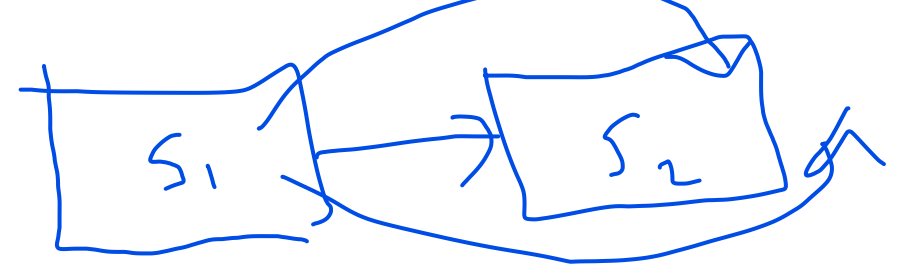


Inheritance in Java



Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

J

Why use inheritance?

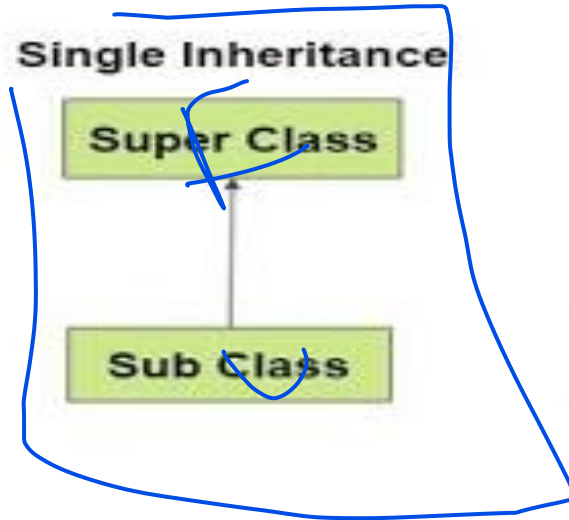
- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

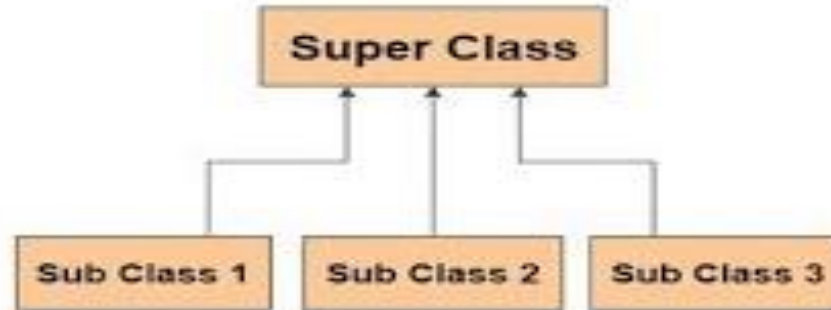
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Types of Inheritance

Single Inheritance



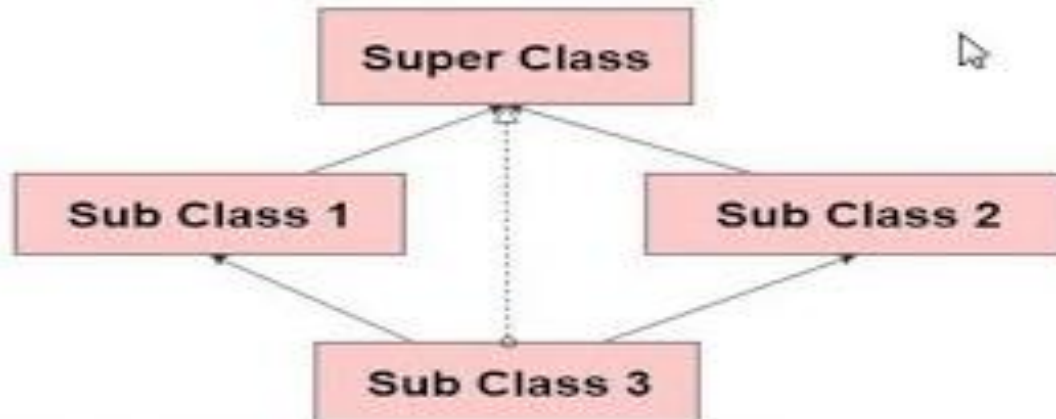
Hierarchical Inheritance



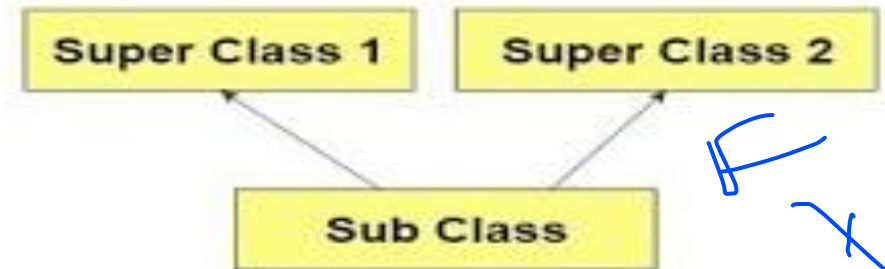
MultiLevel Inheritance



Hybrid Inheritance



Multiple Inheritance



FXM

Multiple Inheritance

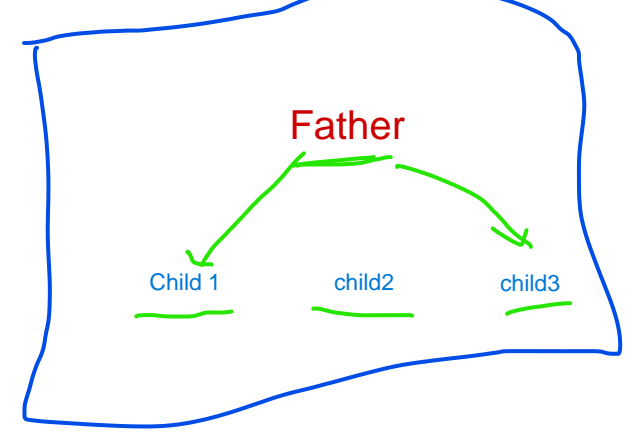
In Java, you cannot inherit from more than one class directly:

```
java
```

```
class A {}
```

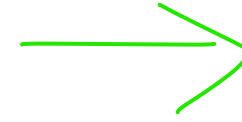
```
class B {}
```

```
class C extends A, B {} // X Not allowed!
```



So to implement Multiple Inheritance in Java, we use interfaces.

ABSTRACT CLASSES IN JAVA



A class is said to be an abstract class in which it's having at least one abstract method.

A method that is just declared without definition is said to be an abstract method.

```
abstract class ex
{
    void display() // concrete method
    {
        System.out.println("hello");
    }

    abstract void show(); //abstract method
}
```

Cgovt

ApGovt

TsGovt

INTERFACES IN JAVA

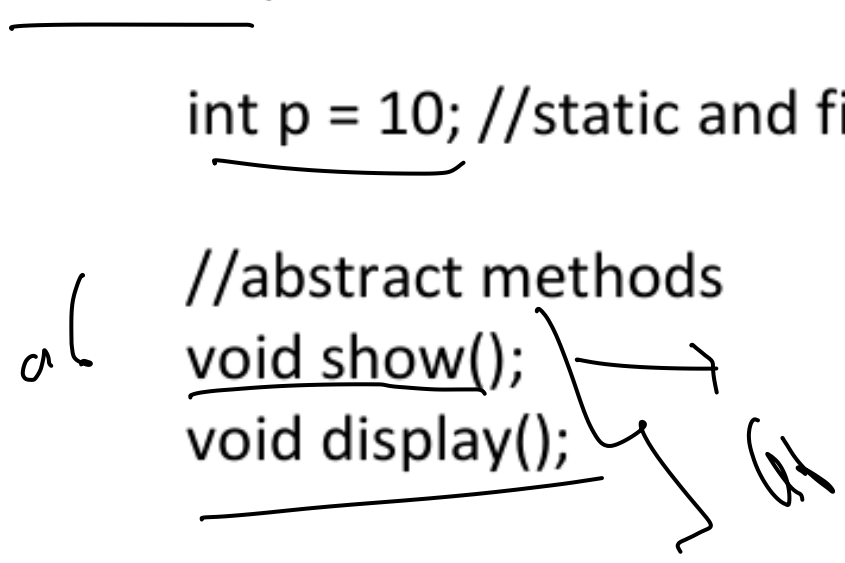
An Interface is similar to a class which contains collection of variables and methods where all the methods in interface are by default abstract methods and all variables in methods are static and final variables.

We no need to specify any method or interface with abstract keyword because by default these are abstract. All the methods in interface are implemented in sub classes of this interface.

As no methods are having definition we cannot create objects for interfaces.

```
interface (A)
{
    int p = 10; //static and final variables

    //abstract methods
    void show();
    void display();
}
```



Interfaces can be extended

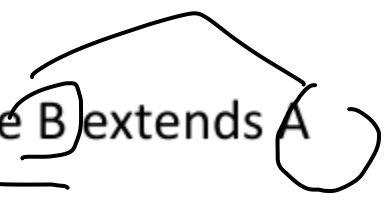
While inheriting an interface from other interface we use extends keyword. In below example interface B is inherited from interface A. As these two are interfaces we used extends keyword here.

Example

```
//program to demonstrate interfaces
interface A{
    int p = 10; //static

    void showA();
}

interface B extends A
{
    //int p = 10;
    //showA from A
    void showB();
}
```





1. What is Association?



Association is a relationship between **two classes**.
It is mainly of two types:

- "HAS-A" (Car has an Engine)
- "IS-A" (Doctor is a Human)

Association (relationship)

HAS A
(Weak and Strong)

IS_A

Composition (stronger)

Aggregation (weak)

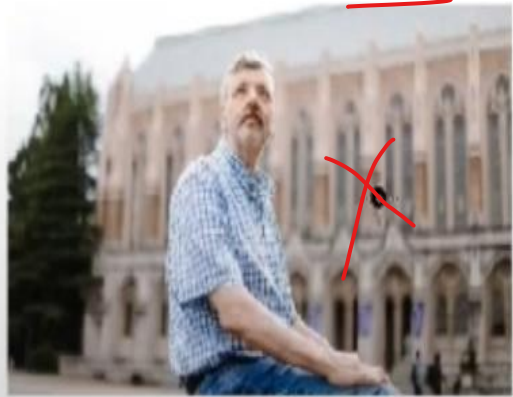
Inheritance

Engine
car
main

```
HAS A:  
class A  
{  
    B b = new B();  
}
```



Car has an Engine



University has a Professors



Doctor is a Human

```
IS A:  
class A  
{  
}  
class B extends A  
{  
}
```

PART 1: HAS-A Relationship

Means one class uses another class.

It has two types:

- Composition (Strong)
- Aggregation (Weak)

✓ A. Composition (Strong "HAS-A")

If Class A owns Class B — when A is destroyed, B is also destroyed.

📌 Example: A `Car` has an `Engine`

If Car is destroyed, Engine is destroyed too.

💡 Think: Without a Car, Engine has no life here — tightly coupled.

✅ B. Aggregation (Weak "HAS-A")

Class A uses Class B, but does not own it.

If A is destroyed, B can live independently.


📌 **Example:** University has Professors


But professors can exist without the university.

Think: Even if University is closed, the professor still exists.

PART 2: IS-A Relationship

Means Inheritance (Class A extends Class B)

 Example: `Doctor` is a `Human`

 Think: Doctor is a special type of Human — so inherits behavior.

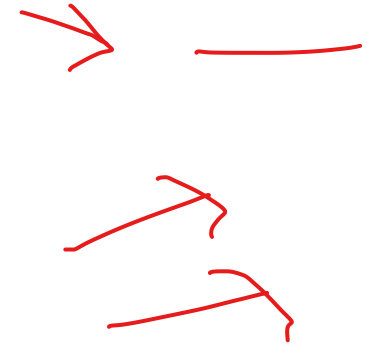
```
class Human {  
    void walk() {  
        System.out.println("Human walks");  
    }  
}  
  
class Doctor extends Human {  
    void treat() {  
        System.out.println("Doctor treats patient");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Doctor doc = new Doctor();  
        doc.walk();    // inherited method  
        doc.treat();   // own method  
    }  
}
```

EXCEPTION HANDLING IN JAVA

Errors in Java Program

There are 3 types of errors

mistake done prog
syntax
logic errors illogical



- 1) compile time errors : These are syntactical errors found in the code

Examples:

- 1) if we write `system.out.println()` instead of `System.out.println()` it will generate a compile time error.
- 2) if we forgot semicolon at the end of any statement in our program

2) runtime errors : These errors will occur due to inefficiency of the computer system (ex : ~~insufficient memory~~)

Example :

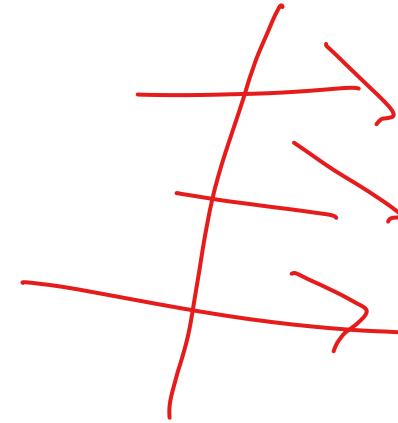


if we forgot to write string array inside main method as follows

```
class A
{
    public static void main()
    {

        System.out.println("Welcome");

    }
}
```





do

- 3) logical errors: These errors depict flaws in the logic of the program. It may give some output but not as expected.

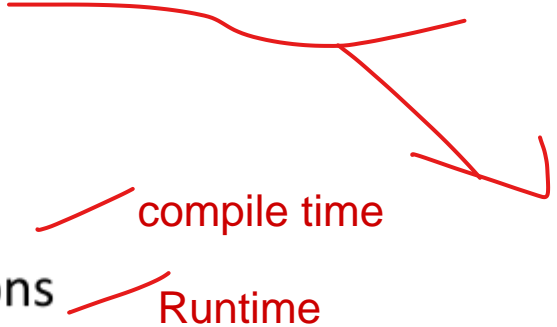
```
void add()  
{  
    int a =10;  
    int b=20;  
    System.out.println(a*b);  
  
}
```


Exceptions:

Exceptions are Runtime errors. Whenever an exception occurs then the java system stops the program's execution. so, exception is defined as

“An Exception is an abnormal condition which stops the normal execution of a program “.

Exceptions are Two types:

- 1) Checked Exceptions
 - 2) Unchecked Exceptions
- 
- compile time
- Runtime



All exceptions are occurred at runtime only. But some are detected at compile time and some are detected at runtime. The exceptions that are checked at compile time are said to be checked exceptions and the exceptions that are checked at runtime are said to be unchecked exceptions.

All these exception related classes and interfaces are belongs to java.lang package

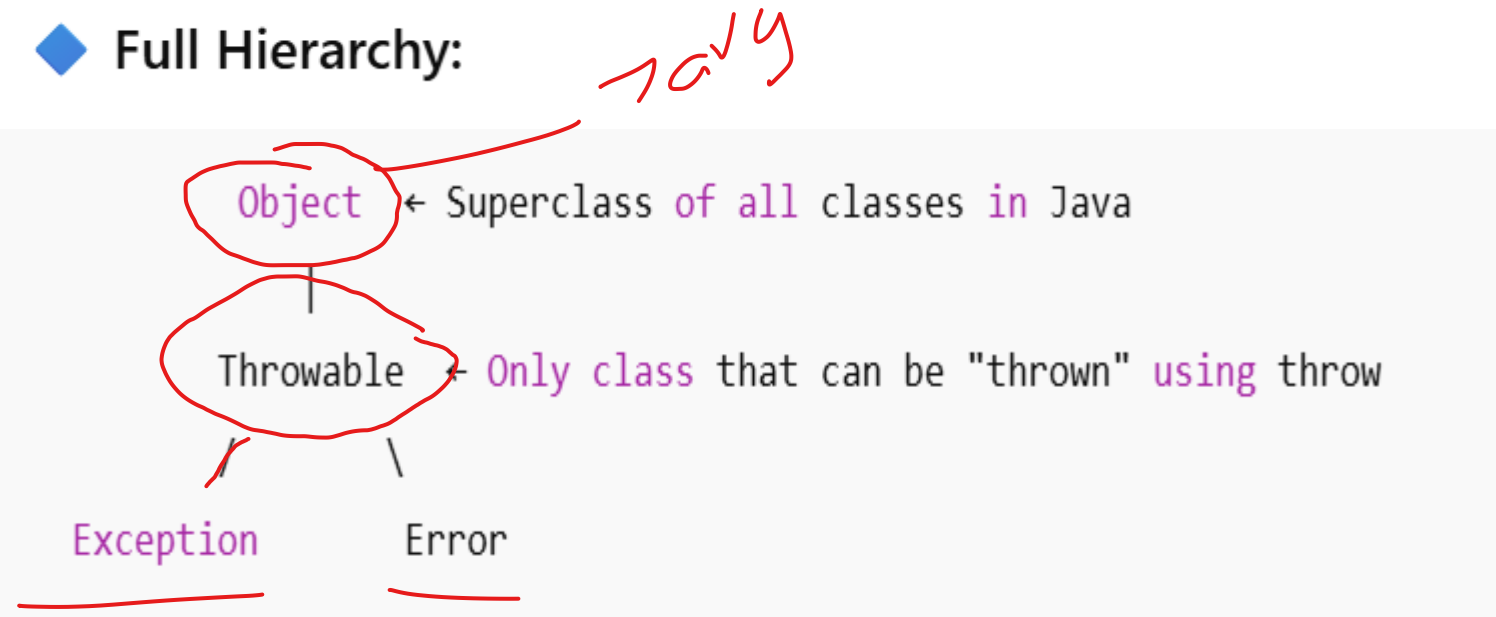
All these exception related classes and interfaces are belongs to java.lang package

The following classes are top in the errors and exceptions class hierarchy.

Throwable: Represents all errors and exceptions in java

Exception: This is super class of all exceptions in java.

◆ Full Hierarchy:



ArrayIndexOutOfBoundsException


```
public class ArrayExample {  
    public static void main(String[] args) {  
        int[] numbers = {10, 20, 30, 40, 50};  
        // Let's print all elements properly  
        for (int i = 0; i <= numbers.length; i++) {  
            System.out.println("Element at index " + i + ": " + numbers[i]);  
        }  
    }  
}
```

Handwritten annotations in red:

- Under the array values: 0, 1, 2, 3, 4, 5 (with a bracket under 5)
- Under the loop condition: $i <=$

Exception Handling:

We can handle exceptions in java by using 5 keywords

- 1) try
 - 2) catch
 - 3) throw
 - 4) throws
 - 5) finally
- 

try
catch
finally

The following are the list of checked exceptions and unchecked exceptions.

A → B

Unchecked Exceptions

R X +

ArithmeticException

IndexOutOfBoundsException

ArrayIndexOutOfBoundsException

StringIndexOutOfBoundsException

NullPointerException

Checked Exceptions

ClassNotFoundException

InstantiationException

NoSuchMethodException

IOException

FileNotFoundException

InterruptedIOException



User Defined Exceptions

We can create our own user defined exception classes and use them as per our requirements in our programs.

To create user defined exceptions

- 1) create a class that extends Exception class

class `MyException` extends `Exception`

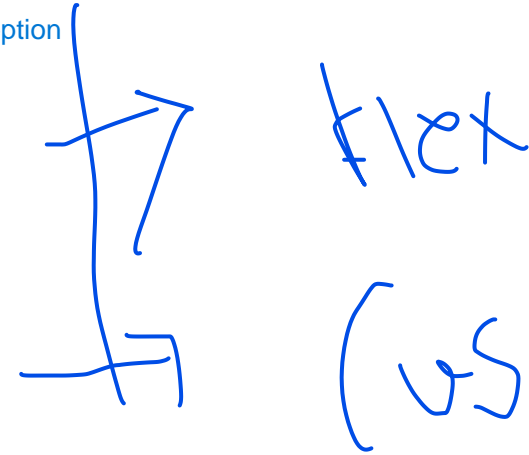
{

- 2) We may define constructor / toString method in it

```
MyException(){  
    System.out.println("Our own message");  
}
```

or


```
public String toString(){  
    System.out.println("Our own message");  
}
```




3) Now as per our conditions, throw our own exception by using throw keyword

2. Ways to Create Custom Exception

`extends Exception`



`extends RuntimeException`



1 Create a class that extends `Exception`

java

```
class MyCheckedException extends Exception {
```



2 We may define constructor / toString method in it

java

// Option 1: Constructor with default message

```
MyCheckedException() {  
    System.out.println("This is a default checked exception message.");  
}
```

// OR

// Option 2: Constructor with custom message

```
MyCheckedException(String msg) {  
    super(msg);  
}
```

// OR

// Option 3: Override toString()

```
public String toString() {  
    return "Custom Checked Exception occurred";  
}
```

}

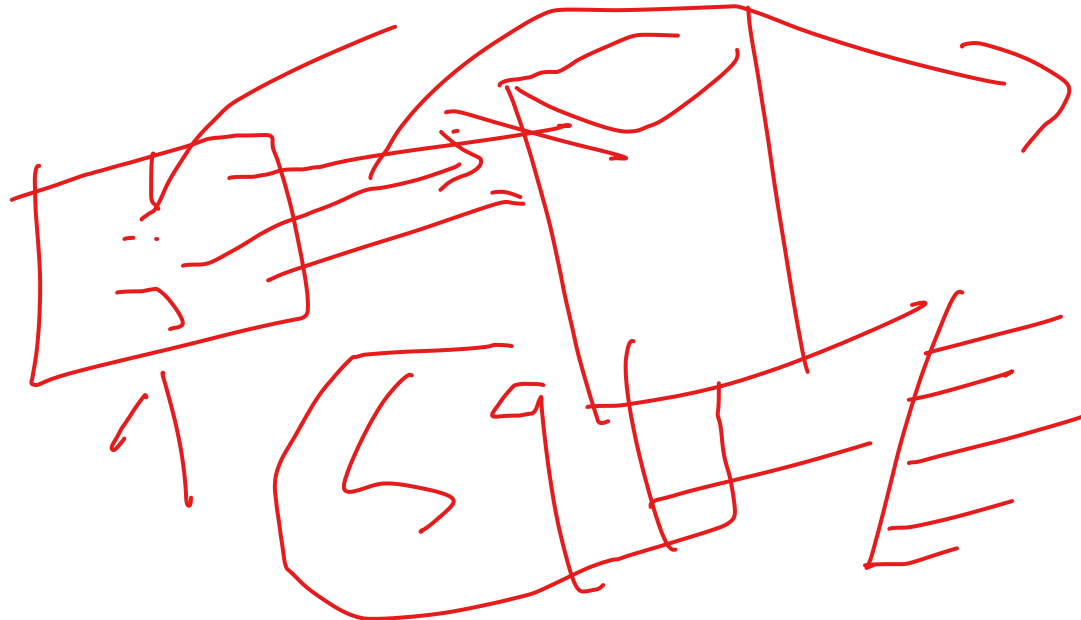

Log

```
log.error("FGHHH ");  
log.warn(" ");  
log.info(" ");  
log.debug(" ");
```

3 Now as per your condition, throw it manually using `throw` keyword

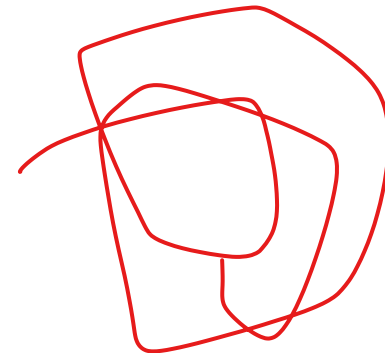
java

```
if (age > 10) {  
    throw new MyCheckedException("Age should be 10 or below.");  
}
```



✓ Java Error Hierarchy – Like a Family Tree

halt



java.lang.Throwable ← Top-level parent of all errors and exceptions

├─ java.lang.Error ← ⚡ Serious issues (JVM errors) → DO NOT HANDLE

│

│ └─ OutOfMemoryError ← No memory left

│ └─ StackOverflowError ← Infinite recursion

└─ java.lang.Exception ← ✅ Things you CAN handle (via try-catch)

├─ Checked Exceptions ← ! Must handle using try-catch or throws

│ └─ IOException ← File/stream issues

│ └─ SQLException ← Database issues

└─ Unchecked Exceptions ← ? Optional to handle (logic errors)

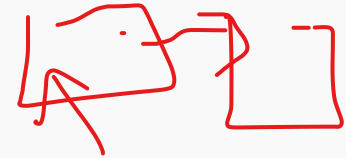
└─ RuntimeException

├─ NullPointerException ← Accessing null object

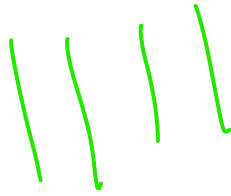
├─ ArithmeticException ← Divide by zero

├─ ArrayIndexOutOfBoundsException

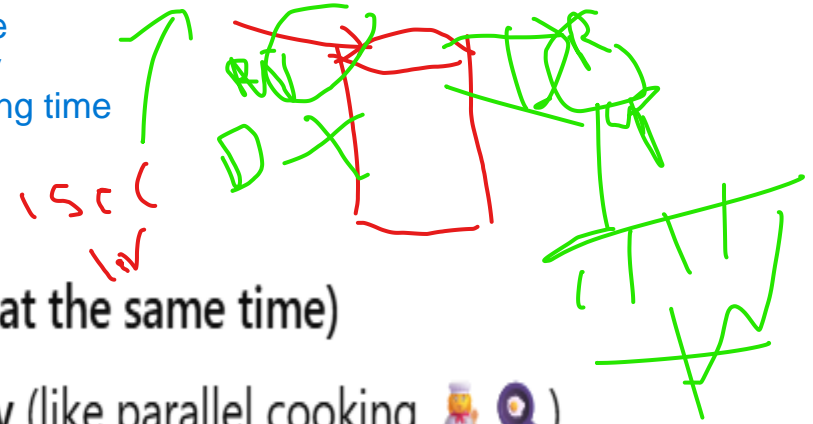
└─ IllegalArgumentException



What is Concurrency?



Save time
Efficiency
less waiting time

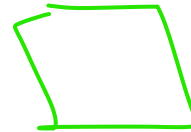


Concurrency = doing multiple things at the same time (or seemingly at the same time)

In Java, this means running multiple parts of a program independently (like parallel cooking 🍳👩🍳).



Thread = Small independent task



- In Java, `Thread` is the unit of concurrency.
- Each thread runs **separately** from others.

Example:

1 thread reads a file

1 thread sends an email

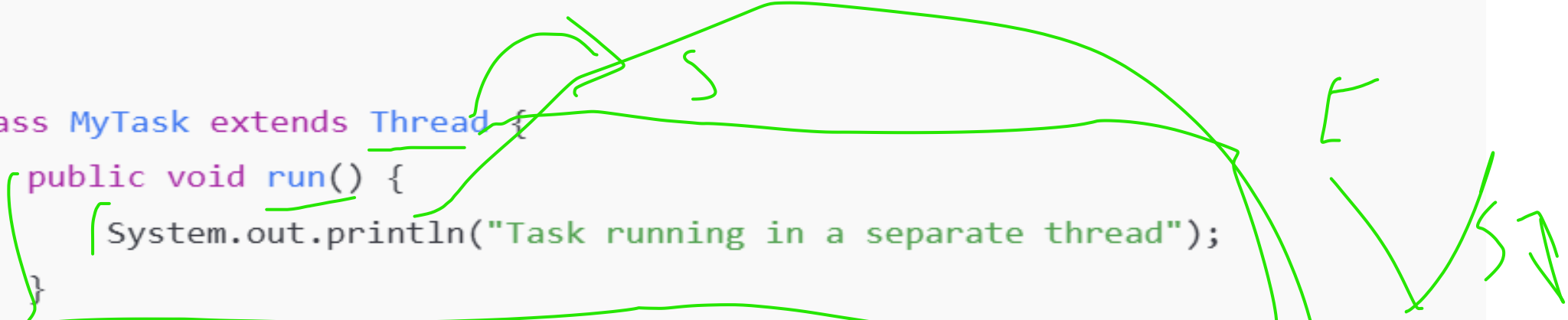
1 thread updates the UI

➡ All at once.

1. Using Thread Class

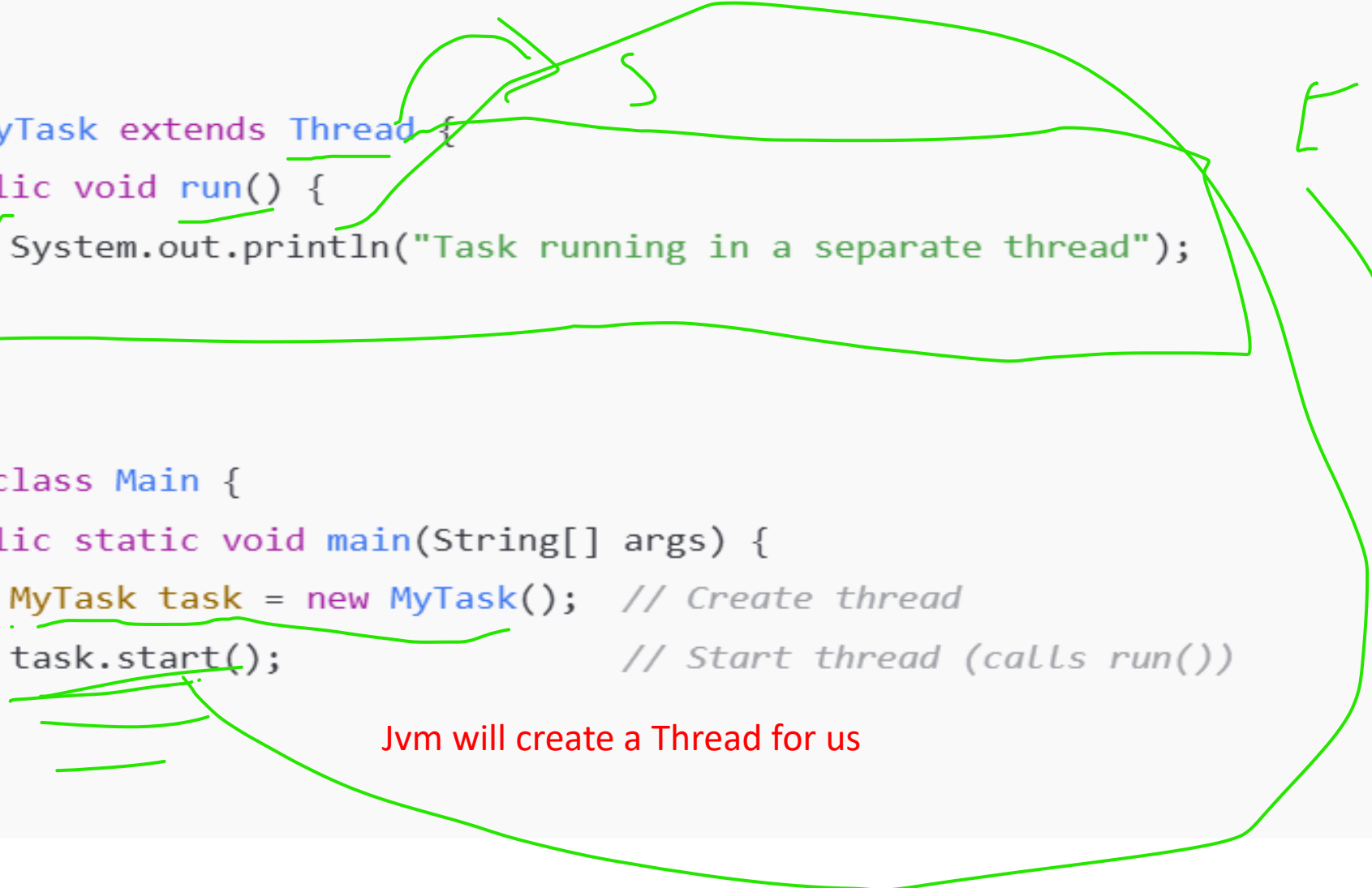
java

```
class MyTask extends Thread {  
    public void run() {  
        System.out.println("Task running in a separate thread");  
    }  
}
```



```
public class Main {  
    public static void main(String[] args) {  
        MyTask task = new MyTask(); // Create thread  
        task.start();                // Start thread (calls run())  
    }  
}
```

Jvm will create a Thread for us



2. Using Runnable Interface

java

```
class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Runnable task is running");  
    }  
}
```

Step 1: implements Runnable

System.out.println("Runnable task is running");

Step2: override Run method

```
public class Main {  
    public static void main(String[] args) {  
        Thread t = new Thread(new MyRunnable());  
        t.start();  
    }  
}
```

Step 3: create object for Thread class

Thread t = new Thread(new MyRunnable()); // Pass runnable to thread

Step 4:- pass your custom thread object to Thread Object

Step 5: start the thread

3. Using ExecutorService (Advanced, Better Way)

java

```
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;
```

```
public class Main {
```

create 2 worker threads

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

```
        ExecutorService service = Executors.newFixedThreadPool(2);
```

Giving 2 tasks

```
        service.submit(() -> System.out.println("Task 1 running"));
```

```
        service.submit(() -> System.out.println("Task 2 running"));
```

```
        service.shutdown();
```

```
    }
```

After shutdown

```
}
```

! Important Concepts in Concurrency

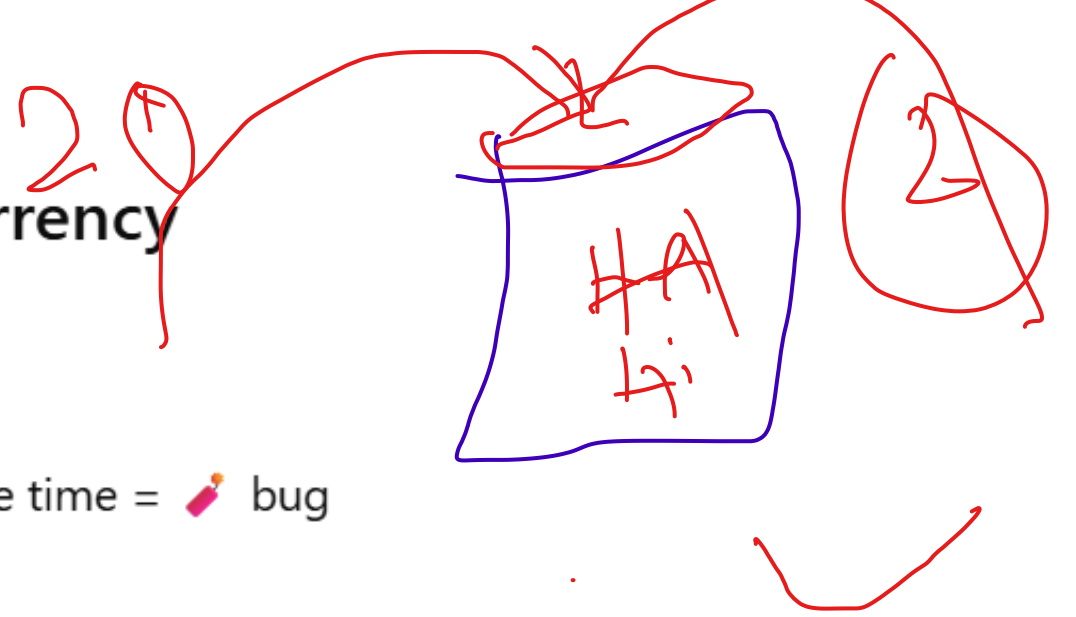
1. Race Condition

Two threads changing the same data at the same time = 🚫 bug

Solution: Use `synchronized` keyword

java

```
public synchronized void increment() {  
    count++;  
}
```



2. Deadlock

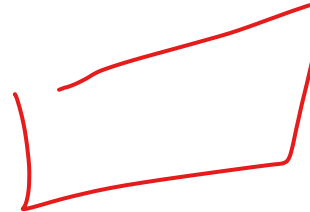
Two threads wait for each other → both stuck forever

3. Thread Safety

Code that works fine even when used by multiple threads at once

Use:



- synchronized keyword
- AtomicInteger, ConcurrentHashMap
- Thread-safe collections




Java Files

✓ 1. CreateFile.java

java

```
import java.io.FileWriter;    
  
public class CreateFile {  
    public static void main(String[] args) throws Exception {  
        FileWriter writer = new FileWriter("sample.txt");  
        writer.write("Hello, this is a sample file.");  
        writer.close();  
        System.out.println("✓ File created and written.");  
    }  
}
```

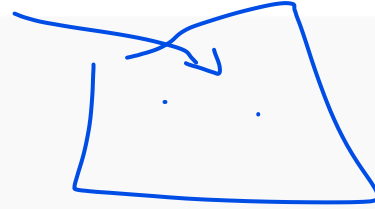


2. ReadFile.java

java

```
import java.io.FileReader;

public class ReadFile {
    public static void main(String[] args) throws Exception {
        FileReader reader = new FileReader("sample.txt");
        int ch;
        System.out.print("📖 File content: ");
        while ((ch = reader.read()) != -1) {
            System.out.print((char) ch);
        }
        reader.close();
    }
}
```



Reads **one character** at a time
,until last character

 `reader.read()` doesn't return a `char` — it returns an `int`

Even though the file has characters (like `A`, `B`, `C`), Java reads them **one-by-one** as their ASCII/Unicode number.

❌ 3. DeleteFile.java

java

```
import java.io.File;

public class DeleteFile {
    public static void main(String[] args) {
        File file = new File("sample.txt");
        if (file.delete()) {
            System.out.println("❌ File deleted successfully.");
        } else {
            System.out.println("File not found or can't delete.");
        }
    }
}
```


What is **Try with Resources**?

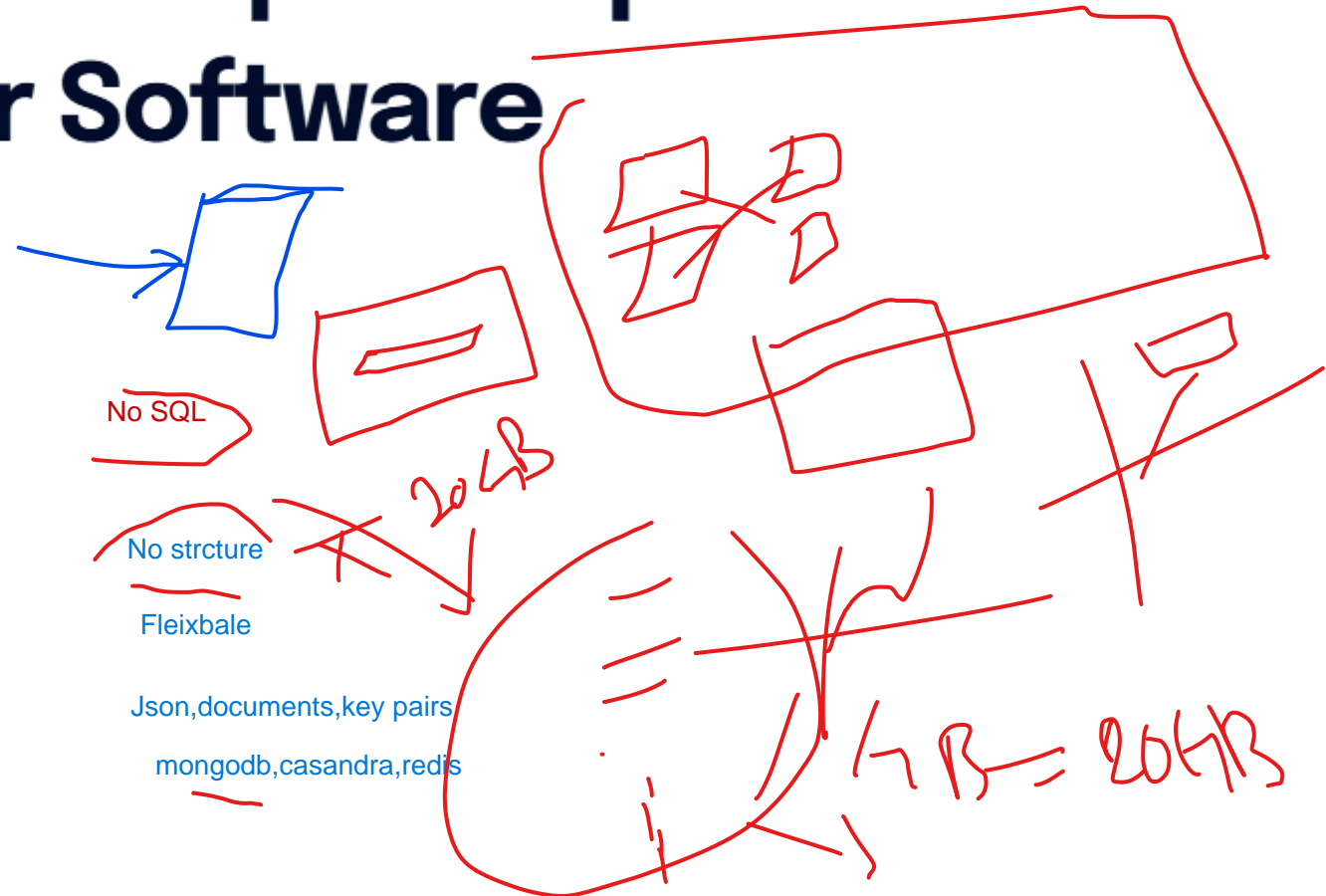
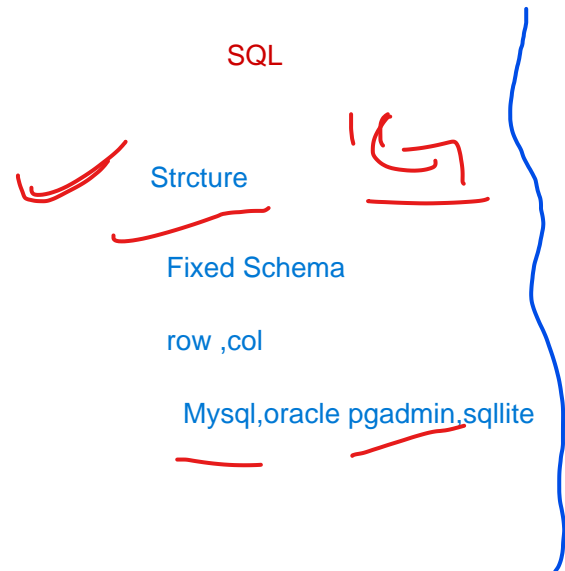
Try with Resources is a special way in Java to automatically close files, readers, writers, etc.

So you don't need to manually write `close()` — Java does it for you 

```
import java.io.FileReader;

public class ReadFileWithTryResources {
    public static void main(String[] args) throws Exception {
        // 📌 Try with resources – reader auto-closes after this block
        try (FileReader reader = new FileReader("sample.txt")) {
            int ch;
            System.out.print("📖 File content: ");
            while ((ch = reader.read()) != -1) {
                System.out.print((char) ch); // 🔄 Convert int to char
            }
        } // ✅ FileReader is closed automatically here!
    }
}
```

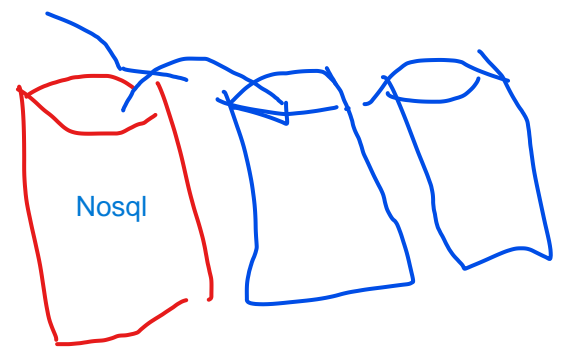
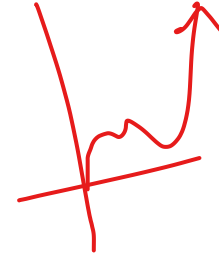
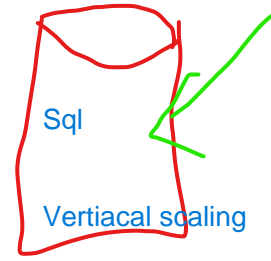
SOLID Design Principles Explained: Building Better Software Architecture





SOLID stands for:

- **S** - Single-responsibility Principle
- **O** - Open-closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle



Single-Responsibility Principle



Single-responsibility Principle (SRP) states:

A class should have one and only one reason to change, meaning that a class should have only one job.

Open-Closed Principle

Open-closed Principle (OCP) states:

Objects or entities should be open for extension but closed for modification.

This means that a class should be extendable without modifying the class itself.

Liskov Substitution Principle



Liskov Substitution Principle states:

Let $q(x)$ be a property provable about objects of x of type T .
Then $q(y)$ should be provable for objects y of type S where S is
a subtype of T .

This means that every subclass or derived class should be substitutable for their base or parent class.

Interface Segregation Principle

The interface segregation principle states:

A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use.

This principle emphasizes that large, general-purpose interfaces should be broken down into smaller, more specific ones. This way, client classes only need to know about the methods that are relevant to them.

Dependency Inversion Principle



Dependency inversion principle states:

Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

This principle allows for decoupling.



