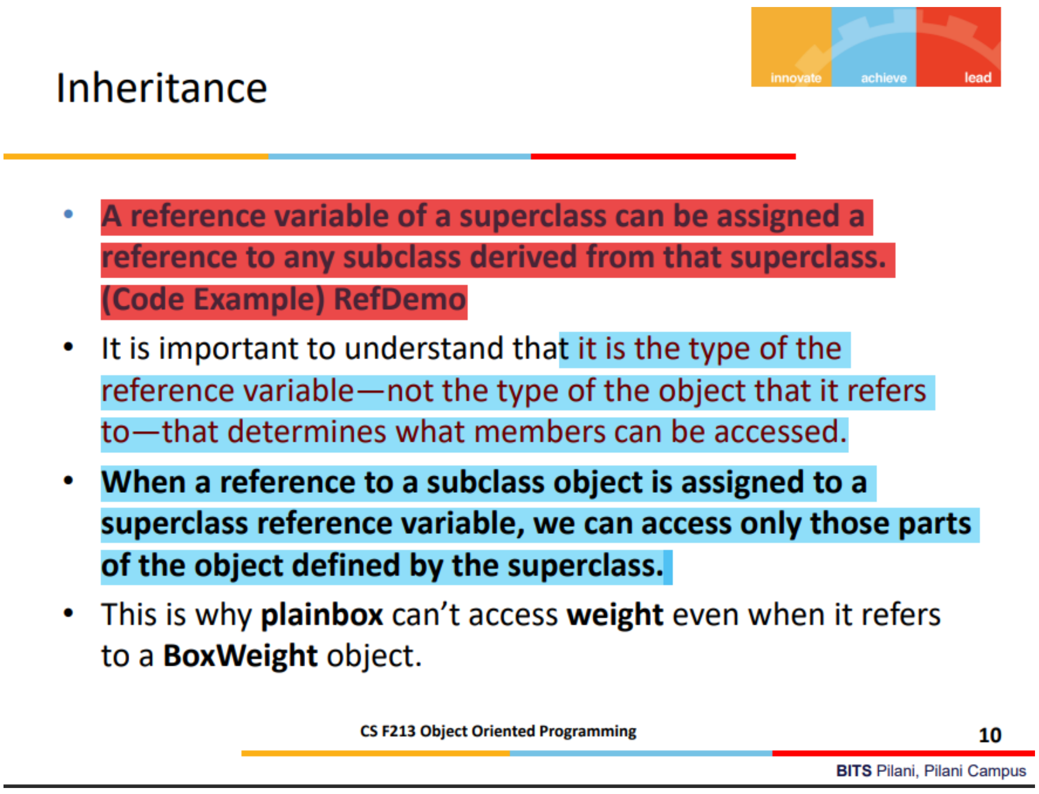
Advantages of static blocks

If you need to do computation in order to initialize your static variables, you can declare a static block which gets executed exactly once, when the class is first loaded.

Disadvantages for static blocks

There is a limitation of JVM that a static initialization block should not exceed 64K.
You cannot throw Checked Exceptions. (will be discussed later)
You cannot use this keyword since there is no instance.
You shouldn't try to access super since there is no such a thing for static blocks.
You should not return anything from this block.
Static blocks make testing a nightmare.

Lecture 9



Inheritance

- **A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. (Code Example) RefDemo**
- It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.
- **When a reference to a subclass object is assigned to a superclass reference variable, we can access only those parts of the object defined by the superclass.**
- This is why **plainbox** can't access **weight** even when it refers to a **BoxWeight** object.

CS F213 Object Oriented Programming 10

BITS Pilani, Pilani Campus

- **super( )** must always be the first statement executed inside a subclass' constructor.

Super() to call constructors

1. When a subclass calls **super()**, it is calling the constructor of its immediate superclass.
2. Thus, **super( )** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy.
3. **super( )** must always be the first statement executed inside a subclass constructor.

L10

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

A superclass reference variable can refer to a subclass object.

- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- **This determination is made at run time.** When different types of objects are referred to, different versions of an overridden method will be called.

It is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

- To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.
- **Methods declared as final cannot be overridden**

Methods declared as final – early binding as it cannot be overridden

- Sometimes we want to prevent a class from being inherited.
- To do this precede the class declaration with **final**.
- **Declaring a class as final implicitly declares all of its methods as final, too.**

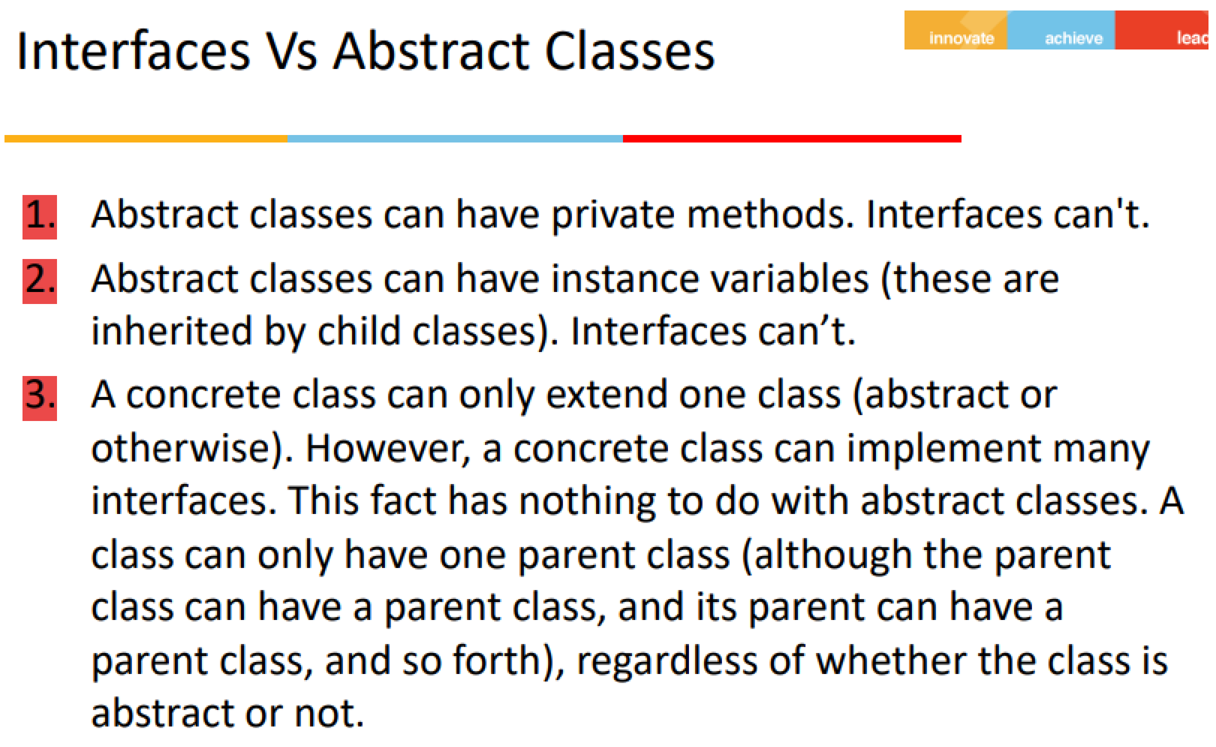
L11

1. Variables can be declared inside interface declarations.
2. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class.
3. They must also be initialized.
4. All **methods** and **variables** are implicitly **public**

Interfaces Vs Abstract Classes

- Abstract classes can have non-abstract methods that have method definitions. This gets inherited.
- Interfaces do not allow any method definitions.
- Abstract classes can have constructors. This may seem a little silly because we can't construct objects from an abstract class. However, when we write child classes, it calls the constructor of the parent class, even if the parent class is abstract.
- Interfaces can't have constructors.

https://www.cs.umd.edu/~clin/MoreJava/Objects/absclass.html



Interfaces Vs Abstract Classes

1. Abstract classes can have private methods. Interfaces can't.
2. Abstract classes can have instance variables (these are inherited by child classes). Interfaces can't.
3. A concrete class can only extend one class (abstract or otherwise). However, a concrete class can implement many interfaces. This fact has nothing to do with abstract classes. A class can only have one parent class (although the parent class can have a parent class, and its parent can have a parent class, and so forth), regardless of whether the class is abstract or not.

L14

- Lanbda- anonymous methods
- Can be implemented only by functional interfaces – a functional interface is an interface which has one and only one **abstract** method.
- Functional interface typically represents a single action
- In the parameter list – it is possible to explicitly specify the type of the parameter, but in many cases it can be inferred as well. – eg: (n)->(n%2)==0;
- An interface method is abstract only if it does not specify an implementation
- Because non-default non-static, non-private interface methods are implicitly abstract, there is no need to use the abstract modifier
- Expression lambdas : have expression bodies instead of a single line of code
- Block lambdas : have a block body
- **Block lambdas must use a return statement to return a value**
- **Lambda expressions can be passed as arguments : useful as we can now pass code as arguments to a method –** refer code LambdaAsArgumentsDemo.java

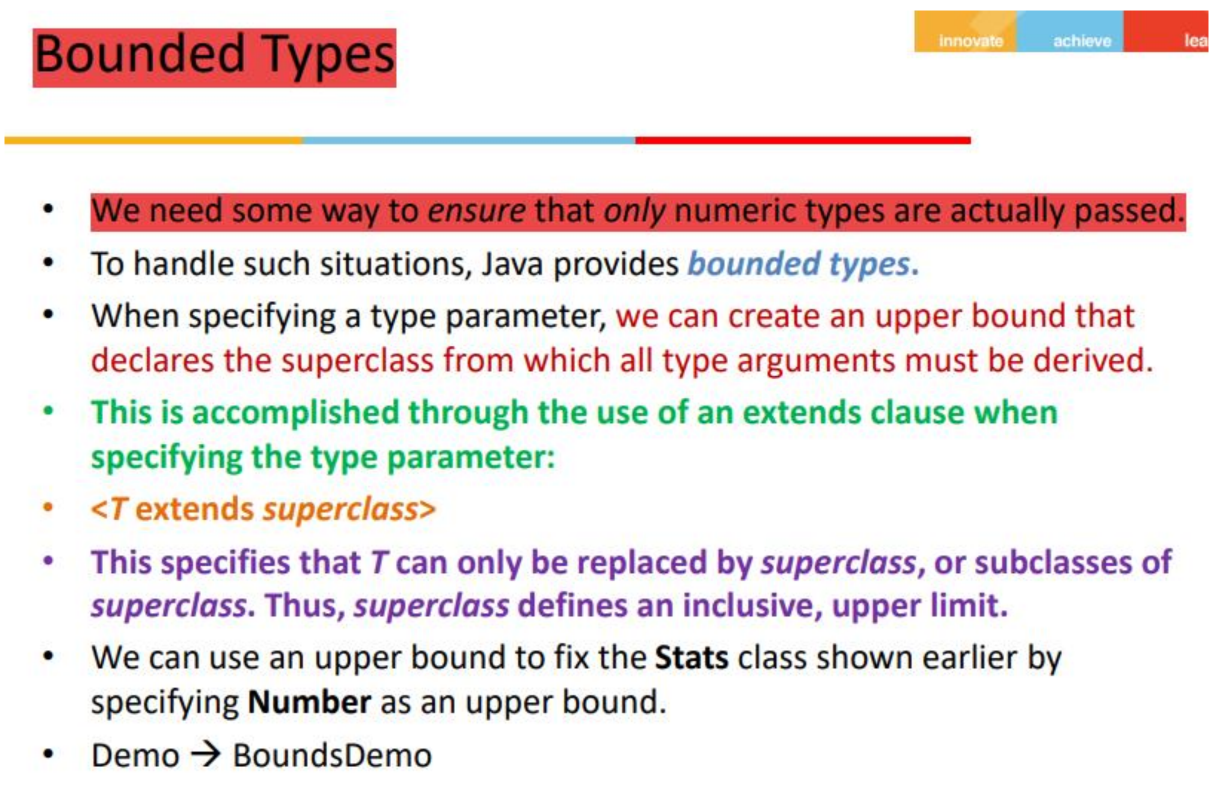
Method Reference: Static Methods

- A method reference provides a way to refer to a method without executing it.
- It relates to lambda expressions because it, too, requires a target type context that consists of a compatible functional interface.
- When evaluated, a method reference also creates an instance of the functional interface.
- To create a **static** method reference:
  - *ClassName::methodName*
- The class name is separated from the method name by a double colon.
- The :: is a separator that was added to Java by JDK 8 expressly for this purpose.
- This method reference can be used anywhere in which it is compatible with its target type. (**Demo** → MethodRefDemo)

Lambdas & Variable Capture

- Variables defined by the enclosing scope of a lambda expression are accessible within the lambda expression.
- For example, a lambda expression can use an instance or **static** variable defined by its enclosing class.

- **When a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a *variable capture*.**
- **A lambda expression may only use local variables that are *effectively final*.**
- **An effectively final variable is one whose value does not change after it is first assigned.**



Bounded Types

- We need some way to *ensure* that *only* numeric types are actually passed.
- To handle such situations, Java provides ***bounded types***.
- When specifying a type parameter, we can create an upper bound that declares the superclass from which all type arguments must be derived.
- **This is accomplished through the use of an extends clause when specifying the type parameter:**
- **<*T* extends *superclass*>**
- **This specifies that *T* can only be replaced by *superclass*, or subclasses of *superclass*. Thus, *superclass* defines an inclusive, upper limit.**
- We can use an upper bound to fix the **Stats** class shown earlier by specifying **Number** as an upper bound.
- Demo → BoundsDemo

Refer Bounded wild card in Generics (Codes)

T - Type

E - Element

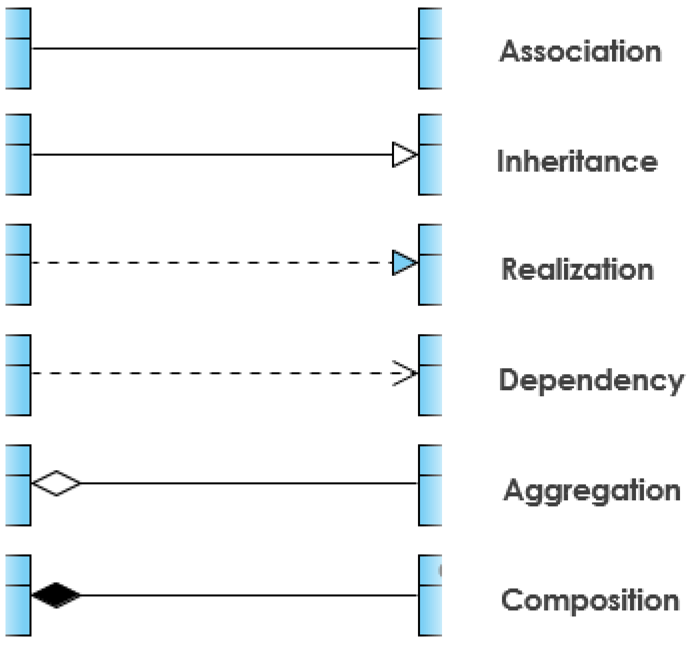
K - Key

N - Number

V – Value

Note: Let there be an interface T;
class Identity<T extends T> cannot be done.

L19 – UML Diagrams



Inheritance => L inherit R

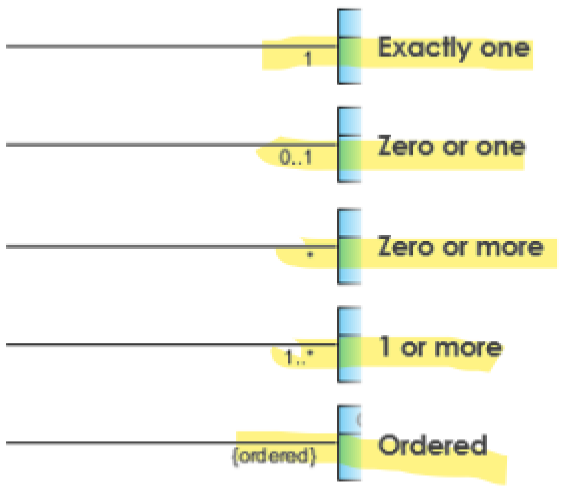
Realization => interface implementation

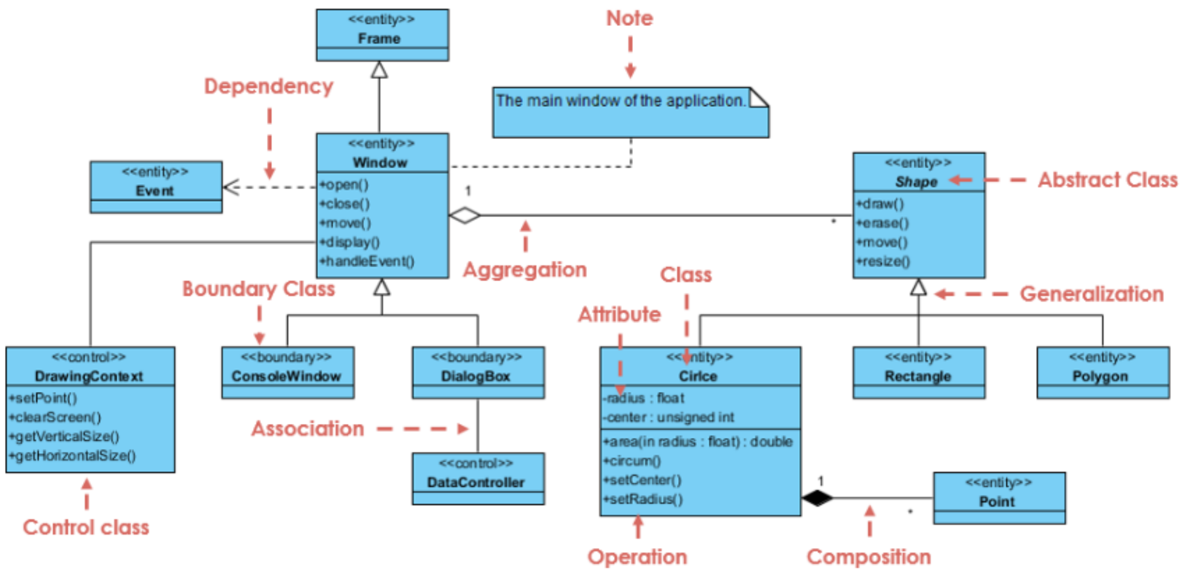
Dependency => Lclass depends on Rclass

Aggre – R is a part of L

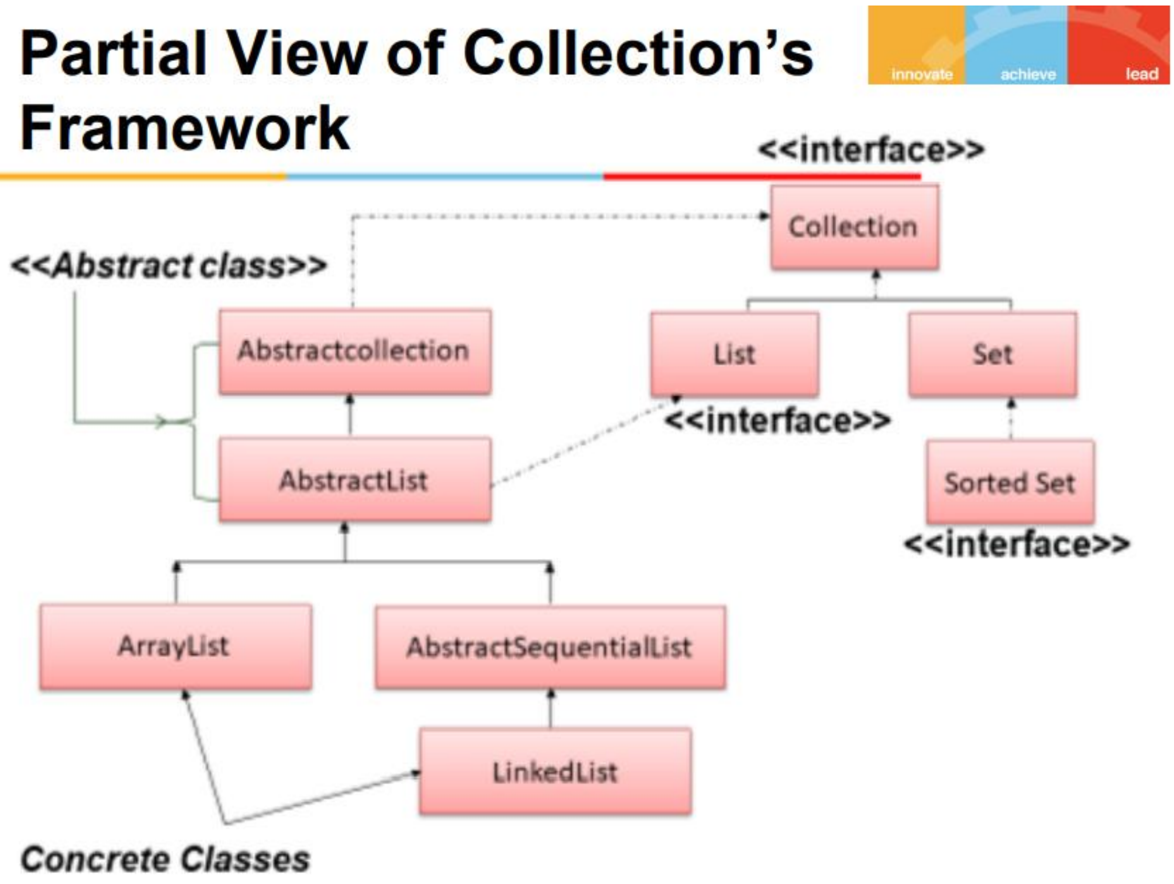
Compo – R cannot exist without L

Cardinality





Partial View of Collection's Framework



Comparable

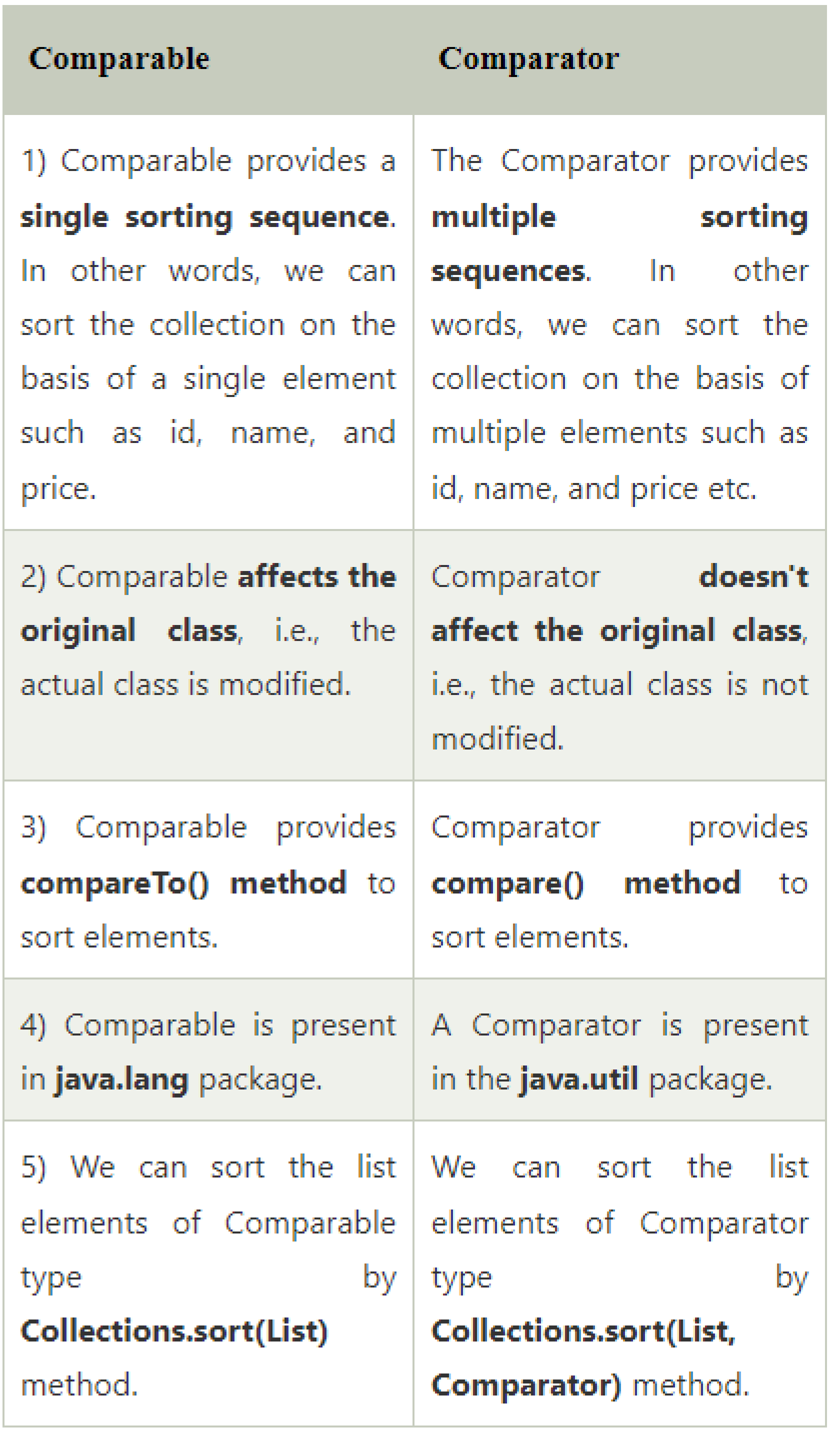
It is found in java.lang package and contains only one method named compareTo(Object)

We can sort the elements of:

- String objects
- Wrapper class objects
- User-defined class objects

COMPARABLE IS A CLASS

COMPARATOR IS AN INTERFACE



Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

Comparator Interface

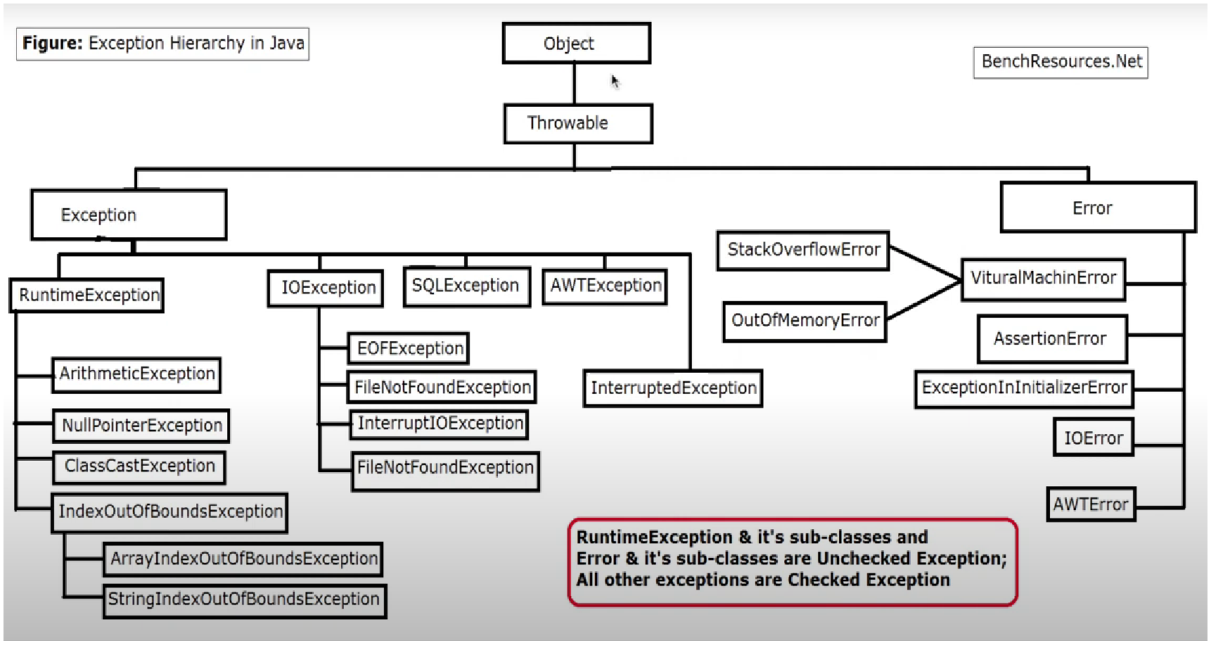
This interface is found in java.util package and contains 2 methods

- compare(Object obj1,Object obj2)
- equals(Object element)

It provides multiple sorting sequence

- Elements can be sorted based on any data member

Exception handling

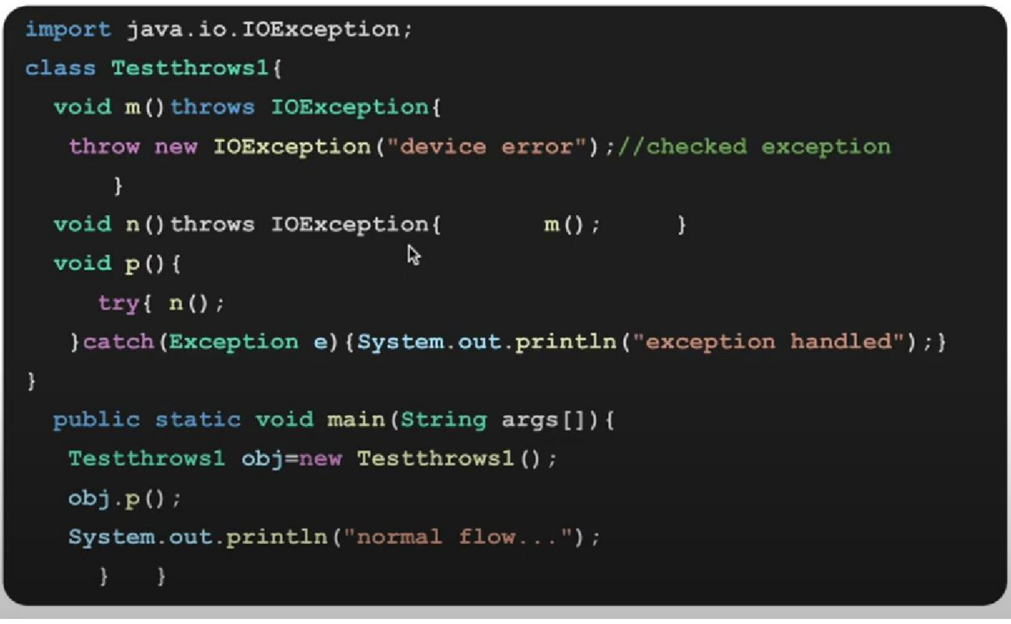


Key Concepts

- 'throw' keyword is used to throw an exception explicitly.
- But object thrown should be of type Throwable or subclass of throwable. Primitive types such as int, char … and non throwable classes like String cannot be used.
- Many built in runtime exceptions have at least two constructors, one that takes a string parameter describing the exception and one no argument constructor.
  - e.getMessage() can also be used.

'throws' keyword is used to specify the callers of the method, when it is capable of causing an exception and it does not handle.

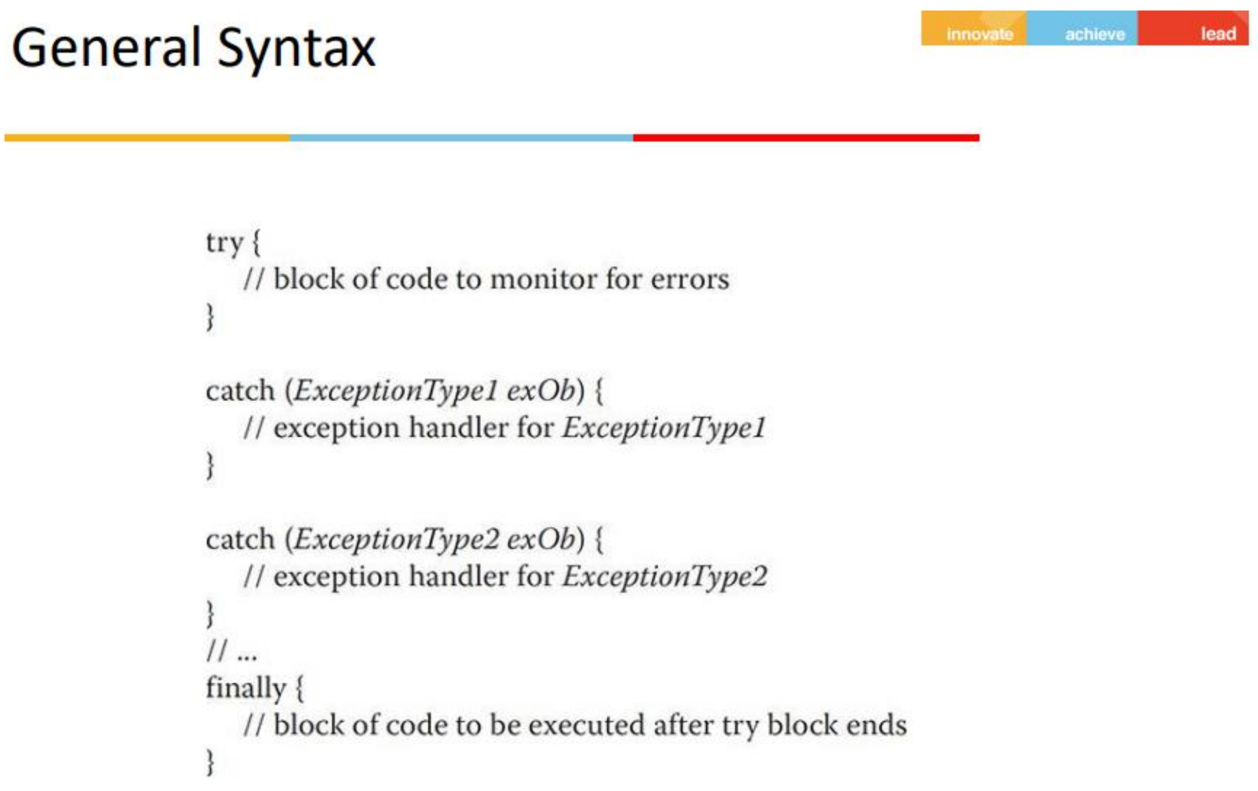
Throws



```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
   throw new IOException("device error");//checked exception
      }
  void n()throws IOException{        m();      }
  void p(){
     try{ n();
   }catch(Exception e){System.out.println("exception handled");}
}
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
     }   }
```

Difference between C++ and JAVA

- In C++, all types (including primitive and pointer) can be thrown as exception. But in Java only throwable objects (Throwable objects are instances of any subclass of the Throwable class) can be thrown as exception.
- In C++, there is a special catch called "catch all" that can catch all kind of exceptions. In Java, we can catch Exception object to catch all kind of exceptions.
- In Java, there is a block called finally that is always executed after the try-catch block. This block can be used to do cleanup work.
- In C++, all exceptions are unchecked. In Java, there are two types of exceptions – checked and unchecked.
- In Java, a new keyword throws is used to list exceptions that can be thrown by a function. In C++, there is no throws keyword, the same keyword throw is used for this purpose also.



General Syntax

```
try {
    // block of code to monitor for errors
}

catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}

catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
// ...
finally {
    // block of code to be executed after try block ends
}
```

Uncaught Exceptions : Uncaught Exception occurs **when an exception is not caught by a programming construct or by the programmer**, it results in an uncaught exception.

- First, it allows us to fix the error.
- Second, it prevents the program from automatically terminating.
- To guard against and handle a run-time error, simply enclose the code to be monitored inside a **try** block.
- Immediately following the **try** block, include a **catch** clause that specifies the exception type that we wish to catch.

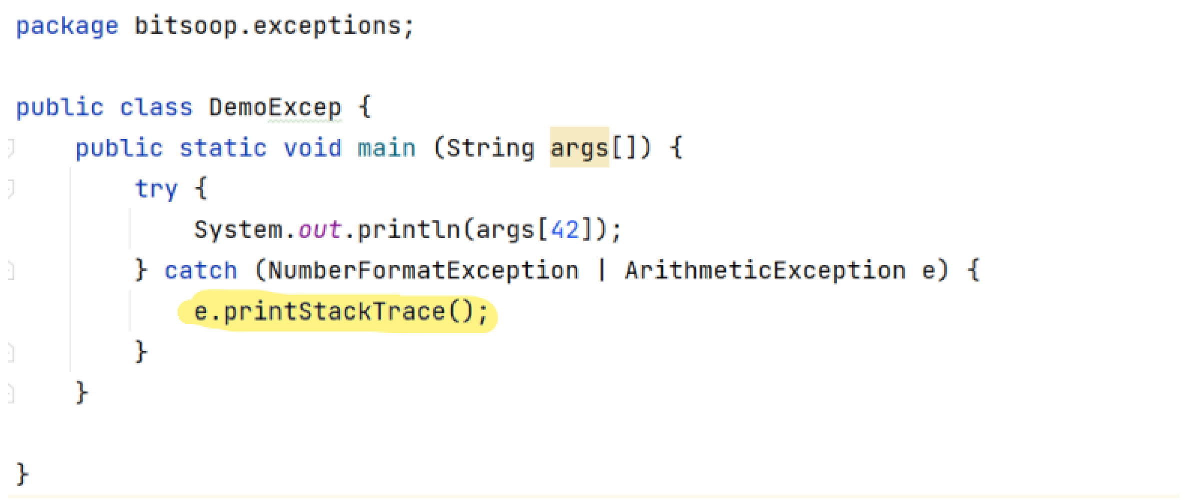
- A **try** and its **catch** statement form a unit.
- The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.
- A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements).
- The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) We cannot use **try** on a single statement.

The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

Multiple Catch Statements

- When we use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses.
- This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
- Thus, a subclass would never be reached if it came after its superclass.
- In Java, unreachable code is an error.

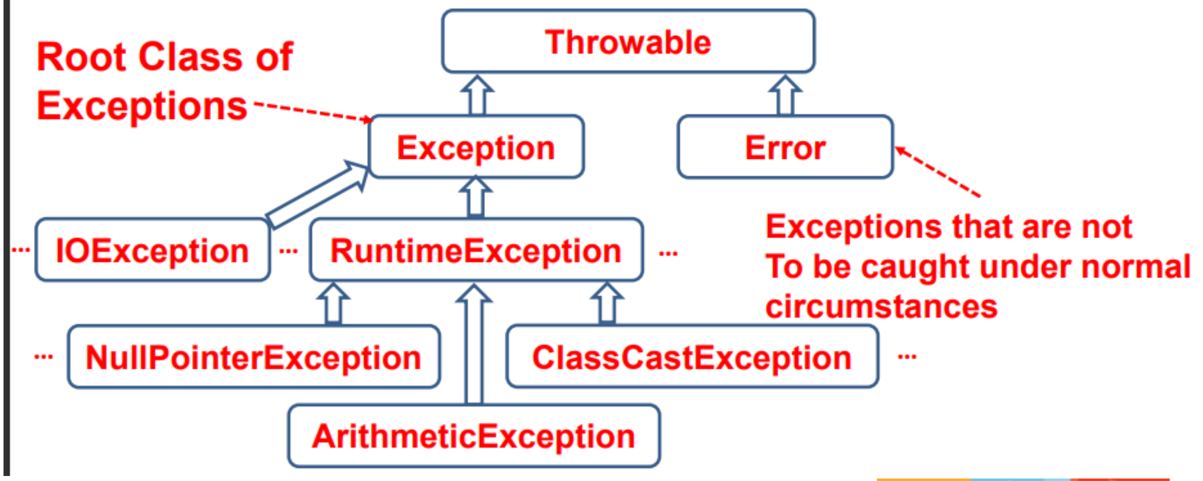
Catching multiple exceptions using one catch block



```java
package bitsoop.exceptions;

public class DemoExcep {
    public static void main (String args[]) {
        try {
            System.out.println(args[42]);
        } catch (NumberFormatException | ArithmeticException e) {
            e.printStackTrace();
        }
    }

}
```

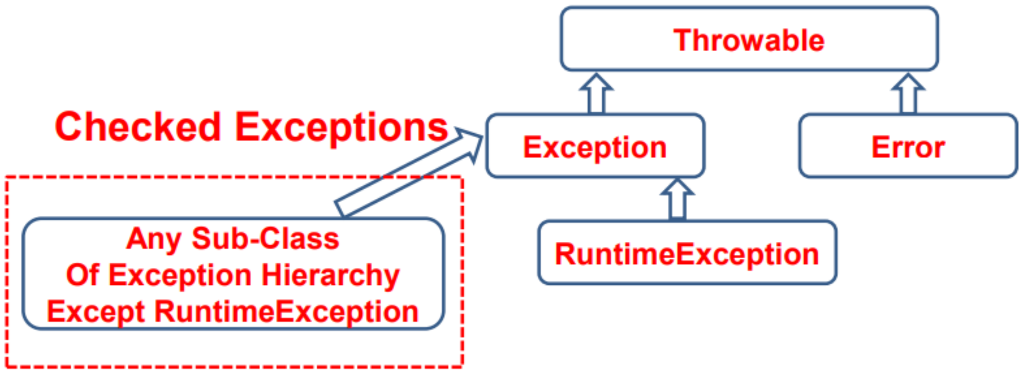
Exception Class Hierarchy

- ❑ *Throwable is the root class of Exception Hierarchy.*
- ❑ *Throwable has two direct subclasses namely Exception and Error*



Exception Types : Checked Exceptions

- ❑ Checked at Compile-Time
- ❑ Any Sub-class of Exception Except RuntimeException
- ❑ If a method throws any Checked Exception then Either it has to Handle the Exception or It must specify Exception Using throws clause


Checked Exceptions
Throwable
Exception
Error
RuntimeException
Any Sub-Class Of Exception Hierarchy Except RuntimeException

Red dotted box : Un-Checked Exceptions

Some Built-in Exceptions

Un-Checked Exceptions	Checked Exceptions
1. ArithmeticException	1. IOException
2. ArrayIndexOutofBoundsException	2. CloneNotSupportedException
3. ArrayStoreException	3. ClassNotFoundException
4. ClassCastException	
5. NullPointerException	
6. StringIndexOutofBoundsException	
7. NumberFormatException	
8. IndexOutofBoundsException	

Nested Try Statements

Try{ } statements can be nested. One try block may contain another try block

In case of nested try blocks, context of that exception is pushed onto stack.

Inner try block may/or may not have catch statements associated with it.

If an exception is thrown from inner try block then first inner catch statements are matched (if present) . If no match is found then outer try block are matched. If there also no match found then default handler will be invoked.

However, if outer try block throws the exception then only outer try blocks are matched.

SOLID Principles

S Stands for **Single responsibility principle** which states that there should be only one reason to change the class and nothing more than that.

O Stands for "**Open to Extension Closed for Modification**" which means that if a class is already in prod/live, do not touch it. Modification of that class is closed and any required changes should be in classes that extend this class.

L is for Liscov Substitution Principle which states that

"if a class A is a superclass of Class B, and an object of A is defined, then a replacement of object A with Object B should not break the code"
In simple terms, Class B should have all functionalities of Class A and should not narrow down the functionalities of Class A.

I is for **Interface Segmented Principle** which means that the Interfaces should only have aspects that are required and unnecessary parts of it should not be there. In other words, Keep only required functions in an interface and move the other functions to different interfaces.

Eg: a waiter need not have a washDishes/CookFood function which has to be defined in it.

D is for Dependency Inversion Principle which means that classes should depend on interfaces rather than concrete classes.

All this is done to

- Avoid duplicate code
- Ease of maintenance
- Ease of understanding
- Flexible software
- Reduce complexity

Creational Design Patterns-

https://www.youtube.com/watch?v=OuNOyFg942M

Prototype Pattern: Removes the overhead of creating costly objects and enables us to create a clone and make required changes from the existing object.

Singleton Pattern: Ensures a class has only one instance- eg: only one db connection, ensured in 4 ways:

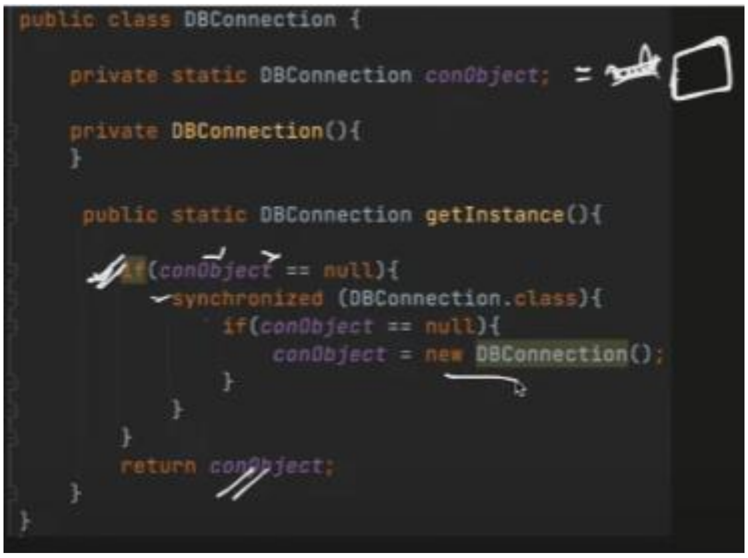
Eager, lazy, Synchronized Method, Double Locking

Eager: Make the "dbConnection" static so that only one method exists irrespective of number of classes(static loading)

Lazy: Method creates the object only if required. (Runtime initialization)(If 2 threads come and enter the function simultaneously, 2 dbconnections will be made : this is the issue)

Synchronized: make the method as synchronized to tackle the issue stated above.(Expensive as all attempts to get the dbconnection variable will use a lock coz of the synchronized method which is expensive)

Double checking:



```
public class DBConnection {

    private static DBConnection conObject;

    private DBConnection(){
    }

     public static DBConnection getInstance(){

        if(conObject == null){
            synchronized (DBConnection.class){
                if(conObject == null){
                    conObject = new DBConnection();
                }
            }
        }
        return conObject;
    }
}
```

This tackles the issue from the synchronized part.

Factory Method: Provides an interface for creating objects in a superclass which the subclasses can override and alter. Keeps all the object creation and business login in the same place.

Abstract Factory: Produces family of related objects without providing concrete classes (factory of factories)

Builder Method: to create an object step by step.

Structural Design Patterns-

https://www.youtube.com/watch?v=WxGtmIBZszk&t

Way to combine or arrange diff classes and obj to form a complex or bigger structure to solve a particular requirement.

(is a/ has a relations…)

Decorator Pattern

Adds more functionality to existing obj without changing its structure.

Keep adding extra features on the base layer.

Saves us from class explosion

Proxy Pattern

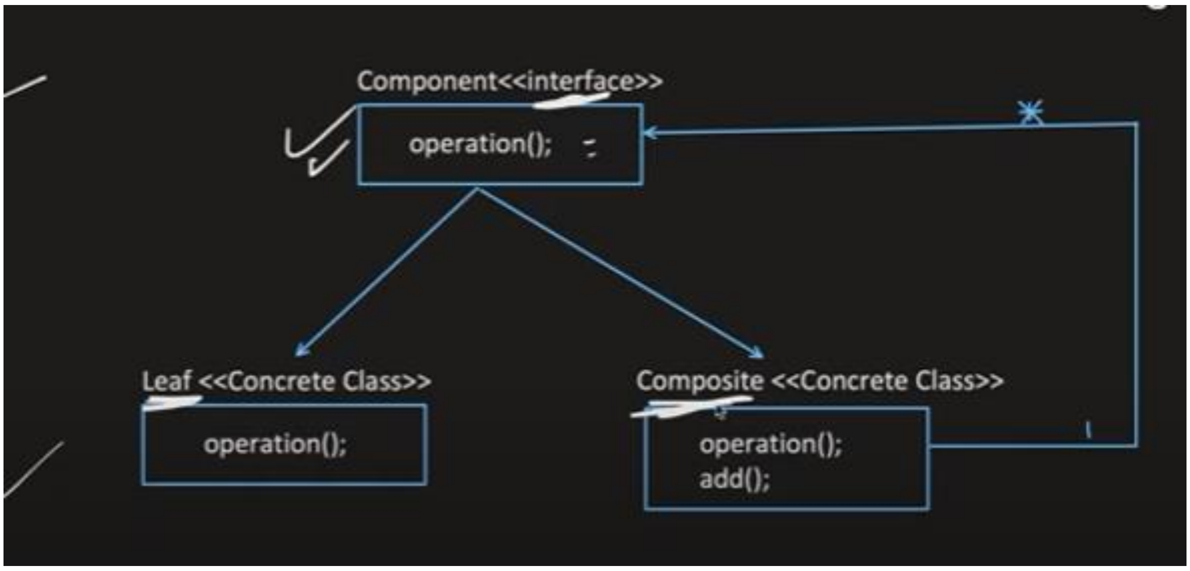
Helps provide a control access to an original object.

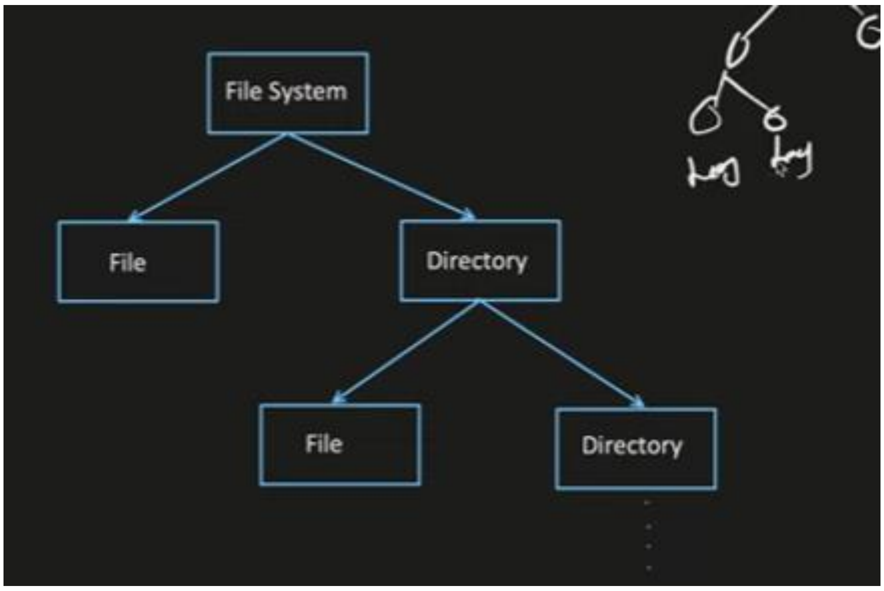
Its like only of some conditions are met you do some actions, else do something else;

Eg: if admin calls this function, then give back the object, else throw an error

Composite Pattern

Helps in cases where we have objects inside objects(tree like structure)

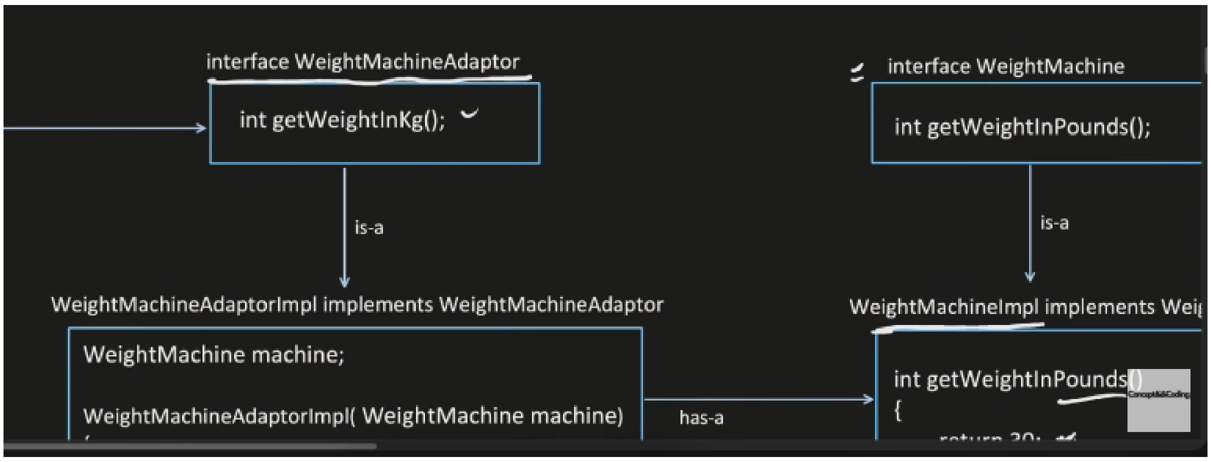


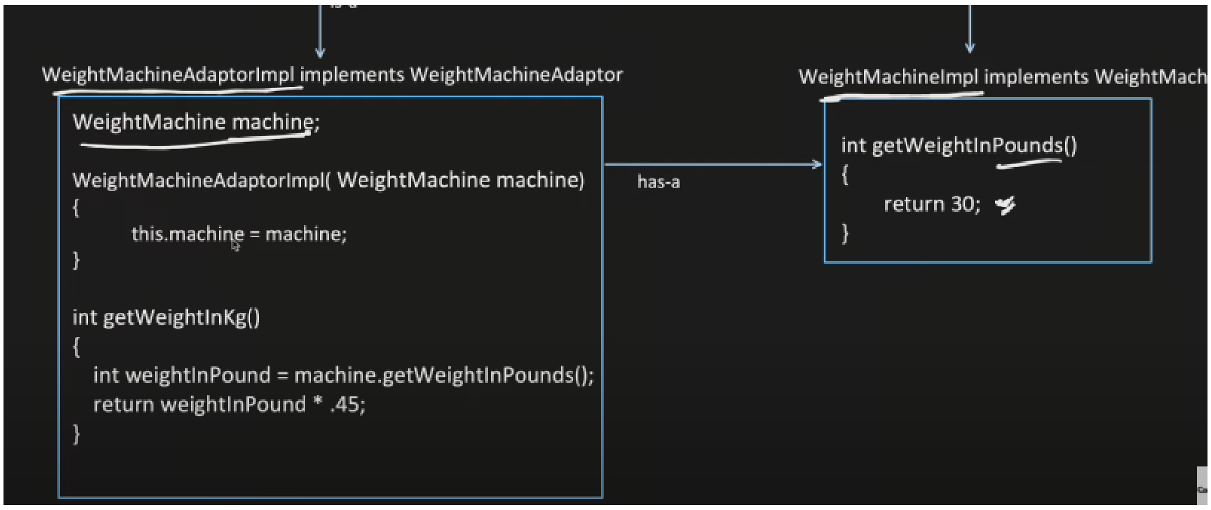


File is a leaf node and Directory is a composite node.

Adaptor Pattern:

Acts as a bridge or intermediate between 2 incompatible interfaces


interface WeightMachineAdaptor
int getWeightInKg();
is-a
WeightMachineAdaptorImpl implements WeightMachineAdaptor
WeightMachine machine;
WeightMachineAdaptorImpl(WeightMachine machine)
has-a
interface WeightMachine
int getWeightInPounds();
is-a
WeightMachineImpl implements Wei
int getWeightInPounds()
{



```
WeightMachineAdaptorImpl implements WeightMachineAdaptor
WeightMachine machine;

WeightMachineAdaptorImpl( WeightMachine machine)
{
        this.machine = machine;
}

int getWeightInKg()
{
   int weightInPound = machine.getWeightInPounds();
   return weightInPound * .45;
}
```

```
WeightMachineImpl implements WeightMach
int getWeightInPounds()
{
      return 30;
}
```

One interface wants weight in pounds and another wants it in kg.

Bridge Pattern

Helps in decoupling abstraction from implementation so that they can vary independently

Façade Pattern

Explore only required details to the client.

Hides system complexity from client.

Flyweight Pattern

Reduces memory used by sharing data among multiple objects.

Intrinsic data: shared among objects and remain same once defined with a value.

Extrinsic data: changes based on client input and defers from one object to another.

From object, remove all the Extrinsic data and keep only intrinsic data (the object is called Flyweight Object)

Extrinsic data can be passed in the parameter to the flyweight class

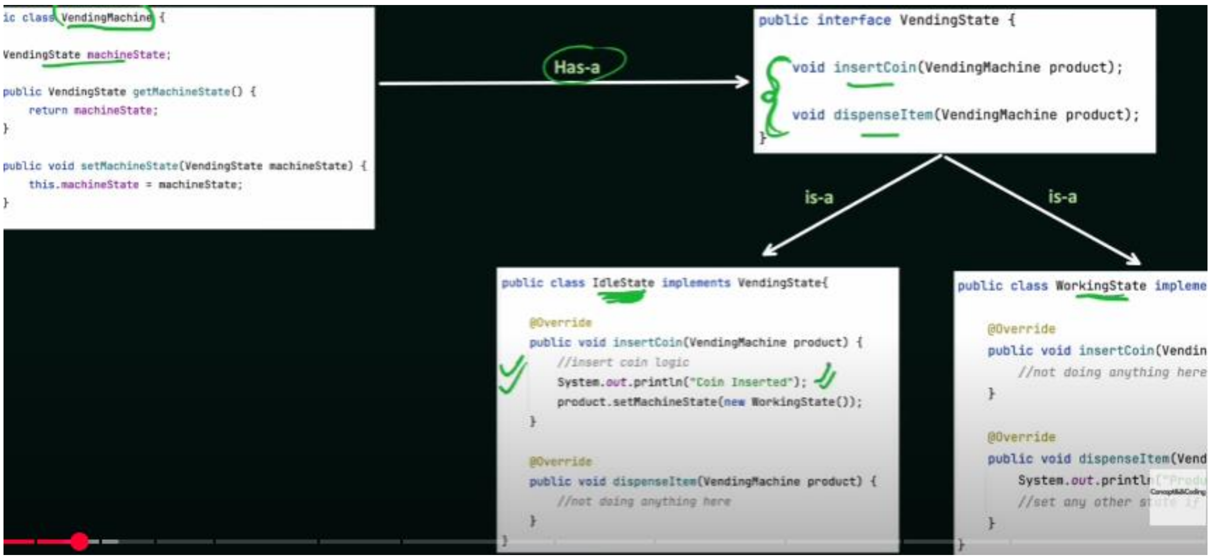
Caching can be used for the flyweight object and used whenever required.

Behavioral Design Pattern

Guides how different objects communicate with each other efficiently and distribute tasks efficiently, making software system flexible and easy to maintain.

State Pattern

Allows an object its behaviour when its internal state changes.



```java
ic class VendingMachine {

VendingState machineState;

public VendingState getMachineState() {
    return machineState;
}

public void setMachineState(VendingState machineState) {
    this.machineState = machineState;
}
```

```java
public interface VendingState {

    void insertCoin(VendingMachine product);

    void dispenseItem(VendingMachine product);
}
```

```java
public class IdleState implements VendingState{

    @Override
    public void insertCoin(VendingMachine product) {
        //insert coin logic
        System.out.println("Coin Inserted");
        product.setMachineState(new WorkingState());
    }

    @Override
    public void dispenseItem(VendingMachine product) {
        //not doing anything here
    }
}
```

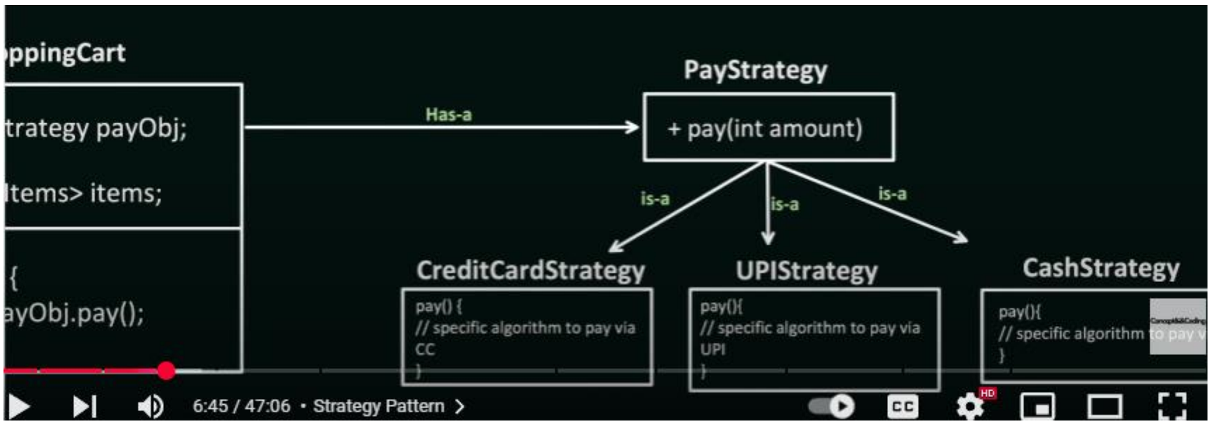
```java
public class WorkingState impleme

    @Override
    public void insertCoin(Vendin
        //not doing anything here
    }

    @Override
    public void dispenseItem(Vend
        System.out.printl
        //set any other s
    }
}
```

Observer Pattern

In this an object (Observable) maintains a list of its dependents(observers) and notifies all of its dependents.

Strategy Pattern:

Helps to define multiple algos for the task and we can select any algo depending on the situation.



```
ppingCart
trategy payObj;
Items> items;

{
ayObj.pay();
```

```
PayStrategy
+ pay(int amount)
```

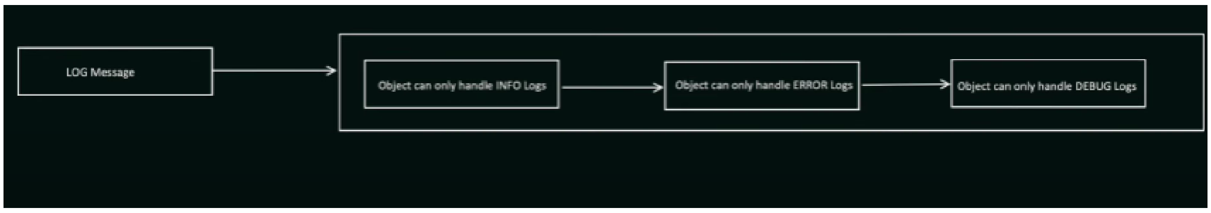
```
CreditCardStrategy
pay() {
// specific algorithm to pay via
CC
}
```

```
UPIStrategy
pay(){
// specific algorithm to pay via
UPI
}
```

```
CashStrategy
pay(){
// specific algorithm
}
```

Chain of Responsibility Pattern:

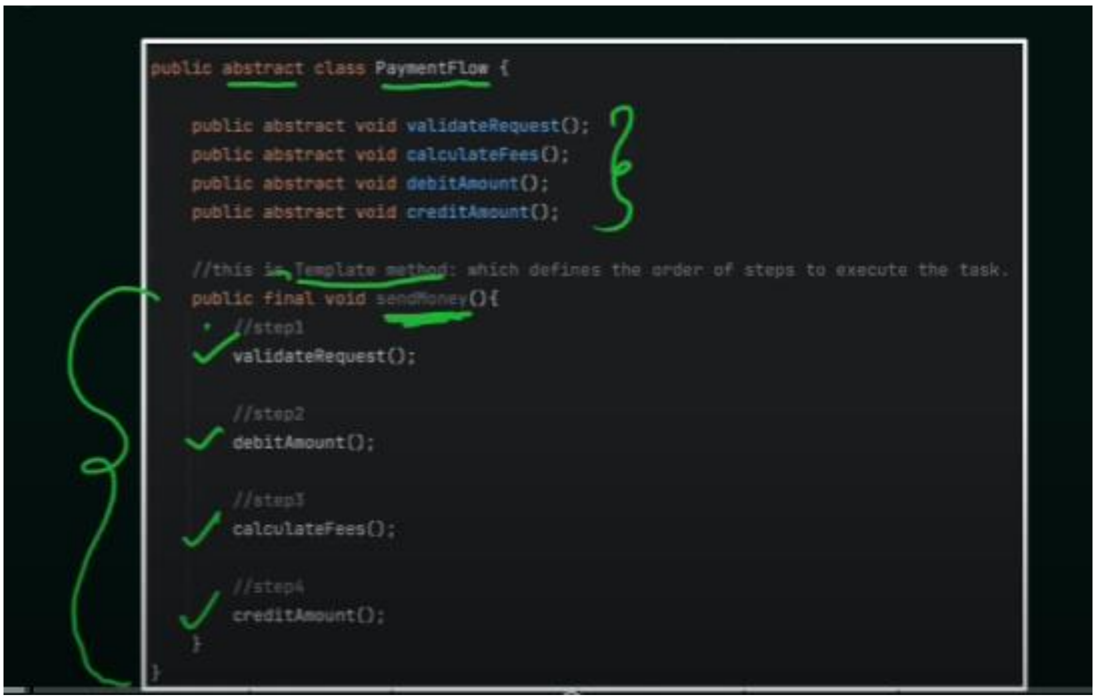
Allows multiple objects to handle a request without the sender needing to know which object will ultimately process it.


LOG Message
Object can only handle INFO Logs
Object can only handle ERROR Logs
Object can only handle DEBUG Logs

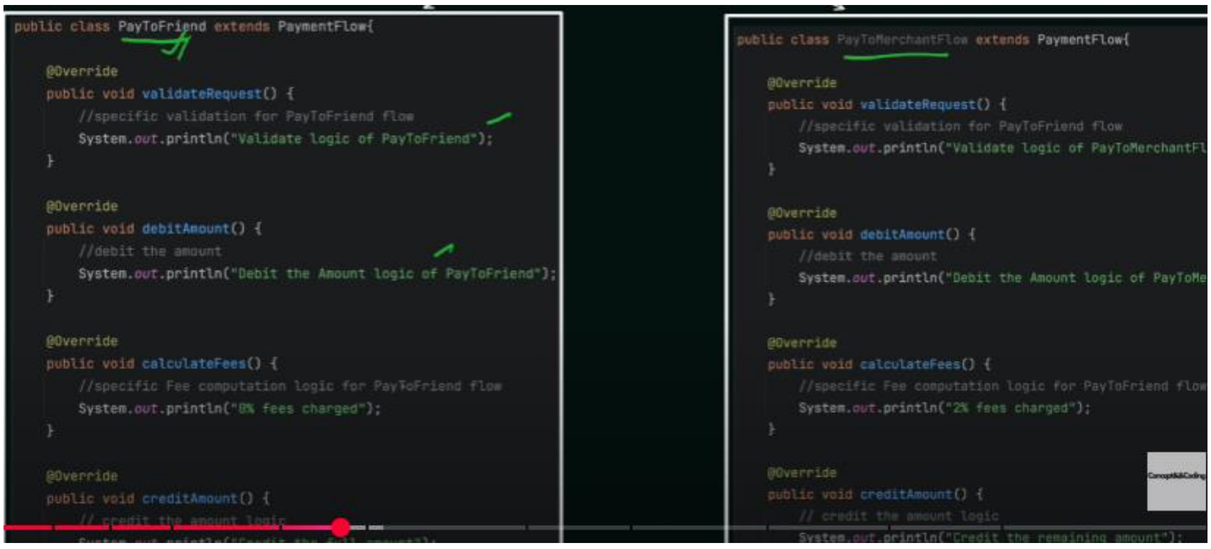
Log messages are passed internally to the correct object

Template Method pattern :

When you want all classes to follow specific steps to process the tasks but provide flexibility that each class can have their own logic in that specific step.



```
public abstract class PaymentFlow {

    public abstract void validateRequest();
    public abstract void calculateFees();
    public abstract void debitAmount();
    public abstract void creditAmount();

    //this is Template method: which defines the order of steps to execute the task.
    public final void sendMoney(){
        //step1
        validateRequest();

        //step2
        debitAmount();

        //step3
        calculateFees();

        //step4
        creditAmount();
    }
}
```

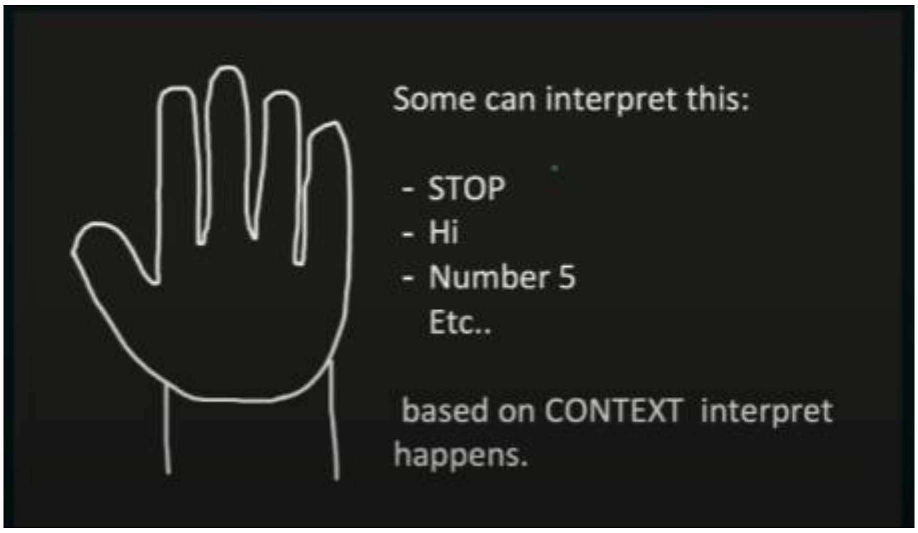


```java
public class PayToFriend extends PaymentFlow{

    @Override
    public void validateRequest() {
        //specific validation for PayToFriend flow
        System.out.println("Validate logic of PayToFriend");
    }

    @Override
    public void debitAmount() {
        //debit the amount
        System.out.println("Debit the Amount logic of PayToFriend");
    }

    @Override
    public void calculateFees() {
        //specific Fee computation logic for PayToFriend flow
        System.out.println("0% fees charged");
    }

    @Override
    public void creditAmount() {
        // credit the amount logic
        System.out.println("Credit the full amount");
```

```java
public class PayToMerchantFlow extends PaymentFlow{

    @Override
    public void validateRequest() {
        //specific validation for PayToFriend flow
        System.out.println("Validate logic of PayToMerchantFl
    }

    @Override
    public void debitAmount() {
        //debit the amount
        System.out.println("Debit the Amount logic of PayToMe
    }

    @Override
    public void calculateFees() {
        //specific Fee computation logic for PayToFriend flow
        System.out.println("2% fees charged");
    }

    @Override
    public void creditAmount() {
        // credit the amount logic
        System.out.println("Credit the remaining amount");
```

Interpreter Pattern

Defines a grammar for interpreting and evaluating an expression



Some can interpret this:

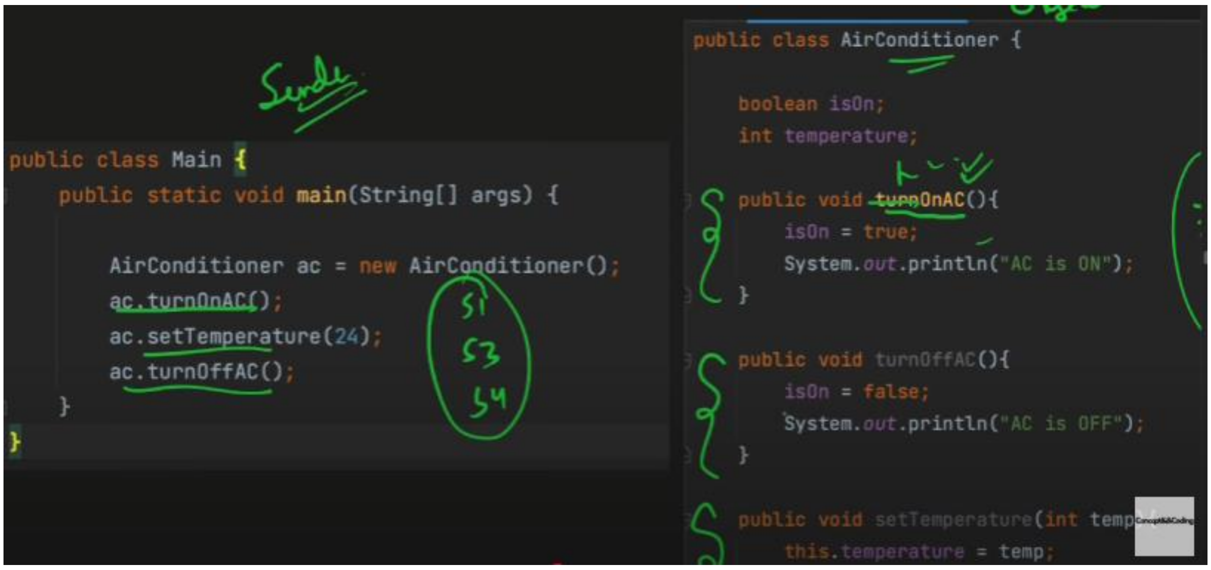
- STOP
- Hi
- Number 5

  Etc..

based on CONTEXT interpret happens.

Command Pattern:

Turns the requests into objects, allowing you to either parameterize or queue them. This will help to decouple the request sender and receiver.



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

        AirConditioner ac = new AirConditioner();
        ac.turnOnAC();
        ac.setTemperature(24);
        ac.turnOffAC();
    }
}
```

```
public class AirConditioner {

    boolean isOn;
    int temperature;

    public void turnOnAC(){
        isOn = true;
        System.out.println("AC is ON");
    }

    public void turnOffAC(){
        isOn = false;
        System.out.println("AC is OFF");
    }

    public void setTemperature(int temp
        this.temperature = temp;
```

In the image above, sender needs to know how to turn on the ac. This is what we wanna decouple. Above is tightly coupled

Turning on ac might involve multiple steps and sender need not know all the steps.

How to handle:

Create an object for each command

Iterator Pattern:

Provide a way to access elements of a collection sequentially without knowing the underlying representation of the collection.

Eg: collections in java

Visitor Pattern

Allows adding new operations to existing classes without modifying them and encourages OPEN/CLOSE Principle

Didn't understand this one

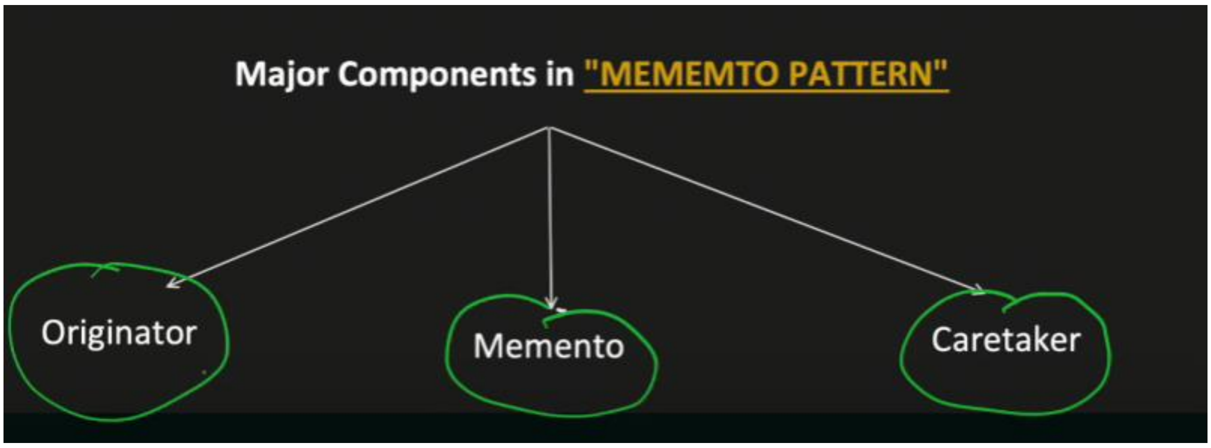
Mediator Pattern:

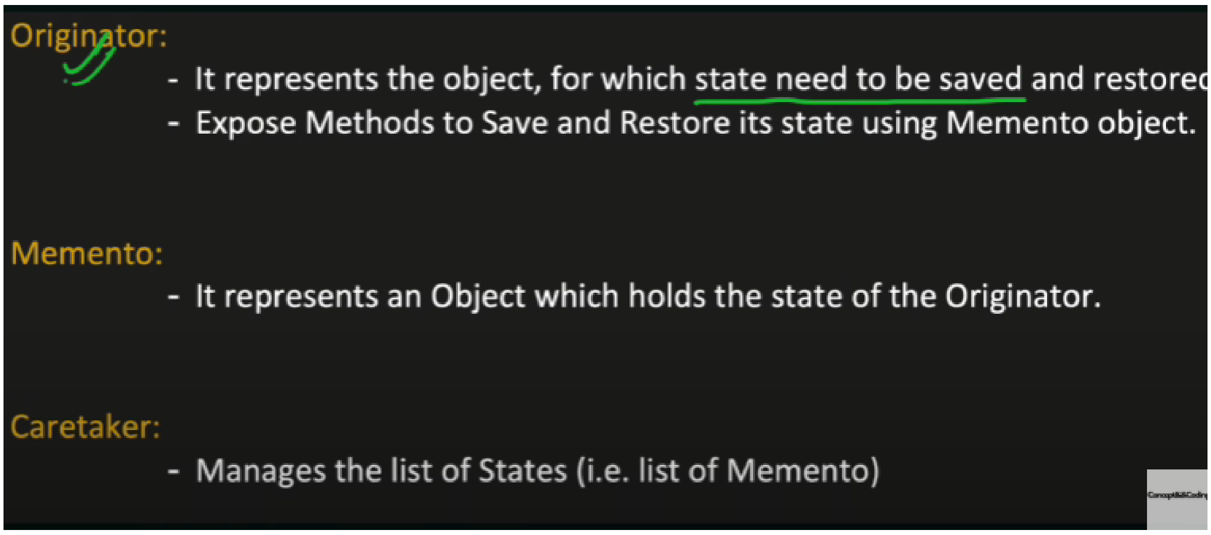
It encourages loose coupling by keeping objects from referring to each other explicitly and allows them to communicate through a mediator object.

Eg: bidding, people talk to the bidder and not each other.

Memento Pattern: Enables "UNDO" Capability

Provides an ability to revert an object to a previous state and it does not expose the object internal implementation.





Originator:
- It represents the object, for which state need to be saved and restored
- Expose Methods to Save and Restore its state using Memento object.

Memento:
- It represents an Object which holds the state of the Originator.

Caretaker:
- Manages the list of States (i.e. list of Memento)