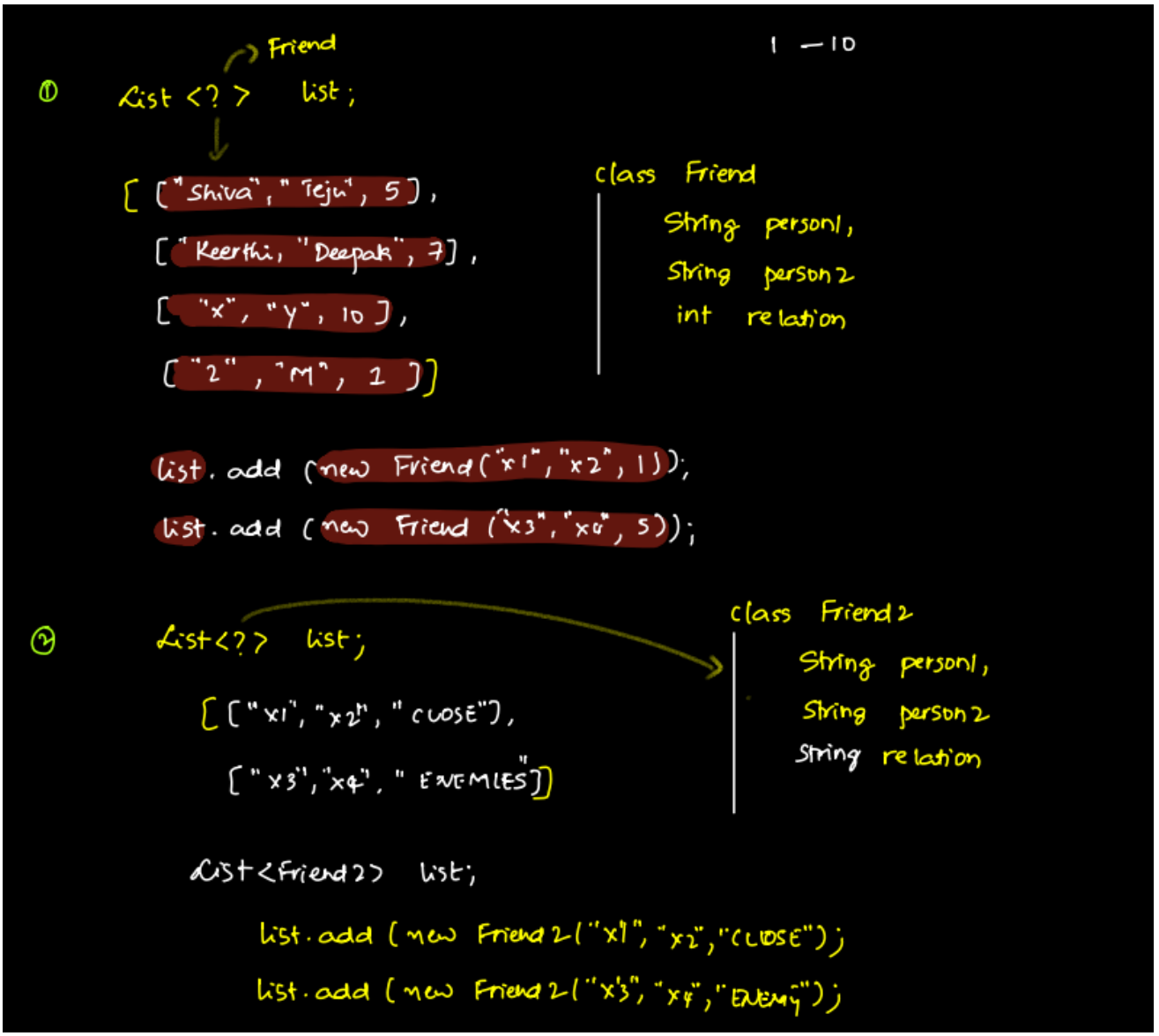
Explnation -> Print Zero Even Odd

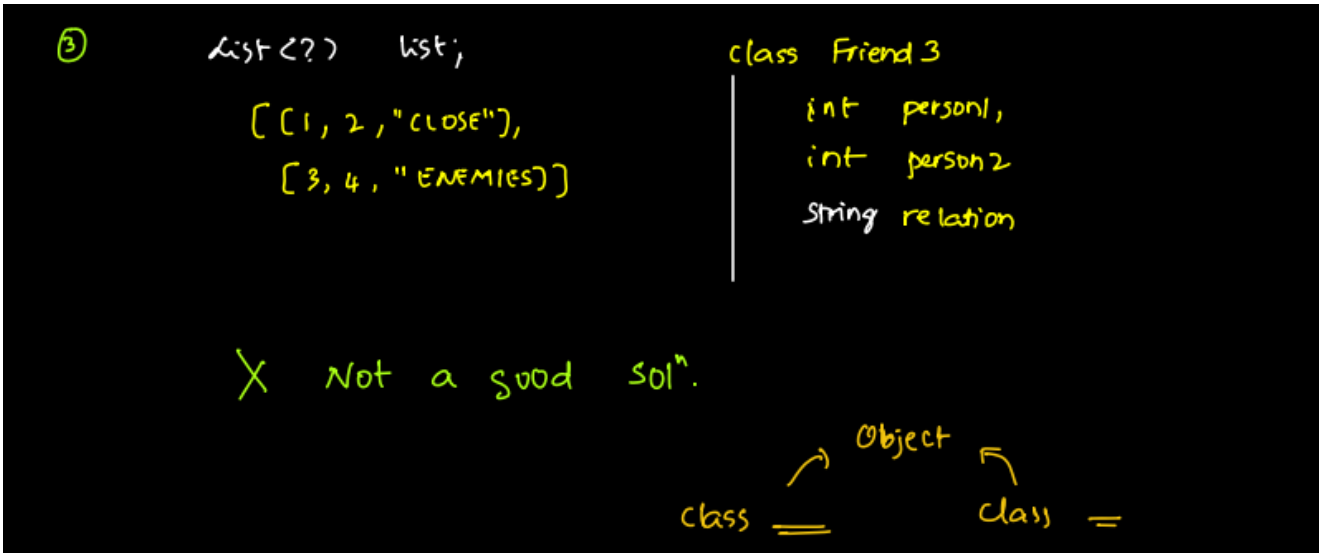


Java Advanced Concepts - 1 [Generics] - Dec 23

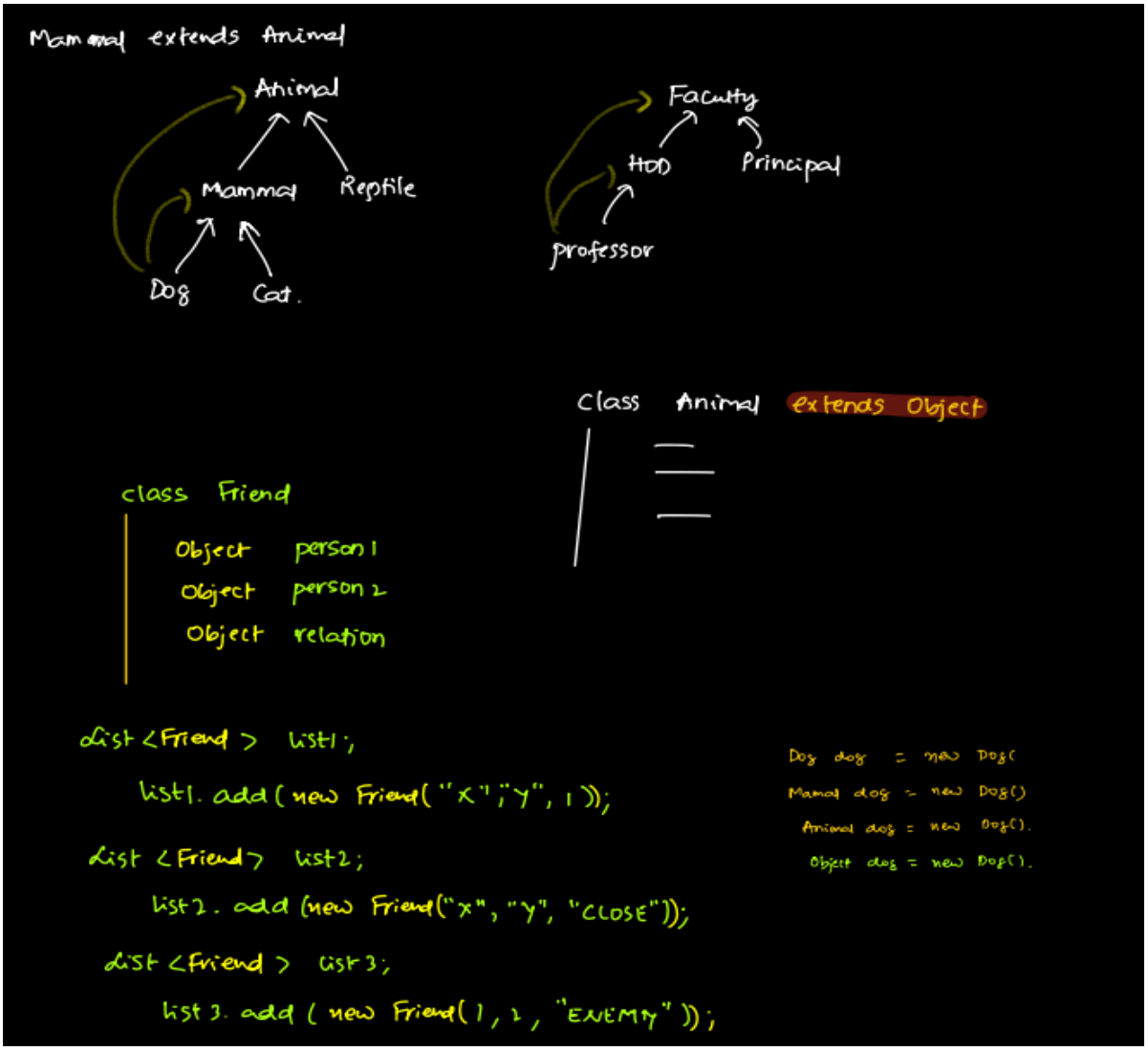
- **The problem which lead to introduction of generics**
- **Genrics Types**
- **Raw Types**
- **Genric methods**
- **Bounds**
- **Inheritance in generics**
- **Inheritance super class interface only in generics**
- **Static Inheritance in generics**
- **Type Erasure**

The problem which lead to introduction of generics.
Is creating another class a viable solution ? -> Making lots of classes for custom types is hard to manage, especially in big projects where there might be thousands of them. If you need to make even a small change, you'd have to create a new class each time. To make this easier, we use something called generics. It's a way to handle different types without having to create a specific class for each one, making things more flexible and easier to handle.



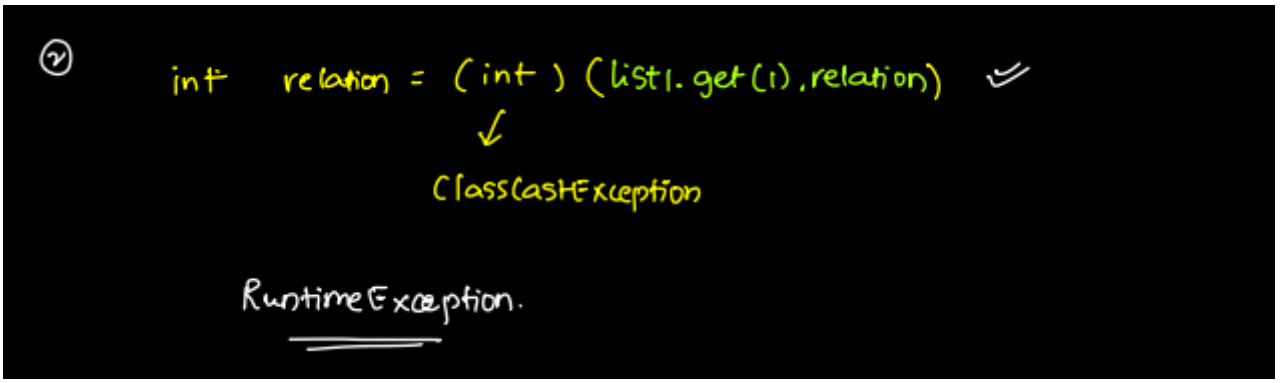


Is using the Object type the solution to everything? Well, it does address certain issues, but it also introduces some drawbacks. Since every class extends the Object class, we can use Object as a data type. This allows us to create multiple objects using just one class and pass different types of data. However, this approach, while seemingly solving the problem of creating multiple classes for different types, is not a perfect solution. The major drawback is that we lose type safety. This means we can pass any type of data, and it won't show errors during compile time, but it may result in runtime errors, which can be risky



The problem with using objects as generics is that while we can retrieve specific fields, we cannot ensure or enforce the desired data types

It won't result in a compile-time error, but instead, we will encounter a runtime error while we will use it, as illustrated below.



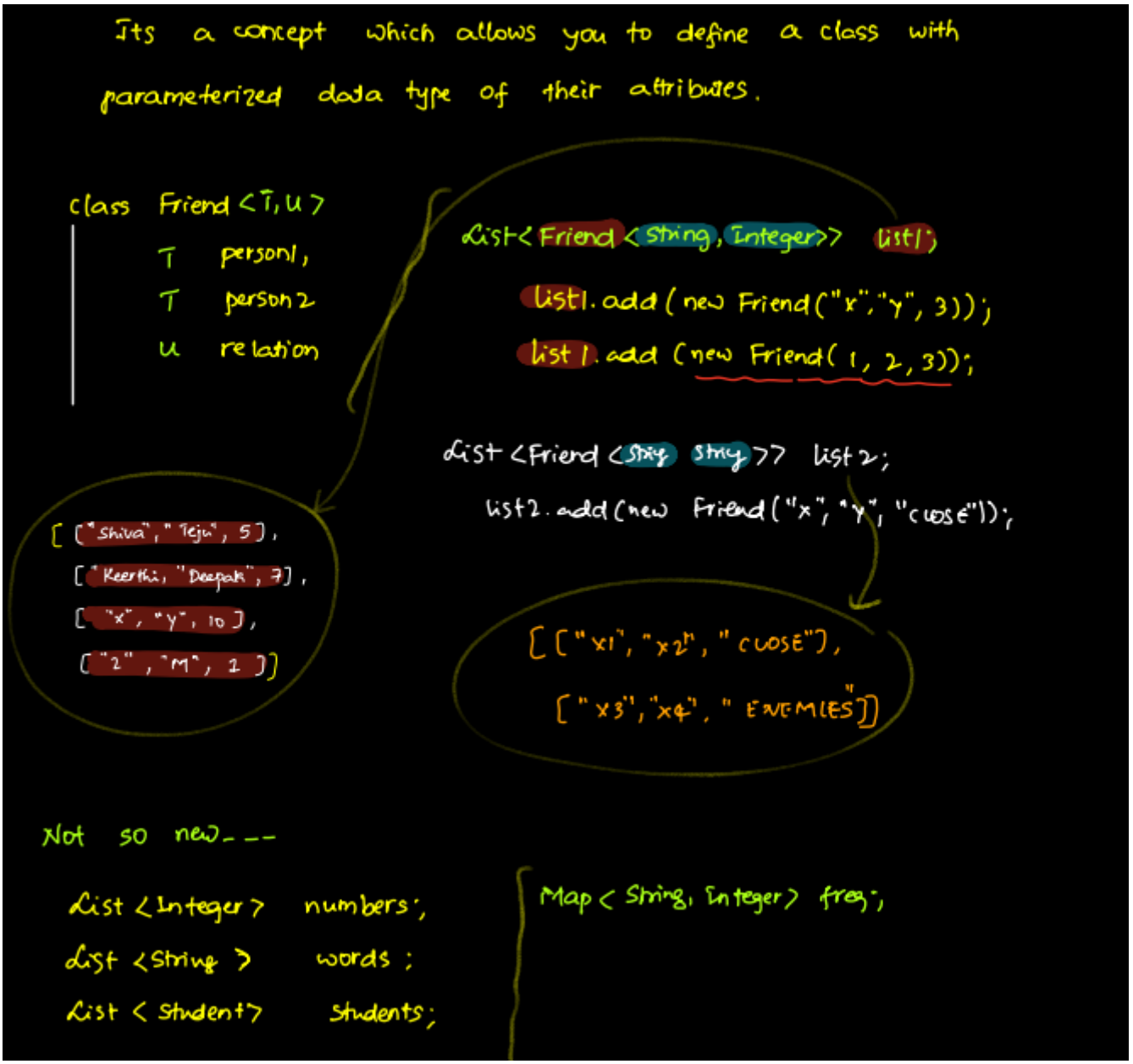
Genrics -> Generics in Java provide a way to create classes, interfaces, and methods with placeholder types. This allows you to design classes and methods that can work with any data type, providing type safety and reducing the need for casting.

Here's a brief overview of generic types:

Class with Generic Type Parameter:

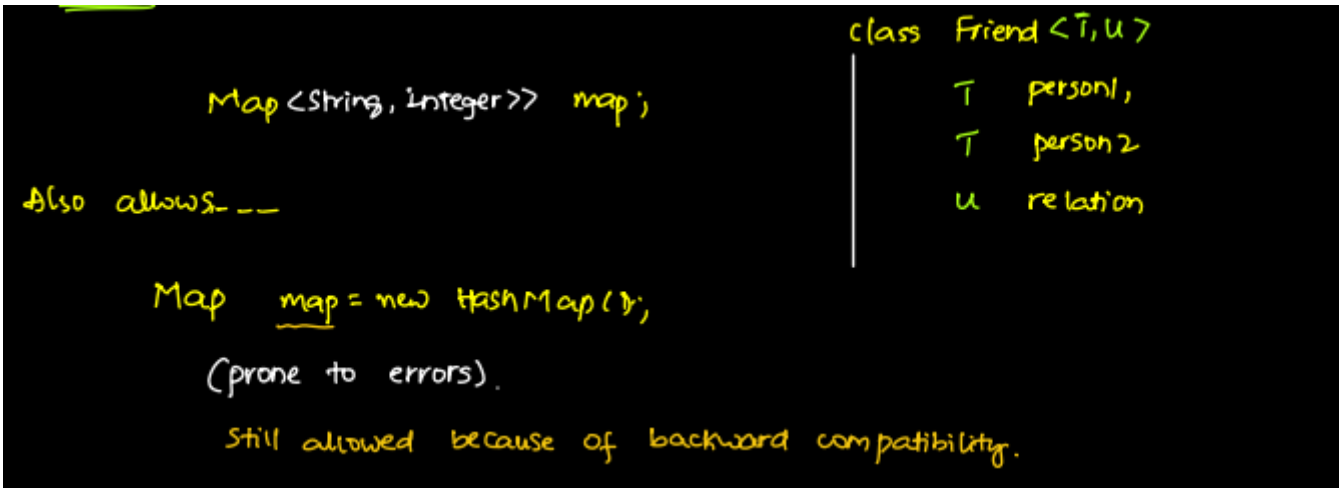
```
public class Box<T> {
    private T value;
    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

Here, T is a type parameter, and you can use any valid Java identifier as a type parameter.



Raw Types -> raw types refer to the use of a generic class or interface without specifying any type arguments for its generic parameters. Raw types exist for compatibility with legacy code that was written before the introduction of generics in Java 5. While raw types allow you to use generic classes without providing type parameters, they sacrifice the benefits of type safety and can lead to runtime issues.

Typically, we create a HashMap by specifying the types for both the key and value. However, in the List interface, you can observe that it's possible to create a HashMap without explicitly defining the types of key and value. This usage, relying solely on the Map interface without specifying key and value types, is referred to as raw types.



Here's an example to illustrate raw types:

```
// Generic class Box with a type parameter T
public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }}

// Using raw types (without specifying type parameter)
public class RawTypeExample {
    public static void main(String[] args) {
        // Creating a raw type of Box (without specifying type parameter)
        Box rawBox = new Box();

        // Assigning a String to rawBox, but the type information is lost
        rawBox.setValue("Hello, Raw Types!");

        // Retrieving the value without type safety
        Object value = rawBox.getValue();
        System.out.println("Value: " + value);

        // This can lead to runtime ClassCastException if the wrong type is assumed
        // String stringValue = (String) rawBox.getValue(); // Potential runtime issue
    }
}
```

In the example above, Box rawBox = new Box(); creates a raw type of the Box class without specifying the type parameter. This means you can assign any type of object to rawBox, but the type information is lost, and you need to be careful when retrieving values.

It's generally recommended to avoid using raw types in new code, as they bypass the type-checking provided by generics, leading to potential runtime errors. Instead, it's better to use parameterized types with explicit type arguments for better type safety.

Note: Generics do not permit the use of primitive types such as (int, float, double, etc.). Always utilize wrapper classes like (Integer, Float, Double) when working with generics.

Generic Method ->

```
public <T> T findFirst(List<T> list) {
    if (list != null && !list.isEmpty()) {
        return list.get(0);
    }
    return null;
}
```

The <T> before the return type specifies that this method is a generic method with type parameter T.

Bounded Type Parameters ->
You can restrict the types that can be used as generic parameters using bounded type parameters:

```
public <T extends Number> double sumOfList(List<T> list) {
    double sum = 0.0;
    for (T item : list) {
        sum += item.doubleValue();
    }
    return sum;
}
```
Here, <T extends Number> means that T must be a subtype of Number.

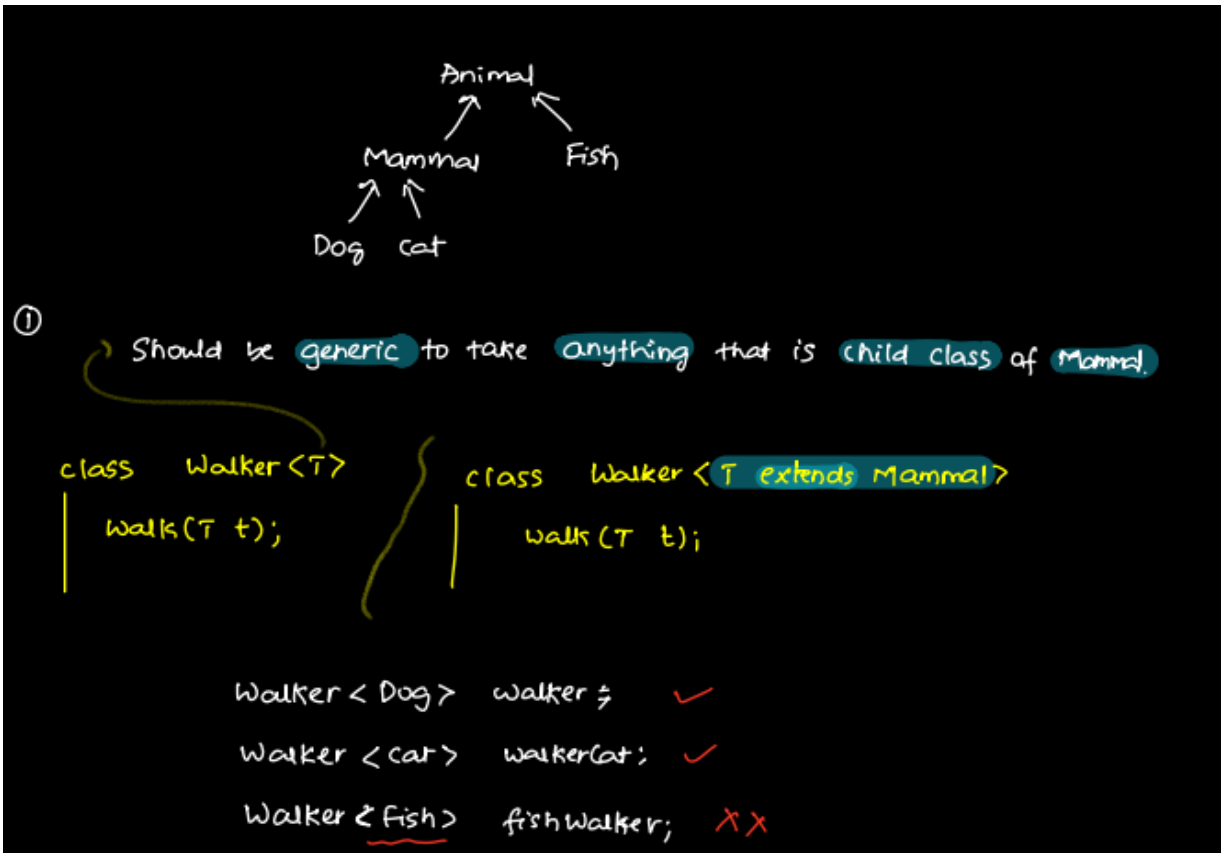
Wildcards:
Wildcards allow you to work with unknown types in a flexible way:

```
public void printList(List<?> list) {
    for (Object item : list) {
        System.out.println(item);
    }
}
```

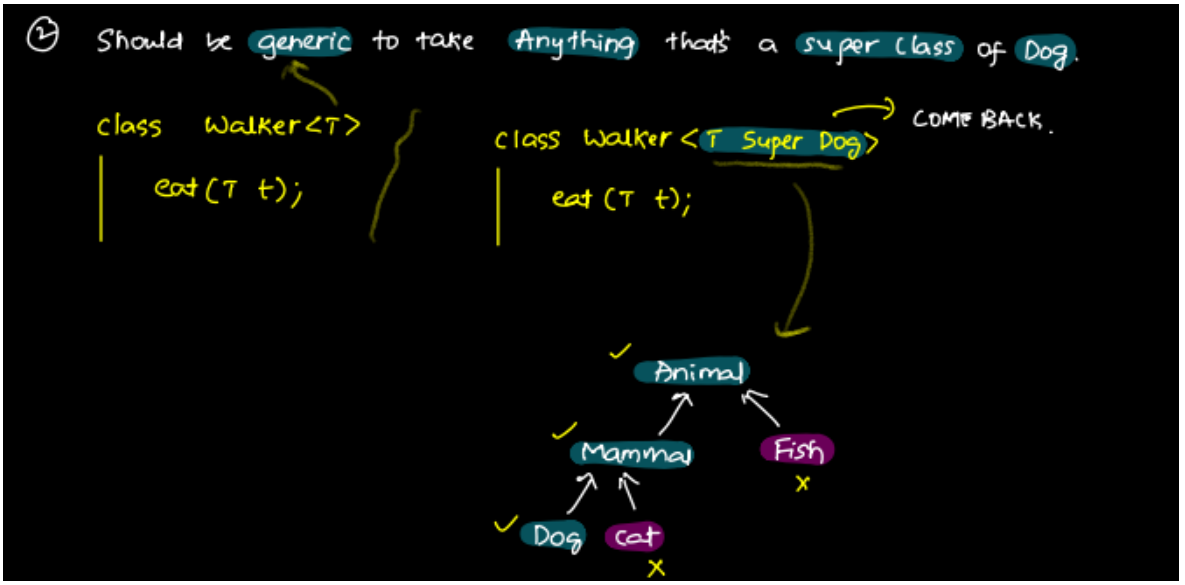
The <?> is a wildcard that represents an unknown type.

Usage of generics helps improve code reusability, maintainability, and type safety in Java. It enables you to create classes and methods that are not tied to specific data types, promoting more flexible and robust code.

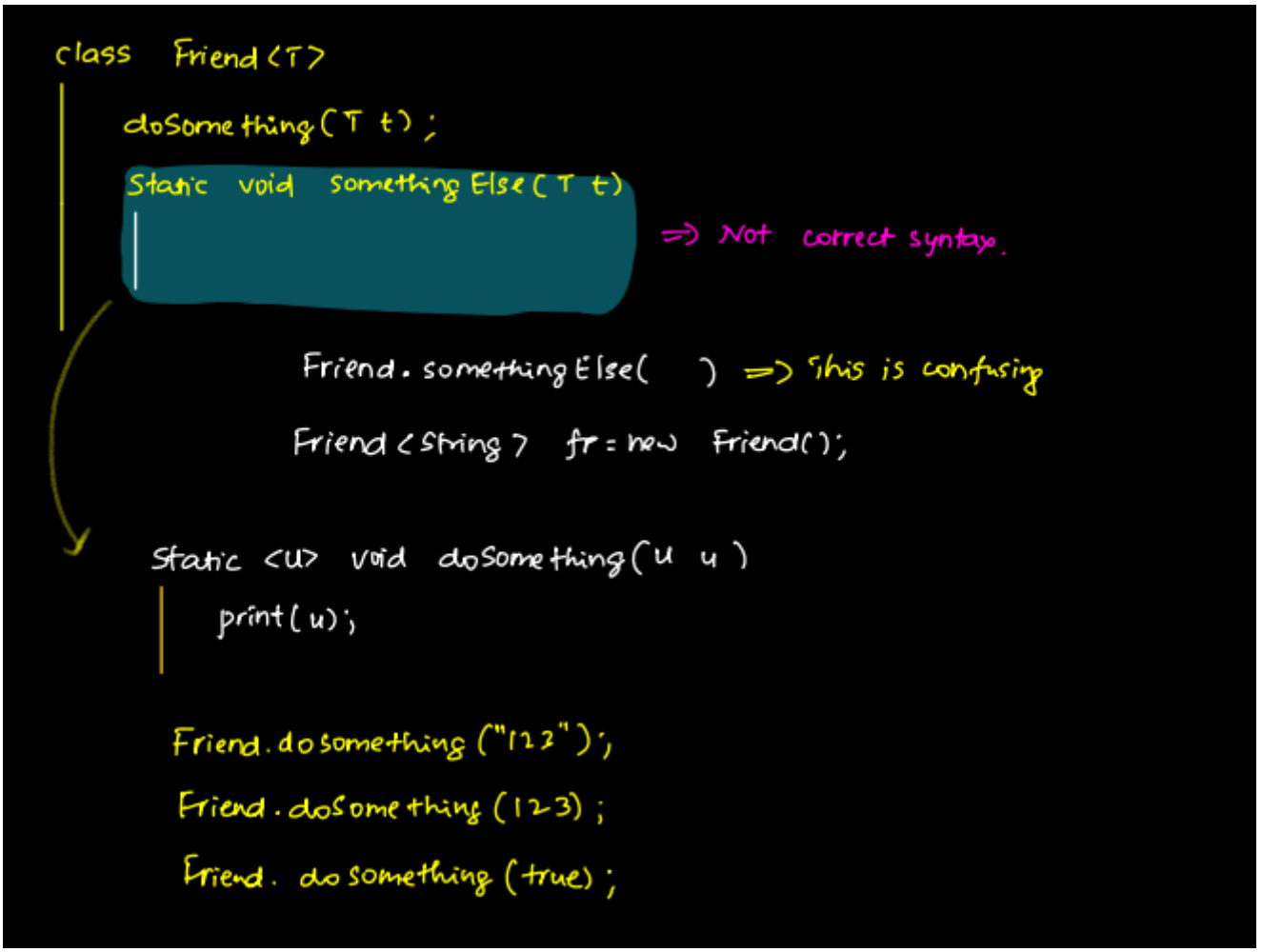
Inheritance in generics



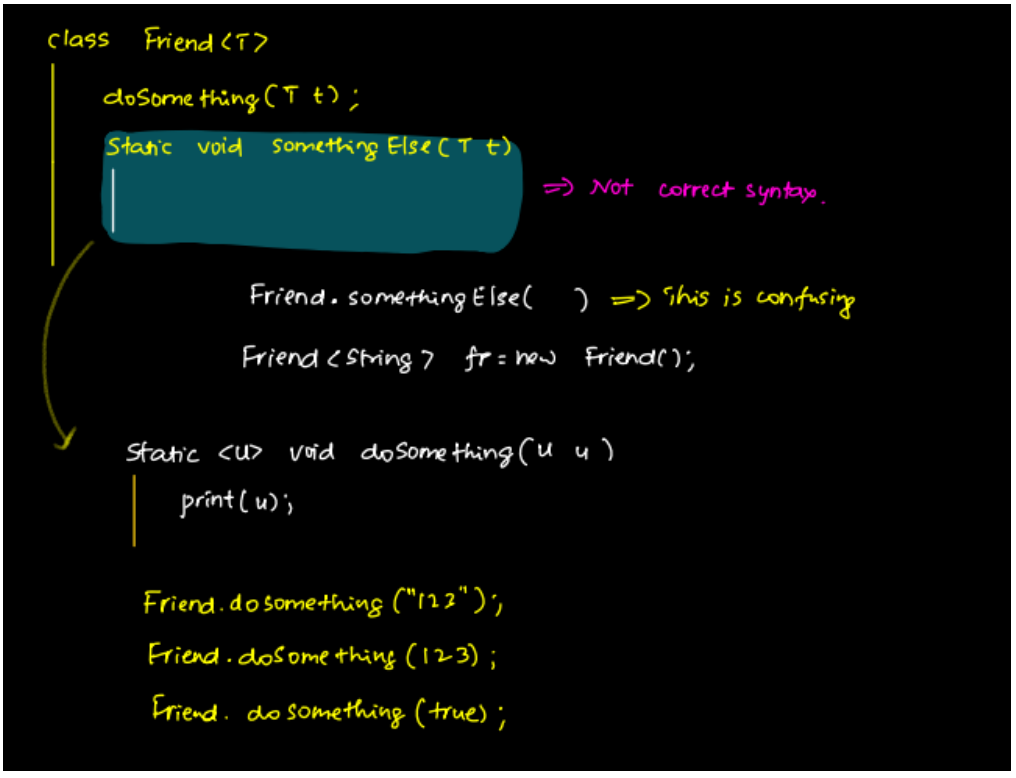
Inheritance super class interface only in generics



Inheritance super class interface only in generics



Static in inheritance -> We can not instantiate static methods because they belong to the class. Therefore, we can not use generics using the regular generic syntax to obtain generics in methods, as shown below.



Type Erasure -> The type information is erased during compilation to ensure backward compatibility with older code that doesn't use generics. At runtime, generic types are replaced with their bound or with Object. This process is known as type erasure.

```
public class MyGenericClass<T> {
    private T data;
    public void setData(T data) {
        this.data = data;
    }

    public T getData() {
        return data;
    }
}
```

At runtime, the type T is erased, and the compiled code effectively behaves as if it were:

```
public class MyGenericClass {
    private Object data;

    public void setData(Object data) {
        this.data = data;
    }

    public Object getData() {
        return data;
    }
}
```

Java Advanced Concepts - 2 [Collections] - Dec 30

- **Collections intro**
- **Collection interface**
- **Collection heirarchy**
- **List interface**
- **ArrayList**
- **Concurrent Modification Exception**
- **Linked List**
- **Vector and Stack**
- **Comparable and Comparator**

Collection Framework -> The Collection Framework in Java is a set of classes and interfaces that provide a comprehensive and unified architecture for handling and manipulating collections of objects.
It includes interfaces like Collection, List, Set, Queue, and Map, along with their various implementations, such as ArrayList, LinkedList, HashSet, HashMap, and more.
The framework provides a common set of methods and conventions for working with different types of collections.

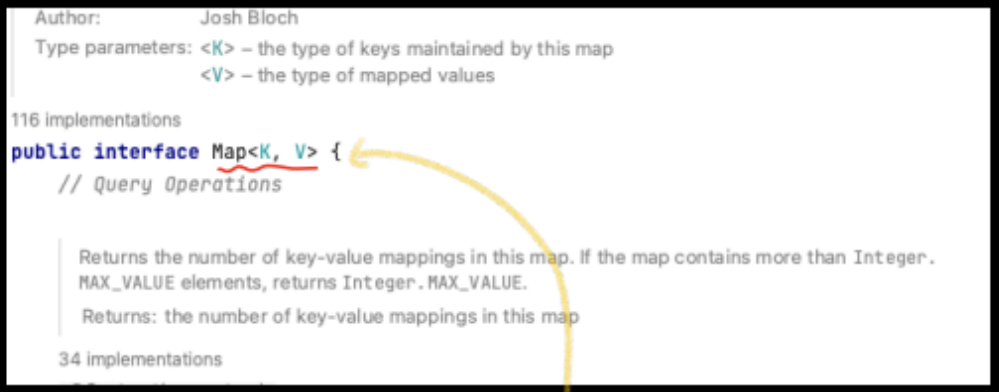
Collection Interface -> The Collection interface is a part of the Java Collections Framework.
It is the root interface for all collection classes.
It defines the basic methods that all collections will have, such as add, remove, contains, and size.
Other interfaces like List, Set, and Queue extend the Collection interface, inheriting its methods and adding additional ones specific to their functionalities.
Classes like ArrayList or HashSet implement these interfaces, providing concrete implementations of the methods defined in the Collection interface.

Why collection framework ?

Java implemented most commonly DS's, so that it's tested and can be re-used by everyone.

Array, List, Set, Map.
Heaps, queues, stacks

Collection is a container of **one** type.



This is a container of **two** types.

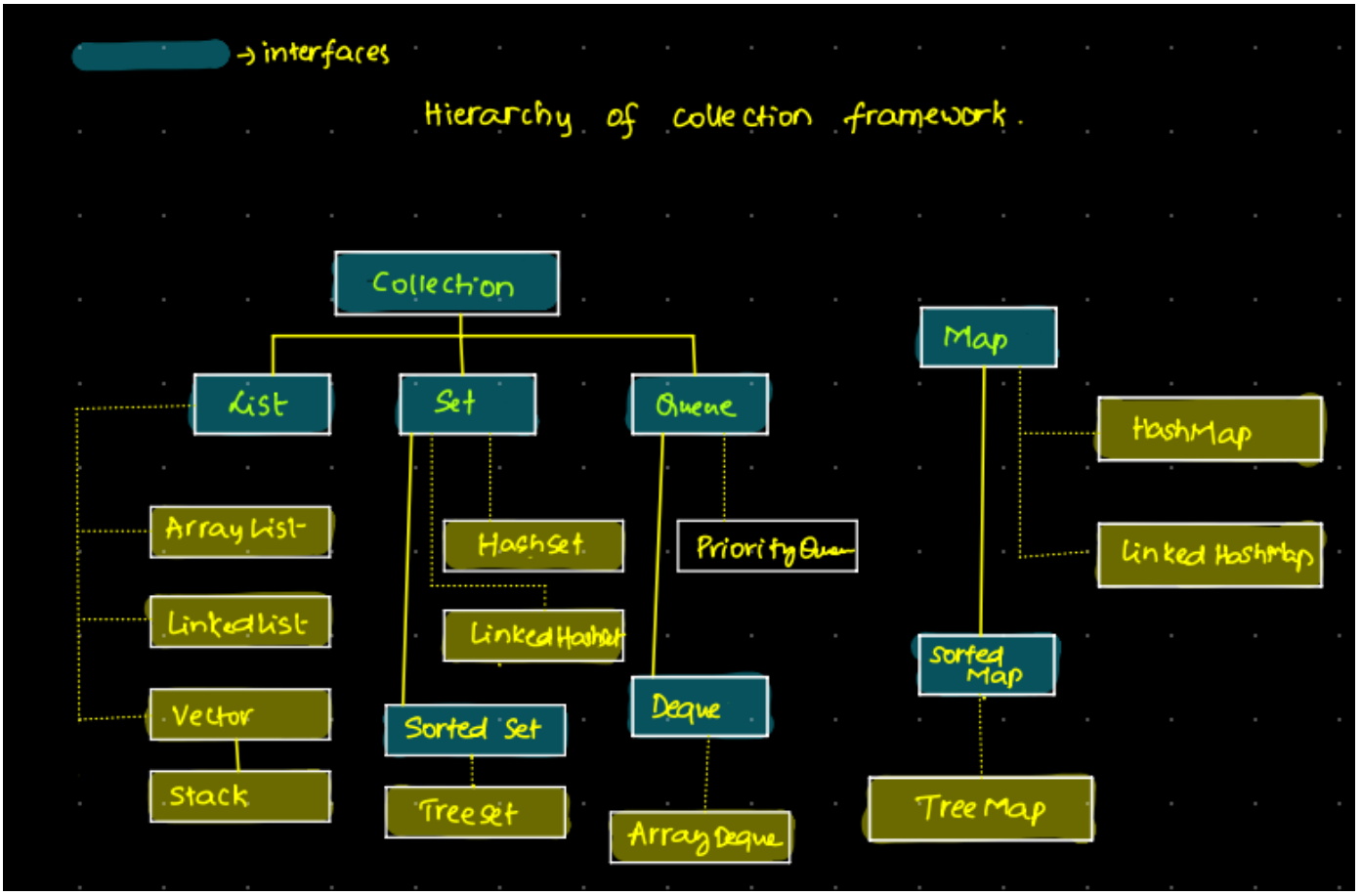
=> Hence map doesn't come under collection interface. However, its part of collection framework.

Methods In Collection Interface

Collection.java

Inherited members (Ctrl+F12) Anonymous Classes (Ctrl+I) Lambdas (Ctrl+L)

- Collection
  - add(E): boolean
  - addAll(Collection<? extends E>): boolean
  - clear(): void
  - contains(Object): boolean
  - containsAll(Collection<?>): boolean
  - equals(Object): boolean ↑Object
  - hashCode(): int ↑Object
  - isEmpty(): boolean
  - iterator(): Iterator<E> ↑Iterable
  - parallelStream(): Stream<E>
  - remove(Object): boolean
  - removeAll(Collection<?>): boolean
  - removeIf(Predicate<? super E>): boolean
  - retainAll(Collection<?>): boolean
  - size(): int
  - spliterator(): Spliterator<E> ↑Iterable
  - stream(): Stream<E>
  - toArray(): Object[]
  - toArray(IntFunction<T[]>): T[]
  - toArray(T[]): T[]



List Interface -> A list is an ordered collection that allows duplicate elements. It is 0-based, meaning the indexing starts from 0. The elements in the list can be accessed using their index, and the list interface provides methods to search for elements within the collection

---

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements `e1` and `e2` such that `e1.equals(e2)`, and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

The `List` interface places additional stipulations, beyond those specified in the `Collection` interface, on the contracts of the `iterator`, `add`, `remove`, `equals`, and `hashCode` methods. Declarations for other inherited methods are also included here for convenience.

The `List` interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based. Note that these operations may execute in time proportional to the index value for some implementations (the `LinkedList` class, for example). Thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.

The `List` interface provides a special iterator, called a `ListIterator`, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the `Iterator` interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

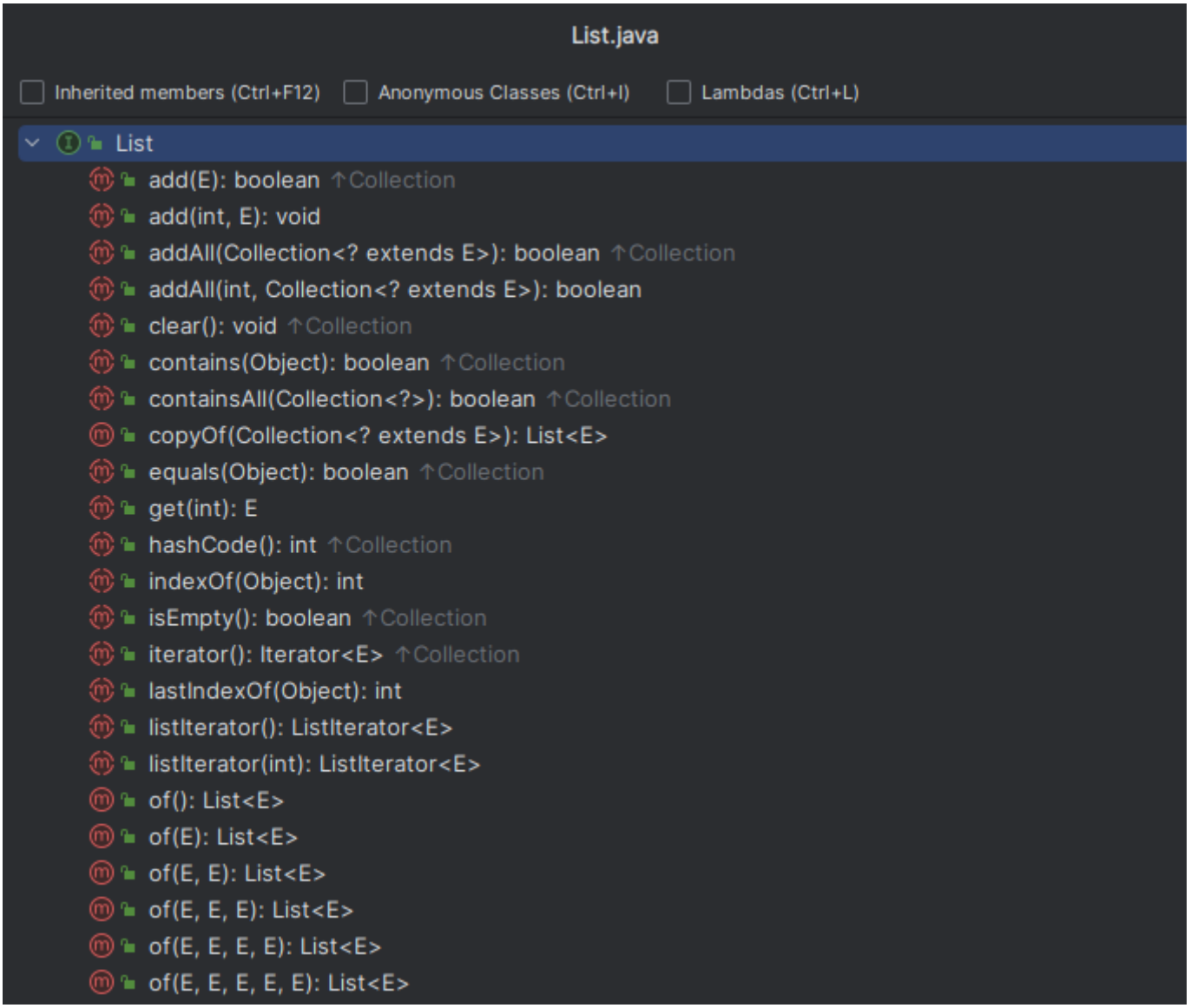
The `List` interface provides two methods to search for a specified object. From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.

The `List` interface provides two methods to efficiently insert and remove multiple elements at an arbitrary point in the list.

Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a list.

Some list implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the list may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

List Interface Method



ArrayList -> The amortized time complexity of ArrayList is on average O(1) for adding elements. However, if additional space is needed, and the current capacity is reached, the operation becomes O(n) because it involves copying and creating a new array in the ArrayList. It's important to note that ArrayList is not synchronized.

List<Integer> numbers = new ArrayList<>();

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    Default initial capacity.
    2 usages
    private static final int DEFAULT_CAPACITY = 10;
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. This class is roughly equivalent to `Vector`, except that it is unsynchronized.

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

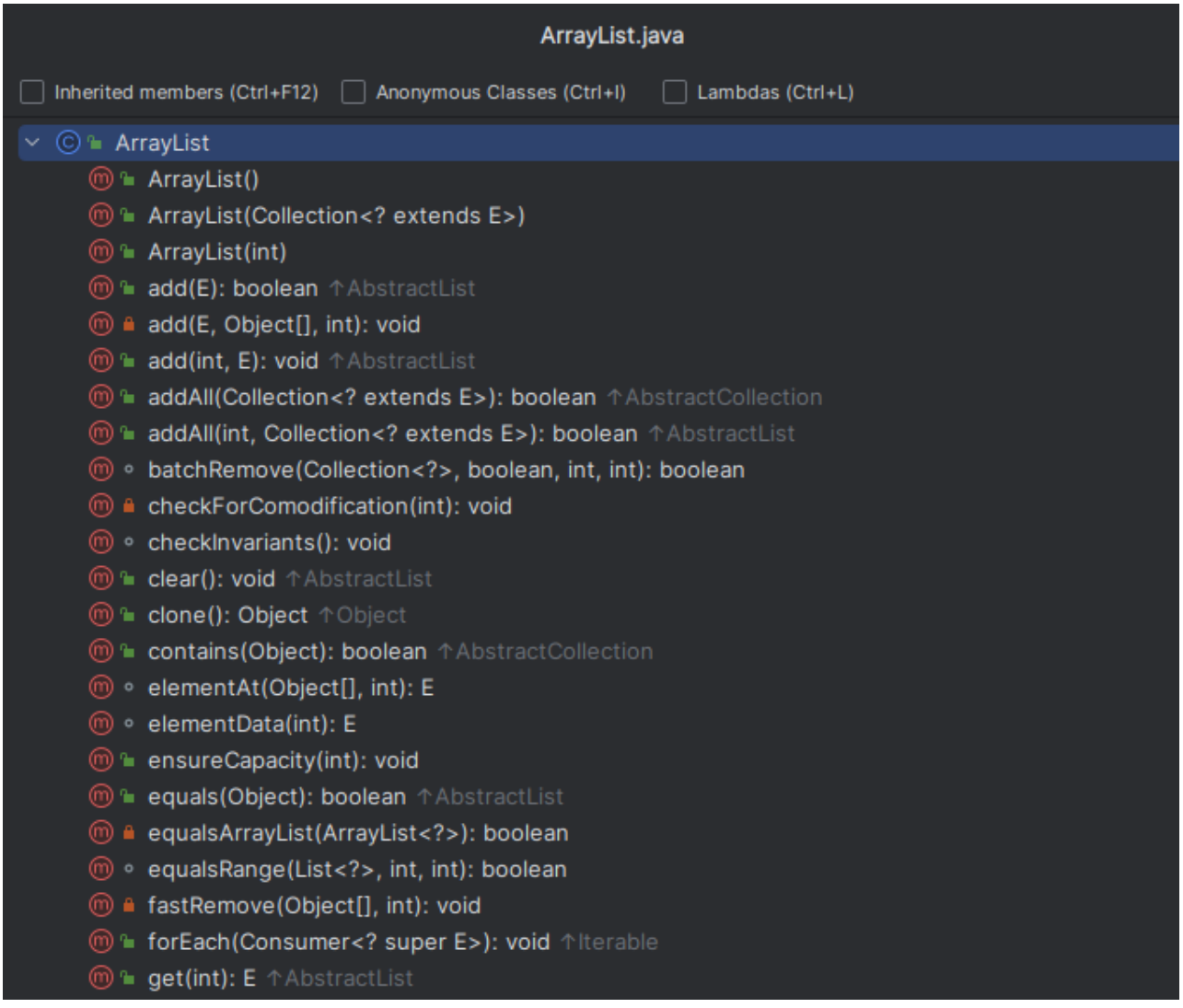
Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an `ArrayList` instance before adding a large number of elements using the `ensureCapacity` operation. This may reduce the amount of incremental reallocation.

Note that this implementation is not synchronized. If multiple threads access an `ArrayList` instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

Create ArrayList Press (CTRL+Click) for entering in ArrayList Class and CTRL + F12 to see list of methods



LinkedList -> A LinkedList is a data structure in Java that represents a linear collection of elements. Unlike an ArrayList, a LinkedList does not use a contiguous block of memory for storage. Instead, it consists of nodes, each containing data and a reference (or link) to the next node in the sequence. It is not syncronized (Not Thread safe)

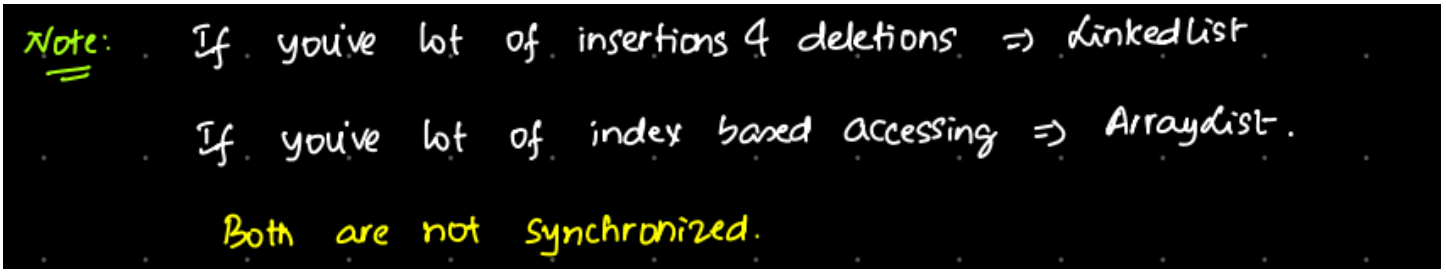
```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
    29 usages
    transient int size = 0;
```

Doubly-linked list implementation of the `List` and `Deque` interfaces. Implements all optional list operations, and permits all elements (including `null`).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Note that this implementation is not synchronized. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new LinkedList(...));
```



Vector -> A Vector in Java is a dynamic array-like data structure that belongs to the Java Collections Framework. It is part of the legacy collections and is synchronized, which means it is thread-safe. However, due to its synchronization, Vector operations might be slower than those of non-synchronized collections.

```
public class Vector<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
```

The `Vector` class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a `Vector` can grow or shrink as needed to accommodate adding and removing items after the `Vector` has been created.

Each vector tries to optimize storage management by maintaining a `capacity` and a `capacityIncrement`. The `capacity` is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of `capacityIncrement`. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation.

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the vector is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. The `Enumerations` returned by the `elements` method are *not* fail-fast; if the Vector is structurally modified at any time after the enumeration is created then the results of enumerating are undefined.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

As of the Java 2 platform v1.2, this class was retrofitted to implement the `List` interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, `Vector` is synchronized. If a thread-safe implementation is not needed, it is recommended to use `ArrayList` in place of `Vector`.

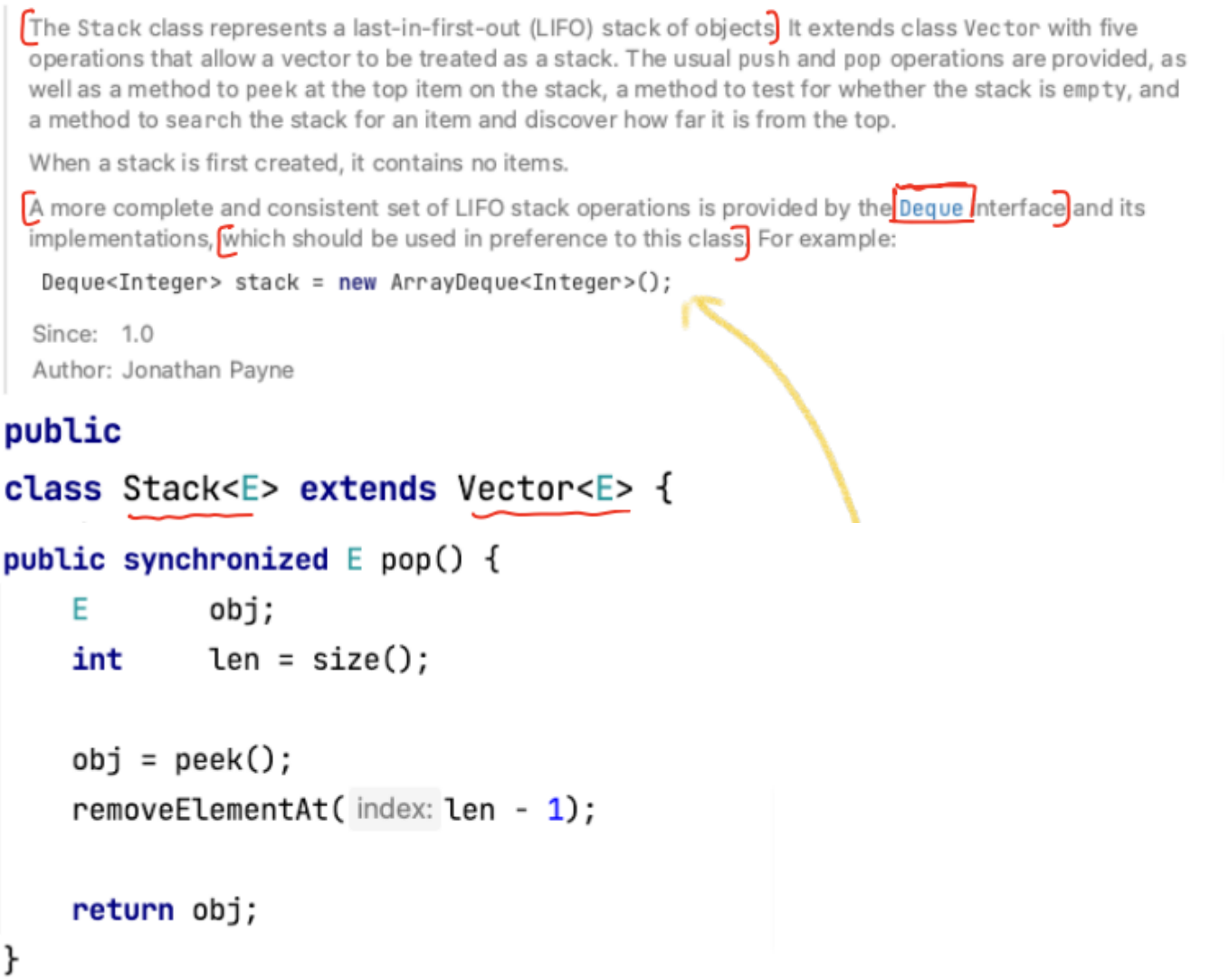
```
public synchronized void copyInto(Object[] anArray) { System.arraycopy(elementData, srcPos:
```

Trims the capacity of this vector to be the vector's current size. If the capacity of this vector is larger than its current size, then the capacity is changed to equal the size by replacing its internal data array, kept in the field `elementData`, with a smaller one. An application can use this operation to minimize the storage of a vector.

```
public synchronized void trimToSize() {
    modCount++;
    int oldCapacity = elementData.length;
    if (elementCount < oldCapacity) {
        elementData = Arrays.copyOf(elementData, elementCount);
    }
}
```

Note -> Synchronization will slow down performance, If not needed, use ArrayList

Stack -> It is a collection that implements the Last-In, First-Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. The Stack class extends the Vector class and is part of the Java Collections Framework.



```
The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five
operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as
well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and
a method to search the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its
implementations, which should be used in preference to this class. For example:

 Deque<Integer> stack = new ArrayDeque<Integer>();

 Since:  1.0
 Author: Jonathan Payne
public
class Stack<E> extends Vector<E> {
public synchronized E pop() {
    E       obj;
    int     len = size();

    obj = peek();
    removeElementAt( index: len - 1);

    return obj;
}
```

Note -> Keep in mind that Stack is a legacy class and It is Thread Safe, **for more modern usage, Deque (e.g., LinkedList) is often preferred.**

Set => The Set interface in Java is part of the Java Collections Framework and extends the Collection interface. It represents a collection of unique elements, meaning that no two elements in a Set can be equal.

Here are some key characteristics of the Set interface:

- **No Duplicate Elements**
- **Unordered Collection**
- **Common Implementations**
  Common implementations of the Set interface include HashSet, TreeSet, and LinkedHashSet. Each has its characteristics, such as the use of hashing (HashSet), natural ordering (TreeSet), and maintaining insertion order (LinkedHashSet).

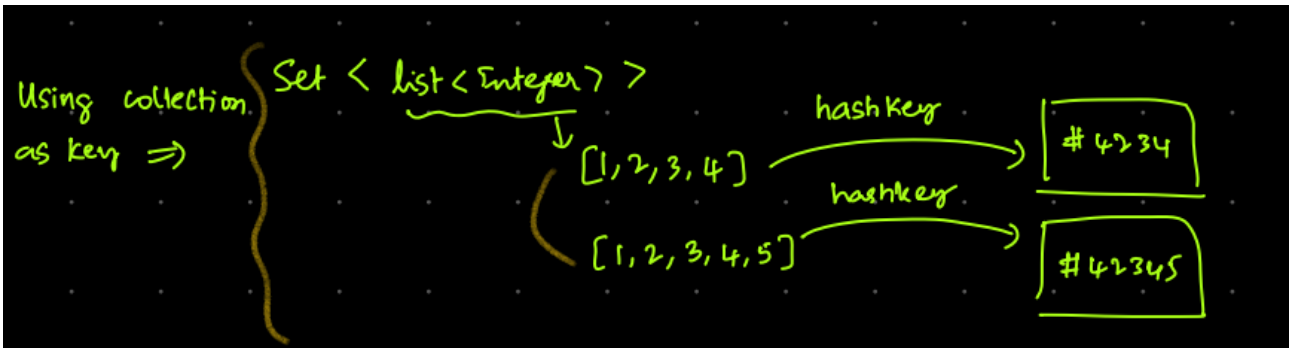
```
public interface Set<E> extends Collection<E> {
```

A collection that contains no duplicate elements. More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

The `Set` interface places additional stipulations, beyond those inherited from the `Collection` interface, on the contracts of all constructors and on the contracts of the `add`, `equals` and `hashCode` methods. Declarations for other inherited methods are also included here for convenience. (The specifications accompanying these declarations have been tailored to the `Set` interface, but they do not contain any additional stipulations.)

The additional stipulation on constructors is, not surprisingly, that all constructors must create a set that contains no duplicate elements (as defined above).

Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.



- Set
  - add(E): boolean ↑Collection
  - addAll(Collection<? extends E>): boolean ↑Collection
  - clear(): void ↑Collection
  - contains(Object): boolean ↑Collection
  - containsAll(Collection<?>): boolean ↑Collection
  - copyOf(Collection<? extends E>): Set<E>
  - equals(Object): boolean ↑Collection
  - hashCode(): int ↑Collection
  - isEmpty(): boolean ↑Collection
  - iterator(): Iterator<E> ↑Collection
  - of(): Set<E>
  - of(E): Set<E>
  - of(E, E): Set<E>
  - of(E, E, E): Set<E>
  - of(E, E, E, E): Set<E>
  - of(E, E, E, E, E): Set<E>
  - of(E, E, E, E, E, E): Set<E>
  - of(E, E, E, E, E, E, E, ...): Set<E>
  - of(E, E, E, E, E, E, E, ...): Set<E>
  - of(E, E, E, E, E, E, E, ...): Set<E>
  - of(E...): Set<E>

Hashset -> The HashSet class in Java is an implementation of the Set interface, part of the Java Collections Framework. It uses a hash table to store elements, providing constant-time performance for basic operations such as adding, removing, and checking for the presence of elements.

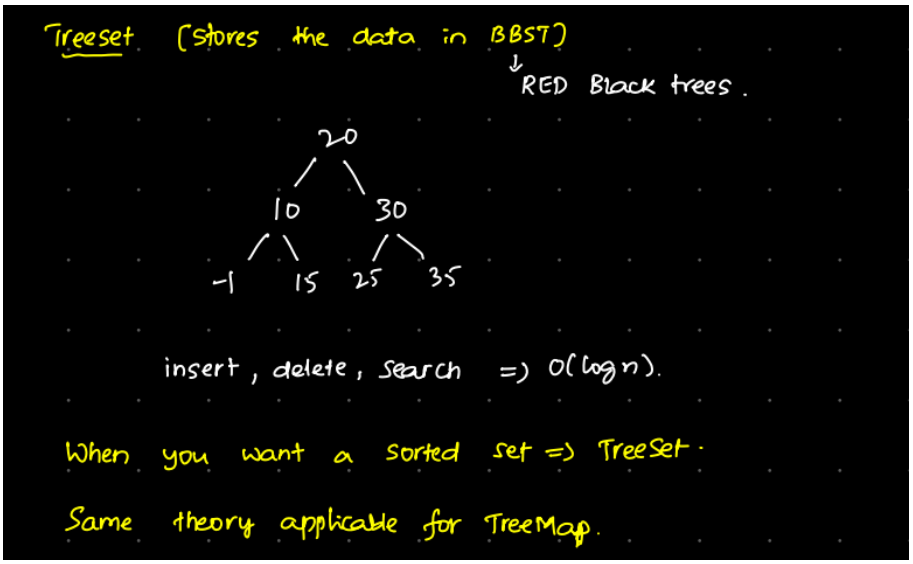
The HashSet implementation is chosen for its fast retrieval and efficient storage of unique elements. It is not maintend order but LinkedHashSet Maintain order

```
HashSet<Integer> integerHashSet = new HashSet<>();
integerHashSet.add(10);
integerHashSet.add(20);
integerHashSet.add(30);
integerHashSet.add(40);

LinkedHashSet<Integer> integerLinkedHashSet = new LinkedHashSet<>();
integerLinkedHashSet.add(10);
integerLinkedHashSet.add(20);
integerLinkedHashSet.add(30);
integerLinkedHashSet.add(40);

System.out.println("integerHashSet = " + integerHashSet);
System.out.println("integerLinkedHashSet = " + integerLinkedHashSet);
```

```
/Users/keerthikumarsg/Library/Java/JavaVirtualM
integerHashSet = [20, 40, 10, 30]
integerLinkedHashSet = [10, 20, 30, 40]

Process finished with exit code 0
```

Treeset -> The TreeSet class in Java is an implementation of the SortedSet interface, which extends the Set interface. It uses a red-black tree data structure to store elements, maintaining them in sorted order. Elements are ordered based on their natural ordering or a custom comparator provided at the time of instantiation.

The TreeSet implementation automatically maintains the elements in sorted order based on their natural ordering (lexicographical order for strings in this case).



Que -> Queue is an interface that represents a collection designed for holding elements prior to processing. Queues typically follow the First-In-First-Out (FIFO) principle, meaning that the first element added is the first one to be removed. The Queue interface extends the Collection interface and is part of the Java Collections Framework.

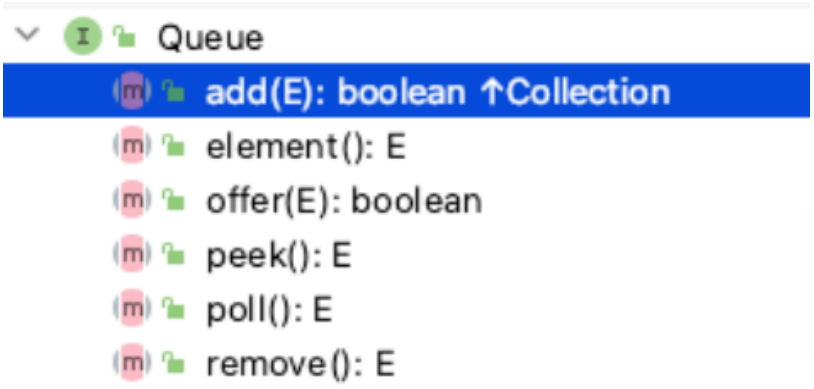
A collection designed for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either `null` or `false`, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted `Queue` implementations; in most implementations, insert operations cannot fail.

Summary of Queue methods

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the *head* of the queue is that element which would be removed by a call to `remove()` or `poll()`. In a FIFO queue, all new elements are inserted at the *tail* of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties.

```
public interface Queue<E> extends Collection<E> {
```



Deque -> The Deque interface in Java represents a double-ended queue, which allows the insertion and removal of elements from both ends. The name "Deque" is an abbreviation of "double-ended queue." It extends the Queue interface and provides additional methods for insertion, removal, and inspection of elements at both ends.

```
public interface Deque<E> extends Queue<E> {
```

A linear collection that supports element insertion and removal at both ends. The name *deque* is short for "double ended queue" and is usually pronounced "deck". Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

This interface defines methods to access the elements at both ends of the deque. Methods are provided to insert, remove, and examine the element. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either `null` or `false`, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Deque implementations; in most implementations, insert operations cannot fail.

The twelve methods described above are summarized in the following table:

Summary of Deque methods

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Dequeue as a Queue and Stack

This interface extends the `Queue` interface. When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. Elements are added at the end of the deque and removed from the beginning. The methods inherited from the `Queue` interface are precisely equivalent to `Deque` methods as indicated in the following table:

Comparison of Queue and Deque methods

`Queue` Method	Equivalent `Deque` Method
`add(e)`	`addLast(e)`
`offer(e)`	`offerLast(e)`
`remove()`	`removeFirst()`
`poll()`	`pollFirst()`
`element()`	`getFirst()`
`peek()`	`peekFirst()`

Queue

Deques can also be used as LIFO (Last-In-First-Out) stacks. This interface should be used in preference to the legacy `Stack` class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque. Stack methods are equivalent to `Deque` methods as indicated in the table below:

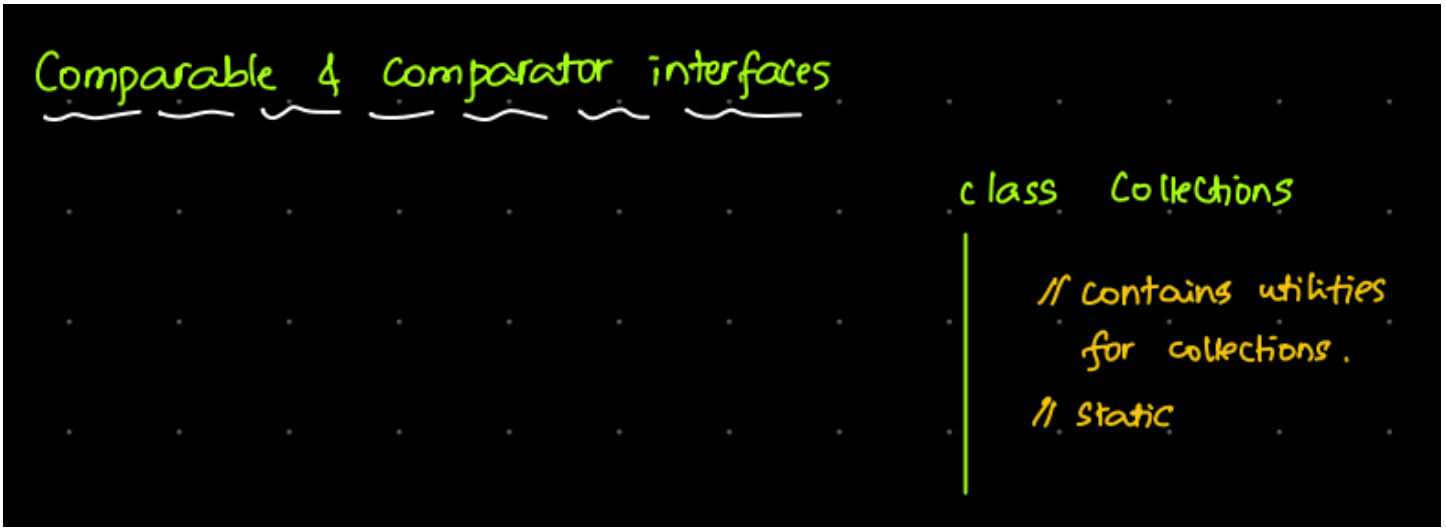
Comparison of Stack and Deque methods

Stack Method	Equivalent `Deque` Method
`push(e)`	`addFirst(e)`
`pop()`	`removeFirst()`
`peek()`	`getFirst()`

Stack

Note that the `peek` method works equally well when a deque is used as a queue or a stack; in either case, elements are drawn from the beginning of the deque.

Comparable and Comparator



Comparable -> The Comparable interface in Java is used to define the natural ordering of objects. It contains a single method, compareTo, which compares the current object with another object, specifying the default sorting behavior.

When a class implements the Comparable interface, it can be sorted using the natural order defined by the compareTo method. This allows objects of that class to be easily sorted using methods like Collections.sort().

Here's a simple example of a class implementing Comparable:

```
public class Student implements Comparable<Student> {
    private int id;
    private String name;

    // Constructor and other methods

    @Override
    public int compareTo(Student otherStudent) {
        return this.id - otherStudent.id;
    }
}
```

In Main class

```
List<Student> studentList = new ArrayList<>();
// Add Student objects to the list

Collections.sort(studentList); // Sorts the list using natural order (based on ID in this case)
```

Comparator -> The Comparator interface is used for defining custom ordering logic for objects. It allows developers to implement various comparison strategies for sorting elements in a collection. The compare method in the Comparator interface is responsible for comparing two objects based on a specified criterion.

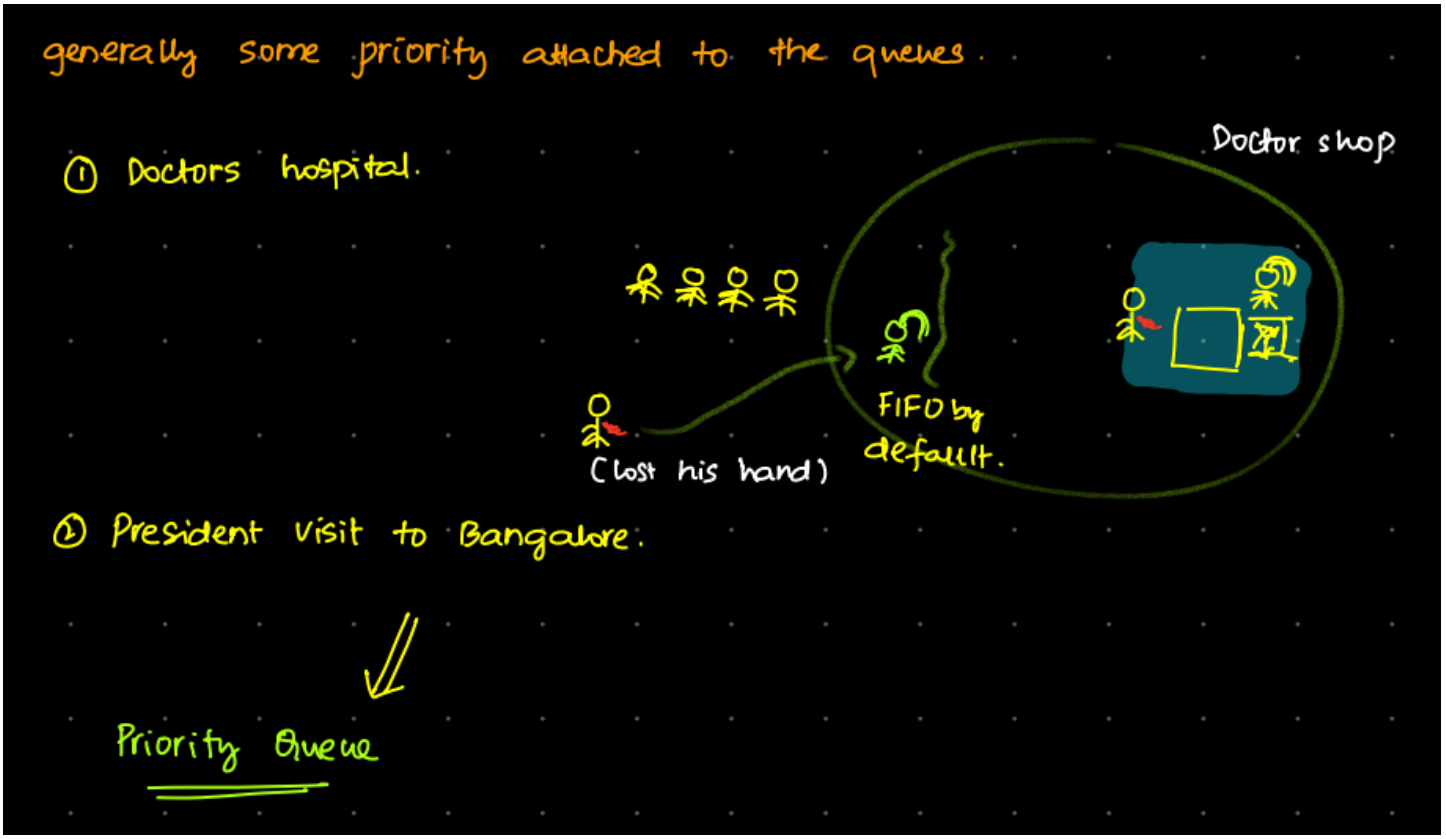
Here's an example of using Comparator to sort a list of Student objects based on their names:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class StudentComparatorExample {
    public static void main(String[] args) {
        List<Student> studentList = new ArrayList<>();
        // Add Student objects to the list

        // Sorting based on custom comparator (by name)
        Collections.sort(studentList, new NameComparator());

        // Now, the studentList is sorted based on the custom comparator
    }
}

class Student {
    private int id;
    private String name;

    // Constructor and other methods

    // Getter and Setter methods

    // Other fields and methods

}

class NameComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.getName().compareTo(s2.getName());
    }
}
```

What the difference between comparable and comparator?
Comparable is limited to a single sorting logic, offering default sorting. It called natural sorting order of the class. However, when a custom sorting logic other than the default is required, we turn to Comparator. Utilizing Comparator, we can provide a custom sorting method or class. For example:

```
Collections.sort(studentList, new Comparator<Student>() {
    public int compare(Student s1, Student s2) {
        return s1.getId() - s2.getId();
    }
});
```

In this example, the Comparator is used to sort a list of Student objects based on their IDs using a custom comparison logic.



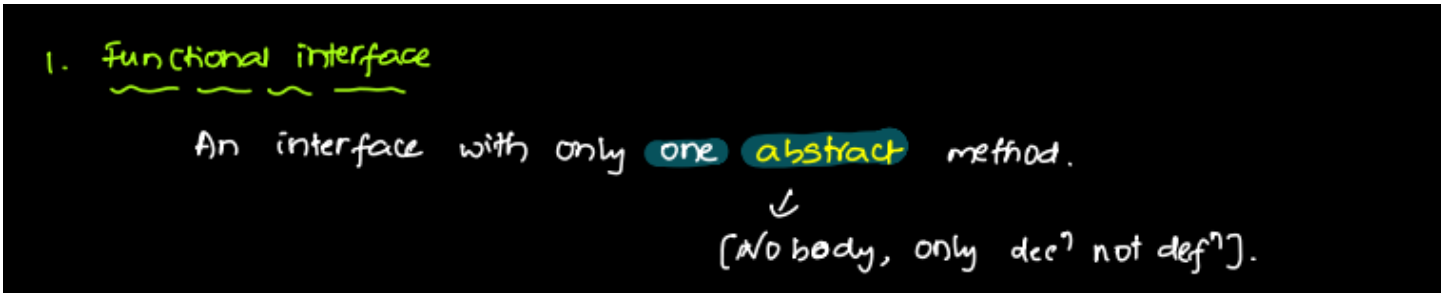
Java Advanced Concepts - 3 [Streams and Lamdas] - Jan 2 - 24

- **Functional interfaces**
  - Can we have body in methods of interfaces ? - default and static methods from java 8
  - @FunctionalInterface annotation
  - Functional interfaces that you already know - Runnable, Callable, Comparable, Comparator
  - Functional interfaces that you should know - Consumer and Bi, Predicate and Bi, Function and Bi
- **Can you create an instance of Interface ?**

- **Lambda introduced here**
  - Predicate and Lambdas clubbed together
  - Threads using Lambdas
- **Streams intro**
  - .filter(predicate) Intermediate
  - .map(function) intermediate
  - Terminal operation, it'll close the stream

Functional interfaces -> a functional interface is an interface that contains only one abstract method. Functional interfaces are a key concept in functional programming and are often used in conjunction with lambda expressions. The @FunctionalInterface annotation is optional but can be used to indicate that an interface is intended to be functional.

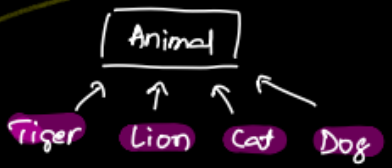
```
@FunctionalInterface
interface MyFunctionalInterface {
    void myAbstractMethod();

    // Additional default or static methods are allowed
    default void myDefaultMethod() {
        System.out.println("Default method implementation");
    }

    static void myStaticMethod() {
        System.out.println("Static method implementation");
    }
}
```



Q: Can we have body in the methods of interfaces?...

From java 8: It's possible using "default" & "static" method.

Why body is required?




```
interface Animal
    void eat()
    void run()
    default void putAnimalInZoo()
        print("Animal is in zoo now");
```

Default methods can be overridden, static methods cannot be overridden.

Used for kind of utility methods.

No need to create an implementation to use static methods.

Q: Can we have default or static methods in FI's?

Yes! You can have any no. of default/static methods.

@FunctionalInterface (Annotation) → functional interface

```
interface I1 {
    void f1();
    default f2()
        // body
```

NEW DEV

```
interface I1 {
    void f1();
    void f3();
    default f2()
        // body
```

This is not a FI.

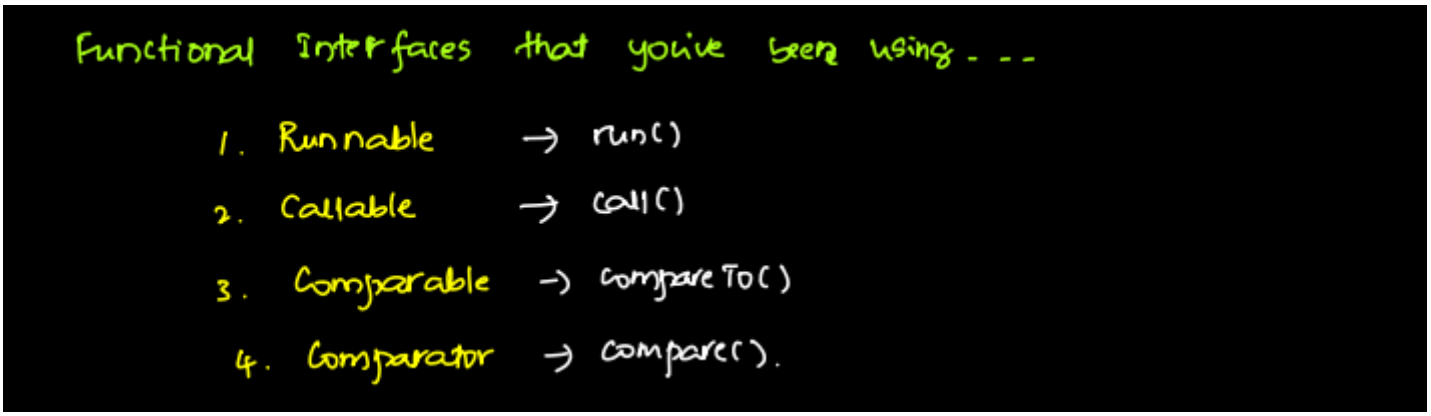
To make sure someone doesn't convert FI's to Non-FI's we use

@FunctionalInterface annotation

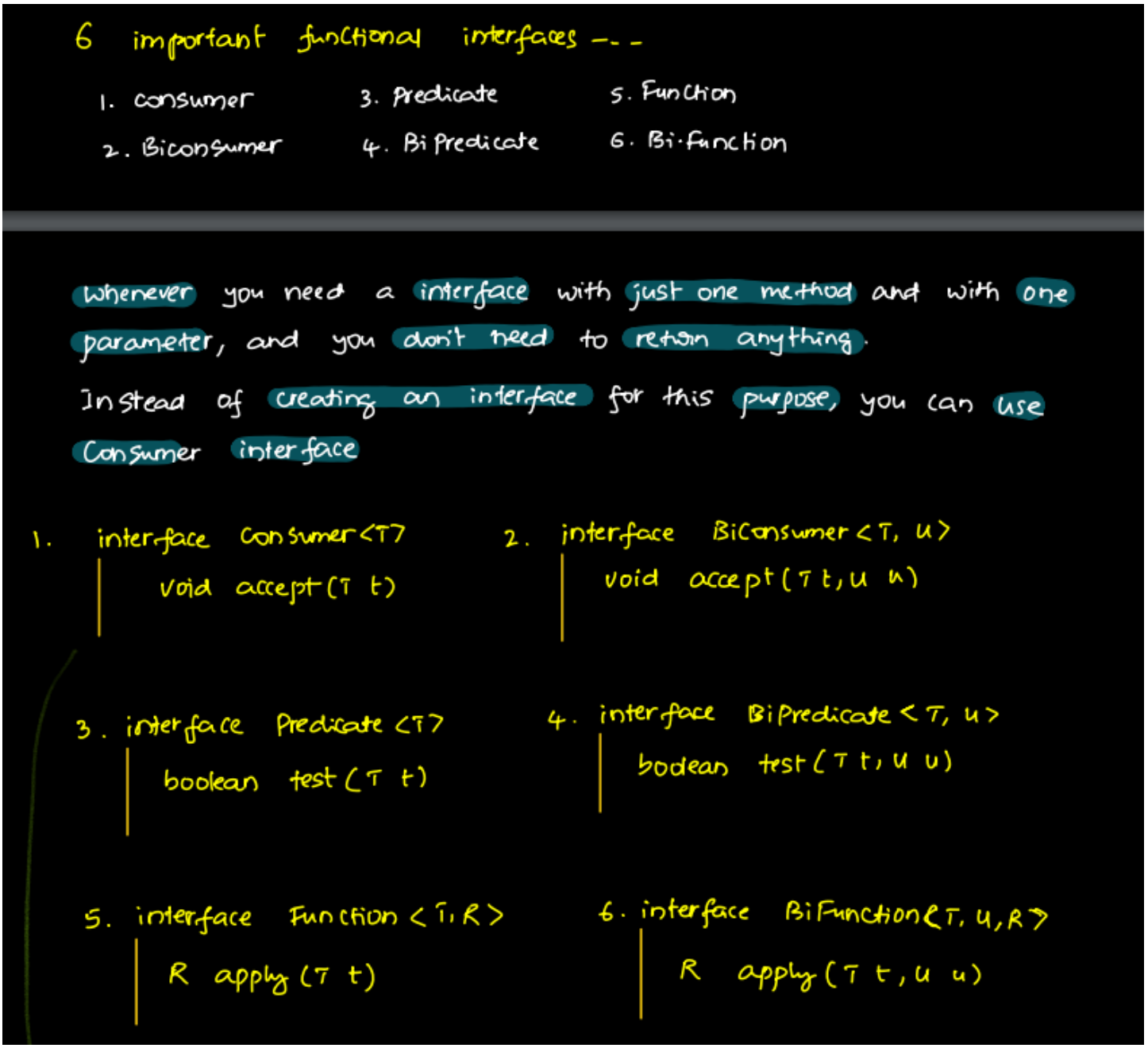
This is not mandatory

Good practice.

Functional interfaces that you already know



Functional interfaces that you should know



Streams -> the Stream API is a powerful and functional programming feature introduced in Java 8 that allows developers to perform operations on sequences of elements. Streams provide a concise and expressive way to process collections of data.

Source: Streams are created from a source, which can be a collection, an array, an I/O channel, or even a generator function.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
Stream<Integer> stream = numbers.stream();
```

Intermediate Operations: Intermediate operations transform a stream into another stream. Common intermediate operations include filter, map, distinct, sorted, and peek.

```
List<Integer> evenSquares = numbers.stream()
                            .filter(n -> n % 2 == 0)
                            .map(n -> n * n)
                            .collect(Collectors.toList());
```

Terminal Operations: Terminal operations produce a result or a side-effect. Examples of terminal operations are forEach, collect, reduce, and count.

```
long count = numbers.stream().filter(n -> n > 2).count();
```

Collectors: The Collectors class provides a set of convenient reduction operations, such as grouping elements, joining them into strings, or aggregating into collections.

```
List<String> strings = Arrays.asList("apple", "banana", "orange");
String result = strings.stream().collect(Collectors.joining(", "));
```

Parallel Streams: Streams can be processed in parallel to take advantage of multi-core processors, improving performance. This is achieved using the parallel() method.

```
long count = numbers.parallelStream().filter(n -> n > 2).count();
```

Streams provide a functional and declarative approach to processing data, promoting cleaner and more concise code. They are particularly useful for handling large datasets, and they seamlessly integrate with lambda expressions, making code more expressive and readable.

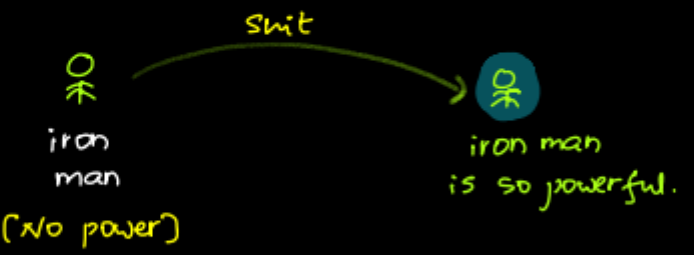
Streams

Wrappers over data source

↓

Collections.

(List, Set, map, etc)



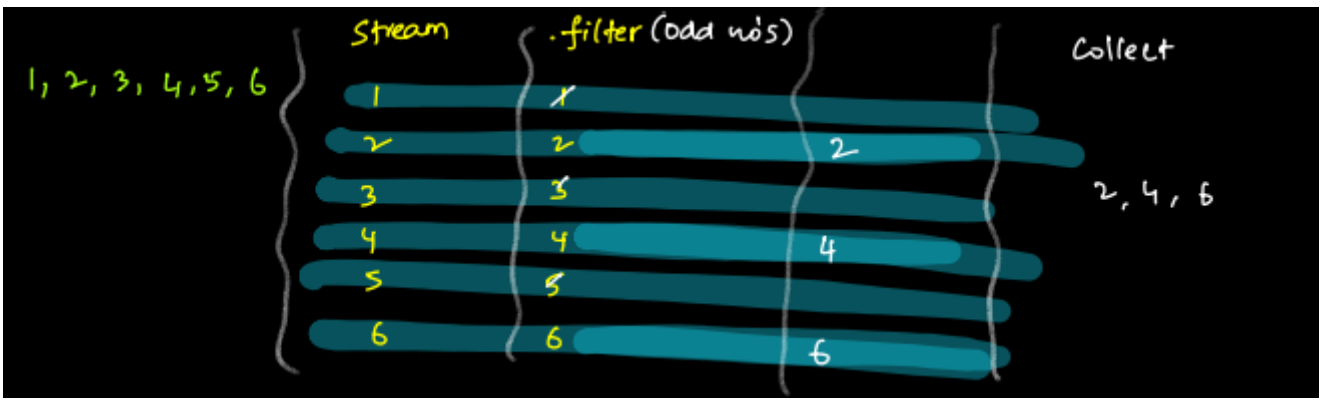
List<Integer> numbers = ——————

numbers. ——— } Methods of list interface.

numbers.stream(). ——— } Methods of Stream.

.filter

```
numbers.stream()  // putting the suit
    .filter(takes a predicate)
    .collect(Collectors.toList()) // removing the suit.
```



To filter odd no's.

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6);
List<Integer> evenNumbers = _____
for(Integer number: numbers)
    if (number%2==0)
        evenNumbers.add(number)
```

evenNos -> [2, 4, 6].

With streams.

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6);

Predicate<Integer> predicate = new Predicate() {
                                    public boolean test(int number)
                                        return number%2==0
                                };

List<Integer> evenNos = numbers.stream()
                          .filter(predicate)
                          .collect(Collectors.toList())
```

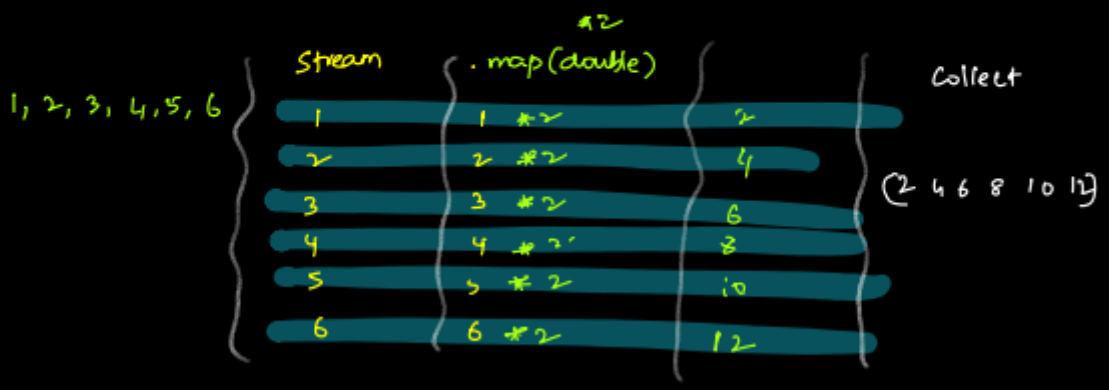
Simplify

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6);

List<Integer> evenNos = numbers.stream()
                          .filter(ele -> ele%2==0)
                          .collect(Collectors.toList())
```

. map.

· filter

```
numbers.stream()  // putting the suit
    · map (takes a function )
    · collect ( Collectors.toList()) // removing the suit.
```

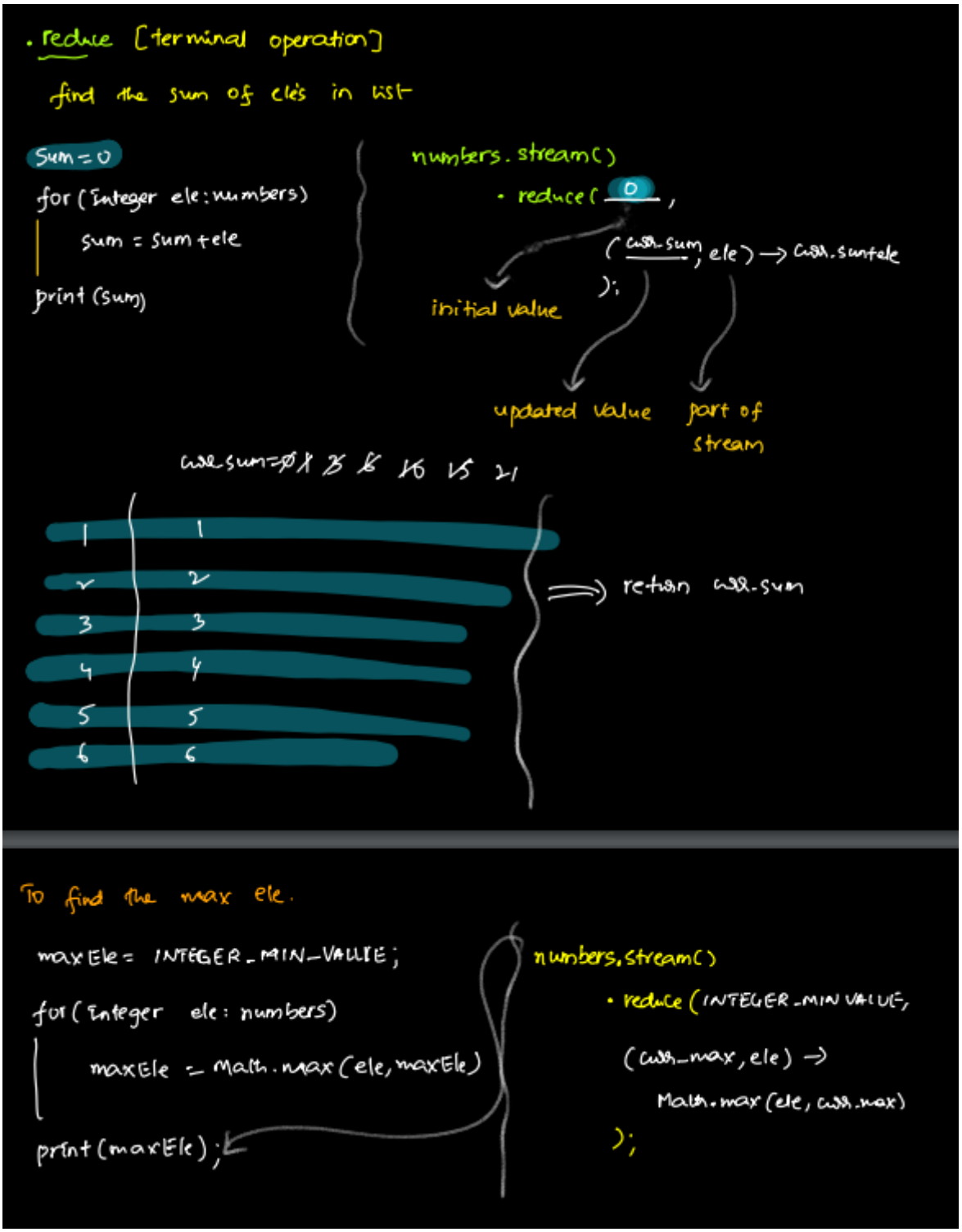



```
Function<Integer, Integer> doubler = new Function() {
    public Integer apply(int n)
    |   return n*2
};
```

```
numbers.stream()
    · map ( doubler)
    · collect(Collectors.toList());
```

Simplify.

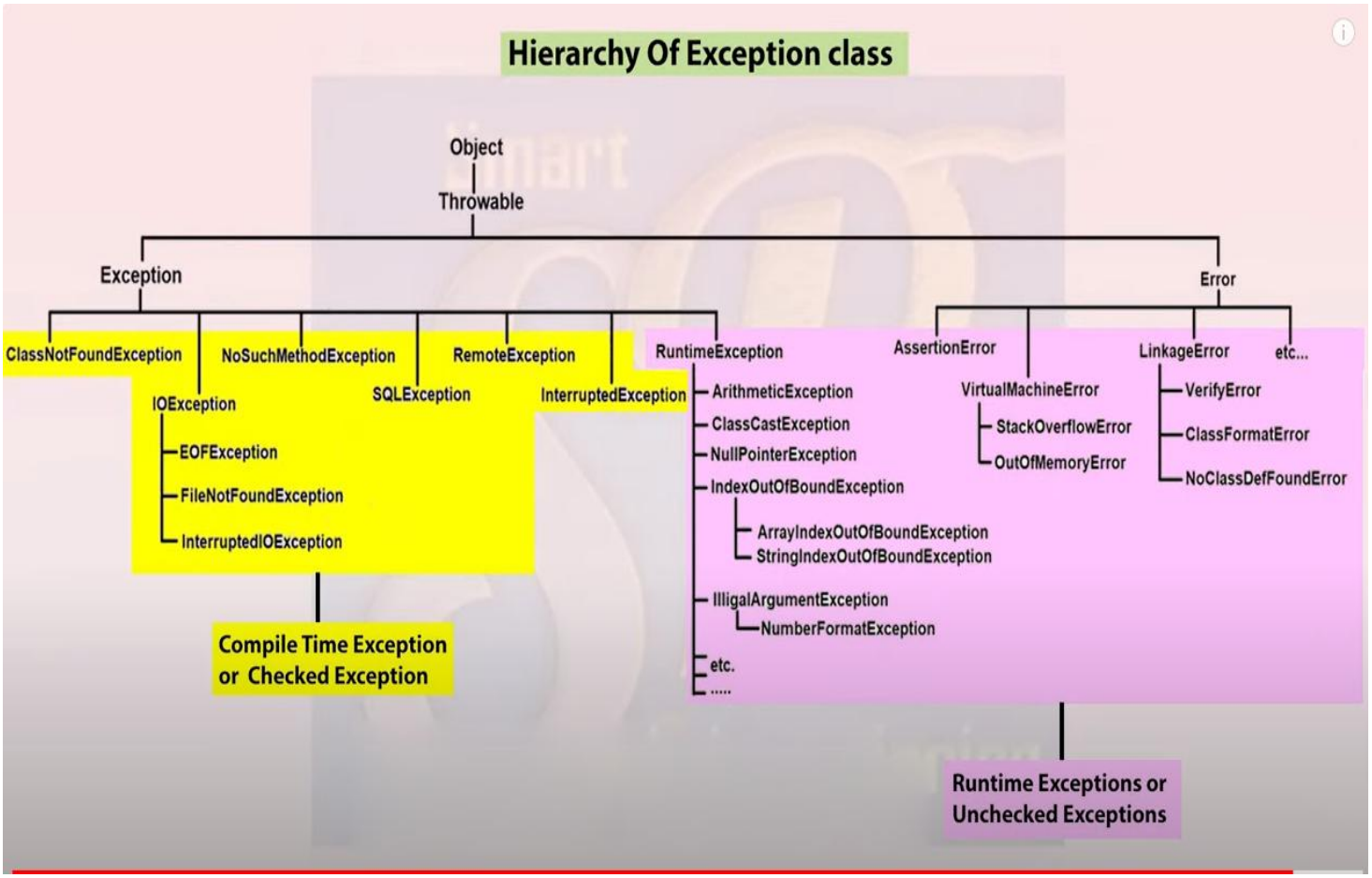
```
numbers.stream()
    - map ( ele -> ele*2)
    · collect(Collectors.toList());
```


. reduce [terminal operation]
find the sum of ele's in list
Sum = 0
for (Integer ele : numbers)
sum = sum + ele
print (sum)
numbers.stream()
. reduce (0 ,
(curr_sum , ele) -> curr.sum+ele
);
initial value
updated value
part of stream
curr_sum = 0 1 3 6 10 15 21
1 1
2 2
3 3
4 4
5 5
6 6
return curr.sum
To find the max ele.
maxEle = INTEGER_MIN_VALUE;
for (Integer ele : numbers)
maxEle = Math.max (ele, maxEle)
print (maxEle);
numbers.stream()
. reduce (INTEGER_MIN VALUE,
(curr_max, ele) ->
Math.max (ele, curr.max)
);

Java Advanced Concepts - 4 [Exception Handling and Miscellaneous Topics] - Jan 4

- **Exception**
  - Exception hierarchy
- **Throw & Throws keywords**
  - Runtime exception
  - Custom exceptions
- **Error - ex: stackoverflow**
  - How to handle exceptions
- **Try & Catch**
  - Should you catch all exceptions ?
- **Inheritance in exceptions**
  - In what order should the catch blocks be return ?
  - catch (Exception e) : is it good ?
- **Finally**

Exception -> an exception is an event that occurs during the execution of a program, disrupting the normal flow of instructions. When an exceptional situation arises, the Java runtime system creates an exception object and hands it off to the nearest suitable exception handler.



What are exceptions?.

Something that is not common / unexpected.

Def^{n}: Unexpected / unwanted event that happens during the execution of program.

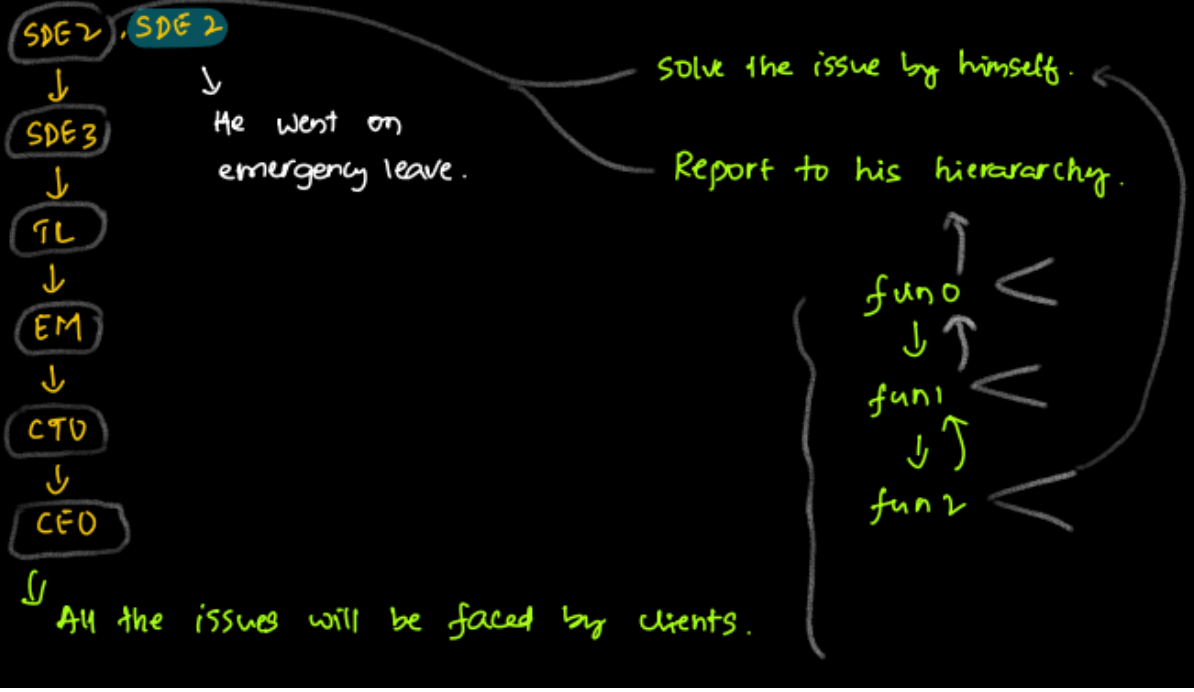
ex:

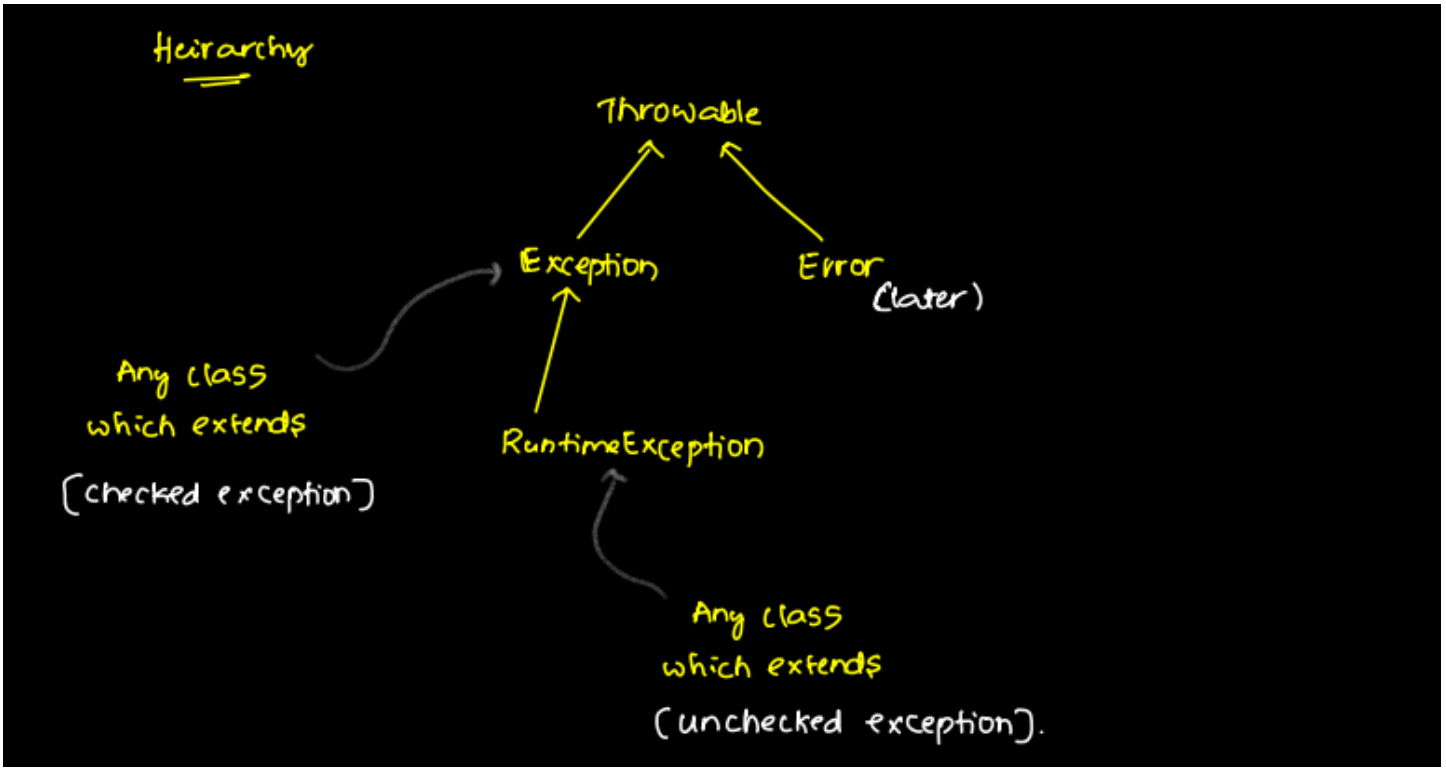
```
Student getStudent(String rollNo)
    db.executeQuery(___);
        Network failure. [unexpected].
```

Purpose:

To understand what went wrong and provides fixes if possible.

Unexpected event happens ...





Throw & Throws keywords ->

Throw keyword: The throw keyword is used to explicitly throw an exception. It is followed by an instance of an exception or a throwable expression. This keyword is typically used in methods or blocks to indicate that a specific exception condition has occurred, and the program flow should be interrupted.

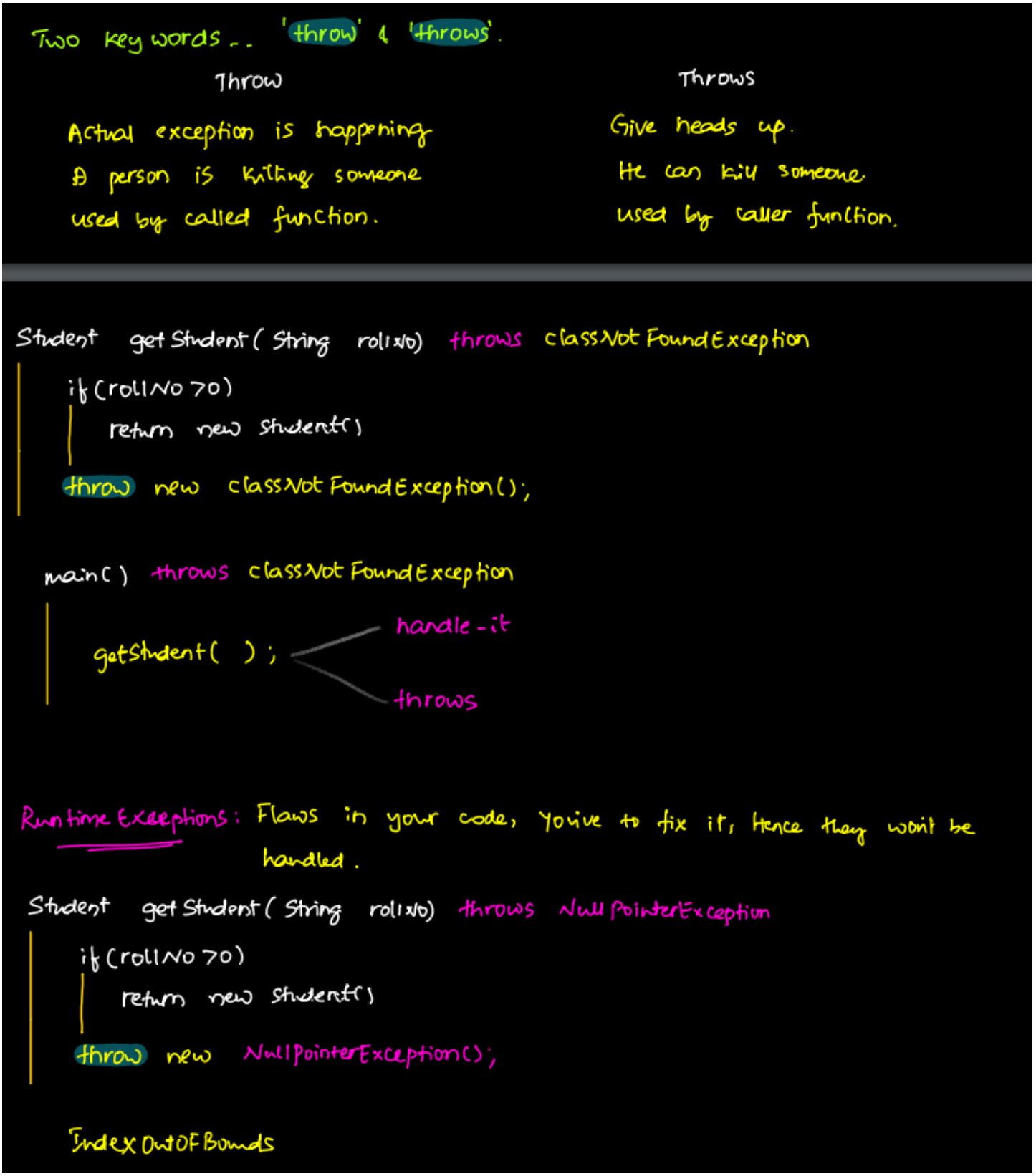
```
public void exampleMethod() {
    if (someCondition) {
        throw new SomeException("This is a custom exception message.");
    }
}
```

Throws keyword:

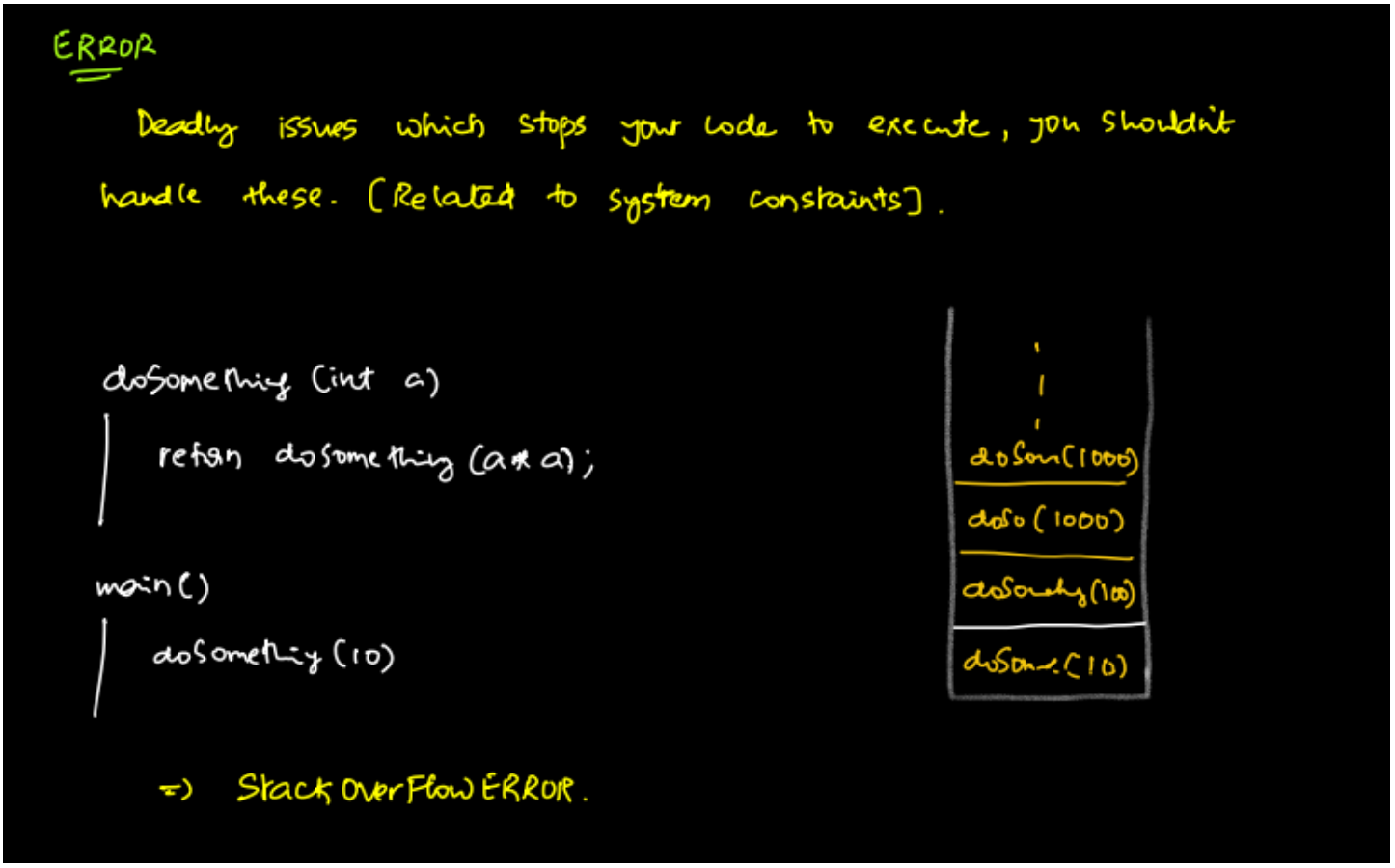
The throws keyword is used in the method declaration to specify that the method might throw certain exceptions.
It alerts the calling method or the code that uses this method that they need to handle or propagate these exceptions.
Multiple exceptions can be declared using a comma.

```
public void anotherMethod() throws IOException, CustomException {
    // Method code that might throw IOException or CustomException
}
```

When using methods that declare exceptions with throws, the calling code needs to handle these exceptions using a try-catch block or propagate them further with the throws clause.



Error -
errors are unexpected issues that occur during the execution of a Java program. Errors are different from exceptions; they are typically unrecoverable and indicate severe problems that might prevent the program from running correctly.



Common types of Java errors include:

OutOfMemoryError: Occurs when the Java Virtual Machine (JVM) cannot allocate enough memory for an object. Often caused by a memory leak or excessive memory consumption.

StackOverflowError: Happens when the call stack grows beyond the available stack space. Typically caused by infinite recursion.

NoSuchMethodError or NoSuchFieldError: Indicates that the Java Virtual Machine (JVM) cannot find a method or field at runtime that was expected to be present.

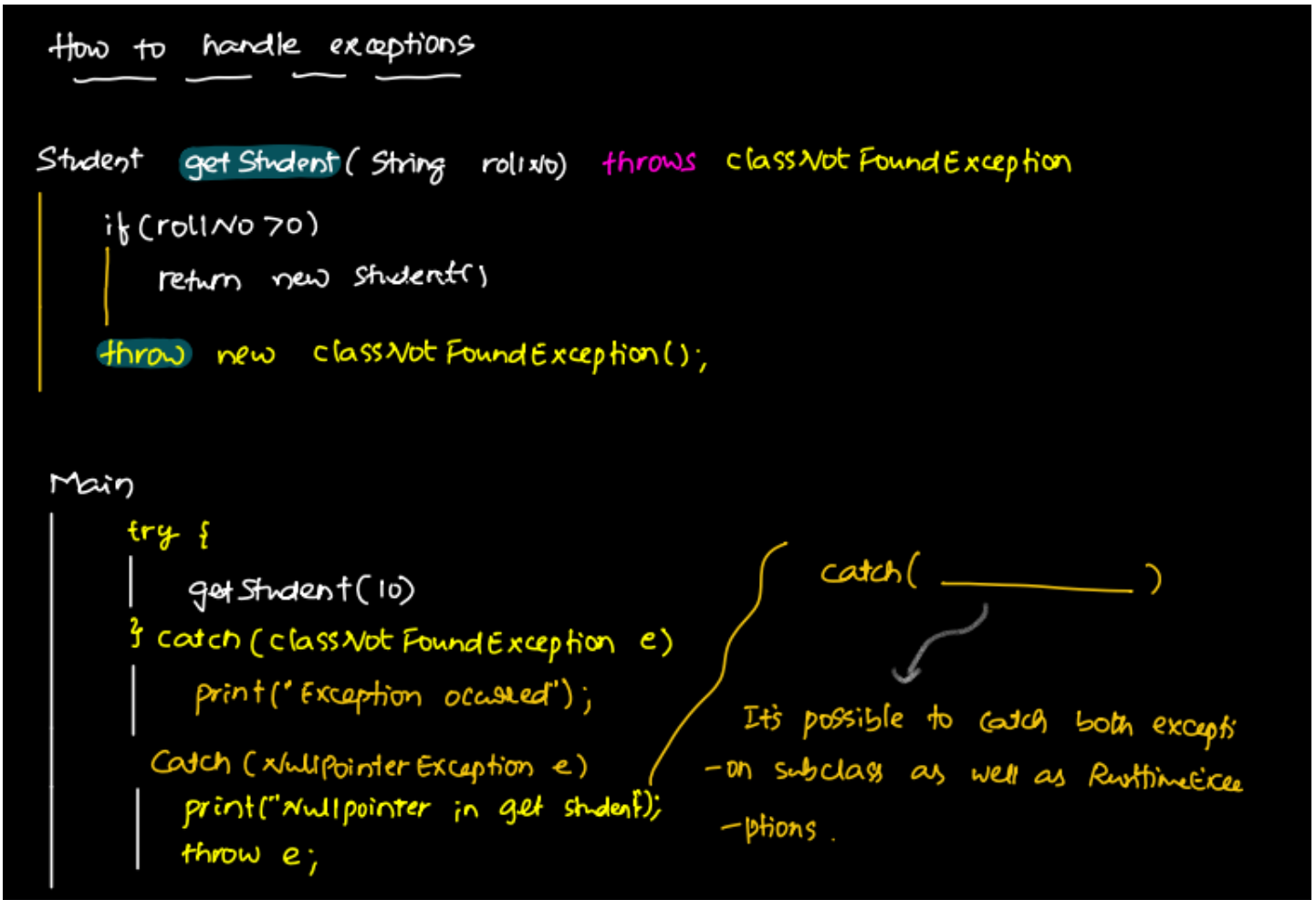
ClassCastException: Occurs when an attempt is made to cast an object to a subclass type that is not an actual instance of that class.

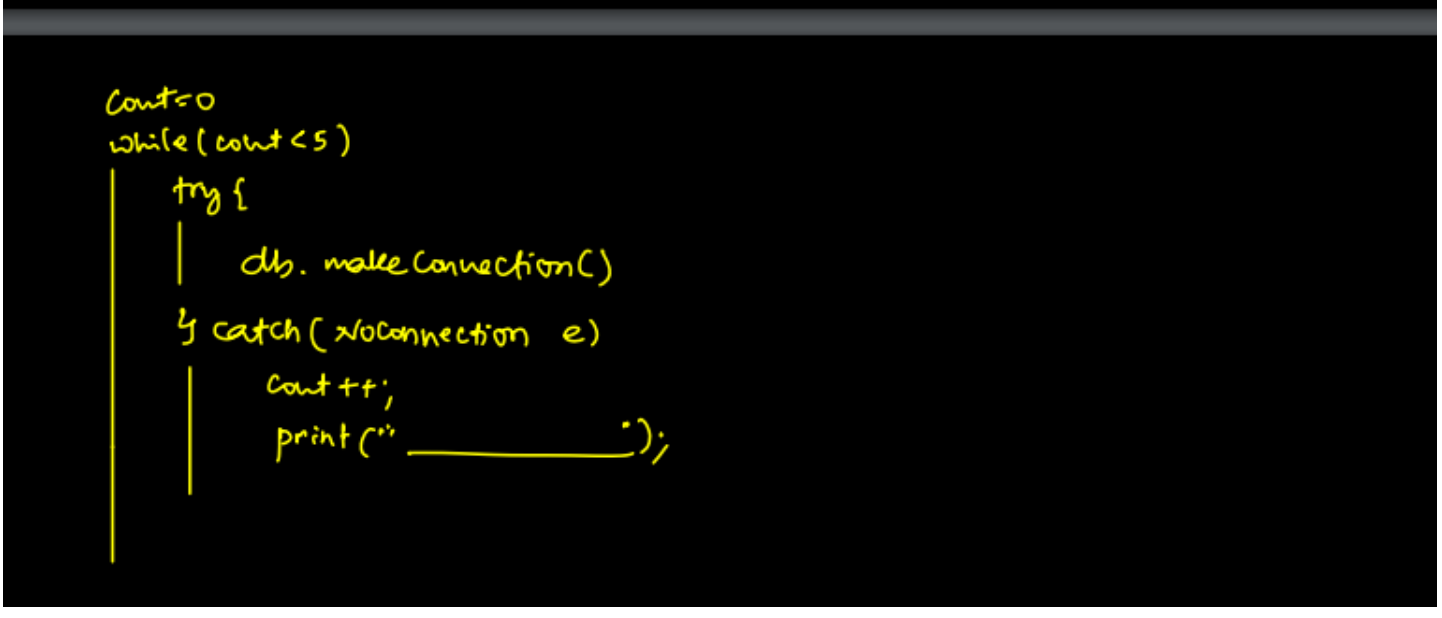
NoClassDefFoundError: Indicates that the Java Virtual Machine (JVM) cannot find the definition for a class during runtime.

ArithmeticException: Occurs when an arithmetic operation, such as division by zero, is performed. It's important to understand the type of error and the context in which it occurs to diagnose and fix the issue. Typically, errors require modification to the code or configuration to resolve.

In below example, attempting to divide by zero in the divideByZero method will result in an ArithmeticException. To handle such errors, appropriate checks and exception handling mechanisms should be implemented.

```
public class Example {
    public static void main(String[] args) {
        int result = divideByZero(); // Causes ArithmeticException
        System.out.println(result);
    }

    public static int divideByZero() {
        return 5 / 0; // Causes ArithmeticException
    }
}
```





Exceptions

class A extends class B

class B extends class C

class C extends class D.

D
↑
C
↑
B
↑
A

```
main
    try
    | doSomething()
    catch (AException a)
    | print("exception a");
    catch (BException b)
    |
    |
    ;
    Catch (DException d)
    |
```

```
main
    try
    | doSomething()
    Catch (DException d)
    |
    Catch (BException b)
    |
    |
    ;

    catch (AException a)
    | print("exception a");
```

this you shouldn't do!

```
Void doSomething()
    try{
        SomeFun()
    } catch(Exception e)
    |
```

Make it return customException.

Not good, instead catch customExceptions.

Finally

finally

You want this to happen — when an exception / when no exception

(You'll turn it off whether food is cooked or not].

```
try {
    db.establishConnection()
    db.insert()
}
catch (InsertionError e)
    InsertionFailed
    throw new e.
finally
{
    db.close().
}
```

```
try {
    db.establishConnection()
    db.insert()
     return
}
catch (InsertionError e)
    InsertionFailed
    throw new e.
finally
{
```