Design Patterns

→ write reusable and extensible object oriented code

3 categories ① creational → ways to create objects
② structural :→ relationships between objects
③ Behavioural :→ communication / interaction between objects

*) classes (Package / namespace determines where the given class will be declared and defined)
static → these members can be called directly without creating an instance / object of the class, classes consists of fields / attributes / data members, and operations / methods / member functions which can be performed on that data

Constructor : gets called automatically when a new instance of a class is created [you can overload a constructor, constructor with no arguments is called default constructor]

Coupling : tells us how much a given class is coupled or dependent on another class

example :

```
public class main {
    public static void main() {
        User user = new User("abc");
        user.sayHello();
    }
}
```

main class is dependent on the user class

if we make changes to the constructor of user class we will have to make changes to the main class too

→ all the classes dependent on a particular class must be recompiled when we make changes to a particular class.
→ thus coupling adds a lot of overhead our application must allow us to change components independently thus we must reduce coupling

to reduce coupling we use Interfaces
Interfaces: A contract that specifies the capabilities a class should provide
example: consider a hotel which wants a chef (interface) a chef is someone with specific capabilities [you are not dependent on a particular chef, you will be satisfied with any chef] loosely coupled system
Example

```
public Interface Topic
{
    void understand();
}

class Topic1 implements Topic {
    public void understand()
    {
        System.out.println("Got it");
    }
}
class Topic2 implements Topic {
    public void understand() {
        System.out.println("understood");
    }
}
```

```
public class subject {
    public static void main (String[] args) {
        Topic t = new Topic1 ();
        t.understand ();
        t = new Topic2 ();
        t.understand
    }
}
```

objects get resolved at runtime

resolved at runtime

runtime polymorphism

objects get resolved on run time

*) loose coupling allows us to use dependency injection while writing unit tests for our code, some times we may want to test a very small functionality we added to a class which needs complex classes to work properly (dependent on complex classes / database queries), we leave the creation/injection of the objects on the framework and test the code we wrote

Encapsulation: bundling data, methods together in a class and hiding state of the object inside a class, this allows us to create robust applications by preventing objects from going into invalid state.

example [getters/setters]

```
public class Account {
    private float balance;

    public void setBalance (float amount) {
        if (balance > 0)
            this.balance = amount;
    }
```

we can use setters to validate the data before setting it which will prevent the object from going into

LOOSE Coupling, Dependency Injection And unit tests are written in the next page

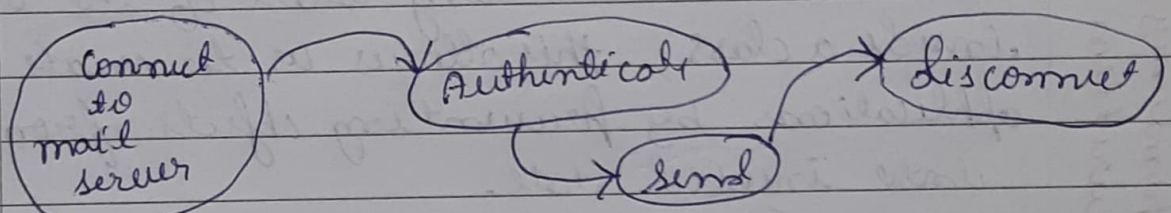
Invalid state, allows us to check if ~~Error~~ Incoming data is valid (Robust Application)

```
                    ↑ float
public get Balance () {
        return this.balance;
}                                   } getter
}
```

Abstraction : Reduce complexity by hiding unnecessary details

Example: lets say we are developing a mail service class, and we want a functionality to send an email

flow for sending Email



which is a very complex flow, we don't want others to get a complicated view of the functionality (when a programmer uses our mail service library to send mails)

*) we add 3 methods Authenticate, connect, Disconnect to the mail service ~~class~~ class to implement the functionality, they add a layer of complexity so we want to hide the 3 methods and reduce complexity (Abstraction)

we hide the unnecessary implementation details by declaring the ~~class~~ methods as private and reduce complexity

```
public class MailService {

    public void SendEmail() {
        connect();
        authenticate();
        // Send email
        disconnect();
    }

    private void connect(int timeout) {
        System.out.println("connect");
    }

    private void Disconnect() {
        System.out.println("Disconnect");
    }

    private void Authenticate() {
        system.out.println("Authenticate");
    }
}
```

*) abstraction helps us hide any changes we make to the implementation details.

__/__/____

Inheritence: Mechanism for reusing code (Let's say we want to build a UI framework with 2 objects like Textbox, button, check box

→ All these objects will have some common behaviour: Enable() / Disable(), focus() on the object, setPosition()

→ when implementing these classes we don't want to implement these methods in every single class (this will cause a lot of code duplication

we can implement all the common behaviours in the parent/base class and have all the other classes inherit this behaviour

```
public class UI Control {
    public void Enable() {
        System.out.println("Enable");
    }
}

public class TextBox extends UI Control {
    // Text Box behaviour
}

public class CheckBox extends UI control {
    // check Box behaviour
}
```