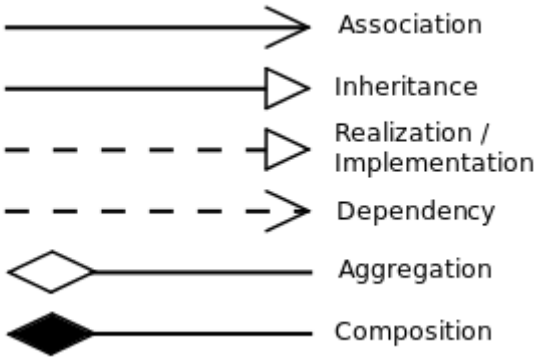
LLD

- **Solid Principles**
- **Design Patterns**
- **Case Studies**



SOLID Principles

The solid principles are a set of basic guidelines for designing Object Oriented Systems so that they are:

1. Easy to maintain
2. Easy to understand
3. Flexible
4. Reduce complexity
5. Avoid duplication of code

The 5 SOLID principles

1. S - Single Responsibility principle
2. O - Open / Closed Principle
3. L - Liskov Substitution Principle
4. I - Interface Segmented Principle
5. D - Dependency Inversion Principle

Single Responsibility Pattern
A class should have only one reason to change, i.e. each class should have a single dedicated responsibility.

```
class Marker {
    String color;
    String name;
    int price;

    public Marker(String color, String name, int price) {
        this.color = color;
        this.name = name;
        this.price = price;
    }
}

class Invoice {
    private Marker marker;
    private int quantity;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }
```

```
    public int calculateTotalPrice() {
        return (this.marker.price) * this.quantity;
    }

    public void printInvoice() {
        // print Invoice
    }

    public void saveToDB() {
        // Save to DB
    }
}
```

In the above example we have 2 classes Marker and Invoice, the Invoice class does not follow the single responsibility principle as it has multiple reasons to change, for example the Invoice class can change if we wish to add a discount and GST to our final price calculation logic, it can change if we wish to change how the invoice is printed, further it can change if we need to change the DB updation logic (eg SQL to NoSQL transition), or maybe we want to save the data to a file now.

A design which follows the single responsibility principle would split this functionality into 3 parts:
=> Invoice Price Calculator
=> Invoice Printer
=> Invoice DAO

```
class Marker {
    String color;
    String name;
    int price;

    public Marker(String color, String name, int price) {
        this.color = color;
        this.name = name;
        this.price = price;
    }
}

class Invoice {
```

```
    private Marker marker;
    private int quantity;
    private Timestamp createdAt;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }
}

class InvoiceCalculator {
    private Invoice invoice;
    public InvoiceCalculator(Invoice invoice) {
        this.invoice = invoice;
    }

    public int calculateTotalPrice() {
        return (this.marker.price) * this.quantity;
    }
}

class InvoicePrinter {
    private Invoice invoice;
    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }

    public void printInvoice() {
        // print Invoice
    }
}

class InvoiceDAO {
    private Invoice invoice;
    public InvoiceDAO(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToDB() {
        // Save to DB
    }
}
```

Now each class has a single responsibility and hence a single reason to change.

Open / Closed Principle

The Open / Closed principle states that software entities (classes, modules or functions) should be open for extension, but closed for modification.

For example suppose we want to add a method to the invoice DAO class to save the invoice to the file.

We can do so by adding a saveToFile method straight into the invoiceDAO class, however that would be a violation of the Open / Closed principle. Think of it this way, the InvoiceDAO class might already be in production and handling traffic so modifying it is not a safe option as it can generate errors, if we want to use some of its functionality a better way would be to extend it.

```
interface InvoiceDAOInterface {
    public void save(Invoice invoice);
}

class DataInvoiceDAO implements InvoiceDAOInterface {
    @Override
    public void save(Invoice invoice) {
        // Save to DB
    }
}

class FileInvoiceDAO implements InvoiceDAOInterface {
    @Override
    public void save(Invoice invoice) {
        // Write to file
    }
}
```

This means we can extend our existing classes, with any behaviour we want, however it is closed for modification so we cannot alter the existing code.

Liskov Substitution Principle

If class B is a subtype of class A, then we should be able to replace object of A with B without breaking the behavior of the program. In other words the subclass should extend the capability of the parent class not narrow it down.

Suppose we have a Bike interface, and two concrete classes MotorCycle, and Bicycle

```
interface Bike {
    public void turnOnEngine();
    public void accelerate();
}

class MotorCycle implements Bike {
    @Override
    public void turnOnEngine() {
        // turn on engine
    }

    @Override
    public void accelerate() {
        // increases speed
    }
}

class Bicyle implements Bike {
    @Override
    public void turnOnEngine() {
        throw new UnsupportedOperationException("Unsupported
method 'turnOnEngine'");
    }

    @Override
    public void accelerate() {
        // increase speed
    }
}
```

The above example is a violation of Liskov Substitution principle, as the class Bicycle is reducing the capability of the parent class (by not supporting the turnOnEngine method), as a result we cannot replace object A with B without breaking the behavior of the program. Think of it

this way that we have been providing an object of Motorcycle to the Bike interface thus far, this works because MotorCycle does not narrow down the capability of the parent, however now we decide to pass objects of Bicycle, this will cause problems as the parent will try to turn on the engine which will lead to breakage in behavior of the program.

Technical definition - If the property P(x) is provable for any object x of type A, then P(y) should also be provable for any object y of type B; If B is a subtype of A. So whatever is true about the objects of the supertype, must also be true about the objects of the subtype.

Remember Covariant Return types are acceptable.
Note: No new exceptions, only subtypes

Interface Segregation Principle

Interfaces should be designed in such a way that clients should not implement any unnecessary functions that they do not need.

```
interface RestaurantEmployee {
    void cookFood();
    void serveCustomer();
    void washDishes();
}

class Waiter implements RestaurantEmployee {

    @Override
    public void cookFood() {
        // The waiter can't cook
    }

    @Override
    public void serveCustomer() {
        // Serve customers
    }

    @Override
    public void washDishes() {
        // That's not the waiter's job
```

```
    }
}
```

The above example is a violation of Interface Segmented Principle, because the waiter class (client) needs to implement unnecessary functions like cookFood() and washDishes() which are not relevant to them.

As the name suggests the solution is to segment the interfaces as much as possible so that the clients need not implement unnecessary functions.

```
interface WaiterIterface {
    void serveCustomer();
    void takeOrder();
}

interface ChefInterface {
    void cookFood();
    void decideMenu();
}

class Waiter implements WaiterIterface {
    @Override
    public void serveCustomer() {
        // serve
    }

    @Override
    public void takeOrder() {
        // Take Order now
    }
}

class Chef implements ChefInterface {
    @Override
    public void cookFood() {
        // cook now
    }

    @Override
    public void decideMenu() {
```

```
        // decide menu right now
    }
}
```

The above design divides the RestaurantEmployee interface into 2 parts - ChefInterface and WaiterInterface allowing the chef and waiter class to implement only the necessary functions.

In the Parent class put only very generic methods which are common for all the child classes.

Dependency Inversion Principle

Classes should depend on interfaces rather than concrete classes.

```
interface Keyboard {
    void onKeyPress();
}

interface Mouse {
    void onScroll();
}

class WiredKeyBoard implements Keyboard {
    @Override
    public void onKeyPress() {
        // key pressed
    }
}

class BluetoothKeyBoard implements Keyboard {
    @Override
    public void onKeyPress() {
        // key pressed
    }
}

class WiredMouse implements Mouse {
    @Override
    public void onScroll() {
```

```
        // scroll
    }
}

class BluetoothMouse implements Mouse {
    @Override
    public void onScroll() {
        // scroll
    }
}

class Laptop {
    WiredKeyBoard wiredKeyBoard;
    WiredMouse wiredMouse;

    public Laptop() {
        this.wiredKeyBoard = new WiredKeyBoard();
        this.wiredMouse = new WiredMouse();
    }
}
```

Consider the example below, in our Laptop design we are tied in to using a Wired keyboard and wired mouse, hence we cannot use a Bluetooth keyboard or mouse, this is an inflexible design, as we are using the concrete classes, a better approach would be for the class to depend on the interfaces rather than the concrete classes.

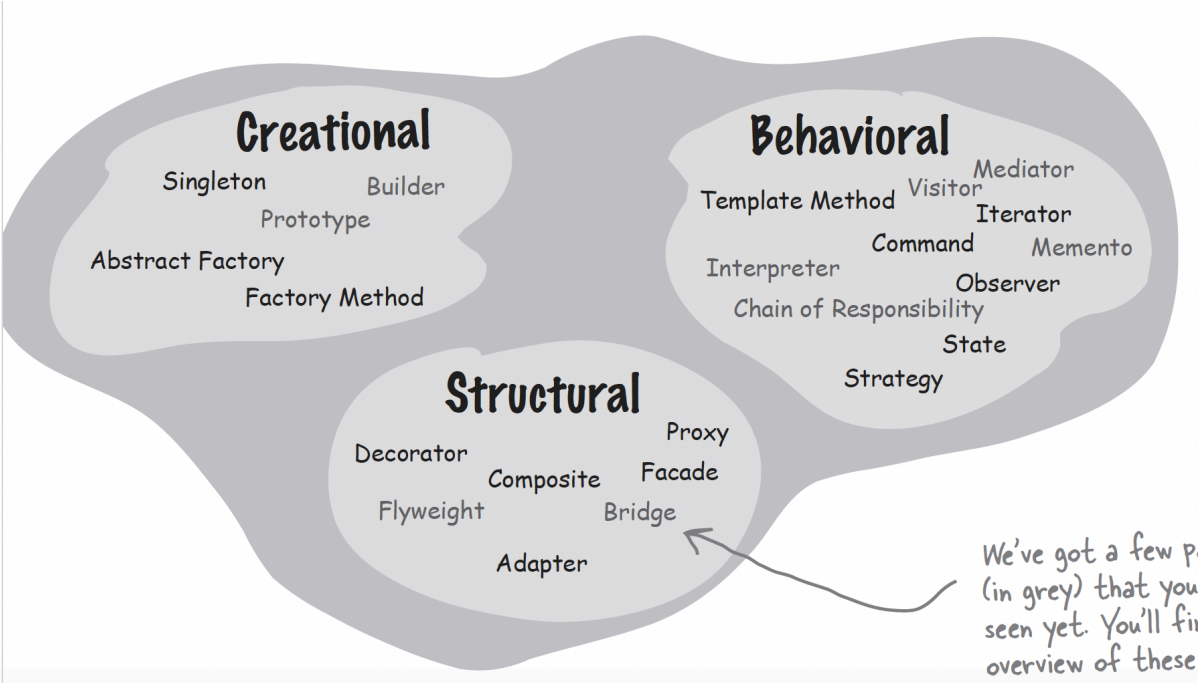
```
class Laptop {
    Keyboard keyboard;
    Mouse mouse;

    public Laptop(Keyboard keyboard, Mouse mouse) {
        this.keyboard = keyboard;
        this.mouse = mouse;
    }
}

class Device {
    Laptop laptop;
    Device() {
```

```
        laptop = new Laptop(new BluetoothKeyBoard(), new
BluetoothMouse());
    }
}
```

In this design the laptop class can have any combination of Keyboard and Mouse.

It can also be stated as: “Program to an interface, not to an implementation”.

Design Patterns



Design Patterns are principles of Object Oriented Programming which help in designing manageable, scalable and reusable software, consider design patterns as solutions to common problems encountered in the development process, design patterns serve as general templates and best practices for software design and development.

Types of Design Patterns

- Creational Design Patterns

- Structural Design Patterns
- Behavioural Design Patterns

Creational Design Patterns

Creational design patterns control the object creation, i.e. their responsibility is to create objects.

Creational Design patterns can be divided into 2 parts Object Creational and Class Creational.
Class Creational patterns use Inheritance to vary the class that's instantiated while Object creational patterns delegate instantiation to another object.

- Prototype Design Pattern
- Singleton Design Pattern
- Factory Design Pattern
- Abstract Factory Design Pattern
- Builder Design Pattern

Prototype Design Pattern

Prototype design pattern is used when we have to create a copy or clone of an existing object.

According to this pattern the responsibility of creating the clone should be taken by the class of which we are trying to create the clone. The alternative is for the client to handle the object cloning, however this method has some shortcomings. What if the class has some private variables how will they be cloned by the client? What if we want to perform some kind of conditional cloning, where not all the attributes should be cloned?

In such cases the class in question should implement a 'clone' method which performs the object cloning, this is beneficial since the class has total control over how the clone is created and now private variables can be copied as well (as the cloning method resides in the class itself).

The cloning method internally will create a new object of the class and copy all of the attributes of the original class to it, this is a deep copy.

Singleton Design Pattern

Singleton Design Pattern is used when we have to create one and only instance of the class, for example we would like to have only one instance of the DB Connection class.

The Singleton Design Pattern ensures a class has only one instance, and provides a global point of access to it.

Refer to the Java section for details on this.

Factory Design Pattern

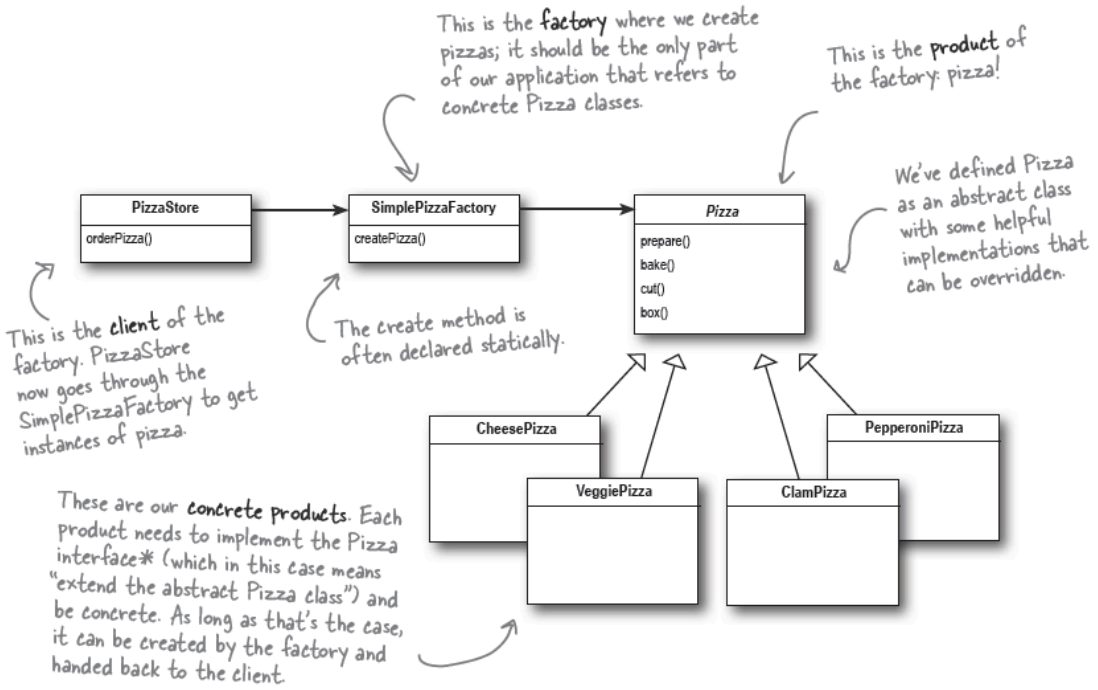
Factory is an object which handles the details of object creation. Other parts of the code or methods which need a concrete object will be the clients of this factory.

Factory design pattern prevents code duplication, as we have a separate and dedicated factory to take care of object creation, hence all the methods don't need to handle the details of object creation themselves,

i.e. the factory will handle all the concrete instantiations and the other parts of the application don’t need to worry about object creation.

Whenever they need a concrete object, the clients can just request the factory to get one.

Also for example, say we want to make some changes in the object creation logic, without a factory in place we’ll need to make changes in all the places where the object is being instantiated, however with a factory this change only needs to happen at one place. If the object creation logic is spread across the application, then even a minor change in the logic will require us to make changes in many places across the application.



It is a creational design pattern which is used for Object creation and it hides the complexity of the object creation from other modules.

Client code which is actually using the Factory Method has no idea about object creation. This pattern is useful when we need to create objects based on some conditions.

The client code will call a method exposed by the Factory Interface, this method will return the object the client is expecting, however this interface does not expose the object creation complexity to the client. The

object creation logic itself is implemented by the factory.

=> The factory design pattern is useful for creation of objects that fall under the same categorization but still have different properties.
=> It helps in hiding complexity.

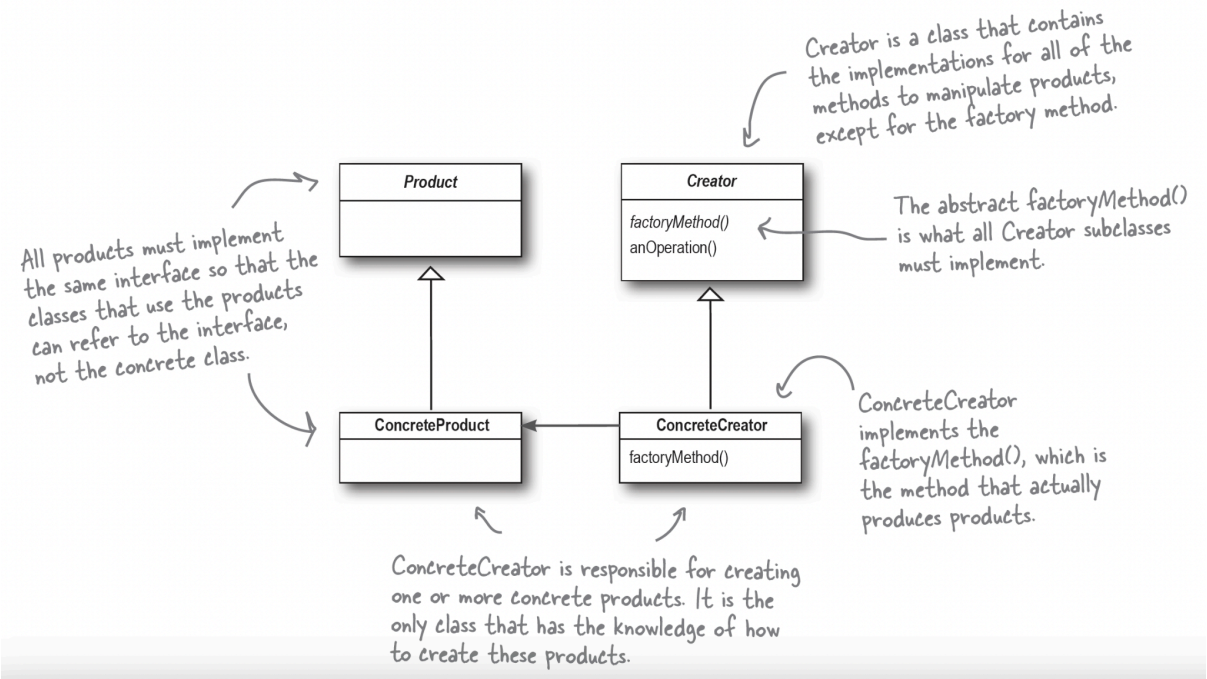
Factory Method Design Pattern

In the Simple Factory Design Pattern we delegate the responsibility of object creation to another object. The Factory method design pattern allows a class to defer the object instantiation to its subclasses, in other words the Factory Method design pattern allows the subclasses to decide which concrete class to instantiate.

This pattern uses a factoryMethod which takes care of instantiating the right object, this method is implemented by the subclasses, hence the object creation is deferred to the subclasses.

This pattern divides the design into Creators and Products, the Creator is an abstract class with an abstract factoryMethod, and other default methods which operate on the object returned by the factoryMethod. The subclasses need to implement this factoryMethod, which decouples the client of the object (in this case the superclass methods) from the actual details of object creation (in the subclass).

Concrete Creator is responsible for creating one or more concrete products, it is the only class which has the knowledge of how to create these concrete products. Concrete Creator is the class which implements the factoryMethod, which is the method that produces products.



Abstract Factory Design Pattern

Factory of Factory, in this design pattern we'll have 2 levels of factories, the first level will actually return a factory to the client. The second factory is the one which will actually create objects.

We use an Abstract factory producer to return the appropriate factory to the client, and the client can use this factory to generate the desired objects, and obviously the Abstract producer is itself a factory too.

This pattern can be seen as a Factory of Factory Pattern. Like Factory pattern this is a creational design pattern. It is used for creating families of related or dependent products.

The Abstract Factory pattern is useful when we have different types of products and we can group certain products together.

For example, instead of having a single Car factory which generates objects for all types of cars, we can have multiple factories each responsible for generating different kinds of cars, for example a factory responsible for generating luxury cars, another one responsible for General purpose cars etc.

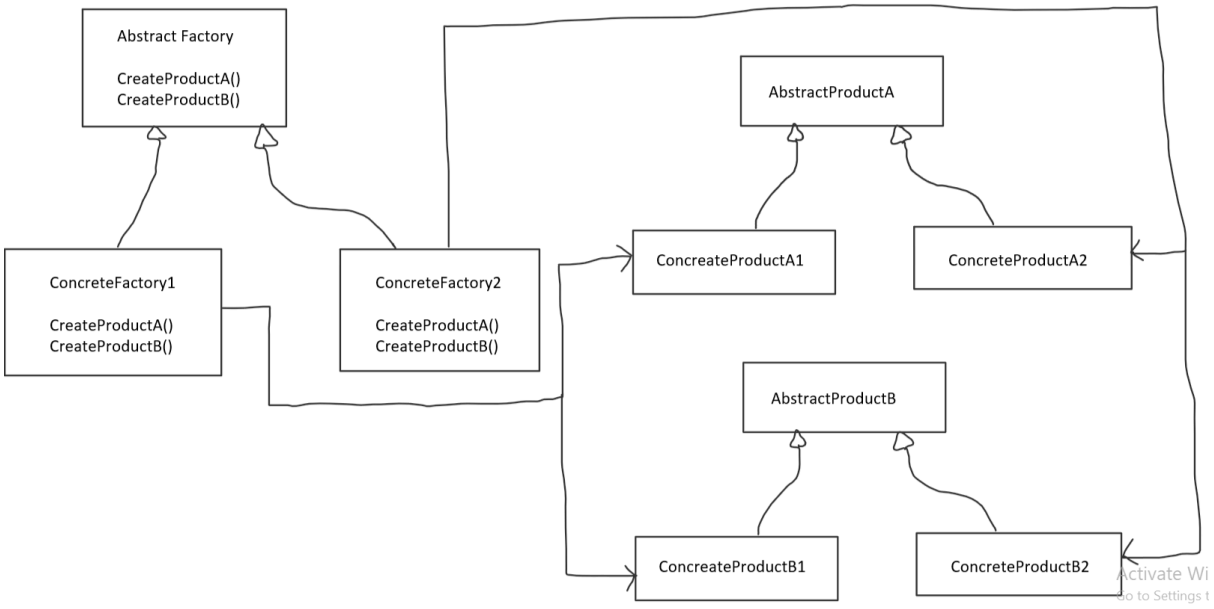
To do this we define a Vehicle Factory interface which is subclassed by

different types of car manufacturing factories, this factory is responsible for returning the actual car object. In addition we have another factory which returns an object of the required Vehicle factory.

The outer factory (the one which returns a factory) has some logic to decide which factory object to return, in this case some logic which evaluates the input and determines whether to return a luxury car Vehicle factory object or Ordinary car Vehicle factory object.

We can call methods on the returned factory object to get the actual product.

Factory Method Pattern uses Inheritance
Simple Factory and Abstract Factory use Composition.



Builder Design Pattern

The Builder design pattern is used when we need to create an object step by step.
Example: StringBuilder

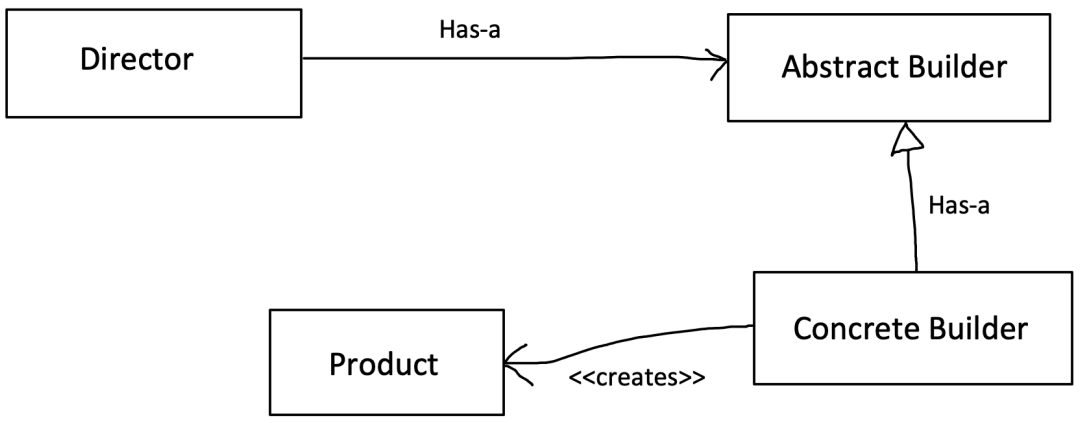
This pattern uses a director to orchestrate the steps, after all the steps are completed the director can call the build method on the builder to get the actual object.

Builder Design pattern is useful when the object has a lot of optional parameters or attributes. This can lead to the constructor becoming very large, we can try to divide the attributes across multiple constructors and then choose one while creating an object however this would result in a very high number of constructors getting created.

The builder design pattern can solve this problem, the StringBuilder uses the builder design pattern.

Builder design pattern is a creational design pattern as well as a Step by step design pattern, i.e. it creates the object step by step, each step corresponds to a method which returns an intermediate Builder Object, eventually we call the build method on the builder object to get the actual object we want.

For example we have a house object and house Builder object, which contain methods like addWall, addRoof, addDoor, addWindow we can call these houseBuilder methods successively and then call the build() method to get an actual house object.



Structural Design Patterns

Structural Design patterns provide a way to combine or arrange different objects to form a complex or bigger structure, which is needed to solve particular requirements.

- Decorator Design Pattern
- Proxy Design Pattern
- Composite Design Pattern
- Adapter Design Pattern
- Bridge Design Pattern
- Facade Design Pattern
- Flyweight Design Pattern

Decorator Design Pattern

This pattern adds additional functionality to existing objects, without changing their structure.

This pattern helps to prevent class Explosion.

The Decorator Design pattern allows us to attach additional responsibilities to an object dynamically, decorators provide a flexible alternative to subclassing for extending functionality.

Consider we are dealing with designing a Coffee Store, which serves a few varieties of Coffee (like Latter, Cappuccino, Roast, Dark Roast, Caramel, chocolate etc). In addition to this the shop provides the customer the option to add any toppings or add-ons they choose, this could include items like Extra milk, soy milk, honey, cream etc.

So for example a customer can order

Latte + Soy Milk + Whipped Cream

How can we model this? Modeling each of the items individually is straightforward by defining a Beverage Base Class.

Beverage Base Class (Abstract)

description getDescription() cost()

Here the Beverage class is abstract, the variable description is set in each subclass and holds a description of the beverage, similarly the cost method is abstract and each class needs to provide their own implementation.

The Beverage class is subclassed by all beverage types offered in the coffee shop.

Latte	Cappuccino	Decaf	Espresso

cost()	cost()	cost()	cost()

How do we model the add-ons, one option is to create subclasses for all the possible combinations of toppings with the basic coffee offerings, i.e. we can create subclasses like:
Espresso + Soy Milk
Decaf + Cream
Latte + Soya + Chocolate
Decaf + Caramel + Extra milk

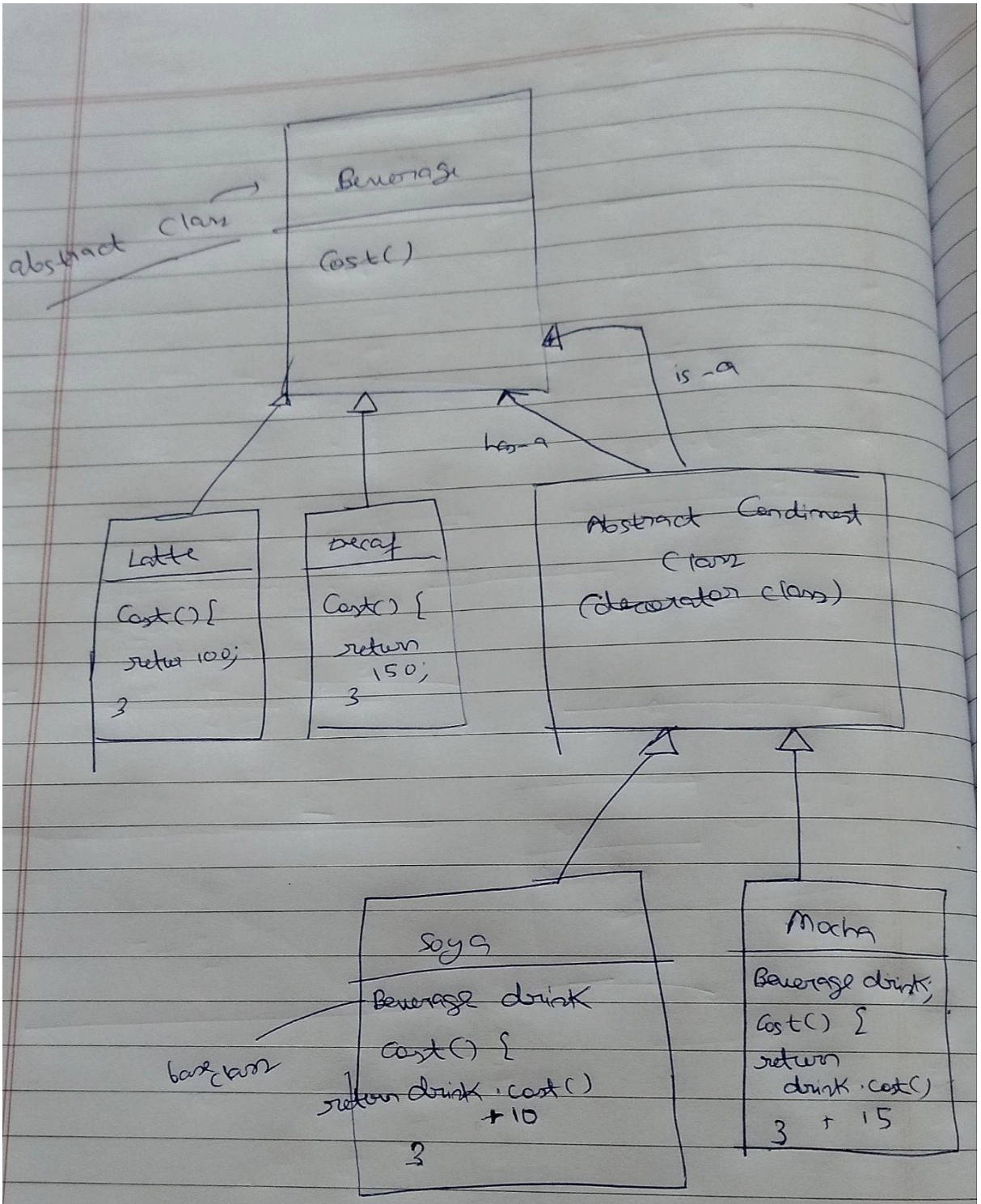
As can be guessed, this will result in a very high number of possible combinations resulting in a class-explosion, which makes maintenance very difficult, imagine if we need to add another topping in this scenario, we'll need to create all possible permutations and combinations just for adding a single topping, hence this is a bad design.

The idea behind the Decorator pattern is to start with a base object (here base Beverage class) and decorate (add) it with properties at run time.

In this example we can start with a Beverage Base Class Object and add condiments (properties) to it dynamically.

=> A decorator must have the same supertype as the object it decorates.
=> We can use multiple decorators to wrap an object.
=> As the decorator has the same supertype as the object it decorates, hence the resulting decorated object will also have the same type, i.e. if we decorate latte (Beverage) with Caramel (Beverage), the resultant decorated object is still a Beverage and can be decorated further.

Cost calculation: How do we calculate the price of the resultant beverage? We call the cost method of the outermost object (decorator), which will add its own cost and delegate the task of computing the price of the remaining beverage to the object it decorates.



* As can be seen the condiment (decorator) class has a "is-a" as well as "has-a" relationship with the Beverage class.

One important point regarding Decorator design pattern and adding new functionality / responsibility to objects in general is that we'll need to override certain methods of the existing object, to override methods the straightforward solution seems to be to use inheritance. But using inheritance can lead to class explosions, directly modifying classes is out of the question as it is a violation of Open / Closed principle. Hence the solution is to use Decorator design pattern which uses Composition and Delegation to override methods. The decorator delegates certain responsibility to the object it is decorating and also adds its own behavior.

For example:

In the case of coffee store: To deal with toppings, we create a toppings decorator, in our example the methods to be overridden include: cost() and getDescription().
Let's consider the Mocha decorator, it'll add behavior by:
=> Adding the cost of Mocha condiments to the cost method.
=> Adding the string "Mocha" to the description.

Similar is the case for InputStream in java, if not for the Decorator pattern we'll need to have separate classes for complex streams like: BufferedInputStreamOnTopOfFileInputStream, adding our sub-classes of the InputStream would have been very difficult and inflexible too.

The methods to be overridden are read() and read(bytes[], offset, len), instead of relying on inheritance, the decorator pattern uses composition and delegation, i.e. it'll delegate to the InputStream it is decorating the task of reading the stream, and before returning the output to the client, the Decorator will add its own behavior (for example turning all characters to UpperCase).

So in general a decorator does two things: Delegation and Adding its own behavior.

Composition + Delegation

=> Decorators allow us to extend behavior without having to modify the existing code.
=> Inheritance allows extension, but it is not flexible and it involves code changes - Compile time.
=> Composition and delegation can be used to add new behaviors at

runtime.

Proxy Design Pattern

The proxy sits between the client and the resource, and all the requests from the client to the resource go through the proxy.
Proxy helps to control access to the resource, like who can access the resource, or does the client need to be authenticated before accessing a resource.

In the Proxy Design pattern the requests from the client to the real object are funneled through the Proxy Object, i.e. when the client sends a request to the real object, the request needs to pass through a proxy object.

Uses of Proxy Design Pattern:

Access Control, the proxy can act as a firewall preventing the client from sending certain requests, thus providing a mechanism of access control. For example a proxy might have a block list configured to prevent the user from accessing certain websites over the internet.

Caching, the proxy object can perform Caching and quickly reply to client requests.

Preprocessing and Postprocessing, Sometimes before sending the request to the real object we might want to do some sort of processing on the request, and then send it to the Real Object, similarly before returning the response to the Client, the proxy might need to do some post processing. Example converting the data to another format.

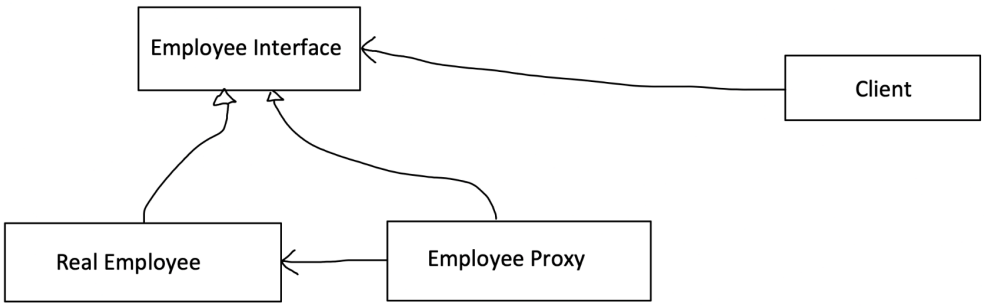
We can have multiple proxies chained together b/w the client and the Real Object.

The proxy pattern provides a surrogate or placeholder for another object to control access to it, in other words the proxy pattern allows us to create representative objects, using which we can control access to the original object, which can be a remote object (i.e. lives in a different JVM), maybe expensive to create or in need of securing.

Types of proxies:
Remote proxy - Controls access to a remote object.
Virtual Proxy - Controls access to a resource that is expensive to create.
Protection Proxy - Controls access based on access rights.

Proxy has the same interface as the thing it is proxying.

Proxy Design Pattern
Sunday, 3 November 2024 1:26 PM

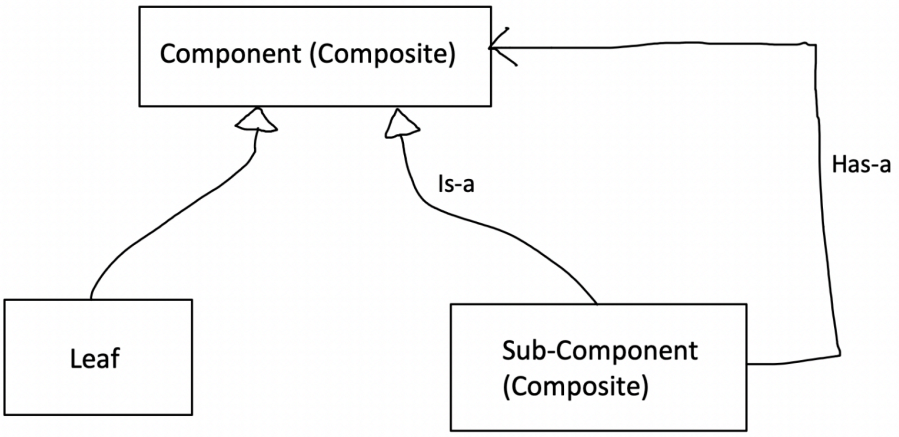


Composite Design Pattern

Composite Design Pattern is useful when dealing with Nested Objects (Object under object), or when the objects can be organized in a Tree hierarchy. For example a File System has 2 components file and directories, a directory itself can contain either a file or another directory, in the composite design pattern the objects have a number of sub-objects hence resulting in the tree structure, additionally similar to a tree data structure we have leaf nodes, i.e. Objects which don’t have any child or sub objects.

Composite Design Pattern

Friday, 1 November 2024 8:09 PM

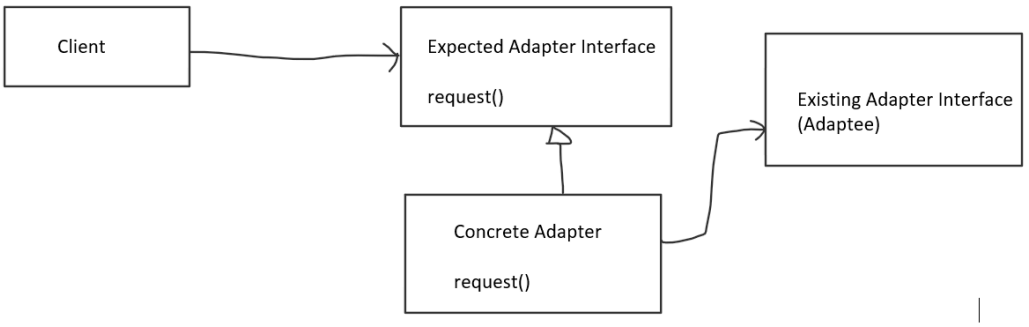


Adapter Design Pattern

The Adapter Design pattern bridges the gap between the Existing interface (Adaptee) and the expected interface, i.e. it acts as an intermediary between two incompatible interfaces.

Adapter Design Pattern

17 August 2024 17:59



Example Use Cases of Adapter Design Pattern and analogies:
Consider the example of having a socket pin and switch plug of different types, in this case we need an adapter to which the socket plug can

connect to, and the adapter should be able to connect to the switch plug.

So, adapters take an interface and adapt it into an interface that the client is expecting. The client is implemented against the target interface, hence the Adapter implements the Target Interface and holds an instance of the adaptee.
The Adaptee is the interface which needs to be adapted.

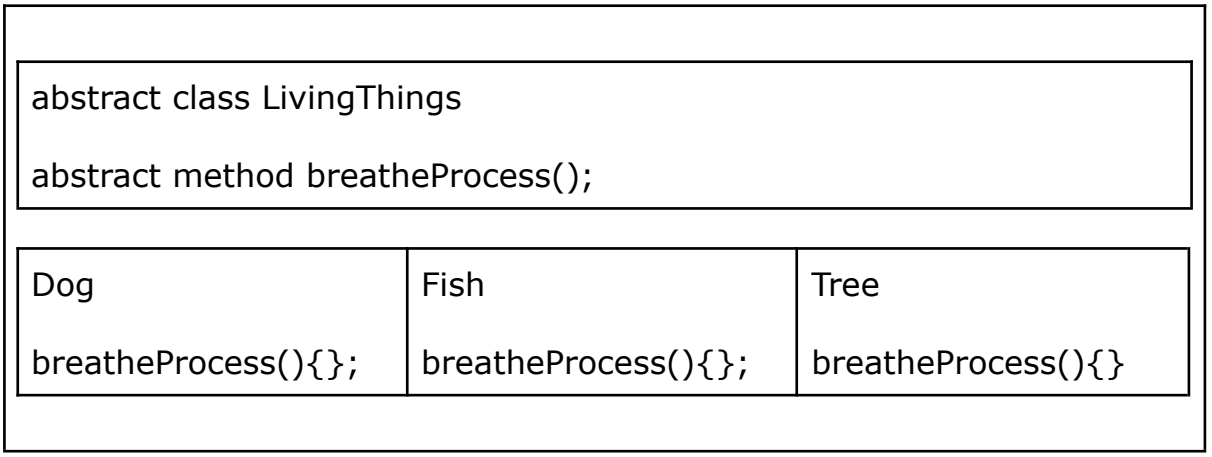
Final definition: Adapter Design pattern converts the interface of a class into another interface that the client expects. Adapter lets classes work together that couldn't otherwise due to incompatible interfaces.

Adapter adapts the adaptee.

Bridge Design Pattern

The Bridge design pattern decouples an abstraction from its implementation, so that the two can vary independently.

Suppose we have the following class hierarchy, we have a class called the LivingThings, which is an abstract class containing the abstract method: breatheProcess();



The dog, fish and tree classes are subclasses of LivingThings, and each of them provides their implementation for the breathProcess method, however can we add an additional implementation for breathProcess here independently of the abstraction? The answer is no, we cannot independently add any additional implementations (breathing process), as the breathing process is closely tied in with the abstraction (abstract

classes). The only way we can add an implementation in the above design is by adding a child concrete class to LivingThings which uses that implementation of breathProcess.

For example we might create a new child ‘Bird’ which can use the new implementation we want to add, however in this case as can be seen the abstractions and implementation do not independently vary, cause if that were the case we should have been able to add the implementation without creating a child class of the abstraction.

To fix this problem we move the breathProcess method to a separate interface, called Implementor which will handle all the implementation details, and remove it from the abstraction, so that the child concrete classes no longer need to implement this method themselves.

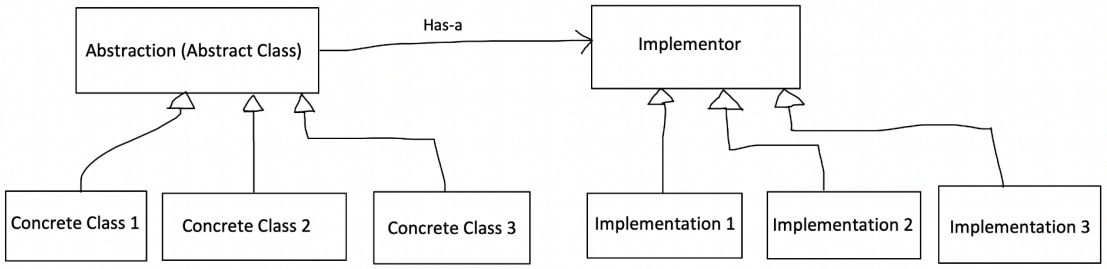
Since the implementation is no longer strongly tied to the abstraction hence, hence we can vary the implementation independently and for example easily add another implementation (breathingProcess) without having to create a concrete class of the abstraction, as we had to before.

Also since the implementation is no longer hardcoded in the concrete class, hence it can easily be swapped for any other implementation the class wants. In the previous design, we cannot have an implementation if no child is using it, however in this design we can have as many implementations as needed, and the abstraction won’t be affected, without needing to create any child concrete class which implements that implementation. The implementation can exist independently now, in other words unused implementations can exist in our design now.

In summary the problem we are trying to solve with the Bridge Design pattern is how can we add a new implementation, without adding any class of the abstraction, this design pattern adds a lot of flexibility to our design, the client can mix and match implementations for the various concrete classes and easily swap one for the other, as the implementations and abstractions are no longer strongly coupled (as implementation is no longer hardcoded in the concrete class itself), now we have a separate interface called Implementor, and we can subclass this interface with all the implementations we need.

Bridge Design Pattern

Monday, 7 October 2024 7:15 AM



The Bridge design pattern allows the abstractions and implementations to vary independently by placing them in separate class hierarchies. The link b/w the 2 class hierarchies is known as the bridge.

https://stackoverflow.com/a/10689017

One major advantage of Bridge Design Pattern is that it prevents class explosions, i.e. it minimizes the number of classes created. In general it reduces the classes needed from m * n to m + n, in case of pairwise matching. For example suppose we have 4 shape Classes (Rectangle, Circle, Square and Rhombus) and 5 color classes (Red, Black, Green, Blue, Yellow), if we perform a pairwise matching then we'll end up with 4 *5 = 20 classes [Cartesian product] (RedRhomubs, BlueRectangle, YellowCircle, GreenSquare etc..)

However with the Bridge design pattern we'll just need to create 9 classes, we model the Shapes and Colors as separate class hierarchies rather than modelling them in a single class hierarchy, i.e we separate the Colors into their own class hierarchy. Finally we compose each Shape with a color. This way we don't need to create classes like RedRhombus, instead we have a Rectangle class composed with the Red color class.

Facade Design Pattern

Facade design pattern is used when we need to hide the system complexity from the client.

Facade Design Pattern takes a complex subsystem and makes it easier to use by implementing a Facade class / layer that provides one or more reasonable interfaces.

The facade still leaves the subsystem accessible, i.e. any client that wishes to access the components / classes of the sub-system directly can still do that. Facade pattern merely provides a simplified interface.

Hence, the intent of the facade design pattern is to provide a simplified interface to a subsystem.

In addition the Facade decouples a client from the subsystem components.

Instead of communicating directly with the system, the client communicates with the Facade layer which'll take care of interacting with the system.

Use case of the facade design Pattern:

When we want to expose only a certain subset of methods to the client, instead of exposing all of them, i.e. expose only the necessary details to the client.

EmployeeDAO

```
insertUser();
getUserDetailById();
getUserDetailByEmail();
updateEmployeeName();
```

EmployeeFacade

```
EmployeeDao employeeDAO;

EmployeeFacade() {
   this.employeeDAO = new EmployeeDAO();
}

insertUser() {
   employeeDAO.insertUser();
}
```

```
getUserDetailById() {
    employeeDAO.getUserDetailById();
}
```

Client

```
EmployeeFacade employeeFacade = new EmployeeFacade();
employeeFacade.getUserDetailById();
```

For example we have a very complex system, which has several moving parts and is spread across lots of classes. If the client is directly interacting with the system, then to perform any action it'll need to interact with multiple classes and methods, know about the various parts of the system and the steps involved, if any of these internal methods are changed then the client will face errors, as it isn't aware that certain methods have been changed / removed, to fix this issue we'll need to change the way client performs the action, since there could be hundreds of clients, hence this approach is inefficient, further having all the logic on the client side will make it heavy.

A better approach in this case will be to have a facade layer in front of our system, any changes in methods or classes involves making changes in the facade layer only and the clients can continue to communicate with the facade layer as they did before. Any changes in the methods / order of execution etc. will result in changes in the facade layer only, clients won't be impacted.

Facade abstracts the complexity of the system from the client.

Flyweight Design Pattern

Flyweight design pattern is a structural and Optimization pattern.

If we have created a lot of objects in our application, then the system performance will be hampered as the RAM will be filled up.

The flyweight pattern helps to reduce memory usage by sharing data among multiple objects.

Examples where the Flyweight pattern can be used: Games, for example car racing games which have a large number of objects repeating cars or trees. Or any application involving repeating objects, e-commerce websites which display a lot of products etc.

The flyweight pattern divides the properties of a class into two categories:

1. Intrinsic Attributes - These are the properties which do not change per object, i.e. all the objects will have the same value for these properties.
2. Extrinsic Attributes - These are the properties which do change with different initializations, i.e. the objects have different values for these properties.

For example suppose we are designing a game involving many tennis balls, which are moving all across the screen.

The TennisBall class will look like:
Color,
radius,
imageUrl

coordX
coordY

// Methods to move the ball on the screen

In this case the color, radius and imageUrl are intrinsic attributes, as they will be the same for all the TennisBall objects. However coordX, coordY will differ across objects, and hence are extrinsic attributes.

Flyweight pattern helps to save memory by sharing and reusing the intrinsic attributes across the objects.

The intrinsic data will be extracted out, i.e. we won't be storing the intrinsic data across all the objects, instead we'll store it in a single place, say a cache, and all objects can use that data, further clients can further customize the object using the extrinsic attributes.

A flyweight object is an object which is used to store the intrinsic data, flyweight objects won't store the extrinsic (object-specific) data, each flyweight object is initialized only once and is stored in a cache, so that it

can be reused.

So in the case of the TennisBall, we'll initialize and store the color, radius and imageURL in the cache via a flyweight object. Wherever we need a new TennisBall object, we'll get it from the cache this way we won't need to reinitialize the intrinsic data, further the object returned by the cache can be enhanced by initializing the extrinsic data (via setters).

```
public class TennisBall {
    private String color;
    private String url;
    private int coordX;
    private int coordY;

    public TennisBall(String color, String url) {
        this.url = url;
        this.color = color;
    }

    public void setCoordX(int coordX) {
        this.coordX = coordX;
    }

    public void setCoordY(int coordY) {
        this.coordY = coordY;
    }

    public void render() {
        // Render the ball to screen
    }
}
```

This is the flyweight TennisBall class, we'll create an object of it and store it in the cache, so the intrinsic data is not re-initialized, the cache is just a hash map in this case, we can use the color + url as the key in to this map and get the corresponding flyweight object if it exists, if not, then one will be created and stored in the cache for future references.

We can create a factory to handle requests for TennisBall objects, it'll check the map to see if a ball with the key already exists in the map, if it

does it’ll simply return that ball, hence preventing the intrinsic data from getting reinitialized.

```
public class TennisBallFactory {
    Map<String, TennisBall> map;

    public TennisBallFactory() {
        map = new HashMap<>();
    }

    public TennisBall getTennisBall(String color, String url) {
        String key = new
StringBuilder(color).append("_").append(url).toString();
        TennisBall tennisBall = map.get(key);

        if(tennisBall == null) {
            System.out.println("Request Served by creating new
object");
            tennisBall = new TennisBall(color, url);
            map.put(key, tennisBall);
        }

        return tennisBall;
    }
}
```

```
public class Driver {
    public static void main(String[] args) {
        TennisBallFactory tennisBallFactory = new
TennisBallFactory();

        TennisBall tennisBall1 =
tennisBallFactory.getTennisBall("Red",
"https://wwww.image.com/eknj2br");
        tennisBall1.setCoordX(3);
        tennisBall1.setCoordY(12);

        TennisBall tennisBall2 =
tennisBallFactory.getTennisBall("Red",
"https://wwww.image.com/eknj2br");
```

```
    }
}
```

Behavioural Design Patterns

Behavioral design patterns guide how the different objects should communicate with each other and how the responsibility should be distributed between the objects.

- State Design Pattern
- Observer Design Pattern
- Strategy Design Pattern
- Chain of Responsibility Design Pattern
- Template Method Design Pattern
- Command Design Pattern
- Iterator Design Pattern
- Mediator Design Pattern
- Memento Design Pattern
- Visitor Design Pattern
- Interpreter Design Pattern

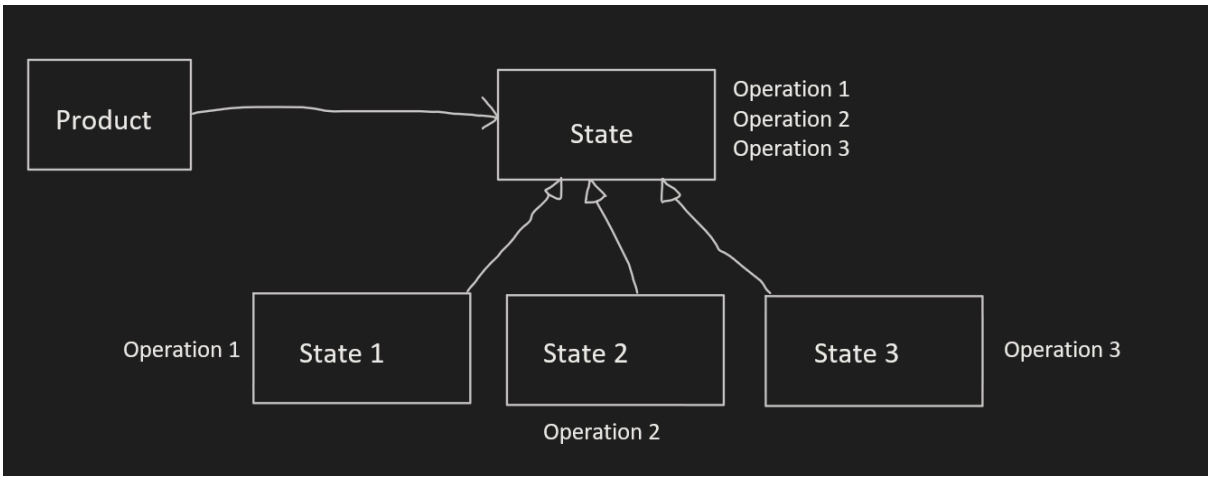
State Design Pattern

The state design pattern allows an object to alter its behavior when its internal state changes.

This pattern represents each state as a class.

In the case of State Design Pattern the product or the object which we are designing or modeling can be broken down into certain well defined states, where each state has a certain set of operations which it permits. This means not all operations are valid for all states. For example suppose we are designing a Vending Machine with m states and a total of n operations, here it is not necessary for each of the m states to implement all of the n operations, instead each state will only implement operations which are applicable for it.

The state is modeled as an interface or abstract class subclassed by various concrete classes, which together encompass all the possible states, each such concrete class represents a state and the state will implement all the operations applicable for it, while providing a default implementation for all of the other operations. So in the state design pattern, the behavior of the object changes when its internal state changes.



Observer Design Pattern

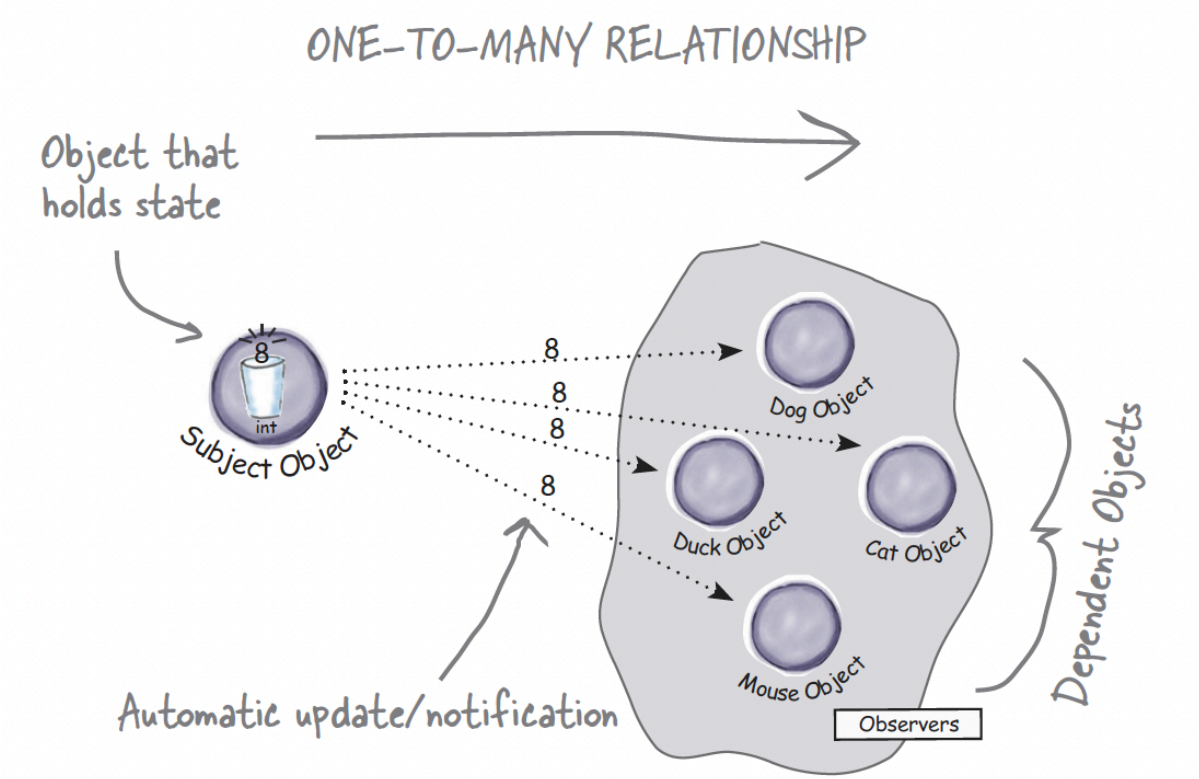
The Observer Design Pattern defines a one-to-many relationship between a set of objects so that when the state of one object changes all of its

dependents are notified.

These dependents are known as observers, which are ‘observing’ the state of an object called the Subject or Observable.

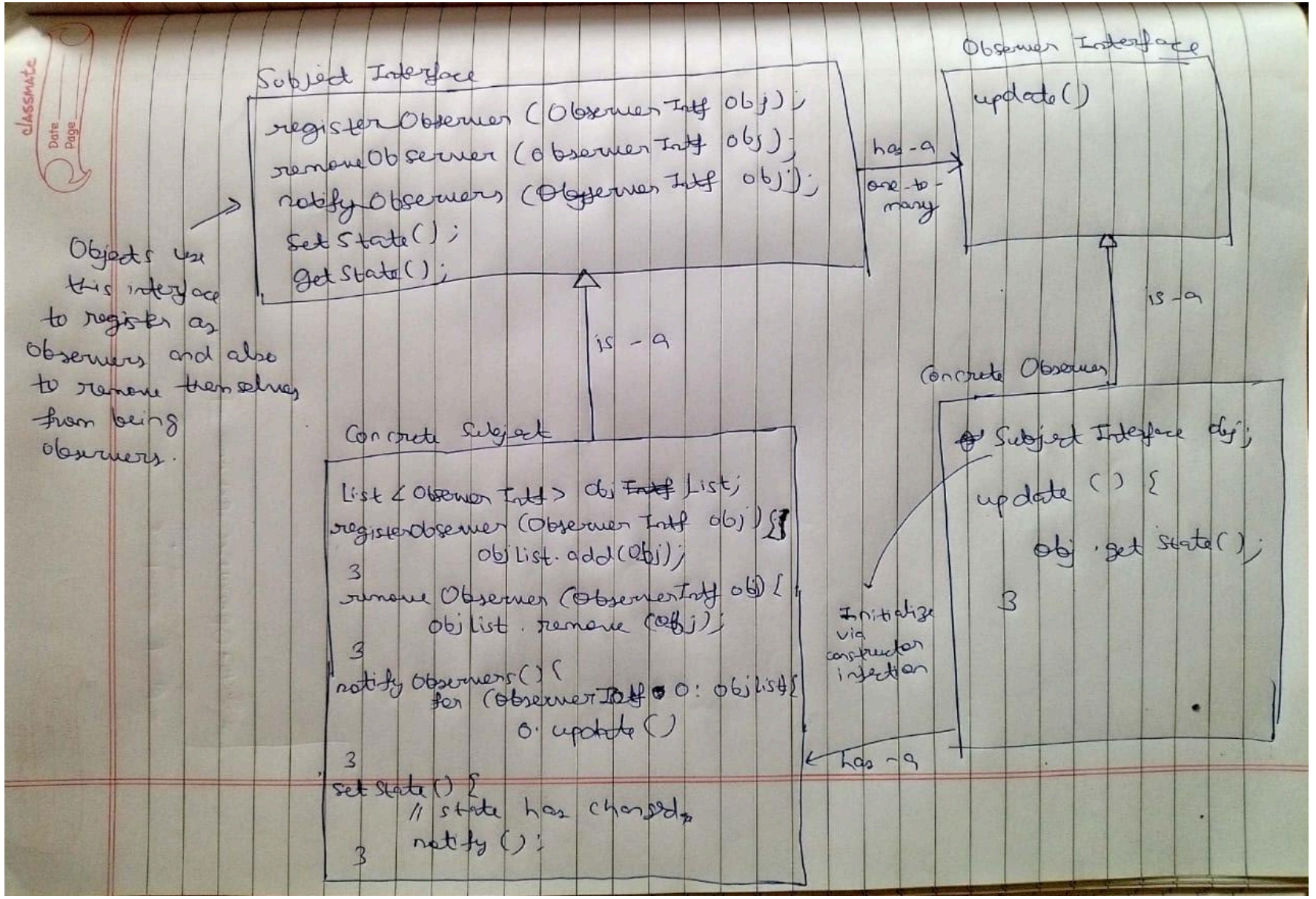
The subject and the observers define the one-to-many relationship, we have one subject which notifies many observers when something in the subject changes. The observers are dependent on the subject.

In the observer design pattern one object called the subject or observable maintains a list of its dependents (observers) and notifies them when its state changes.



Another stolen diagram. This one depicting the Observer Pattern

The Observer pattern is generally implemented by using a Subject Interface and Observer Interface.


Subject Interface
registerObserver(ObserverIntf obj);
removeObserver(ObserverIntf obj);
notifyObservers(ObserverIntf obj);
setState();
getState();
Observer Interface
update()
has-a
one-to-many
Objects use this interface to register as observers and also to remove themselves from being observers.
is-a
is-a
Concrete Subject
List<ObserverIntf> obsIntfList;
registerObserver(ObserverIntf obj){
obsList.add(obj);
}
removeObserver(ObserverIntf obj){
obsList.remove(obj);
}
notifyObservers(){
for(ObserverIntf o: obsList){
o.update()
}
setState(){
// state has changed
notify();
}
Concrete Observer
Subject Interface obj;
update(){
obj.getState();
}
Initialize via constructor injection
has-a

Let’s consider the example of a weather station getting the latest temperatures using physical sensors and Devices (TV, Mobile etc.) which get this data from the weather station. We can use the Observer pattern where the subject is the Weather Station while the TV Display and Mobile Display are observers.

WS = Weather Station

WS Subject Interface

```
=> registerObserver(DisplayObserver obj);
=> removeObserver(DisplayObserver obj);
=> notifyObservers();
=> setTemperature();
=> getTemperature();
```

WS Subject Concrete Class

```
List<DisplayObserver> observersList;
// State
int temp;

registerObserver(DisplayObserver obj)
{
    observersList.add(obj);
}

removeObserver(DisplayObserver obj)
{
    observersList.remove(obj);
}

notifyObservers()
{
   for (DisplayObserver observer: observersList) {
       observer.update();
   }
}

setTemperature(int newTemp)
```

```
{
    temp = newTemp;
    notifyObservers();
}

getTemperature() {
    return temp;
}
```

DisplayObservers Interface

```
=> update();
```

Display Observers Concrete Class, here we have 2 observers for the WS Observable

a. TV Display Observer
b. Mobile Display Observer

TV Display Observer

```
WSSubjectInterface obj
TVDisplayObserver(WSSubjectInterfae obj)
{
    this.obj = obj;
}

update() {
    obj.getTemperature();
}
```

Similarly the Mobile Display Observer can be defined.

Strategy Design Pattern

The Strategy Design Pattern is a behavioral design pattern which allows us to dynamically change the behavior of an object by encapsulating it into different strategies. In other words the pattern allows an object to choose from multiple behaviors at runtime, rather than being restricted to

a single one statically.

In brief the strategy design pattern allows us to define multiple ways to perform a said task, and we can choose one depending on the situation, further we can easily swap one ‘strategy’ with another. A strategy is just a way of performing a task.

Example say we are designing a Payment class, which handles payment related details, we want to support UPI payments, Cash and Card Payments, we can do so by creating a Payment Strategy interface, which will be subclassed by the UPIStrategy, CardStrategy and CashStrategy, the payment object can then dynamically choose any of these strategies, at run-time according to the use-case. This provides the flexibility to the client as they can easily swap out one strategy to another.

For example consider this Payment class

int amount;
payer_name: “”,
date: “”,
UPIPaymentMethod paymentMethod: new UPIPaymentMethod()

This is a bad design as the Payment class is hardcoded to using the UPIPaymentMethod, if we want to support Cash and Card payments then we’ll need to create separate classes for CashPayments and CardPayments.

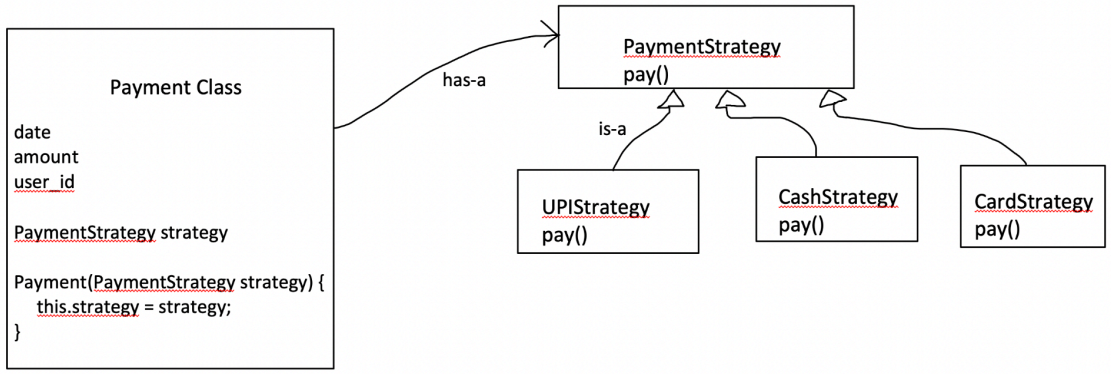
Strategy pattern solves this problem, by creating a Strategy interface which will be subclassed by strategy concrete classes, and we can make our code flexible by depending on the Interface rather than the concrete class (Dependency inversion)

int amount;
payer_name: “”,
date: “”,
PaymentStrategy paymentStrategy: paymentStrategy (pass via constructor injection)

paymentStrategy object will be passed via constructor injection, giving the client the required flexibility.

Strategy Design Pattern

Monday, 7 October 2024 7:38 PM



Strategy Pattern helps to solve the problem of code duplication associated with inheritance, when two child classes end up overriding a method from the parent identically.

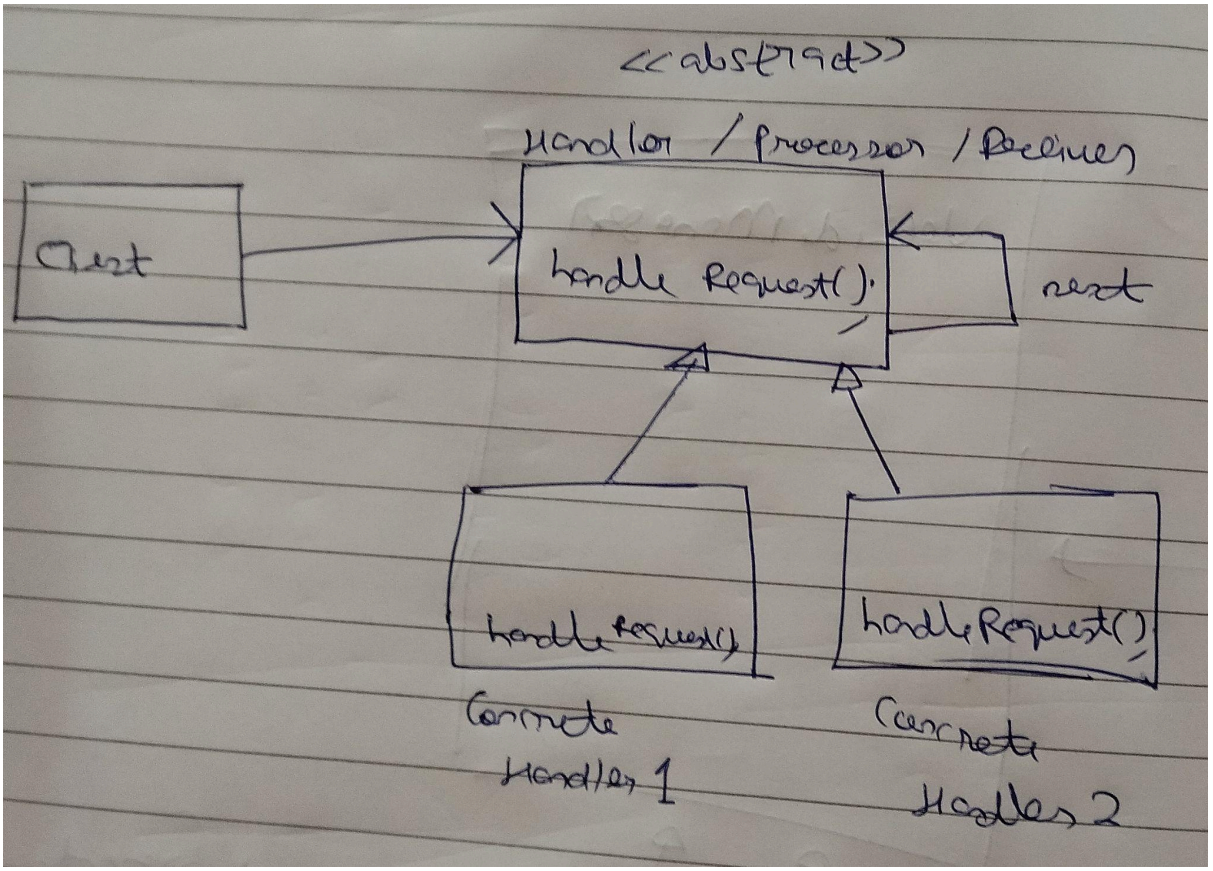
Strategy design pattern defines a family of algorithms, encapsulates each one and makes them interchangeable. Strategy Pattern lets the algorithm vary independently from the clients that use it.

Chain of Responsibility Design Pattern

In this pattern we have a chain of Receiver Objects (or Handlers) which can handle the requests from the client, and the client (the component sending the requests) isn't interested in which receiver is handling the request.

This pattern is useful for real world use-cases like Loggers, ATMs and Vending Machines.

The handlers are structured in a chain (imagine a linked list), where each Handler beginning from the first one will try to process the request, if it is not able to process it, then the request will be forwarded to the next handler in the chain, which will then try to serve the request and so on, like a linked list each Handler has a pointer to the next handler in the chain.



It is possible that none of the receivers are able to handle the request in such cases we need to return an error. We can add a default, catch-all receiver at the end of the handler chain (like a default condition in a switch).

Template Method Design Pattern

The template Method defines the steps of the algorithm and allows the subclasses to provide implementation for one or more steps.
In other words the template method design pattern defines the skeleton of the algorithm in a method, deferring some steps to the subclasses. So this pattern allows the subclasses to re-define certain steps of the algorithm without changing the structure of the algorithm.

A step of the algorithm is basically just a method call.

=> The template method is often declared final because we don't want the subclasses to be able to override it.
=> We declare the class containing the template method as abstract, so that the class can itself provide concrete implementations for some of the

methods which are common across classes, hence preventing code duplication.
=> The methods that need to be supplied (or implemented) by the subclasses are declared abstract.

```
abstract class AbstractClass {
     final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }

    abstract void primitiveOperation1();
    abstract void primitiveOperation2();
    void concreteOperation() {
       // implementation here
    }
}
```

In the above example the Template Method is the method named ‘templateMethod’, which defines the sequence of steps which makes up the algorithm. Since we don’t want this structure to be changed, hence we declare this method as final.

The methods primitiveOperation1 and primitiveOperation2 are declared abstract, hence the subclasses will need to supply these methods. The method ‘concreteOperation’ is implemented by the abstract class itself, this method is known as concrete method, if we want the concrete methods to not be overridden by the child concrete classes then we can make them final too.

Using Hooks: Hooks are concrete methods defined in the abstract class, but only have an empty or default initialization. The subclasses are free to override the hook methods, but they don’t have to necessarily.

The Hooks allow the subclass to ‘hook’ into the algorithm at various points. For example:

```
public void hook() {
```

```
    return true
}
```

We can use hooks to control whether certain parts of the algorithm get executed or not, i.e. hooks can be used to conditionally control the flow of the algorithm in the abstract class.

Side Notes:

Hollywood Principle: Don’t call us, We'll call you.
Hollywood principle is used to prevent dependency rot. A dependency rot happens when we have high level components depending on low-level components which are in turn depending on the high-level components, which in turn might be depending on sideways components. A dependency rot makes the system prone to failures and difficult to understand.

The Hollywood principle allows the low-level components to hook into the high level components and participate in computation, but the high level components control when and how they are needed.
=> A low level component can never directly call a high level component.

The Hollywood principle and the template method design pattern work hand in hand. In the template method design pattern the high level component - the abstract class, has control over the algorithm and calls on the subclasses only when they are needed, to supply a method, the subclasses never call the abstract class directly without being called “first”.

Uses of the Template Method design pattern in other places in Java source Code:

1. Arrays.sort: The method Arrays.sort internally calls the mergeSort method which is the template method. It uses several methods to perform the sorting, the method swap is a concrete method, i.e. it is implemented by the Arrays class.
   However the method compareTo is an abstract method, which needs to be implemented by the object being sorted, i.e. the comparison part of the algorithm has been deferred to the objects being sorted. So for example we are sorting an array of Projects, then the Project class will need to provide an implementation for the compareTo

method.
Note: Java Arrays cannot be subclassed (extended).

2. The JFrame (A Swing Container) inherits a paint() method, by default this paint() method does nothing because it's a hook. The subclass can hook itself into the JFrame's algorithm by overriding the paint() method. JFrame's algorithm calls the paint() method, however by default paint() draws nothing, as it's a hook.

3. The package java.io has the class InputStream which has the template method read(byte[], int, int), this template method uses the 'read()' method, which must be implemented by the subclasses.

Command Design Pattern

The Command Design Pattern allows us to decouple the requester of an action from the object that actually performs the action, i.e. the requestor and the object which performs the action never have to directly communicate.

A command object encapsulates a request to do something, on a specific object, in other words the Command Object consists of a set of actions on a Receiver. The client / requester uses the command object.

The command design pattern allows us to encapsulate a request as an object, called the Command Object. This pattern breaks the design down into the following components:

Client / Requester
Invoker
Command Object
Receiver

=> Client creates the Command Object: As mentioned before the client creates the command object, which is basically a request to perform some actions on the receiver.

=> Client calls setCommand on an Invoker Object: The invoker object exposes a method 'setCommand(Command command)'. The client will call this method and pass the command object it created to the Invoker. The

command will be stored in the Invoker object for as long as needed. Invoker objects can be parametrized with a command, i.e. we can swap the Command associated with the Invoker, by using the setCommand method.

=> The command Object provides one method - execute(), that encapsulates the actions and the receiver, and bounds them together. To invoke the actions on the receiver, this method needs to be called. Who calls this method? The Invoker object calls the command object's execute method.

=> The execute method will invoke the desired actions on the receiver.

To implement the "Undo" Functionality we add an undo method to our Command Interface (which needs to be implemented by the Concrete Command Classes), hence now our Command interface has two methods :- execute, and undo. The undo method encapsulates the logic to reverse the operations performed by the execute method.

Example:
execute() :- Turn the Light Object On
undo() :- Turn the Light Object Off

Advantages of the Command Design Pattern:

1. Makes the client thin, as it does not need to know about the receiver object and the various methods exposed by it. Hence by using the Command Pattern we can keep the client code very simple, this provides decoupling and abstraction.

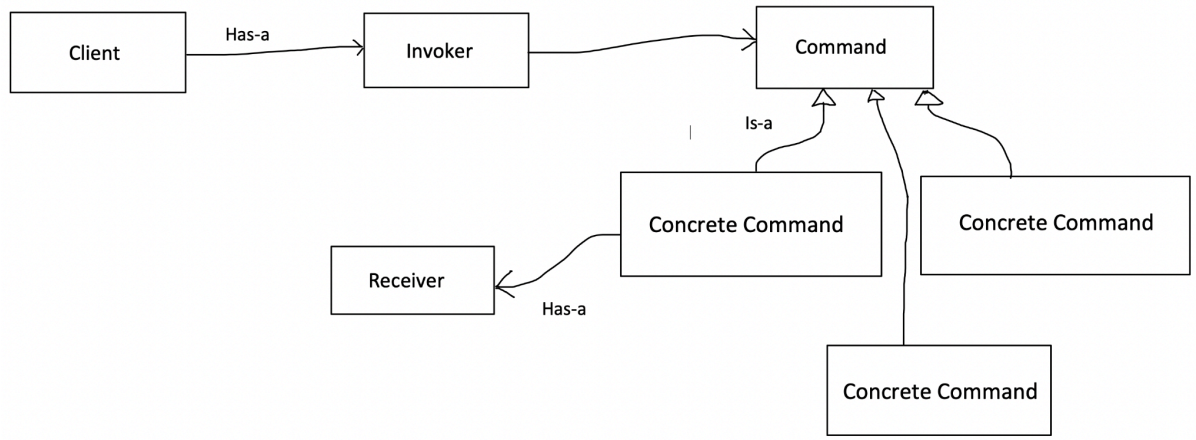
2. Allows us to implement the "undo operation" feature, as well as request logging, queuing requests and failover mechanisms.

3. Helps in creating an extensible design, we can easily add more Commands, and swap commands set on the invoker.

MacroCommand: A MacroCommand is a type of command that can execute multiple other commands. The MacroCommand is initialized with a list of commands, when we call execute on the MacroCommand it'll go over the list of commands it has and call the execute method on each of them individually.

The Command Design pattern helps in designing the following use-cases:

1. Any application which involves the concept of command, for example Remote, Word Editor.
2. Designing undo / redo feature
3. Requests Logging
4. Request Queueing: Designing decoupled components, designing Message Queues.
5. Failover Mechanism / Failure Recovery Mechanism



Iterator Design Pattern

Iterator design pattern is used to encapsulate iteration.

Iterator design pattern provides a way to traverse the elements of an aggregate, sequentially without exposing the underlying implementation, i.e. the client does not need to know how the data is internally stored.

An aggregate basically stores a group of objects, it can also be called a collection. The aggregate will store these objects in some data structure for example: List, Map or Arrays.

The goal of the Iterator design pattern is to provide the clients a way to traverse the elements inside the aggregate without exposing the internal implementation to the client.

Separation of Responsibility: The Iterator design pattern places the responsibility of iteration / traversal on the Iterator object, rather than the aggregate object itself. This approach simplifies the aggregate's interface and implementation, and allows it to focus on storing the objects while

the Iterator object handles iteration.

=> The aggregate should provide the client the iterator object when requested, this iterator object exposes methods: next() and hasNext() which the client uses to iterate the aggregate.

=> Depending on the internal representation of data, different aggregates will need to implement their own Iterators, for example ArrayList and LinkedList will have very different iterators.

=> To provide a consistent interface, we define the Iterator interface which exposes the next() and hastNext() method.

=> In addition the aggregates implement the aggregate interface, which exposes a method: createIterator which creates the iterator and returns it to the client, in Java this interface is called the Iterable interface, and the createIterator() method is just called iterator().

=> The Iterable interface also provides the method: forEach, which is built on top of enhanced-for loop.

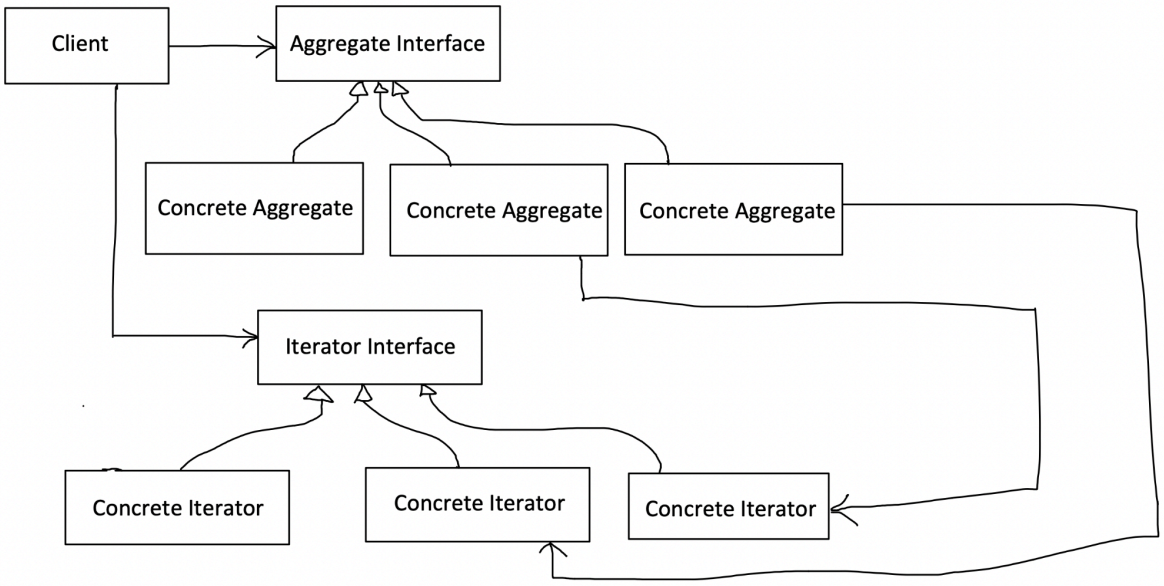
In summary the Iterator design pattern decouples the client from the internal implementation details of how the objects are actually stored in the aggregate. The Iterator object provides a layer of abstraction. The client doesn't need to know about the aggregate, all it knows is to call next() and hasNext() on the iterator object, which implements the Iterator interface.

For example we have a method called print, which takes as input an Iterator and prints all the elements in the corresponding aggregate.

```
print(Iterator it) {
	while(it.hasNext()) System.out.println(it.next());
}
```

To this method we can pass any child of the Iterator interface, in other words any class implementing the Iterator interface, for example LibraryIterator, MenuIteator etc. The print method wouldn’t care. Which aggregate’s elements it’ll print will be decided at run time (Runtime polymorphism).

Final Point:
=> Java provides an Iterator interface, which provides next(), hasNext() and remove().
=> All the aggregates implementing the Iterable interface, provide an iterator() method to get the iterator for that class. If not, then we can define our own Iterator class.
=> Iterator design pattern encapsulates iteration. The aggregate object does not need to modify its implementation / interface to add methods to support traversal of its elements. The iterator object lives outside of the aggregate object, in other words it encapsulates iteration.



Mediator Design Pattern

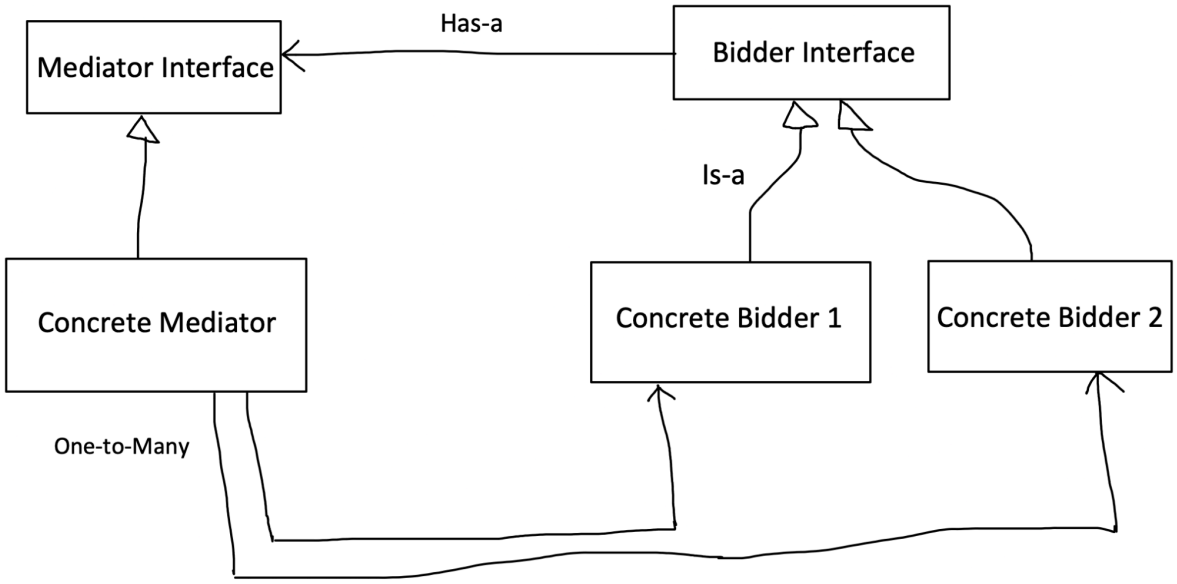
Mediator Design Pattern is a behavioral design pattern, it provides loose coupling by preventing the objects from referring to each other explicitly and instead allows them to communicate through the mediator object.

Hence an object doesn't need to know about the other objects, whenever it wants to send a message it'll pass it to the mediator, and the mediator can relay it appropriately.

No two objects explicitly call each other, they communicate via the mediator object.

Example use-case: Online Auction System, Airline Management System (ATC), Chat Application

For example in an online auction system, the objects representing the people can make bids, when a bid is made all the other objects must be notified about the bid. This is the responsibility of the mediator, the other alternative would be for the object making the bid to inform all the other objects, this leads to strong coupling.



Memento Design Pattern

Memento Design Pattern is a behavioral design pattern.
This pattern is used when we need to save the state of an object, and implement the undo functionality to restore the object to an older state.

The Memento design pattern does not expose the internal implementation of the object.

This pattern is also known as the Snapshot design pattern, as a snapshot is a saved state of the object, which we can restore to if needed.

Components Involved in the Memento design pattern:

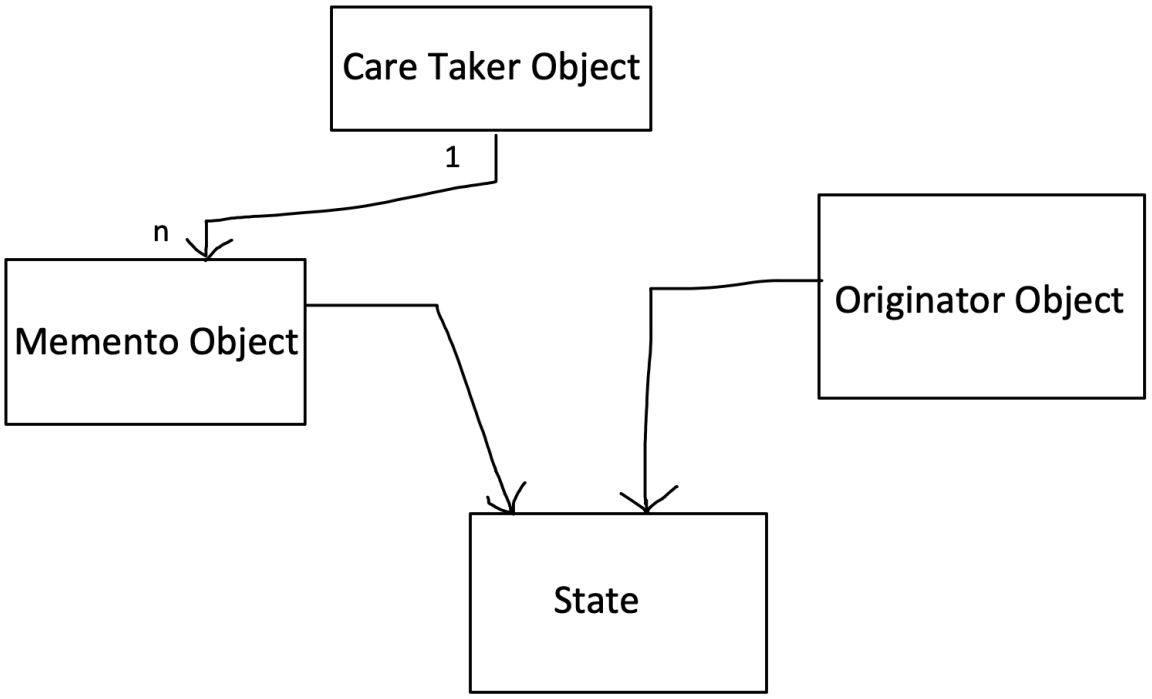
1. Originator
2. Memento
3. Caretaker

Originator Object: This is the object for which the state needs to be saved, and restored.

Memento Object: This is the object which holds the state of the originator.

Caretaker: Manages a list of states or Memento objects.

The Originator object exposes methods to create the memento object and to restore the originator to a previous state (i.e. to a previous memento object), this way Memento design pattern prevents the internal implementation of the originator from getting exposed, the originator will be solely responsible for creating the memento, this allow us to save private variables in the memento or perform selective saving (Save only certain properties of the object).



Visitor Design Pattern

Visitor Design Pattern is a behavioral design pattern.
This design pattern allows us to add new operations / methods to existing classes without changing their structure.

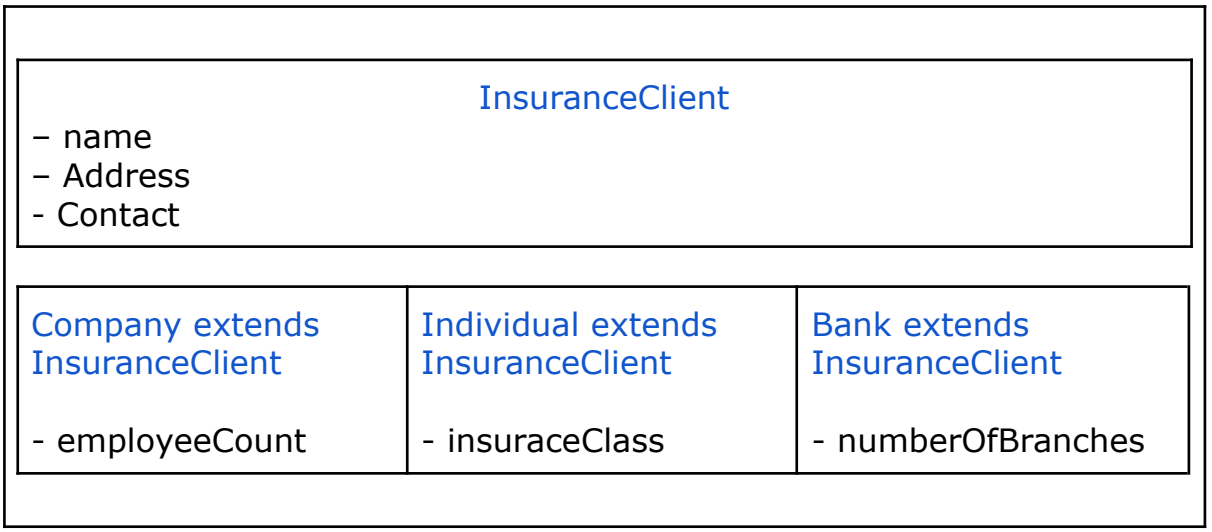
=> If we directly add new methods to existing classes, then that'll be a violation of the Open / Closed principle. As the existing class would

already have been tested.
=> It is a violation of the Single Responsibility principle if we keep adding different methods to the class, having very different effects.

Example:

Consider we have an insurance company, which has multiple clients: Company, Bank, Individual.

These classes have attributes in common, so we can abstract them out in an interface.



Suppose we want to add a new method to each of the client classes, one way would be to add the method to the InsuranceClient interface, and provide an implementation for the method in each of the sub-classes, however this is a violation of the Open / Closed principle, also in the existing design the classes are simple POJOs, and we cannot add functionality to POJOs, now since the classes contain actual functionality, they might break in some situations.

Furthermore, if we decide to alter the method behavior just added, then we'll need to open and change this code in each of the subclasses individually.

The Visitor design pattern allows us to add new operations / methods to existing classes without changing their structure, it does so by extracting the operations / methods outside of the classes on which they operate, in other words Visitor design pattern separates the operations from the objects on which they operate.

How does this pattern work:

Instead of defining the method in each of the sub-classes, We create a Visitor class, which contains the implementation of the method for all of the sub-classes. For example we wanted to implement the method: sendNotification, in this case instead of modifying all the classes, the visitor class alone contains the implementation of this method for all the sub-classes, Bank, Individual and Company. This way, we don't need to modify the existing classes.

Visitor class

=> sendNotificationBank
=> sendNoticationIndividual
=> sendNotifiactionCompany

Problems with this approach, now that the implementation of the method for each subclass is in the visitor class, how do we choose the right version of the method to call for each object?

We can do simple if-checks

```
if(client instanceOf Individual) {
        visitor.sendNotificationIndividual((Individual)client)
} else if(client instanceOf Bank) {
        visitor.sendNotificationBank((Bank) client)
} else {
}
```

This is not a scalable approach, to solve this problem the Visitor pattern uses the concept of Double Dispatch, since the objects know their own classes, hence the pattern delegates the responsibility of choosing the correct method to the object itself, instead of letting the clients select the method.

To do so, each object will accept a visitor and redirect it to the correct method.

We can easily add more visitors to this design, instead of giving specific names like sendNotificationResident, or sendNotificationBank the methods are given the overloaded name 'visit', with each method accepting as parameter an object of its corresponding sub-class.

=> The Visitor interface declares a set of visiting methods that take concrete objects of the classes as arguments.
=> The concrete visitor class implements several versions of the same operation, for the various sub-classes.
=> The Element interface declares a method for accepting visitors.
An element in this context is the object to which we want to add the new method.
=> This method commonly called 'accept' is implemented by each of the concrete Element classes, the accept method redirects the visitor to the correct visitor method (double dispatch) [corresponding to the sub-class].

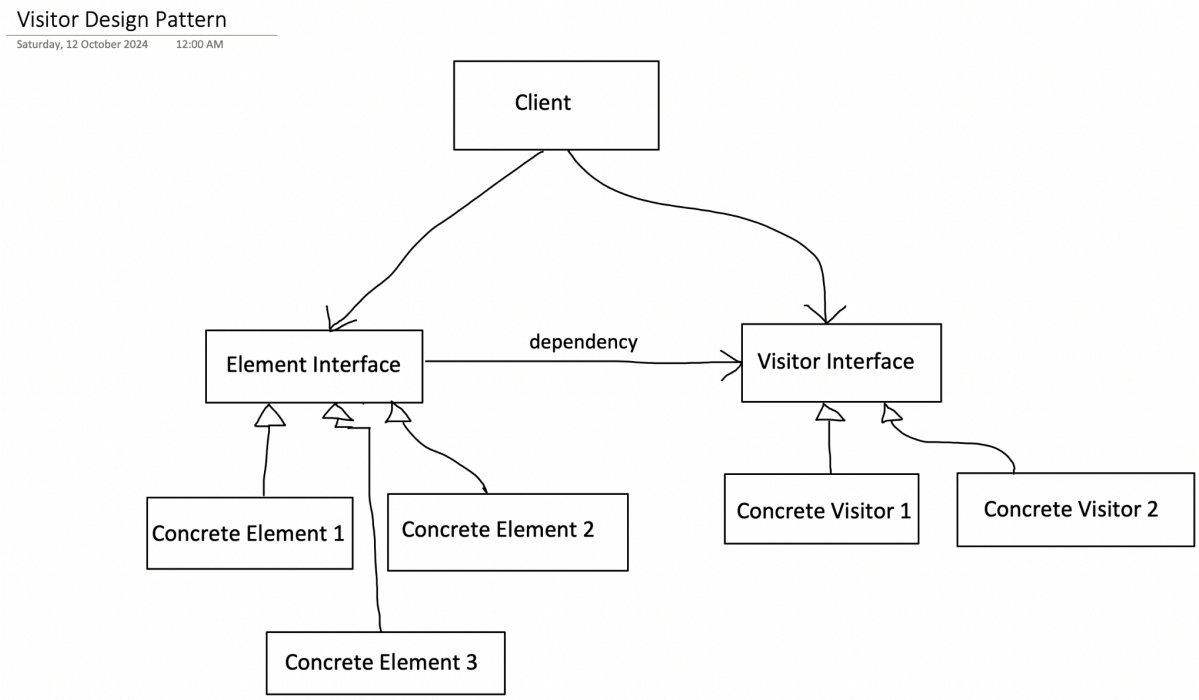
- The Visitor design pattern helps in enforcing the Single Responsibility principle, the classes can focus on their main job.
- Helps in enforcing the Open / Closed principle as new visitors can be added without modifying the existing classes.
- Visitors are interchangeable (of course all the visitors must implement the Visitor interface).

For each operation we create a new Visitor subclass.

How are the classes closed for modification?

With the visitor pattern, If we want to add a new operation / method, then we need to create a new Visitor to support this operation, this visitor class will contain the implementation for each of the sub-classes for the newly added operation. Hence the original source files, and the existing Visitors are untouched. Hence we are not violating the open / closed principle.

The visitors will grow horizontally, however the classes themselves won't become too big.

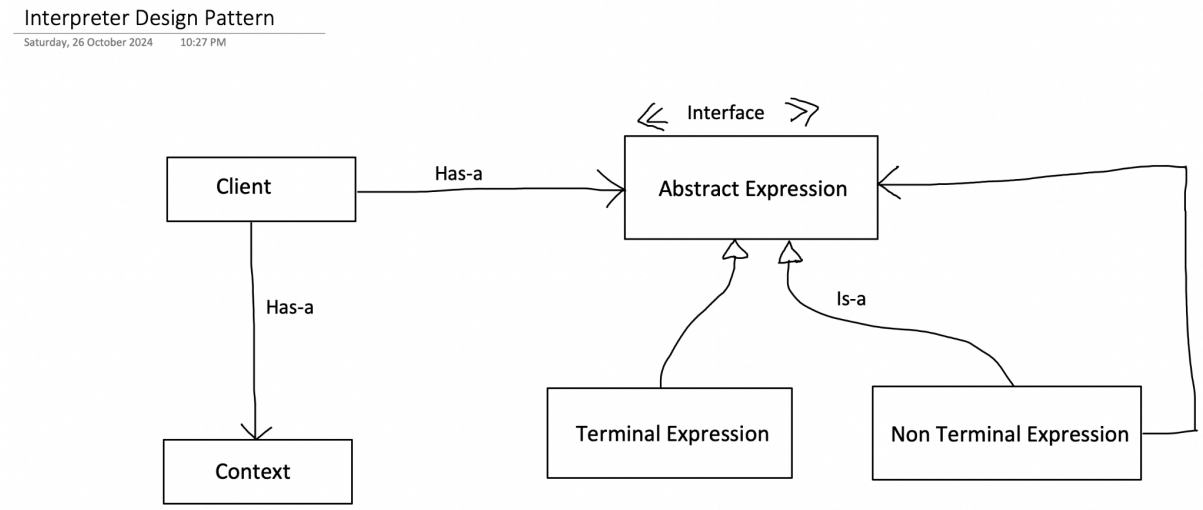


Double Dispatch = Which method needs to be invoked depends both on the caller and argument. In Single Dispatch which method needs to be invoked depends only on the caller.

Interpreter Design Pattern

Interpreter Design Pattern is used to evaluate expressions based on the context.

Based on the context, the same expression might evaluate to different results.



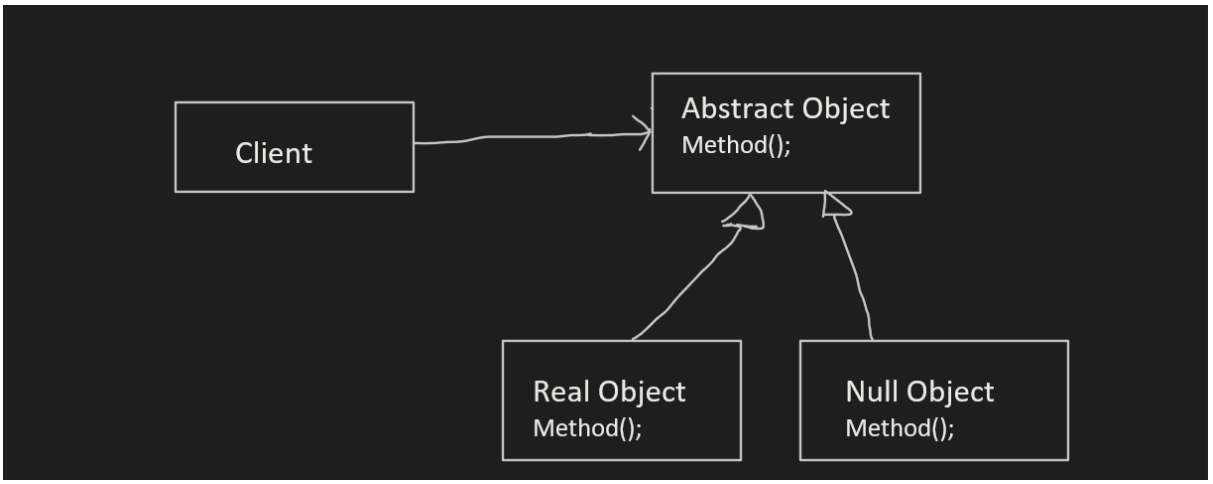
Other Design Patterns

- Null Object Design Pattern
- Model View Controller Design Pattern (MVC)

Null Object Design Pattern

Before performing any actions on an object, or before calling any method on it, we first need to check if the object is non null. If the object is null and we call any method on it then we will get a Null Pointer Exception. To check for null objects, we can simply use an if-else check, however in large codebases this could be problematic as we will have 1000s of such conditions. The Null Object Design Pattern is an alternative.

In the Null Object Design Pattern:
=> A null object replaces the NULL return type.
=> No need to put if checks for checking NULL.
=> Null Object reflects do-nothing or default behavior.



Model View Controller (MVC)

MVC consists of:
1. Model
2. View
3. Controller

Frameworks like Springboot, Django follow the MVC model.

MVC model creates a tiered architecture, where in the View layer communicates with the Controller layer, while the Controller layer communicates with the Model layer.
The Model layer itself communicates with the Databases.
The Client interacts with the View layer.

=> So the controller acts as the mediator between view and model. It accepts the user requests from the View layer, interprets them and redirects them to the Model.

=> The Model holds the data, and a list of entities / POJO Classes. The Model layer handles the data, and can connect to the database.

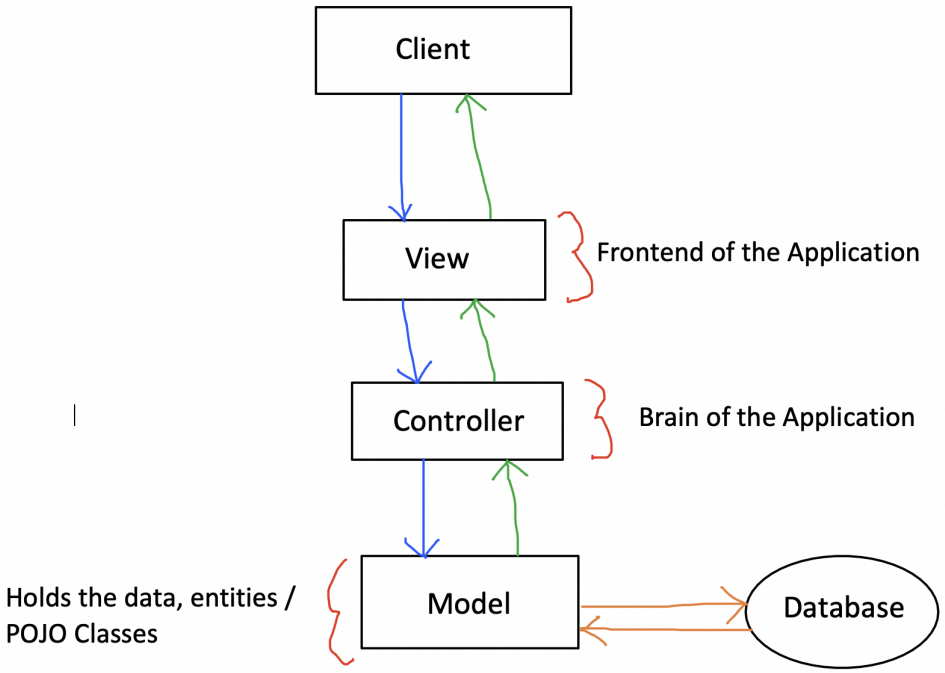
=> Controller is the brain of the application, i.e. it holds the business logic.

=> The Model will return the required data to the controller, which passes it back to the View, the View layer will render the output for the client, since View is the frontend of the application.

Advantages of the MVC pattern:
1. Separation of responsibility
2. Decoupling of the components, the View, Controller or Model can scale independently.
3. We can make changes to each of the components in isolation.

MVC

Saturday, 26 October 2024 7:14 PM



Low Level Design Case Studies

- **Parking Lot**
- **Tic Tac Toe**
- **Elevator**
- **Vehicle Rental System**
- **Snake and Ladders Game**
- **Movie Ticket Booking application**
- **Vending Machine**
- **ATM**
- **File System**
- **Cricbuzz**
- **Inventory Management System**

Designing a Parking Lot

Clarification Questions

1. What are the different types of spots in the Parking Lot?
2. How many entrances and exit gates does the parking lot have?
3. How many slots are there in the parking lot?

Requirements and blueprint of Objects needed

Objects:

1. Parking Slot
2. Vehicle
3. Entrance Gate
4. Exit Gate
5. Ticket

Functions and properties of different components:

1. **Parking Slot** - It is the space where the vehicle will be parked. Depending on the requirements we can have multiple types of parking slots like - 2 Wheeler, or 4 Wheeler.

   Properties : ID (this identifies the slot and helps us to locate it), isEmpty? (is the slot available), type (as mentioned above), vehicle (if occupied), price (each slot can have different prices).

2. **Vehicle** - Properties:
   a. Vehicle Number
   b. Type (2 wheeler or 4 wheeler) [Could be represented as Enum]

3. **Ticket** - Properties:
   a. Entrance Time
   b. Vehicle
   c. Parking Slot (slot allocated to the vehicle)

4. **Entrance Gate** - Functions:
   a. findParkingSlot - Find an empty slot in the parking lot where the incoming vehicle can be parked, if found mark the slot as

taken and remove it from further consideration.

b. Generate Ticket - Print a ticket containing the entrance time and the parking slot id which has been allocated for this vehicle.

5. Exit Gate - Functions:
   a. releaseParkingSlot - Since the vehicle is leaving we can release the parking slot where it was parked, and this slot should now be available for further consideration.

   b. Cost Calculation - Depends on the model of the charging, could be per hour or per minute for example. The exit gate can check the entrance time on the ticket and determine exactly the amount of money to charge the customer.

   c. Accept payments

Approaching the design (Here we do it in a bottom-up manner)

Parking Slot (the most basic element of the parking lot):

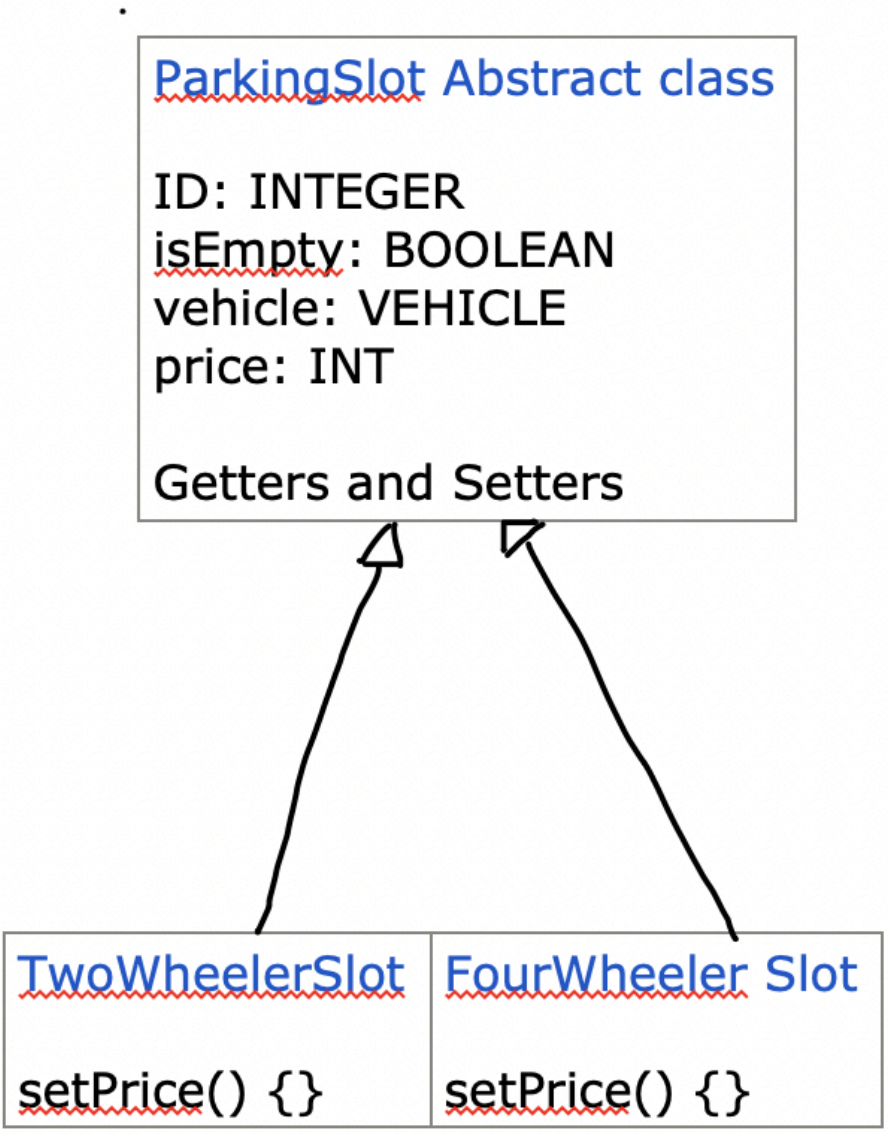
ParkingSlot Abstract class ID: INTEGER isEmpty: BOOLEAN vehicle: VEHICLE price: INT Getters and Setters

This interface will be subclassed by the various types of slots - 2 wheeler, 4 wheelers, heavy vehicles etc.

TwoWheelerSlot setPrice() {}	FourWheeler Slot setPrice() {}

TwoWheelerSlot and FourWheeler are concrete classes which subclass the ParkingSlot (abstract) class and override the setPrice() method as each

type of slot can have a different price. We can add more slot types as need be, hence allowing for a flexible design.



Parking Slot Manager

We define the Parking Slot manager to manage everything related to slots, i.e. it manages the parking slots.

ParkingSlotManager Abstract Class

```
List<ParkingSlot> parkingSlots;
ParkingSlotManager (List<ParkingSlot> parkingSlots) {
    this.parkingSlots = parkingSlots;
}


findParkingSlot()
addParkingSlot()
removeParkingSlot()
parkVehicle()
removeVehicle()
```

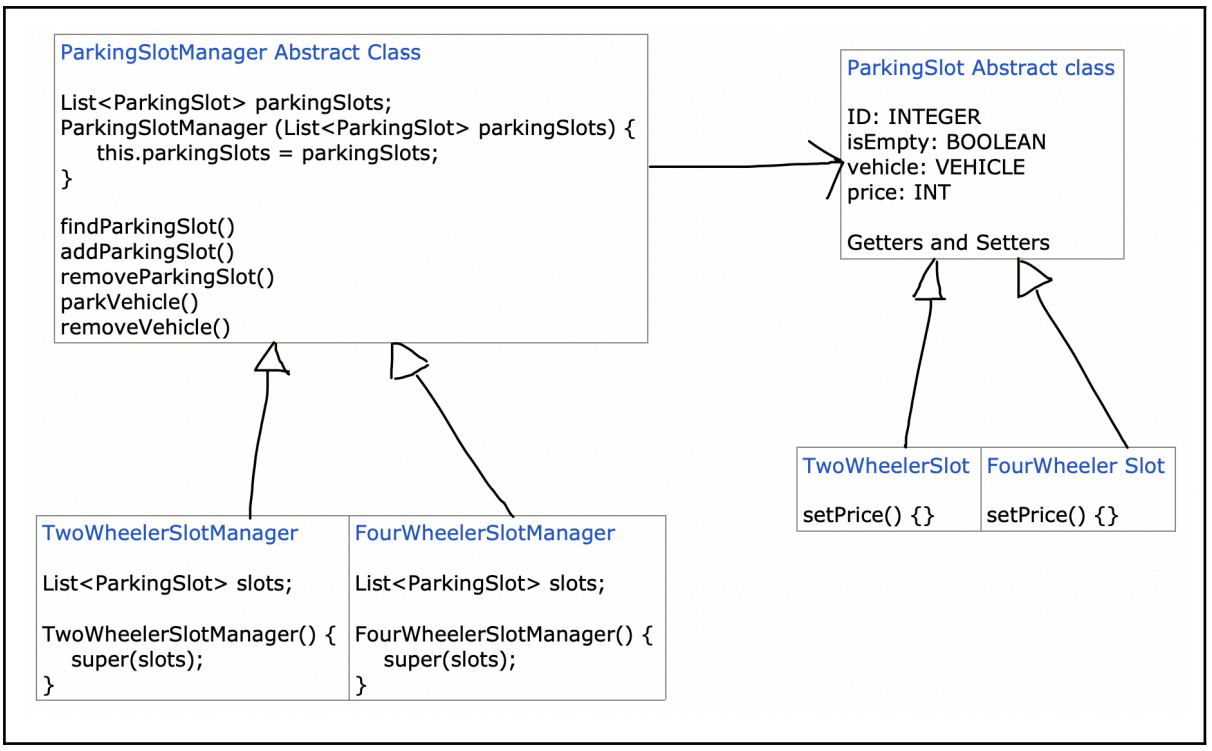
This class will be subclassed by the TwoWheelerSlotManager and FourWheelerSlotManager.

TwoWheelerSlotManager

```
List<ParkingSlot> slots = new
ArrayList<>();

TwoWheelerSlotManager() {
   super(slots);
}
```

FourWheelerSlotManager

```
List<ParkingSlot> slots = new
ArrayList<>();

FourWheelerSlotManager() {
   super(slots);
}
```



Interlude: Parking Strategy

Note on findParkingSlot() - Find parking slot will find an empty slot in the parking area, by default it'll find any empty slot, however we might want to customize its behavior to return the vacant parking slot nearest to the entrance area, or we might want to find one close to the exit gate and other such choices, we can model these choices using the strategy pattern.

We define a ParkingStrategy and subclass it with various concrete classes encompassing the various choices of parking policy.

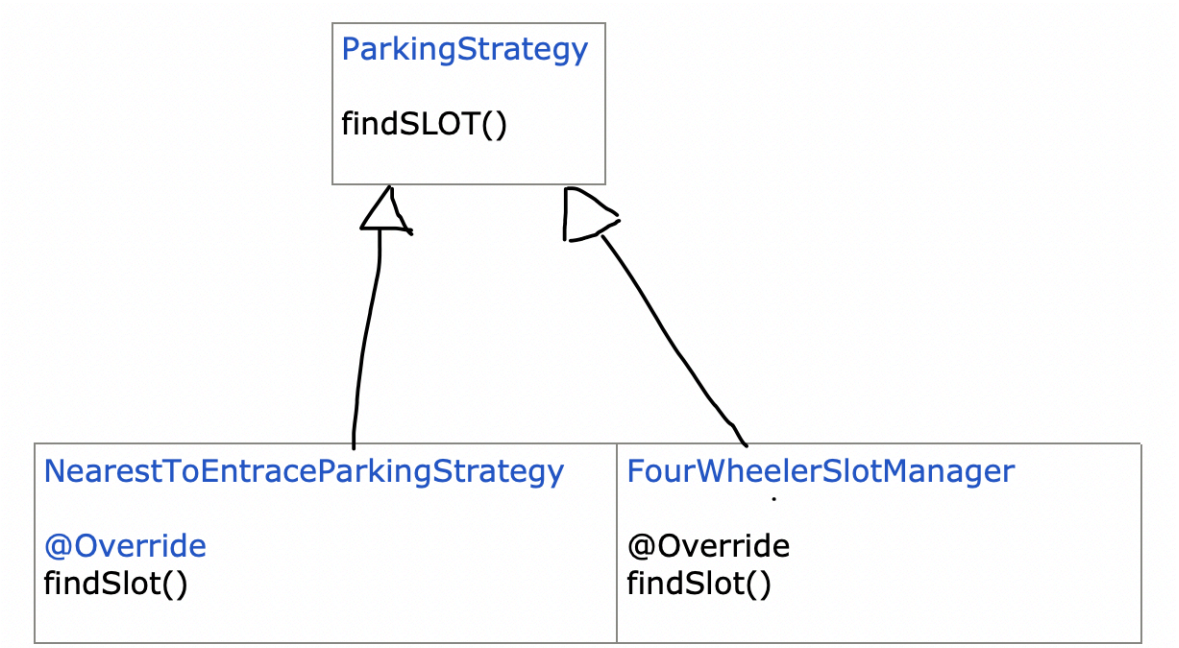
Now in addition to passing the list to the ParkingSlotManager parent class, we'll also pass the parking strategy.

For example:

TwoWheelerSlotManager

```
List<ParkingSlot> slots = new ArrayList<>();
ParkingStrategy parkingStrategy;
```

```
TwoWheelerSlotManager() {
    super(slots, parkingStrategy);
}
```



Vehicle

```
Vehicle

vehicleNumber: INT
vehicleType: VT (ENUM)
```

Here we define an enum ‘VT’ to store various vehicle types.

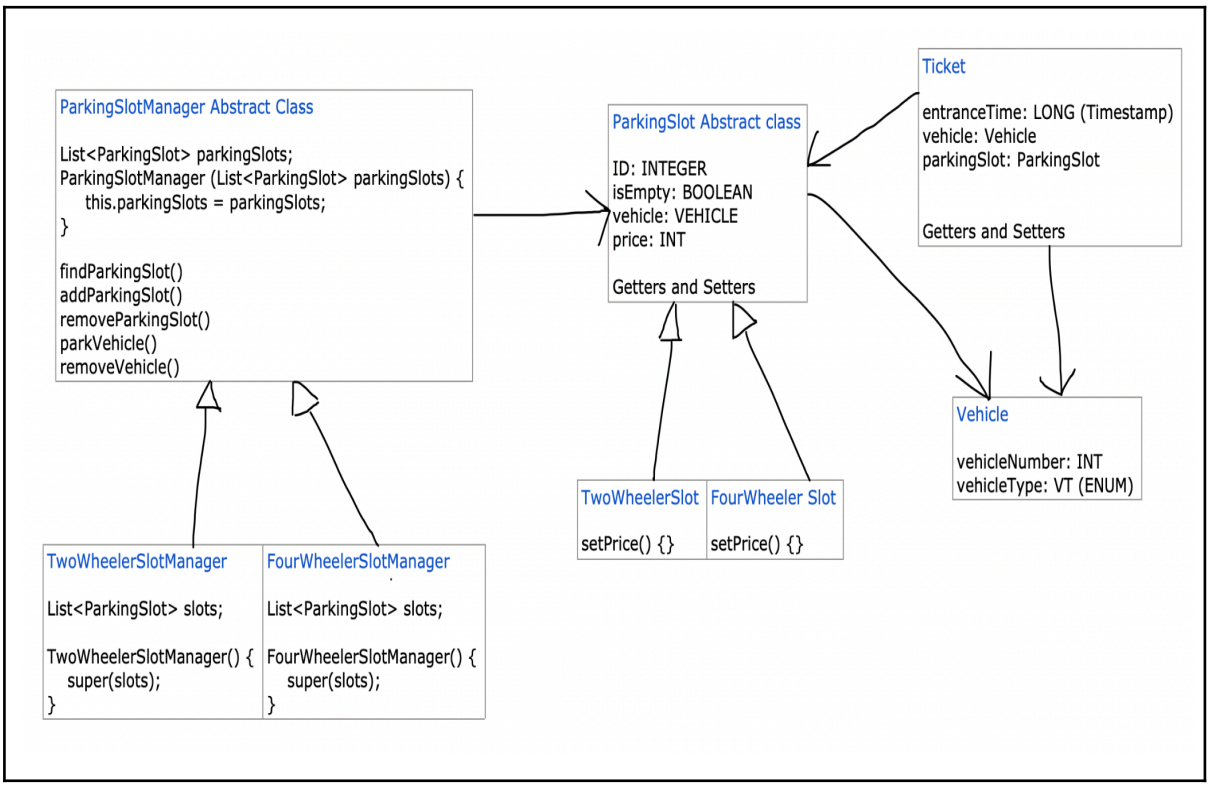
```
enum VT {
    TwoWheeler,
    FourWheeler,
    Truck;
}
```

Ticket

Ticket

entranceTime: LONG (Timestamp)
vehicle: Vehicle
parkingSlot: ParkingSlot

Getters and Setters



Entrance Gate

The EntranceGate class will handle multiple things:
=> It will invoke the findParkingSlot() method provided by the ParkingSlotManager. However which implementation of the ParkingSlotManager should it call, we need to decide that on the basis of the vehicle type (if it's 2 wheeler or 4 wheeler), to do this we define a ParkingSlotManagerFactory which will return an object of the appropriate factory.
=> Once a slot is found it should call parkVehicle (provided by the parkingSlotManager) to actually assign the vehicle to that spot.

=> EntreGate should call generateTicket (provided by the Ticket method).

EntranceGate

```
ParkingSlotManagerFactory parkingSlotManagerFactory;
ParkingSlotManager parkingSlotManager;
Ticket ticket;

EntranceGate(ParkingSlotManagerFactory parkingSlotManagerFactory) {
    this.parkingSlotManagerFactory = parkingSlotManagerFactory;
}


findSlot() {} // calls findParkingSlot provided by ParkingSlotManager
reserveSlot() {} // calls parkVehicle provided by ParkingSlotManager
generateTicket() {} // calls generateTicket provided by Ticket class.
```

All 3 routines should be called as part of single workflow for example by a find_reserve_slot_and_generate_ticket()

findSlot() => Internally calls findParkingSlot provided by ParkingSlotManager, we pass the vehicle type and the entrance gate as arguments. We pass the entrance gate because it might be useful to the policy we are using to find a free slot.
It returns the slot found.

reserverSlot() => Internally calls parkVehicle provided by ParkingSlotManager, we pass the vehicle and the slot returned by findParkingSlot

generateTicket() => Internally calls generateTicket provided by Ticket class. We pass the vehicle and the slot. This method will get the current timestamp and generate the ticket.

The ParkingSlotManagerFactory abstracts the details of ParkingSlotManager object creation, the client does not need to deal with that complexity. Instead ParkingSlotManagerFactory will figure out which object to create based on some conditions (in this case vehicle type).

ParkingSlotManagerFactory

```
factoryMethod()
```

Exit Gate

The major functionality of the exit gate include Cost Computation, Payment and freeing the parking slot.

Cost Computation - We can have multiple payment strategies like charge per minute, charge per hour, static charges etc, and each vehicle type will have a pricing strategy for example 2-Wheeler might be charged per hour and 4-wheeler per minute.

Modeling pricing strategies

PricingStrategy

```
price(Ticket ticket) {}
```

HourlyPricingStrategy

```
price(Ticket ticket) {
    h = hours(cur_time
            - ticket.time)

    return
  h * ticket.ps.price;
}
```

MinutePricingStrategy

```
price(Ticket ticket) {
    m = mins(cur_time
            - ticket.time)

    return
    m * ticket.ps.price;
}
```

StaticPricingStrategy

```
price(Ticket ticket) {
    return
    ticket.ps.price;
}
```

ps = ParkingSpot

Now we can design a CostComputation class which determines the cost of different types of vehicles (2-wheeler or 4-wheeler etc).

Modeling CostComputation

CostComputation

```
PricingStrategy pricingStrategy;
CostComputation(PricingStrategy pricingStrategy) {
  this.pricingStrategy = pricingStrategy;
```

```
}

cost() {
    return pricingStrategy.price();
}
```

TwoWheelerCostComputation

```
TwoWheelerCostComputation() {
    super(new
            HourlyPricingStrategy());
}
```

FourWheelerCostComputation

```
FourWheelerCostComputation() {
    super(new
            MinutePricingStrategy());
}
```

Finally the exit gate will need an object of either TwoWheelerCostComputation or FourWheelerCostComputation. To do this we can create a CostComputationFactory which will return the appropriate CostComputation object.

CostComputationFactory

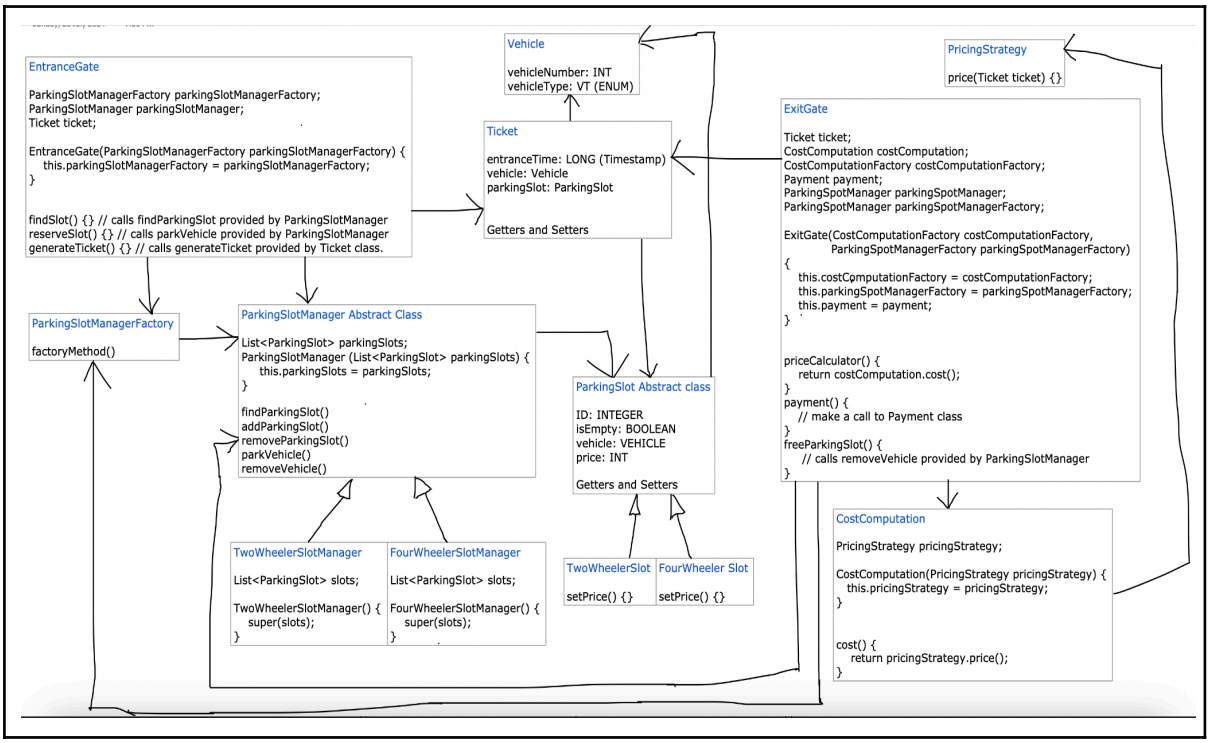
```
factoryMethod()
```

Finally, the Exit Gate class

ExitGate

```
Ticket ticket;
CostComputation costComputation;
CostComputationFactory costComputationFactory;
Payment payment;
ParkingSpotManager parkingSpotManager;
ParkingSpotManager parkingSpotManagerFactory;

ExitGate(CostComputationFactory costComputationFactory,
            ParkingSpotManagerFactory parkingSpotManagerFactory)
{
    this.costComputationFactory = costComputationFactory;
    this.parkingSpotManagerFactory = parkingSpotManagerFactory;
    this.payment = payment;
}
```

```
priceCalculator() {
    return costComputation.cost();
}

payment() {
    // make a call to Payment class
}

freeParkingSlot() {
     // calls removeVehicle provided by ParkingSlotManager
}
```



In addition we'll have a payment class which takes care of accepting payments, as we might have different payment modes so we need to implement a Payment strategy and then create a PaymentFactory to return the correct type of Payment Object to the ExitGate.

LLD_Parking_Lot.png

Designing Tic Tac Toe

Clarification Questions

Q. Are we designing a traditional tic tac toe game which has 2 players and a 3 X 3 grid.

Requirements and blueprint of Objects needed

Objects Needed 1. Piece Object 2. Board 3. Player 4. CentralGameClass

In a traditional tic tac toe game we have 2 types of pieces 'X' and 'O', here we will design an extensible tic tac toe game which can support multiple types of pieces, and a N x N board instead of a fixed 3 x 3 board. The player object represents an actual player, each player will have a piece.

Approaching the design (Here we do it in a bottom-up manner)

Playing Piece

PlayingPiece Abstract class
 	
PieceType pieceType;
 	
PlayingPiece(PieceType pieceType) {
  this.pieceType = pieceType;
}
 	
Getters and Setters

The PlayingPiece abstract class is subclassed by concrete classes encompassing all the piece types.

PlayingPieceX
 	
PlayingPieceX() {
  super(PieceType.TypeX);
	PlayingPieceO
	
	PlayingPieceO() {
	  super(PieceType.TypeO)

}	}

We define an enum to store the actual piece types:

```
enum PieceType {
    TypeX,
    TypeO;
}
```

Board

PlayingBoard

size: INTEGER
board: PlayingPiece[][]

```
PlayingBoard(int size) {
    this.size = size;
    board = new PlayingPiece[size][size]
}
```

addPiece(int X, int Y, PlayingPiece playingPiece)
getFreeCells()
printBoard()

The PlayingBoard class provides the method addPiece(x, y, pieceType) which takes two integer params indicating the location in the grid (2D matrix) where the piece needs to be placed, and ofcourse the piece type itself which needs to be placed.

getFreeCells() returns a list of all empty cells in the grid.

Player

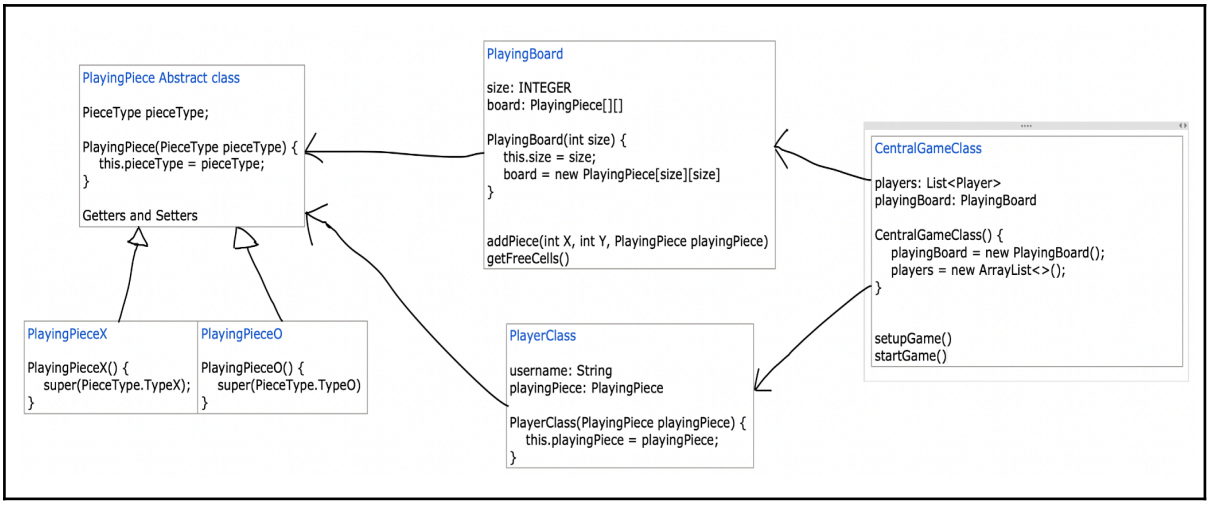
PlayerClass

```
username: String
playingPiece: PlayingPiece

PlayerClass(PlayingPiece playingPiece) {
    this.playingPiece = playingPiece;
}
```

CentralGameClass

Central Game Class is the coordinator which sets up and starts the game and handles all user interactions.

```
CentralGameClass

players: List<Player>
playingBoard: PlayingBoard

CentralGameClass() {
    playingBoard = new PlayingBoard();
    players = new ArrayList<>();
}


setupGame()
startGame()
```



TicTacToe.png

Designing an Elevator System

Clarification Questions

Q. What exactly do you mean by an elevator system Q. How many elevator cars does our system have

Requirements and Objects

Objects: 1. Building 2. Elevator Car 3. Doors 4. Buttons

Building - The building houses all the elevator cars.
Elevator Car - Each Elevator car consists of a display, buttons
Buttons - Internal Buttons (inside elevator cars), External button (present on each floor)

Considerations:
What logic do we use to dispatch lifts to people, i.e. if we have multiple requests and a person requests for a lift then which lift should service the request. Possibilities:
=> Odd / Even: Lift 1 handles even floors and Lift 2 handles odd floors.
=> DistributionL Lift A handles requests 1, 2, 3. Lift B handles 4, 5, 6 and so on.
=> Shortest seek time: lift nearest to the requested floor.

Approaching the design

Display

Display currentFloor: INTEGER direction: Direction curTime: Timestamp helplineNumber: Long

Getters and Setters

Direction is an enum, containing two options UP or DOWN

```
enum Direction {
    UP,
    DOWN;
}
```

ElevatorCar

ElevatorCar id: INTEGER display: Display inService: Boolean currentFloor: INTEGER direction: Direction buttons: InternalButton status: Status   move(int destinationFloor, Direction direction)

Status is an enum, containing two options in-service or out-of-service

```
enum Status {
    IN_SERVICE,
    OUT_OF_SERVICE;
}
```

ElevatorCarController

The Elevator car is a dumb object which provides a move method, however it does not have any logic to determine which floor to move to. It just moves to the specified floor, to control the ElevatorCar we need an ElevatorCarController.

ElevatorCarController

ElevatorCar elevatorCar;

acceptRequests(int destinationFloor, Direction direction);
controlElevatorCar();

acceptRequests - This method accepts all the incoming requests from the users, the requests can come in via the Internal Button or the External Buttons.

Each Elevator Car will have a controller. The ElevatorCarController is the central component which determines where the elevator car should move next, it will accept requests and make a choice.

Which component sends requests to the ElevatorCarController?

There are 2 components
=> Internal Button
=> External Button

InternalButton

InternalButton

InternalButtonDispatcher internalButtonDispatcher;

pressButton(int Button)

The InternalButton contains an object of InternalButtonDispatcher, which handles the logic of sending requests to appropriate ElevatorCarController.

InternalButtonDispatcher

InternalButtonDispatcher

List<ElevatorCarController> elevatorCarControllers;

```
submitRequest(int elevatorCarId);
```

submitRequest - This method sends a request to the appropriate elevator car, suppose there are 5 elevators and the user is inside elevator 2 and presses the button to go to a certain floor. This request should only be sent to ElevatorCarController of elevator car 2, and not to the other elevatorCarControllers.

Hence we pass the elevatorCarId to submitRequest, which determines the correct ElevatorCarController using the elevatorCarId and sends a request to it.

ExternalButton

```
ExternalButton

ExternalButtonDispatch externalButtonDispatches;

pressButton(Direction direction)
```

The ExternalButton is similar to InteralButton, each floor has an ExternalButton using which the user can send Elevator requests. The ExternalButtonDispatcher handles the logic of sending requests to the ElevatorCarController.
The external button has only 2 options (UP or DOWN directions).

ExternalButtonDispatcher

```
ExternalButtonDispatcher

List<ElevatorCarController> elevatorCarControllers;


submitRequests(Direction direction)
```

TheExternalButtonDispatcher needs to send the request to one ElevatorController, i.e. it needs some logic to determine which ElevatorCarController should handle the request and it’ll submit a request to it.

To determine which elevatorCarController to send requests to we need some logic, we can offload this to a separate ElevatorCarSelection Module which can implement multiple strategies, when ExternalButton needs to determine the elevatorCarController, it'll send a request ot this ElevatorCarSelection module which will return the appropriate ElevatorCar to send the request to, using which ExternalButtonDispatcher can determine the ElevatorCarController. For now we'll just subclass the ExternalButtonDispatcher with other classes which implement ElevatorCar selection logic.

OddEvenSelection getElevatorCarId()	StaticSelection getElevatorCarId()

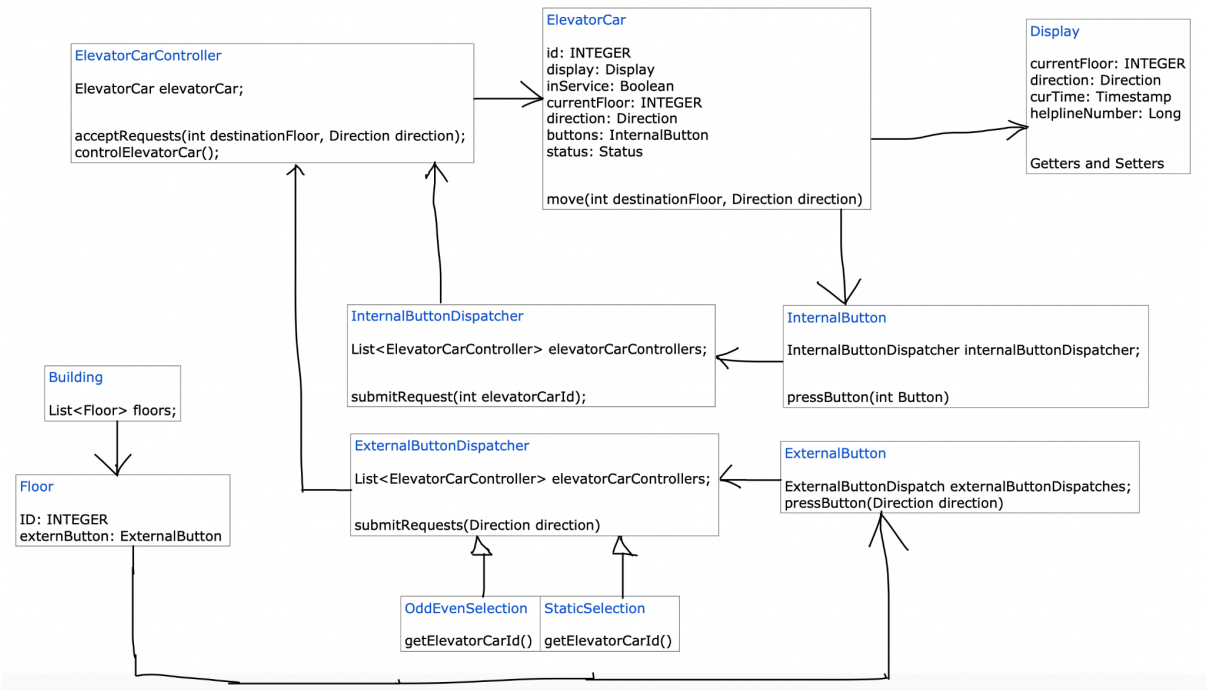
Floor

Each Floor has an ExternalButton

Floor ID: INTEGER externButton: ExternalButton

Building

Building List<Floor> floors;



ElevatorSystem.png

How will the elevator handle the requests? This design ensures whenever a person presses the External Button logic will be executed on the ExternalButtonDispatcher side to ensure that the request is sent to a single ElevactorCarController. However in what order will the Elevator handle the requests.

=> One Option is to use the SCAN (ELEVATOR) Algorithm, where the elevator moves in one direction till the end servicing any requests in its path which are also for movement in that direction, once elevator reaches the end it will reverse its direction of motion and serve the requests in the other direction.

=> The LOOK algorithm is an optimization of SCAN, it prevents the elevator from moving all the way to the end if there are no further requests for movement in that direction.

SCAN and LOOK are Disk Scheduling Algorithms.

Designing an Vehicle Rental System

Requirements and Objects

1. Vehicle
2. User
3. Location
4. ReservationDetails
5. Store
6. Invoice

Approaching the design

Vehicle

Vehicle

```
vehicleID: INTEGER
vehicleNumber: INTEGER
vehicleType: VehicleType
modelName: STRING
reservationStatus: ReservationStatus
hourlyRentalCost: INTEGER

Vehicle(VehicleType vehicleType) {
    this.vehicleType = vehicleType;
}
```

Getters and Setters

The Vehicle class is subclassed by concrete classes encompassing the type of Vehicles, in this case we consider Car and Bike

Here vehicleType and reservationStatus are enums.

```
enum ReservationStatus {
    RESERVED,
    AVAILABLE;
}
```

```
enum VehicleType {
    CAR,
    BIKE;
}
```

Car	Bike

Car() {
    super(VehicleType.CAR);
}
	Bike() {
	    super(VehicleType.BIKE);
	}

Store

Store
storeID: INTEGER vehicleInventory: VechicleInventory location: Location reservationDetails: List<ReservationDetails>

Each store will have some vehicles (a list of vehicles)

VechicleInventory

VechicleInventory
List<Vehicles> vehicles CRUD operations on Vehicles

The Vehicle Inventory will manage the vehicles for a particular store.

Each store will have a location.

Location

Location

cityName: STRING
stateName: STRING
countryName: STRING
pinCode: INTEGER

ReservationDetails

ReservationDetails

reservationID: INTEGER
reservedBy: USER
vehicle: Vehicle
reservedFrom: TIMESTAMP
reservedTill: TIMESTAMP

Getters and Setters

User

User

ID: INTEGER
userName: STRING
phoneNumber: INTEGER
drivingLicense: STRING

CentralVehicleRentalSystem Class

CentralVehicleRentalSystem

users: List<User>
stores: List<Store>

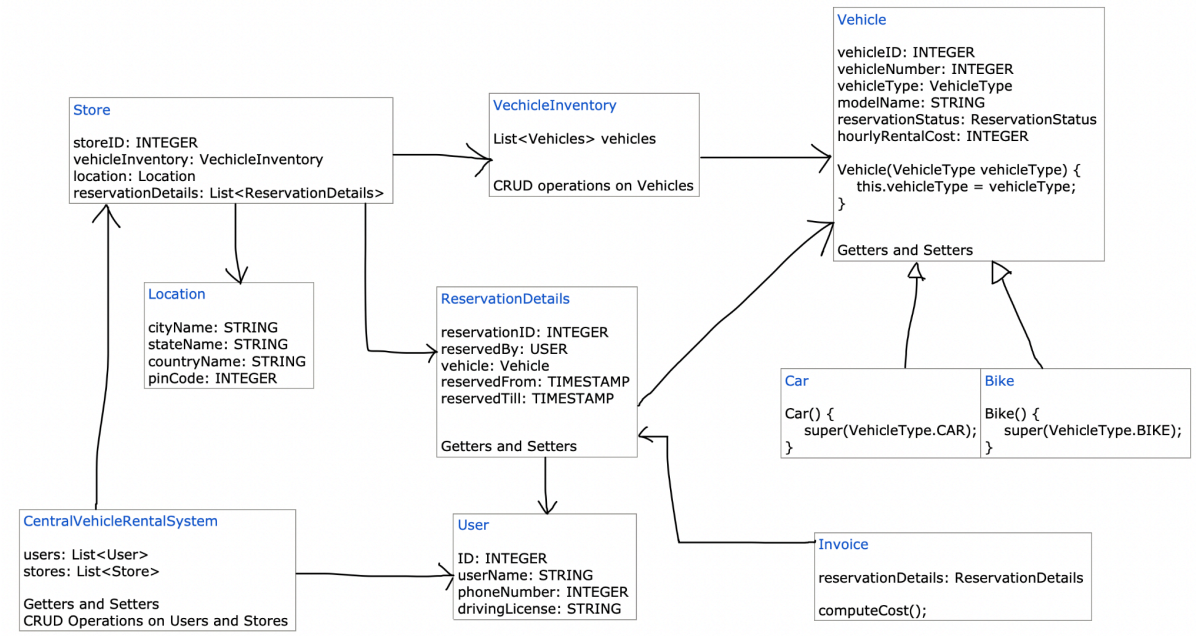
Getters and Setters
CRUD Operations on Users and Stores

Invoice

Invoice

reservationDetails: ReservationDetails

computeCost();



VehicleRental.png

Designing a Snake and Ladders Game

Clarification Questions

Q. Is this a traditional Snake and Ladders game where the number goes up to 100, and the board is 10 x 10?
Q. How many players can we have?
Q. How many dice?
Q. When should the game end?

Requirements and Objects

Objects:
1. Dice
2. Snake
3. Ladder
4. Board
5. Players
6. Cells

Approaching the design (in a Bottom-Up manner)

Player

Player

userName: STRING
currentPosition: INTEGER

Dice

Dice

int diceCount;

rollDice();

Jump

Instead of creating separate Snake and ladder objects, we can create a Jump object, this is because both Snake and Ladder components contain the same data (int start and int end), hence a better design would be to use a single class ‘Jump’ which can represent both.

Jump
start: INTEGER end: INTEGER

Cell

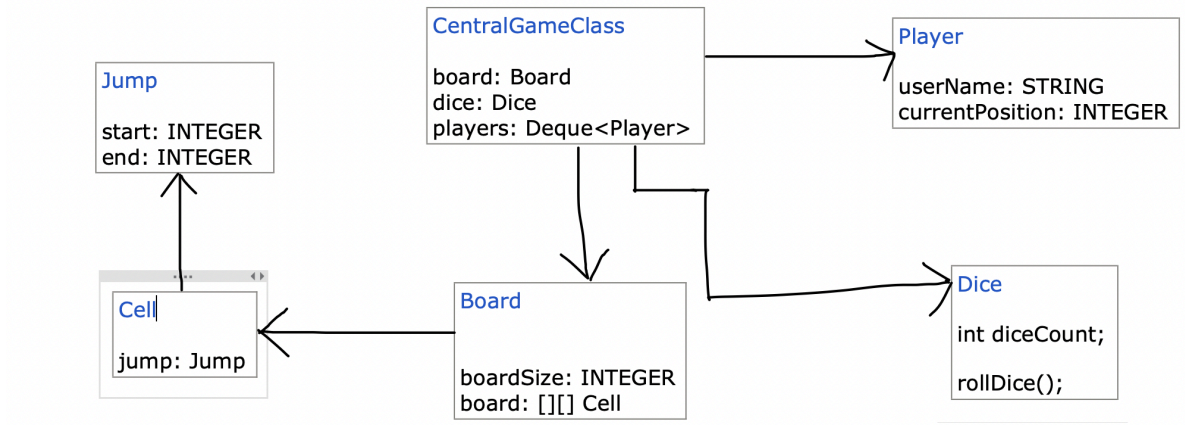
Cell
jump: Jump

Board

Board
board: [][] Cell

CentralGameClass

CentralGameClass
board: Board dice: Dice players: Deque<Player>



SnakeAndLadders.png

Designing a Movie Ticket Booking application

Objects and Requirements

Objects:
1. Movie
2. City
3. Theater
4. Screens
5. User
6. Shows
7. Booking
8. Payment

Approaching the design (in a Top-Down manner)

Movie

Movie

ID: INTEGER
Name: String
durationInMins: INT

MovieController

MovieController

Map<City, List<Movie>> cityMovies
List<Movie> allMovies

CRUD operation on Movies

Theater

Theater

ID: INTEGER
address: String
city: City
screens: List<Screen>

shows: List<Show>

Screen

Screen

ID: INTEGER
seats: List<Seat>

Show

Show

ID: INTEGER
movie: Movie
screen: Screen
startingTime: TIMESTAMP
bookedSeatIds: List<int>

Seat

Seat

ID: INTEGER
rowNum: INTEGER
seatCategory: SeatCategory
price: INTEGER

SeatCategory is an enum

```
enum SeatCategory {
    SILVER,
    GOLD,
    PLATINUM;
}
```

TheaterController

TheaterController

```
Map<City, List<Theater>> cityTheaters;
List<Theater> allTheaters;
```

Booking

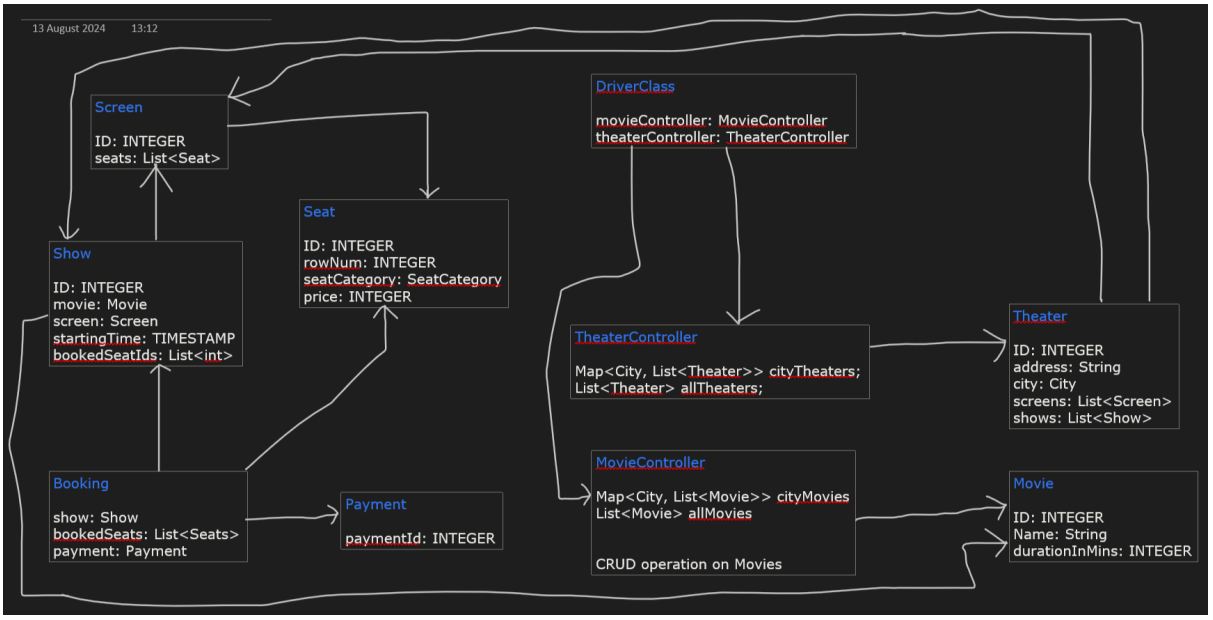
```
Booking

show: Show
bookedSeats: List<Seats>
payment: Payment
```

Payment

```
Payment

paymentId: INTEGER
```

DriverClass

```
DriverClass

movieController: MovieController
theaterController: TheaterController
```



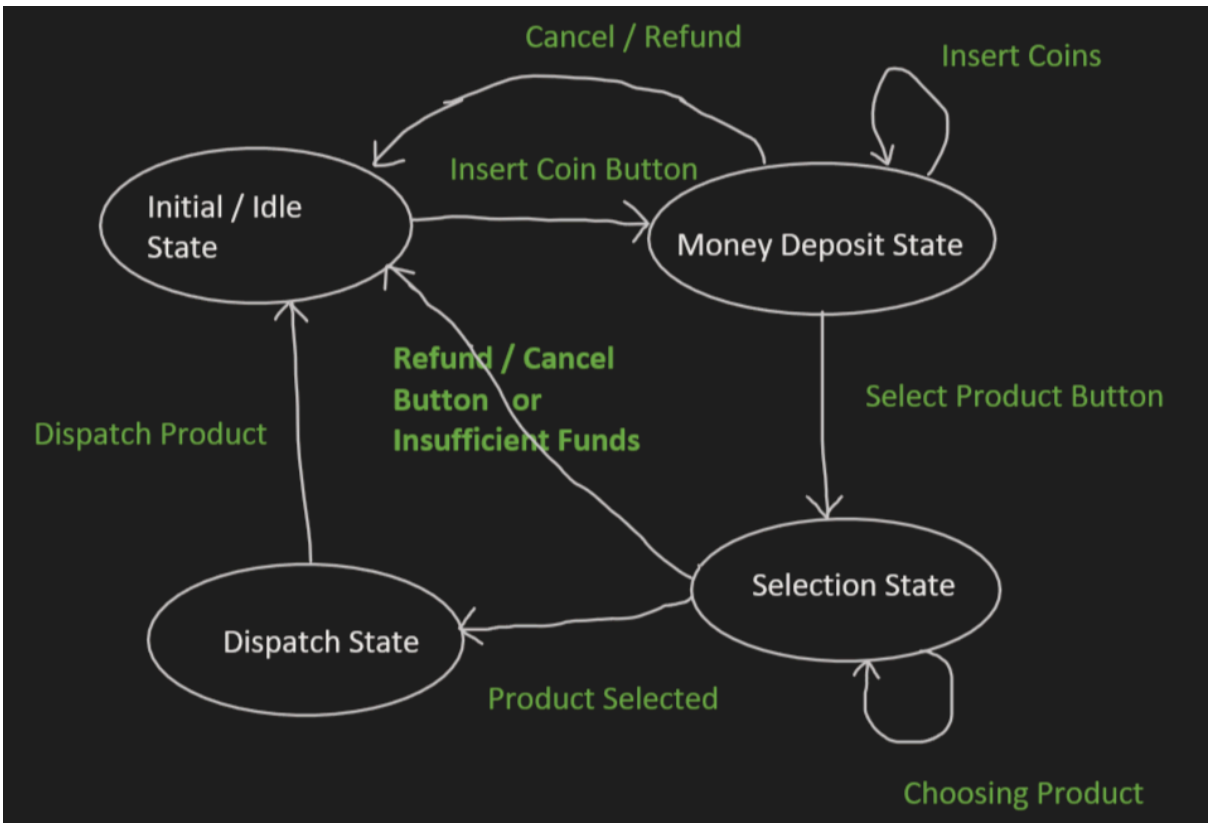
Movie Ticket Booking.png

To handle concurrency we can use Optimistic Locking, Remember in

optimistic locking always acquire and then compare the row version rather than the other way around. Because if we first check the version and acquire the lock based on it, then by the time we enter the critical section it is possible that some other thread has already made an update. To fix this behavior, compare versions post acquiring locks.

Design a Vending Machine

Flow of the Vending Machine



For this design we'll be using the State Design Pattern.

Approaching the Design

State

State Interface

pressInsertCoinButton (VendingMachine vendingMachine)
insertCoin (VendingMachine vendingMachine)
pressSelectProductButton (VendingMachine vendingMachine)
selectProduct (VendingMachine vendingMachine)
getChange(VendingMachine vendingMachine)
refundMoney(VendingMachine vendingMachine)
dispatchProduct(VendingMachine vendingMachine)

The state interface is subclassed by various concrete classes encompassing all the possible states.

IdleState Implement press coin Button	DepositMoneyState Implement insert Coin, Press Select Product Button and refund Money	SelectProductState Implement press select Product, refund Money and get Change	DispatchProductState Implement dispatchProduct

Item

Item itemType: ItemType price: INTEGER

Here ItemType is an enum

```
Enum ItemType {
    COKE,
    PEPSI,
    WATER,
    SODA;
}
```

Each item is present in a slot, let’s call it an ItemShelf

ItemShelf

ItemShelf item: Item price: INTEGER isSoldOut: BOOLEAN

Next, we design an inventory which manages all the ItemShelf

Inventory

Inventory

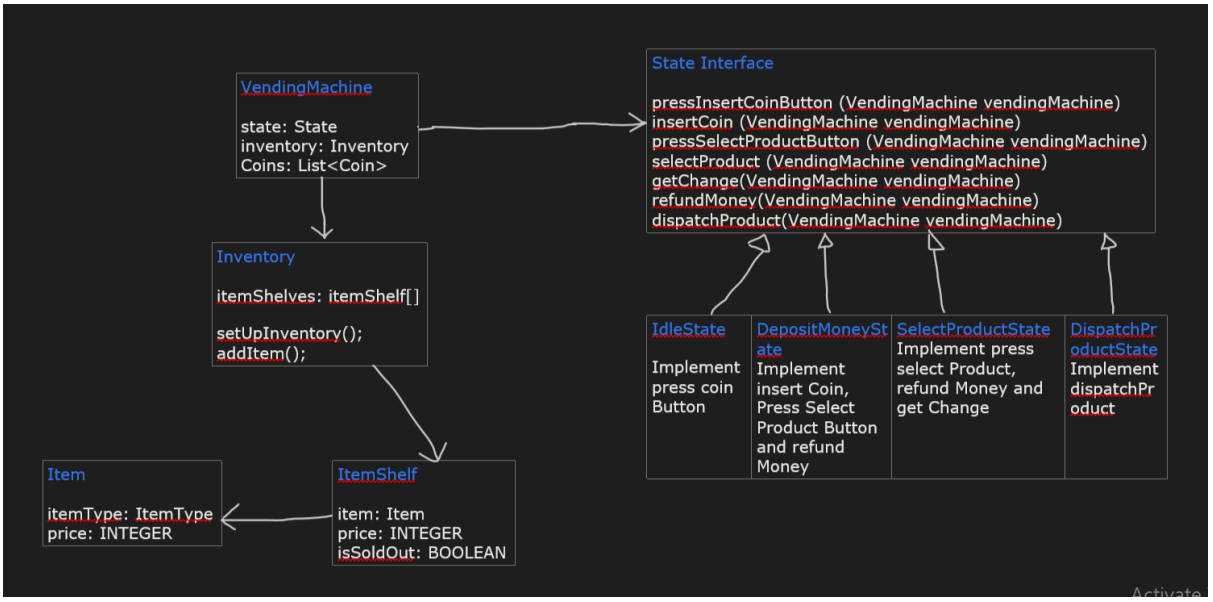
```
itemShelves: itemShelf[]


setUpInventory();
addItem();
```

Coin is an enum.

```
enum Coin {
   PENNY,
   NICKEL,
   DIME,
   QUARTER;
}
```

VendingMachine

```
VendingMachine

state: State
inventory: Inventory
Coins: List<Coin>
```



VendingMachine.png

Design an ATM

Designing an ATM involves using both the State design pattern and Chain of Responsibility Design Pattern.

Similar to Vending Machine design, an ATM can be broken down into multiple well defined states and each state has certain operations which are valid for it.

Example

States	Allowed Operations
IdleState	InsertCard
HasCardState	AuthenticatePin
OperationSelectionState	SelectOperation

etc.

Approaching the design

State

State Interface

insertCard(ATM atm);
authenticatePin(ATM atm);
selectTransactionalOperation(ATM atm);
displayAccountBalance(ATM atm);
withdrawMoney(ATM atm);

IdleState	HasCardState	OperationSelectionState	PerformOperationState
Implements insertCard	Implements authenticatePin	Implements displayAccountBal and "selectTransactionalOperation()"	Implements withdrawMoney()

Inventory

Inventory

denominations: Map<Currency, Integer>
withdrawHandler: WithdrawHandler

CRUD Operations on Currency and Denominations
withdrawMoney();
updateInventory();
displayInventory();
getUserAccountBalance();

Where Currency is an enum

```
enum Currency {
    note2000,
    note500,
    note100;
}
```

WithdrawHandler

WithdrawHandler Abstract class

WithdrawHandler nextWithdrawHandler;

processWithdrawalRequest(int money);

WithdrawHandlerDen2000	WithdrawHandlerDen500	WithdrawHandlerDen100

constructor(nextWH) {
  super(nextWH);
}
	constructor(nextWH) {
	  super(nextWH);
	}
		constructor(nextWH) {
		  super(nextWH);
		}

processWithdrawalRequest(int money);	processWithdrawalRequest(int money);	processWithdrawalRequest(int money);

User

User

userName: String
userPin: INTEGER
accountBalance: INTEGER

GETTERS and SETTERS

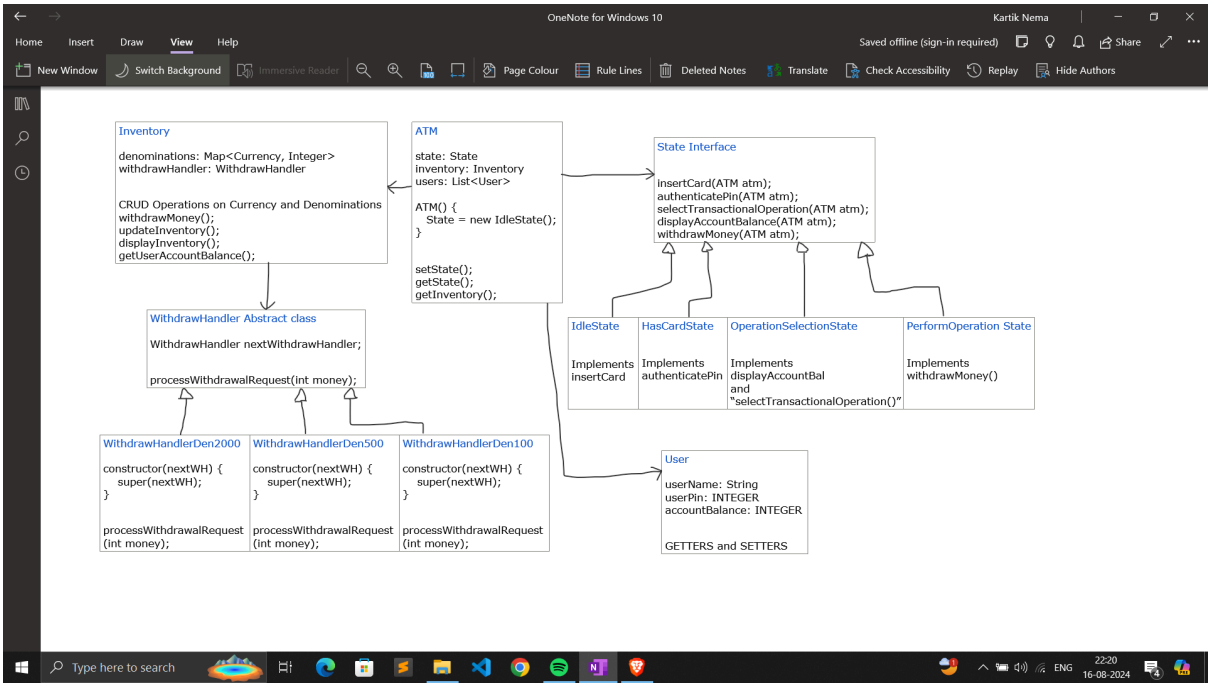
ATM

ATM

state: State
inventory: Inventory
users: List<User>

```
ATM() {
   State = new IdleState();
}
```

setState();
getState();
getInventory();


OneNote for Windows 10
Kartik Nema
Home
Insert
Draw
View
Help
Saved offline (sign-in required)
Share
New Window
Switch Background
Immersive Reader
Page Colour
Rule Lines
Deleted Notes
Translate
Check Accessibility
Replay
Hide Authors
Inventory
denominations: Map<Currency, Integer>
withdrawHandler: WithdrawHandler
CRUD Operations on Currency and Denominations
withdrawMoney();
updateInventory();
displayInventory();
getUserAccountBalance();
ATM
state: State
inventory: Inventory
users: List<User>
ATM() {
State = new IdleState();
}
setState();
getState();
getInventory();
State Interface
insertCard(ATM atm);
authenticatePin(ATM atm);
selectTransactionalOperation(ATM atm);
displayAccountBalance(ATM atm);
withdrawMoney(ATM atm);
WithdrawHandler Abstract class
WithdrawHandler nextWithdrawHandler;
processWithdrawalRequest(int money);
IdleState
Implements insertCard
HasCardState
Implements authenticatePin
OperationSelectionState
Implements displayAccountBal and "selectTransactionalOperation()"
PerformOperation State
Implements withdrawMoney()
WithdrawHandlerDen2000
constructor(nextWH) { super(nextWH); }
processWithdrawalRequest (int money);
WithdrawHandlerDen500
constructor(nextWH) { super(nextWH); }
processWithdrawalRequest (int money);
WithdrawHandlerDen100
constructor(nextWH) { super(nextWH); }
processWithdrawalRequest (int money);
User
userName: String
userPin: INTEGER
accountBalance: INTEGER
GETTERS and SETTERS
Type here to search
ENG
22:20
16-08-2024

ATM.png

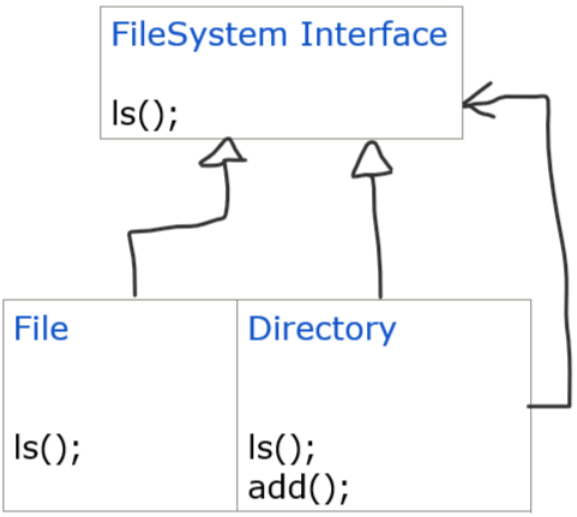
Design a File System

We use the composite design pattern for designing a File System

FileSystem

FileSystem Interface ls();

File
 	
ls();
	Directory
	
	ls();
	add();



Design Cricbuzz

Requirements and Objects

Objects:
1. Match
2. Team
3. Innings
4. Players
5. Over
6. Ball
7. Scorecard

Approaching the design (Top Down)

Match

Match

teamA: Team
teamB: Team
matchType: MatchType
venue: String
innings: Inning[2]
date: Date
tossWinner: Team

startGame();

Team

Team

teamName: String
players: Queue<Player>
reserves: Queue<Player>
playerBattingController: PlayingBattingController
playerBowlingController: PlayerBowlingController

Player

```
Player

person: Person
playerType: PlayerType
battingScorecard: BattingScorecard
bowlingScorecard: BowlingScorecard
```

Here PlayerType is an enum

```
enum PlayerType {
    BATSMAN,
    BOWLER,
    WICKETKEEPER,
    ALLROUNDER;
}
```

Person

```
Person

name: String
age: INTEGER
```

BattingScorecard

```
BattingScorecard

runs: INTEGER
balls: INTEGER
fours: INTEGER
sixes: INTEGER
strikeRate: DOUBLE
```

BowlingScorecard

```
BowlingScorecard

oversBowled: INTEGER
wicketsTaken: INTEGER
runsGiven: INTEGER
noBallsBowled: INTEGER
wideBallsBowled: INTEGER
```

economy: DOUBLE

PlayerBattingController

PlayerBattingController nextPlayers: Queue<Player> striker: Player nonStriker: Player

PlayerBowlingController

PlayerBowlingController bowlers: Deque<Player> oversCount: Map<Player, Integer> currentBowler: Player

MatchType

MatchType Interface
 	
noOfOvers();
maxOversAllowedByBowler();

OneDayMatchType
 	
noOfOvers () {
return 50;
}
 	
maxOversAllowedByBowler() {
return 10;
}
	T20MatchType
	
	noOfOvers () {
	return 20;
	}
	
	maxOversAllowedByBowler() {
	return 4;
	}
		TestMatchType
		
		noOfOvers () {
		return 450;
		}
		
		maxOversAllowedByBowler() {
		return 225;
		}

Innings

Innings

```
battingTeam: Team
bowlingTeam: Team
List<Over> overs


startInnings();
```

Over

```
Over


overNumber: INTEGER
balls: List<Bowl>

startOver();
```

Ball (Subject / Observable)

```
Ball

ballNumber: INTEGER
bowlType: BowlType
runType: RunType
facedBy: Player
bowledBy: Player
observers: List<ScoreUpdateObservers>

deliverBall();
notify();
```

Here bowlType is an enum

```
enum BowlType {
    LEGAL,
    WIDE,
    NOBALL
}
```

and runType is an enum as well

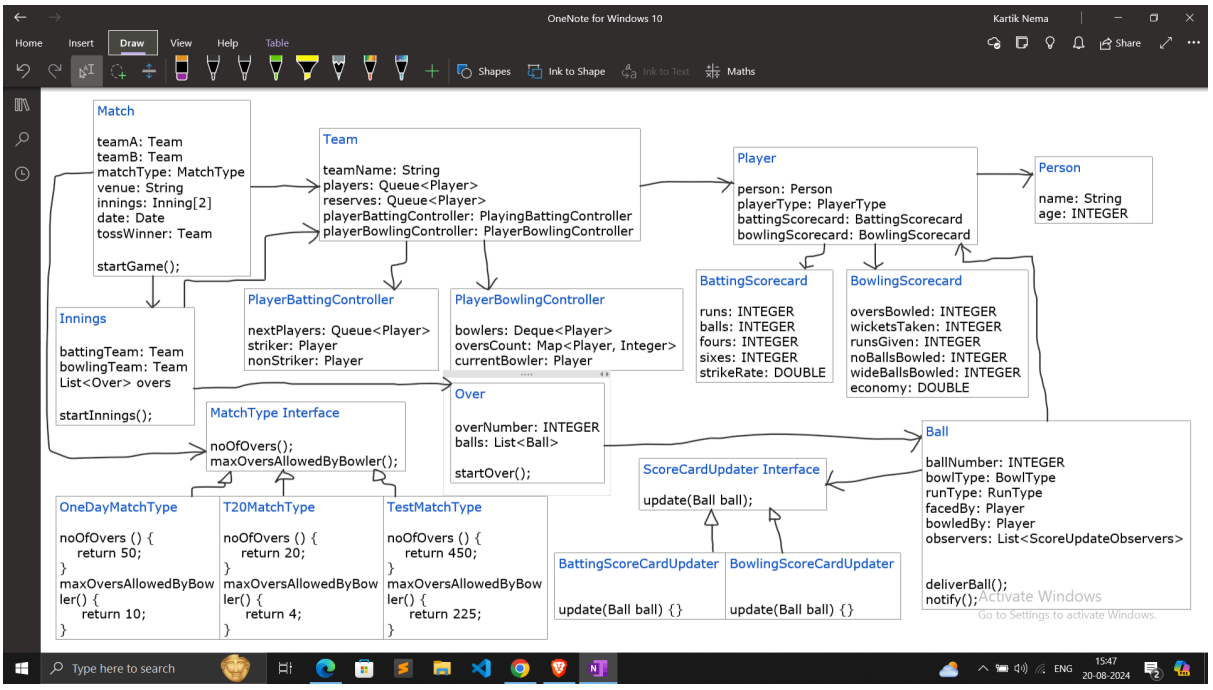
```
enum RunType {
```

```
    ONE,
    TWO,
    THREE,
    FOUR,
    SIX,
    WIDE;
}
```

ScoreCardUpdater (Observer)

ScoreCardUpdater Interface
update(Ball ball);

BattingScoreCardUpdater	BowlingScoreCardUpdater
update(Ball ball) {}	update(Ball ball) {}



Cricbuzz.png

Inventory Management System

We are designing an Inventory Management system and an order management system for a service like Zepto.

Flow:
->User opens the App
->Views a list of products
-> Adds products to cart
-> Places Order, gets Invoice
-> Makes Payment

Objects to be modeled:

1. User
2. Product
3. Cart
4. Order
5. Invoice
6. Payment
7. Inventory
8. Warehouse

A service like Zepto has multiple warehouses in any location, so we'll model them too.

Product

- ID
- name
- price
- cost: double

ProductCategory

- List<Product> products
- categoryName
- description

Methods:
addProduct

removeProduct

Inventory

- List<ProductCategory> products

Methods:
addCategory()
removeCategory()
updateCategory()

Warehouse

- location: Location
- inventory: Inventory
- warehouseOrderManagment: OrderManagement
- isFunctional

WarehouseController

- List<Warehouse> warehouses
- WarehouseSelection strategy strategy

Methods:
selectWarehouse() {strategy.selectWarehouse();}

WarehouseSelectionStrategy

NearestWarehouseSS	LeastTrafficWarehouseSS

Location

- address: String

User

- userName: String
- email: String
- contact: INT,
- address: String,
- cart: Cart
- List<OrderID> orders

Cart

- Map<ProductID, INT> productsInCart

Methods:
addToCart(),
removeFromCart(),
emptyCart()

OrderManagement

- List<Order> orders

CRUD Operations on Order

Order

- user: User
- Map<ProductID, int> products
- orderedAt: TIMESTAMP
- orderID: INT,
- warehouse: Warehouse
- invoice: Invoice
- address: Location
- payment: Payment

- orderStatus: enum {ONGOING, DELIVERED, CANCELED}

Invoice

- amountToBePaid
- tax
- createdAt

Payment

- paymentMode

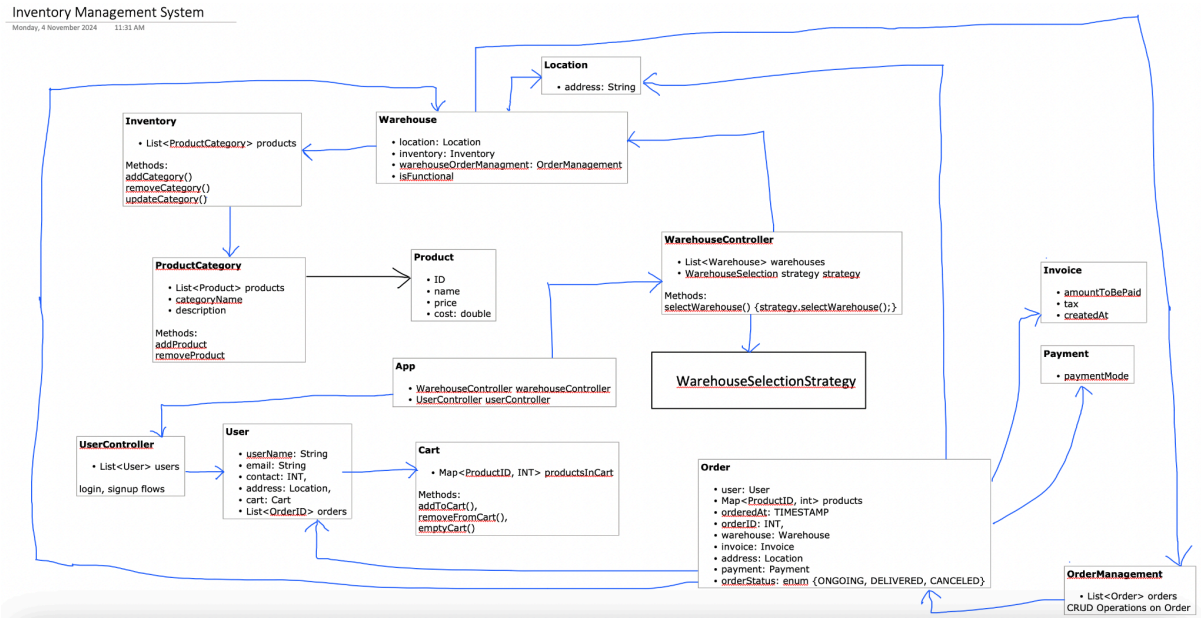
UserController

- List<User> users

login, signup flows

App

- WarehouseController warehouseController
- UserController userController



InventoryManagement.png

Chess

Objects to be Modeled:
- Board
- Player
- Cell
- Piece
- Inventory (Inventory of pieces for each player)

Piece

Piece Abstract class pieceType: PieceType Color: Color Methods: public abstract List<Integer> getAllValidMoves();

PieceType enum KING QUEEN KNIGHT ROCK BISHOP PAWN	Color enum WHITE BLACK

The Piece abstract class is subclassed by the different kinds of Pieces in Chess.

Pawn
 	
Pawn(type, color) {
  super(type, color);
}
	Knight
	
	Knight(type, color) {
	  super(type, color);
	}
		Queen
		
		Queen(type, color) {
		  super(type, color);
		}

Bishop
 	
Bishop(type, color) {
  super(type, color);
}
	Rock
	
	Rock(type, color) {
	  super(type, color);
	}
		King
		
		King(type, color) {
		  super(type, color);
		}

Cell

Cell class

piece: Piece
color: Color
cellID: INTEGER

getPiece()
setPiece(Piece piece)

Board

Board class

Cell [][] board
boardSize: INTEGER

// CRUD on Board

Player

Player class

userName: String
Color: Color
inventoryController: InventoryController
pieceAbstractFactory: PieceAbstractFactory

Inventory

Inventory class

Piece king, queen
List<Piece> pawns
List<Piece> knights, rocks, bishops

InventoryController

InventoryController class Inventory inventory CRUD on Inventory

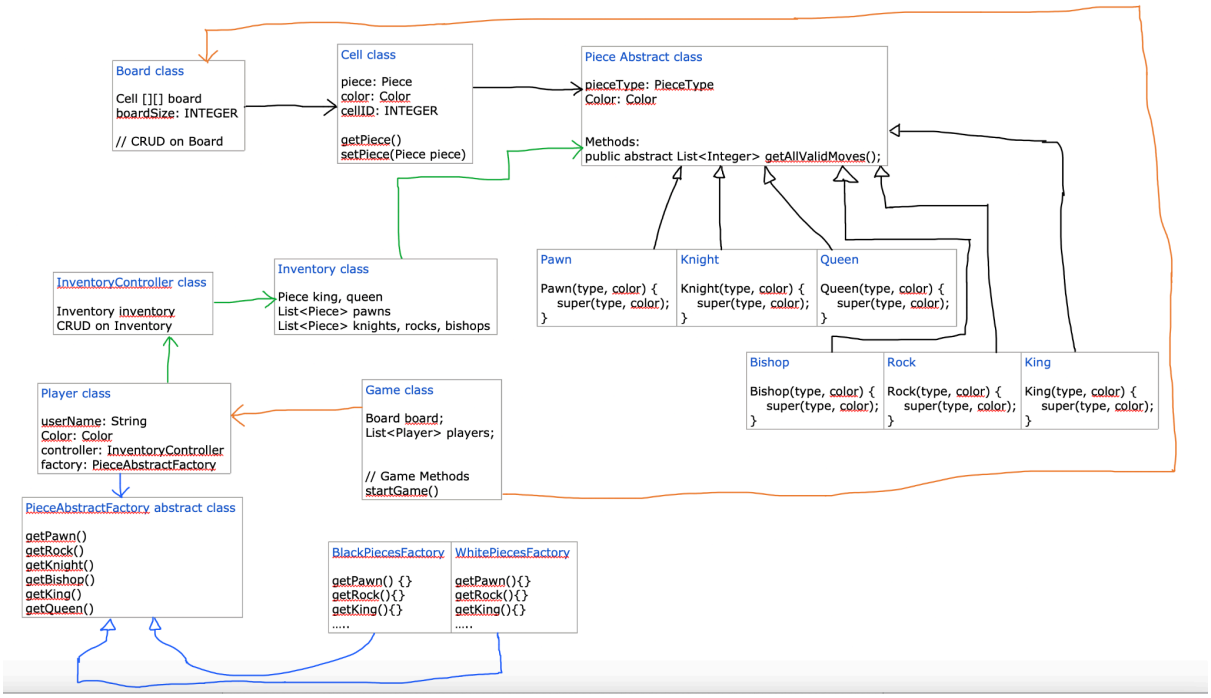
Game

Game class
 	
Board board;
List<Player> players;
 	
// Game Methods
startGame()

PieceAbstractFactory

PieceAbstractFactory abstract class getPawn() getRock() getKnight() getBishop() getKing() getQueen()

BlackPiecesFactory	WhitePiecesFactory
getPawn() {} getRock(){} getKing(){} …..	getPawn(){} getRock(){} getKing(){} …..


Board class
Cell [][] board
boardSize: INTEGER
// CRUD on Board
Cell class
piece: Piece
color: Color
cellID: INTEGER
getPiece()
setPiece(Piece piece)
Piece Abstract class
pieceType: PieceType
Color: Color
Methods:
public abstract List<Integer> getAllValidMoves();
InventoryController class
Inventory inventory
CRUD on Inventory
Inventory class
Piece king, queen
List<Piece> pawns
List<Piece> knights, rocks, bishops
Pawn
Pawn(type, color) {
super(type, color);
}
Knight
Knight(type, color) {
super(type, color);
}
Queen
Queen(type, color) {
super(type, color);
}
Bishop
Bishop(type, color) {
super(type, color);
}
Rock
Rock(type, color) {
super(type, color);
}
King
King(type, color) {
super(type, color);
}
Player class
userName: String
Color: Color
controller: InventoryController
factory: PieceAbstractFactory
Game class
Board board;
List<Player> players;
// Game Methods
startGame()
PieceAbstractFactory abstract class
getPawn()
getRock()
getKnight()
getBishop()
getKing()
getQueen()
BlackPiecesFactory
getPawn() {}
getRock(){}
getKing(){}
.....
WhitePiecesFactory
getPawn(){}
getRock(){}
getKing(){}
.....

Chess.png

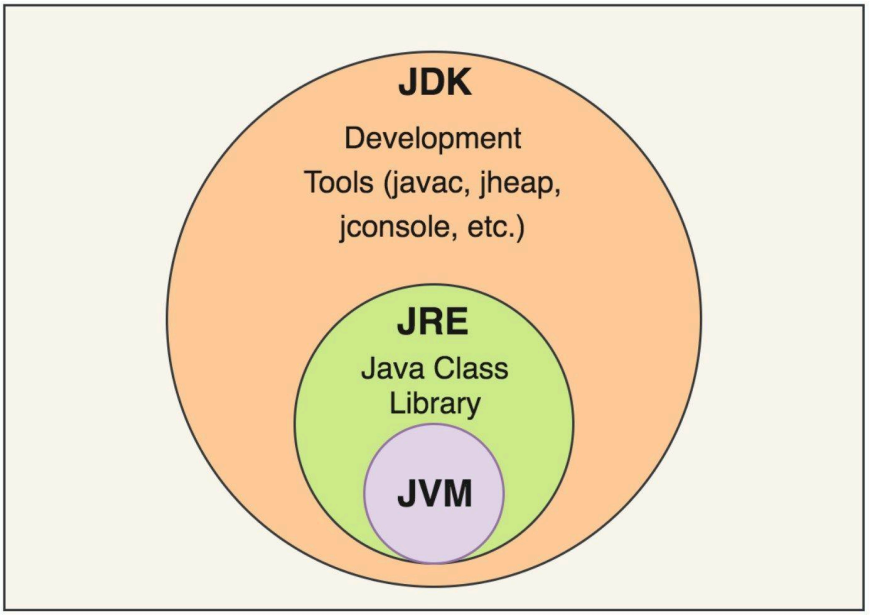
JAVA

- **JRE / JVM / JDK**
- **Threading**
- **Basics**
- **Types of Classes**
- **Primitive / Non-Primitive (includes String Pool)**
- **Methods in Java**
- **Constructors**
- **Interfaces**
- **Interfaces extension: New Features, Functional and Lambda**
- **Memory Management and Garbage Collection**
- **Streams**
- **Exception Handling**
- **Collections Framework**
- **Map Framework**
- **Reflection and Annotations**

JRE / JVM / JDK

Java is a platform Independent, Object Oriented and Portable (WORA - Write once, Run anywhere) programming language.

Java is called a platform Independent language because Java code is compiled into a universal bytecode that can run on any device with a Java virtual machine (JVM).



JVM (Java Virtual Machine)

Java -> Compiler -> Bytecode -> **JVM** -> Machine Code -> CPU -> Output Program

=> JVM is the core of Java programming, It provides Java portability and platform Independence.

When Java code is compiled we don't directly get Machine code, instead we get Bytecode. The bytecode can be run by the JVM which will convert it to Machine code.

Points to consider: Java program is platform independent, however JVM itself is not, i.e. JVM is platform dependent, so for example a Mac OS will need a different JVM version than Windows OS.

JVM has a JIT (just in time) compiler, which will take the bytecode as

input and convert it to the machine code.

Bytecode -> **JVM** -> Machine Code

Once we have a bytecode we can run it on any JVM, so for example we write and compile a Java program on Windows and get the bytecode, now we can take this bytecode and run it on any other system, like Linux or Mac having a JVM. The bytecode is universal hence any JVM will be able to understand it and convert it to Machine Code, this is why Java is called a portable and platform independent language.

Java Program		Bytecode
filename.java	-> Compile ->	filename.class

So for example we have a Student class in a file Student.java, when we compile this file (javac Student.java) we get the bytecode Student.class, Now we can take this bytecode and run it on any machine with a JVM.

Once the compiler has generated the bytecode, JVM loads it and runs the Java Bytecode by calling the main method to start running the program.

Think of the JVM as an OS, it is a VM after all.

JRE (Java Runtime Environment)

JRE is a software layer that contains class libraries, JVM and other resources that a Java program requires to run. JRE is actually responsible for running the bytecode. This is because JRE contains JVM, which converts the bytecode to Machine Code and hands it off to CPU, however the code has other dependencies, libraries etc. which first need to be resolved. JRE consists of components which will take care of resolving the dependencies and finally once done, it will hand over the bytecode to the JVM which comes packaged in as a part of JRE.

JDK and JRE interact with one another to create a suitable environment to run the Java application, the JRE runtime environment has the following attributes.

=> ClassLoader: Java ClassLoader dynamically loads all the classes and libraries needed to run the Java program, Java classes are loaded into memory only when they are needed, i.e. classLoader performs loading on demand.

=> Bytecode Verifier: The Bytecode Verifier verifies the integrity and accuracy of Java code before it passes it to the interpreter. This involves checking if the code violates system integrity or access rights.

=> Interpreter: After the bytecode successfully loads and is verified, the Java interpreter creates an instance of the JVM that allows the Java program to be run natively on the underlying machine.

As mentioned before JRE consists of Class libraries, libraries are basically collections of pre-written code for developer convenience, for example java.Math, java.util are libraries, it is the responsibility of JRE to resolve these dependencies.

So given any bytecode if we have a JRE then we can run it. The JVM comes as a part of the JRE.

Note; the term libraries is broad, and can be segmented as Utility (java.util) libraries, language (java.lang) libraries and Integration Libraries (like JDBC).

JDK (Java Development Kit)

JDK actually contains the actual programming language, JRE, Compiler (javac), Debugger and other developer tools.

Compiler compiles the java program (.java) to Bytecode (.class)

JDK, JRE and JVM are all platform dependent, however the java code itself is platform independent, specifically the bytecode generated can be run on any JVM across systems.

=> One file can have only one public class.

=> The main method must be in a public class.

Why ?

JVM will invoke the main method, hence it needs access to this method. The main method is defined as static so that JVM can just invoke it using the class name without having to create an object of the class. Let's say the main method is defined in a class called Driver, hence the JVM will look for Driver.main, this mandates for the class Driver to be a public class, otherwise JVM won't be able to find the main method

One line answer: We need the class to be accessible outside its package, so that the JVM can call main. (1)

=> main is the entry point to the program, JVM reads the bytecode and will call the main method to start running the program.

=> Name of the public class should be the same as the file name.

Why ?

A file can have hundreds of classes, suppose we have multiple public classes then how will JVM detect which one of these hundreds of classes has the main method?

Hence the name of the public class is mandated to be the same as the file name. (2)

Why can a file have only a single public class?

Refer (1) and (2) above.

Threading

The JVM can be conceptualized as an OS, it consists of memory regions like: Heap, Stack, Data Segment, Code Segment as well as Registers and Program Counter.

What happens internally when we execute java Main (after javac Main.java) -

1. A new process is created, to run the program
2. A new JVM instance is created, and assigned to that process. Any program needs memory to execute - heap, stack etc. JVM provides those. The JVM instance contains all the memory regions mentioned above.

When we run java Main, a new process is created and a new JVM instance will be allocated to that process.

How much heap memory does a process get? Process gets as much heap memory as there is in the JVM instance allocated for that process.

We can configure the heap memory at the time of process creation, by using the command:
java -Xms256m -Xmx2g Main

-Xms<size>: Sets the initial heap size, here it is 256MB
-Xmx<size>: Max heap size the process can get, here it is 2GB. If it tries to allocate more memory, then the "OutOfMemory" error will occur.

Hence we can configure the size of heap memory in the JVM instance by using the Xms and Xmx args.

When we ran java Main, the new process got created and a JVM instance was allocated for that process. The JVM instance will convert the bytecode to Machine Code by using Interpreter or JIT compiler.

Information regarding the different segments in the JVM:

1. Code Segment: Contains the compiled bytecode (Machine Code) of the Java program. This segment is read only.

2. Data Segment: Stores the global and static variables. This segment is shared by all the threads. Since the same data is accessible to multiple threads hence we need proper synchronization.

3. Heap Segment: Used for dynamic memory allocation, objects created

at runtime by using the “new” keyword. The Heap segment is shared among all the threads, again synchronization is needed to prevent race conditions.

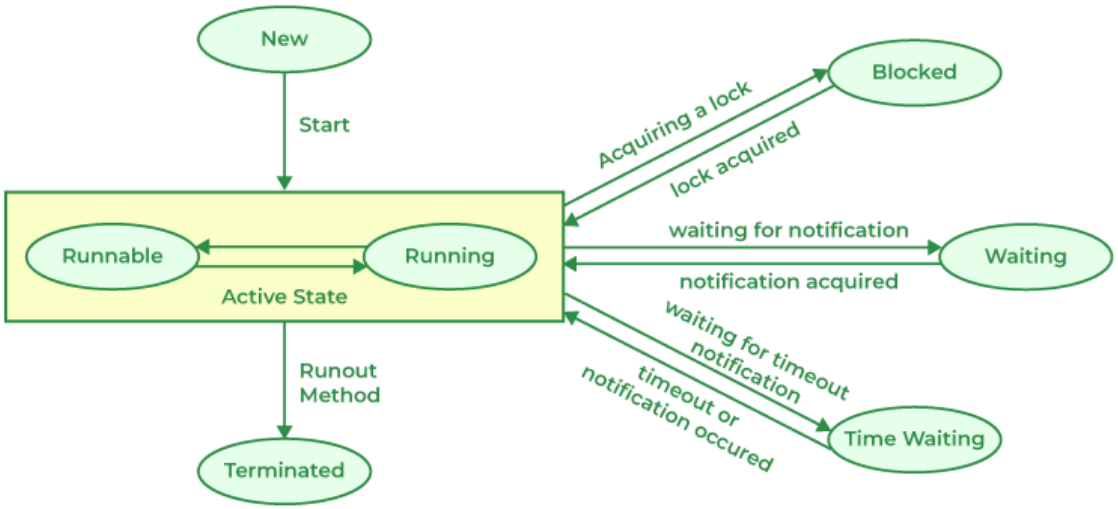
4. Stack: Each thread has its own stack segment. It manages method calls and local variables.

5. Registers: Each thread has its own private registers, registers are used for context switching, also used by the JIT compiler for converting bytecode to native machine code.

6. Program Counter: Points to the instruction which is getting executed. Gets incremented after successful execution of the instruction.
The PC points to an address in the code segment.

How are the different threads scheduled: JVM Scheduler is used for scheduling the threads, however the JVM Scheduler is itself managed by the OS Scheduler.

Thread Lifecycle:



1. New - Thread has been created but not yet started, it’s just an object in memory.

2. Runnable - Thread is ready to run, waiting for CPU time.

3. Running - Thread is running on the CPU, i.e. executing its instructions.

4. Blocked - Thread moves to the blocked state when:
    a. It performs I/O, like reading from disk, or a file
    b. Waiting to acquire a lock

   Whenever the thread goes to the blocked state it will release all of its monitor locks.

5. Waiting - Thread goes into the waiting state, when it calls the wait() method. It goes back to runnable when we call notify() or notifyAll() methods. Whenever the thread moves to the waiting state it will release all of its monitor locks.

6. Timed Waiting - Thread moves to the timed waiting state when it calls join or sleep, it does not release the monitor locks when it moves to this state.

7. Terminated - Life of the thread is completed, it cannot start again.

Monitor locks ensure that only one thread goes inside the critical section, no matter if there are multiple critical sections spread across different methods. This is because Monitor lock is a property of the object.

Monitor Lock: Java has the concept of synchronized blocks / methods, only one thread can execute the code in this section at a time. Synchronization is built around an intrinsic lock, also called monitor lock. Every object has a monitor lock (or intrinsic lock) associated with it. Before accessing a block of code marked as synchronized the thread must acquire the monitor lock. Since the monitor lock is the property of an object, hence only one thread can acquire it at a time. This implies that even if there are multiple synchronized blocks, only one of them can be executed at a time. For example T1 wants to execute synchronized method M1, and T2 wants to execute the synchronized method M2.

Since these are 2 different methods, hence one might expect them to get executed by the respective threads simultaneously. However that is not possible as to execute a synchronized method, the thread needs to acquire the monitor lock and there is only monitor lock associated with the object.

Once the thread acquires a monitor lock, it can execute the critical section code. Once it is done, it'll release the monitor lock. If a thread attempts to acquire the monitor lock while it is being held by another thread then the requesting thread will block (moved to blocked state).

Remember:
When the thread moves to blocked or Waiting state its monitor locks are released.
However if the thread moves to the Time Waiting state, the monitor locks it has acquired are not released.

Thread Methods

A note about deprecated thread methods

Stop: The stop method terminates the thread abruptly, no locks are released and no resource cleanup happens.
Suspend: Similar to wait, however no locks are released, which could lead to deadlocks.
Resume: The resume method is similar to notify / notifyAll, however resume works with the suspend method. Since suspend itself is deprecated hence there is no point in maintaining the resume method.

Stop and Suspend methods can lead to deadlocks and hence have been deprecated.

Thread Join

The join method is used by a thread to wait for another thread to complete execution, for example the parent thread might want to wait for the child thread to finish before it carries on its own execution.

Daemon Thread

There are 2 types of threads:

1. User Thread
2. Daemon Thread

All threads created normally by extending the Thread class or by implementing the Runnable interface are considered User Threads. To

create a daemon thread use the setDaemon(true) method on the existing user thread.

The daemon thread will die once all the user threads die (i.e. complete execution), this is different from User Threads which continue executing even if all the other User Threads (including the parent thread) die.

Daemon Thread runs continuously in the background doing some useful repetitive work.

Uses of Daemon Thread:
=> Garbage Collection - JVM runs the Garbage Collector on a daemon thread, hence the life of the GC is tied to the life of the program.
=> Auto-Save

Locks and Semaphores

Types of Locks (Custom Locks)

=> Reentrant Locks: A class can have multiple critical sections, as mentioned before an object of the class has a monitor lock which controls access to the blocks of code labeled 'synchronized'. However, Monitor locks are associated with the object, so multiple threads working with different objects can still simultaneously access the same piece of critical section code in the class.

For example, suppose a class has a method m1(), marked as synchronized, and we have 3 threads each working with its own object of the class.
T1 -> obj1, T2 -> obj2, T3 -> obj3
If we solely rely on monitor locks then all threads can execute the method m1 on their respective objects simultaneously. If this is not desired and we want only one thread to access the method at a time, regardless of which object they are coming from then we can use Reentrant Locks.

=> ReadWrite Locks: Encapsulates the concept of Shared Locks and Exclusive Locks. (Not a property of the object). ReadWrite locks are used in read-heavy applications.

Note about Optimistic locking - In case of Optimistic locking no locks are actually acquired, optimistic locking is a technique which relies on versioning rather than actual locking like Mutex locks for example.

=> Stamped Locks: Stamped locks provide the functionality of both ReadWrite locks and Optimistic locks.

Again, ReadWrite locks are based on the concept of Shared (S) and Exclusive (X) locks, while Optimistic locking isn't really a locking technique (since no locks are actually acquired or released) and it instead works on the concept of versions.

Stamped Locks like ReadWrite and Reentrant locks are not properties of the object and hence can be used to control access to critical sections across objects.

=> Semaphore Locks: Semaphore locks allow multiple threads to enter the critical section simultaneously, this number can be configured. Semaphores are used to control access to resources with multiple instances.

Condition Variables: wait() and notify() / notifyAll() work with monitor locks (i.e. blocks of code marked synchronized), however these methods don't work with other kinds of locks like Reentrant locks, ReadWrite locks or StampedLocks.

For this reason we use await() and signal().

await(): Similar in functionality to wait()
signal() / signalAll(): Similar in functionality to notify() / notifyAll()

Lock Free Concurrency

There are two ways to achieve concurrency control:
=> Using lock based techniques: Monitor Locks, Reentrant Locks, ReadWrite locks, Stamped locks, Semaphores etc.
=> Lock Free Concurrency - Compare and Swap.

Compare and Swap (CAS)

Compare and Swap is a low level instruction supported by the CPU, it is atomic in nature.

Compare and Swap involves the following 3 parameters:

1. Memory Location - Location where the variable is stored.
2. Expected Value - Value which should be present at that location in the memory.
3. New Value - The value which needs to be written to the memory location. The value is only written if the expected value matches the current value.

If the current value doesn’t match the expected value then that means the memory location has been updated by another thread, and if that is the case the thread in question will not be able to update the memory location.

We can prevent the ABA problem by adding a version ID or timestamp.

Thread Pool

Thread Pool is a collection of threads, called workers which are available to perform the submitter tasks.
=> When a task arrives, it will be assigned to one of the threads from the pool, and the thread will be removed from the pool.
=> Once the task completes the thread will be freed and will move back to the pool, waiting for other tasks.
=> If a task is submitted when all the threads are busy, then the task will be added to a queue associated with the thread pool.

Advantages of Thread Pool:
=> Thread creation time is reduced due to reuse of threads.
=> Overhead of managing the thread life cycle is removed.
=> Improves performance, context switch time is reduced.

In Java, we use the ThreadPoolExecutor to create a Thread Pool.

The threads are part of an auto scaling group, we specify the desired thread count and the maximum thread count when initializing the thread pool, the thread pool can scale based on the tasks submitted.

Flow:
We have task1, task2, ….. task11
And a thread pool with desired thread count = 3, and maximum thread count = 5.
Suppose the queue has a size of 3.

task1 comes in and is assigned to t1
task2 comes in and is assigned to t2
task3 comes in and is assigned to t3
task4 comes in, since all threads are occupied it is added to the queue.
task5 comes in, since all threads are occupied it is added to the queue.
task6 comes in, since all threads are occupied it is added to the queue.
task7 comes in, since all threads are occupied, it tries to go to the queue, however the queue is full as well. Now the thread pool will check if it is possible to create more threads based on the value of Maximum Thread Count, since Maximum thread count is 5 and currently only 3 threads are in use, hence 1 more thread is added to the pool and task7 is assigned to this thread.
task8 comes in, since all threads are occupied, it tries to go to the queue, however the queue is full as well. Now the thread pool will check if it is possible to create more threads based on the value of Maximum Thread

Count, since Maximum thread count is 5 and currently only 4 threads are in use, hence 1 more thread is added to the pool and task8 is assigned to this thread.
Task9 comes in, all threads are occupied, the queue is occupied and no additional threads can be created since the ASG is at its max capacity hence the task is dropped.

The excess threads created this way (beyond the core thread count specified), will remain in the pool.

=> How to remove unused threads from the pool?
The unused threads, including the excess threads created to handle peak traffic can be removed from the thread pool after a specified time interval. This time interval is called 'KeepAliveTime'. If we set the attribute allowCoreThreadTimeOut of the ThreadExecutorPool, then any thread which hasn't been in use for KeepAliveTime units of time will be removed from the pool. Note this does include the threads from the original core pool, for example a thread pool created with a core thread count of 3 but one of the threads hasn't been used for a very long time (specified by KeepAliveTime), in such a case the third thread can be removed.

=> The tasks are stored in a Blocking Queue, generally a bounded blocking queue (bounded = fixed capacity).

Basics

An object consists of two components:

1. Properties or state
2. Behavior or functionality

Object Oriented programming gives importance to the data, and provides data hiding, on the other hand procedural programming languages give more importance to functions.

A Class is a blueprint, from which objects are created, a class has data variables (attributes) and methods which operate on that data.

Pillars of Object Oriented Programming

- Data Abstraction
- Data Encapsulation
- Inheritance
- Polymorphism

Data Abstraction

Real World Example: When we press the brake pedal of the car, its speed decreases. But how does that actually happen, this aspect (complexity of implementation) is abstracted from us.

Abstraction is the technique of hiding the complexities of internal implementation from the user (or client) and exposes only the essential features or information.

=> Abstraction can be achieved via interfaces and abstract classes.

Advantages:
=> Abstraction increases security and confidentiality, abstraction helps to provide a clear and simple interface to the user.

=> Ease of use: User doesn't have to deal with how the methods are actually implemented, hence the user doesn't have to deal with the intricacies of the method (like Return Codes / Error Handling), abstraction simplifies the client code.

=> We can perform changes to the internal system without affecting the

end user, i.e. the user doesn't need to be aware of these changes and can keep using the same abstract method in the same manner as before.

=> There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

Data Encapsulation

Real World Example: Take the example of a capsule, where all the medicine (or data) is encapsulated within the gelatine body of the capsule. The capsule is just a cover outside the data, which is protecting it. Similarly in case of data encapsulation, it is the class which wraps the data together inside it and protects it using the access modifiers.

Encapsulation, also known as data hiding, bundles the data and the code working on that data as a single unit, and the class has full control over them.

How to achieve encapsulation:
=> Declare the variables / attributes of a class as private
=> Provide public getters and setters to fetch / modify these variables.

Encapsulation provides data hiding as the data variables are no longer accessible to the client directly, i.e. they cannot directly fetch / modify the data. If they want to access the data they need to use the public getters and setters provided by the class.

Data Encapsulation allows us to have full control over the data, we can put all the logically related data into the same class, so that the class has full control over these variables, i.e. this class has ownership over this data, and any operations on that data will be performed by this class only.

This is very different from procedural programming where data flows freely and there is no concept of ownership, say we have some data which is being passed b/w multiple functions, here any function in the chain can modify it, there is no concept of ownership here, and hence no control over what the data variables are being modified to.

=> With data encapsulation the class which encapsulates the data takes the ownership and has full control over the variables, if any other class wants to fetch / modify the data, then it'll need to use the public getters / setters provided by the encapsulating class (i.e. it acts as a central authority for any access to the underlying data).

=> The encapsulating class has control as to what data the other classes can access and how (Private data, with public getters / setters, public data (other classes can access directly using obj.var) or private data with no public getters / setters, for internal data which we don't want other classes to access at all).

=> Further the encapsulating class has control as to how the data should be modified as it has the ownership over this data. We certainly don't want a situation where the age attribute of a Person class is being set to -1.

Advantages:
=> Loosely coupled code: Modification of data by one object of a class does not affect the other objects of the class.
=> Better access control and security.

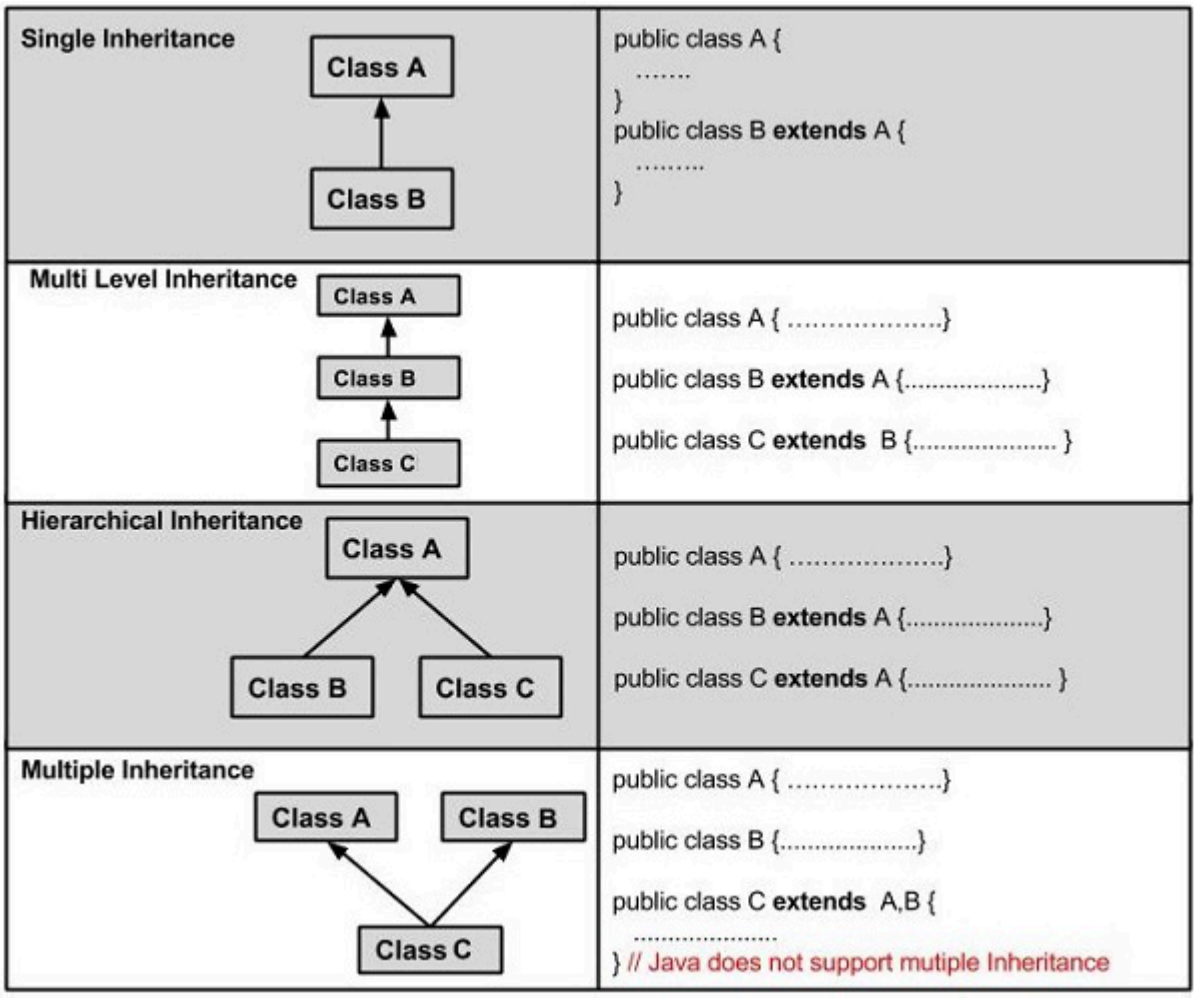
Inheritance

=> Capability of a class (child class or subclass) to inherit from another class (parent class or superclass)
=> The child class inherits both methods and variables.
=> Can be achieved by using the extends keyword or through interfaces (using the implements keyword).

Types of Inheritance:

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance (Can be done only using interfaces, with classes we can't as we'll encounter the Diamond problem)

Advantages Of Inheritance
=> Code Reusability
=> We can achieve polymorphism using Inheritance



Polymorphism

'Poly' means many and 'morphism' means forms.
=> Polymorphism is the ability of a method to behave differently in different situations, i.e. a method has the ability to take on many forms.

Types of Polymorphism

=> Compile Time / Static Polymorphism / Method Overloading
=> Run Time / Dynamic Polymorphism / Method Overriding

What is Method Overloading: We can define multiple methods with the same name, differing by the arguments they take as input, i.e. the type of arguments and the number of arguments.
Note: Methods cannot be overloaded solely based on return type.
Why? Because during compile time the return type is not checked.

Why Method Overloading is known as compile time polymorphism?
This is because at compile time itself Java knows which version of the method to call, based on the arguments passed to the function call.

What is Method Overriding: The child class can override methods of the parent class. The method signatures (method name, arguments) must exactly match. Method Overriding is achieved via Inheritance.

[What about return type?
Before Java 5.0, when you override a method, both parameters and return type must match exactly. Java 5.0 introduced a new feature called covariant return type. You can override a method with the same signature but return a subclass of the object returned.

In other words, a method in a subclass can return an object whose type is a subclass of the type returned by the method with the same signature in the superclass.]

However, obviously the internal implementations differ, which allows the child class to override methods of the parent class, or in other words override the implementation of methods from the parent class.

Why is Method Overriding known as Runtime Polymorphism?
This is because only at Run time does java know which version of the method to call. One from the child class or the parent class. Which method to call, i.e. the one in the parent or the one in the child class will be determined at run time depending on the object which we have created. Hence Method Overriding is also called dynamic polymorphism.

Again the overridden methods must have exactly the same signature.

Overloading is done within the same class, while Overriding is done across different classes via Inheritance.

Object Relationships

Is-a Relationship

=> Achieved through Inheritance
=> Example Dog is-a Animal
=> Inheritance forms an is-a relationship b/w the child and parent class.

Has-a Relationship

=> Whenever an object is used in another class it's called a Has-a relationship.

```
Class B {
        // something
}
```

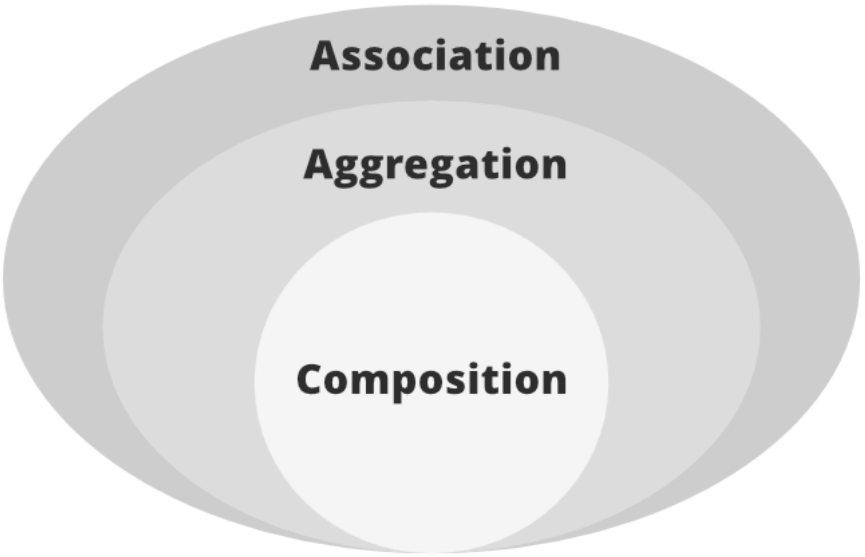
```
Class A {
	int data;
	B box;
}
```

As can be seen, class A has an object of class B, so there is a has-a relationship between class A and class B.

The has-a relationship can be: one-to-one, one-to-many, many-to-many.

one-to-many: School has students
one-to-one: A car has an engine
many-to-many: A student can take multiple courses, and each course has multiple students

Types of Has-a relationship:



=> Association: Relationship between two different classes established through their objects.

Types of Association:

=> Aggregation (Weak Relationship):
Both the objects can survive independently, i.e. ending one entity will not end the other entity. For example the relationship b/w School and Student.

=> Composition (Strong Relationship):
Ending of one object will end another object, in composition both

entities are highly dependent on each other. For example the relationship between House and Rooms, if the house is destroyed then obviously the rooms cannot exist independently. Composition represents a part-of relationship.

Types of Classes

CONCRETE CLASSES

Concrete classes are those classes of which we can create an instance by using the new keyword.

=> All methods in a concrete class must have implementations.
=> A concrete class can extend an abstract class or implement an interface.
=> The class access modifier can be public or package private (no explicit access modifier provided, default) - If no access modifier is provided then the class is package private, i.e. it can only be accessed within its package and not outside it. A public access modifier allows the class to be accessed from anywhere.

```
public class ConcreteClass {
    String name;
    int age;

    public void greet() {
        System.out.println("Hello: " + name + ", you are: " +
                           age + " years old");
    }

    public int getAge() {
        return age;
    }
}
```

ABSTRACT CLASSES

Abstraction refers to the technique of only exposing the important features to the users, and hiding the internal implementation details. In other words users only interact with the methods they require and don't need to care about the internal implementation complexity.

How to achieve abstraction:
- Interface
- Abstract Class

An abstract class allows us to define abstract methods (i.e. methods

without any implementation), these methods will need to be implemented by the children (or sub classes) i.e. the class which extends the aforementioned abstract class.

Note:
Interfaces offer 100% Abstraction
Abstract classes offer 0 - 100% Abstraction (we can have all the methods as abstract - 100% abstraction or have none of them as abstract - 0% abstraction).

As can be seen the major difference b/w Interfaces and abstract classes is that abstract classes do allow us to provide implementations for the methods as well as the option to define them as abstract. This is useful because in many situations we might need a combination of abstract methods as well as implemented methods.

Abstract methods can be public, protected and default (package private) however abstract methods can't be private.

=> Abstract classes can have constructors.

```
public abstract class AbstractClass {
    int mileage;

    public AbstractClass(int mileage) {
        this.mileage = mileage;
    }

    public abstract void pressBrake();

    public abstract void accelerate();

    public int getNumberOfWheels() {
        return 4;
    }
}
```

=> It is possible for an abstract class to be subclassed by another abstract class, which can possibly implement some of the abstract methods or add more abstract methods.

As mentioned before a concrete class can extend an abstract class, it will need to provide implementations for all of the abstract methods in the abstract class.

=> We cannot create an instance of an abstract class, because some of the methods don't have an implementation.

However we can use the reference of the parent class.

SUPERCLASS AND SUBCLASS

The parent class is called the super class, while the child class is called the sub class.

In Java in the absence of any other explicit superclass, every class is implicitly a subclass of the Object class. In other words the Object class is the parent of all classes.

What are the methods provided by the Object Class:- toString(), clone(), notify(), equals(), wait() etc.

Child Class can be stored in the parent reference.

```
public class ChildOfObject {
    public static void main(String[] args) {
        ChildOfObject childOfObject = new ChildOfObject();

        Object obj1 = Integer.valueOf(24);
        Object obj2 = new String("hippo");

        System.out.println(obj1.getClass());
        System.out.println(obj2.getClass());
    }
}
```

NESTED CLASSES

A class within another class is called a Nested class.

When to use a nested class?
If we know a class (A) will be used by only one other class, say class (B) then instead of creating a separate new file A.java for class A, we can create a nested class inside B itself.
This also helps to group logically related classes together.

=> The Scope of the nested class is the same as the parent class.

Types of Nested Class:
- Static Nested Class
- Non Static Nested Class (also called Inner class)
    - Local Inner Class
    - Member Inner Class
    - Anonymous Inner Class

Static Nested Class

=> As the class is static, hence it does not have access to non-static methods and variables of the outer class.
=> We can initialize an object of the nested class without initializing an object of the outer class.
=> The Nested class can be public, package-private (default, no explicit declaration), private or protected.

```
class OuterClass {
    int instanceVariable = 10;
    static int classVariable = 20;

    static class NestedClass {
        public void print() {
            System.out.println(classVariable);
        }
    }
}

public class StaticNestedClass {
    public static void main(String[] args) {
        OuterClass.NestedClass innerClass = new
OuterClass.NestedClass();
        innerClass.print();
    }
}
```

Note: A class can have only public or package-private (no explicit declaration) access modifiers, however nested classes can be private, public, protected or package private. If we define a nested class as private then we cannot create its object outside the Outer Class.

Non Static Nested Class or Inner Class

It has access to all the instance and class variables /methods of the outer class. As the nested class is not static, we need an object of the parent class to access the inner class.

Types of Inner Class

=> Member Inner Class

```
class OuterClass {
    static int classVariable = 20;
    int instanceVariable = 30;

     class InnerClass {
        public void print() {
            System.out.println(instanceVariable + classVariable);
        }
     }
}

public class MemberInnerClass {
    public static void main(String[] args) {
        OuterClass.InnerClass innerClass =
                            new OuterClass().new InnerClass();
        innerClass.print();
    }
}
```

The inner class can be public, private, protected, default (package-private).

=> Local Inner Class

Local Inner classes are defined within any block, for example for loop, while loop, if condition, methods etc.

It cannot be declared as public / private or protected, only default access modifier is allowed.

As the class is defined entirely within a block, hence we cannot instantiate an object of the class outside that particular block. For example we define our Local Inner Class inside a method, the class cannot be accessed

outside the function's scope. Hence the life of the class is tied to the scope of the method, once the method goes out of scope, the local inner class cannot be accessed. In summary it can only be accessed and instantiated within the method. (or more generally the block wherein it is defined).

The Local Inner class can access the methods / variables defined in the outer class, as well as the local data inside the block where it is defined.

```
public class OuterClass {
    private int temperature = 1080;

    public void getTemprature() {
        System.out.println("Today is " + temperature + " degree
Celsius");
    }

    public void weatherStation() {
        String personName = "Park";
        class LocalInnerClass {
            private int data = 20;

            public void sayHello() {
                System.out.println("Hello: " + data + ", " +
personName);
                getTemperature();
            }
        }

        LocalInnerClass localInnerClass = new LocalInnerClass();
        localInnerClass.sayHello();
    }
}
```

=> Anonymous Inner Class

An inner class without a name is called an anonymous class.

```
abstract class Car {
    public abstract void accelerate();
}
```

```
public class AnonymousClass {
    public static void main(String[] args) {
        Car carObj = new Car() {
            public void accelerate() {
                System.err.println("speed is increased by 10");
            }
        };
        carObj.accelerate();
    }
}
```

Here it seems we are creating an object of an Abstract class, however that isn't possible, in reality there are 2 steps happening in the background.

=> A Subclass is created (extends the abstract class), the name of this subclass is decided by the compiler.
=> Creates an instance of the subclass and assigns its reference to the variable carObj.

GENERIC CLASSES

Generics were introduced in Java 5 to provide type safety to collections. There are classes like ArrayList for example which work with multiple types of objects, for example we can define a list of Integers, a list of String or a list of User-defined objects. So, we need to design a class which works with multiple types of objects. One possible option is to use a class like this:

```
class Print {
    Object value;
    public void setValue(Object value) {
        this.value = value;
    }
    public Object getValue() {
        return value;
    }
}
```

The problem in this case is when we call the getValue() method we don’t know what kind of object is returned, hence we don’t know what methods

can be called on it. To overcome this we'll need to use multiple if conditions, checking if the object is an instance of one type or another, and then cast the object into that particular type and eventually call the methods on it.

An Alternative is to use Generic Classes

```
class Print<T> {
    T value;

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

public class GenericClass {
    public static void main(String[] args) {
        Print<Integer> print = new Print<>();
        print.setValue(50);
        Integer value = print.getValue();

        System.out.println(value);
    }
}
```

Where T is the type parameter.
T must be a Reference type (i.e. it cannot be a primitive type like int, char etc.). So, Generics only work with Reference Types.

With Generic classes we don't need to use instanceOf or cast the returned objects, we can use them as is, as Java will know what the type of the object is.
<> is known as diamond tag.

Generic Classes and Inheritance.

=> A non-generic class can inherit from a generic class.

```
class Print<T> {
```

```
    T value;

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

class ChildPrint extends Print<String> {
    public void getChildName() {
        System.out.println("Keane");
    }
}
```

Note we cannot extend directly from Print, as ChildPrint is a non-generic class, hence it cannot extend a Raw type.

=> A generic class can inherit from another generic class.

```
class Print<T> {
    T value;

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

class ChildParentGeneric<T> extends Print<T> {

}
```

Of Course a class can have multiple generic types.

```
public class Pair<K,V> {
    K key;
```

```
    V value;

    public void put(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) {
        Pair<String, Integer> pair = new Pair<String,Integer>();
        pair.put("cart", 54);

        System.out.println(pair.getKey());
        System.out.println(pair.getValue());
    }
}
```

Generic Methods

If we only want to make a few methods generic and not the entire class then we can do so using generic methods.

```
class Print {
    public <T> void setValue(T value) {

    }
}
public class GenericMethods {
    public static void main(String[] args) {
        Print print = new Print();
        print.setValue("Car Bus");
    }
```

```
}
```

=> Scope of the type parameter is limited to the method only.

Raw Type

When we create an object of the Generic class like this:
Print<Integer> print = new Print<>();
This is a parameterized type object.

The other alternative is to create a Raw Type object.
Print print = new Print();

internally the compiler translates it to:
Print<Object> print = new Print<>()

Bounded Generics

Sometimes we want to restrict the types that can be used as the type parameter.
When we create a generic class, we can pass objects of any type to it, for example Integer, String, user-define etc. However what if we want to restrict our generic class to only accept certain objects while rejecting the rest. For example our generic class should only accept Number as the type parameter, and not other object types like String. How can we enforce this?

This is done by using Bounded Generics. Types of Bounded Generics:

=> **Upper Bound Generics**

Using upper bound generics we can restrict the generic class to only accept the Number and its subclasses (child classes) as the type parameter. Nothing else (for example String) will not be accepted.

Syntax: class Generics <T extends Number>

Using this we enforce that the type parameter is either a Number or a subclass of the Number class (Integer, Double, BigInteger etc.). In this case we call the Number class as the upper bound class, as we only accept Number and below as accepted type parameters.
If we pass any other object (say a string) we will get a compile time error.

We mentioned Number to be the upper bound class, but it doesn't actually have to be a class, it can be an interface as well.

```java
class Printer<T extends Number> {
    T value;

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

    public T getValue() {
        return value;
    }
}

public class UpperBound {
    public static void main(String[] args) {
        Printer<Integer> print = new Printer<>();
        print.setValue(8967);
        System.out.println(print.getValue());
    }
}
```

If we pass another type (String) as the type parameter, we will get a compilation error.

Note: In java a class can only extend from one super class, i.e. it cannot inherit from multiple classes.

So for example class ChildClass extends ParentClass1, ParentClass2 is not possible.
Why?
Suppose both classes P1 and P2 have a method m1, and C1 extends both of these classes. Now, when we create an object of the child class and call child.m1(), then which version of m1 should be called? One implemented by P1 or one by implemented P2, to prevent such conflicts, java doesn't allow a child class to inherit from multiple parent classes.
However a single class can implement multiple interfaces.

=> **Multi Bound Generics:**

It is basically a stricter form of Upper bound generics, the type parameter in addition to being a subclass of the specified class (first argument) [or being the class itself], also needs to implement all of the listed interfaces.

```
class Print<T extends Number & Intf1 & Intf2> {
    T value;

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

public class MultiBoundClass {
    public static void main(String[] args) {
        Print<Animal> print = new Print<>();
    }
}
```

```
public class Animal extends Number implements Intf1, Intf2 {

}
```

<T extends ParentClass & Intf1 & Intf2>
=> The type parameter T must be a child class (or subclass) of the ParentClass, and it should implement Intf1 and Intf2, if we try to pass any other object to the generic class it'll generate a Compilation error.

Wildcards

Suppose we have a vehicle class, which the bus class inherits from, hence we can use a Vehicle reference to refer to the child object.

Vehicle vehicle = new Bus();

Hence, we can swap the child (Bus) reference with the Parent (Vehicle) reference.

Wildcards enable us to specify a type that is a subtype of another specified type.

List<Vehicles> vehicleList = new ArrayList<>();
List<Bus> busList = new ArrayList<>();

public void setVehicleList(List<Vehicles> vehicleList) {

}

Considering the above method, we can pass vehicleList to it without any compilation errors, however busList cannot be passed to it. Even though Bus is a child of Vehicle, busList is not a child of vehicleList, hence we cannot pass List<Bus> to a method which is expecting List<Vehicles> even though Bus is a child of Vehicle. In such scenarios we make use of Wildcards.

=> Upper Bound Wildcard: <? extends UpperBoundClassName>

We can modify the above function as follows:

Public void setVehicleList(List<? extends Vehicle> vehicleList);

This method will accept List<Vehicle> as well as List<Obj> where Obj is any subclass of Vehicle, hence List<Bus> will be accepted, since Bus is a subclass of Vehicle.

So anything Vehicle and below will work in our new method specification.

=> Lower Bound Wildcard: <? super LowerBoundClassName>

This wildcard will accept the specified class and all of its super classes, i.e. it will accept the specified class and anything above it.

Code Examples:

```
public class Bus extends Vehicle {

}

public class Print {
    public void setPrintValues(List<? extends Vehicle> vehicles) {

    }

    public void setPrintValues2(List<? super Vehicle> vehicles) {

    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Print print = new Print();
        List<Vehicle> vehicle List = new ArrayList<>();
        List<Bus> busList = new ArrayList<>();

        print.setPrintValues(busList);

        List<Object> objectList = new ArrayList<>();
        print.setPrintValues2(objectList);
    }
}
```

=> Unbounded Wildcard: <?>

We can use this if we aren't sure what the type of the object will be, or if our method performs only very generic functions, like using the methods defined in the Object class.

Differences between Generics and Wildcards

Generics and wildcards do serve pretty much the same purpose, however Generics offer better type safety. For example we have a method which takes a list and an object, and adds that object to the list. Using Type parameter we can write it as:
public <T> addToList(List<T> list, T objToAdd)

With Wildcard, we'll write it as:
public addToList(List<?> list, Object objToAdd)

=> Important observation: The first approach enforces both the types to be the same, while there is no restriction in the second case, so we very easily could be adding a String to a List<Integer>, however approach 1 offers type safely and will detect such an issue at compile time.

Wildcards and Generics both offer a way to filter what objects are acceptable by the class or methods, however Wildcards does offer the additional method of Lower Bound Wildcards, while Generics offer Multi Bound Generics

However if we are using a type parameter only once, it essentially becomes a wildcard pattern, here it does not make sense to use a type parameter.

For example, we have a function which takes a list of objects of any kind, say String, integer, float etc. and just returns the size of the list.

Using generics:

```
public <T> int getSize(List<T> list) {
	return list.size();
}
```

Using Wildcards:

```
public int getSize(List<?> list) {
	return list.size();
}
```

In such a situation using a wildcard makes sense, since there is no use whatsoever of the type parameter, this method doesn't care about the type of the object it just returns the size of the list, and in such conditions the wildcard parameter helps in avoiding unnecessary code bloat.

Primitive / Non-Primitive

Java is a statically typed language and strongly typed language.

Statically typed programming languages, for example C, C++, Java do type checking at compile time, i.e. the type of the variables is known at compile time.

Dynamically typed programming languages, for example Python, Perl, Javascript do type checking at runtime

Types of Variables:
Primitive: int, short, byte, long, float, double, char, boolean

Types of variables in a class:

1. Member Variable: Also called instance variables, a default value of 0 is assigned to these variables. Member variables are associated with the objects, Each object of the class will have its own copy of the Member Variables.

2. Local (or Method Local variables): These variables are defined inside a method, and their scope is restricted to this method itself.

3. Class Variables (or Static Variable): All the objects share the same copy of the Class Variables, in other words only one copy of the static variables exist. The class variables are accessed via the class name and not via an object.

4. Constructor Parameters / Method Parameters

Primitive Data types are predefined in Java, Non-primitive types (or Reference types) are created by the programmer and are not pre-defined by Java (String is an exception, as even though it is defined by Java it is still considered as a non-primitive or Reference type).

In Java everything is passed by value, there is no concept of Pass by reference, because there is no concept of pointers in Java.

Reference data types:

Non primitive data types are also called Reference data types because they refer to objects.
Reference data types hold the reference to the actual objects in the heap, in other words it holds the reference to the actual memory location where

the object is allocated.

Example of reference data types: Class, Strings, Interface, Arrays etc.

Why is String considered a Reference data type?

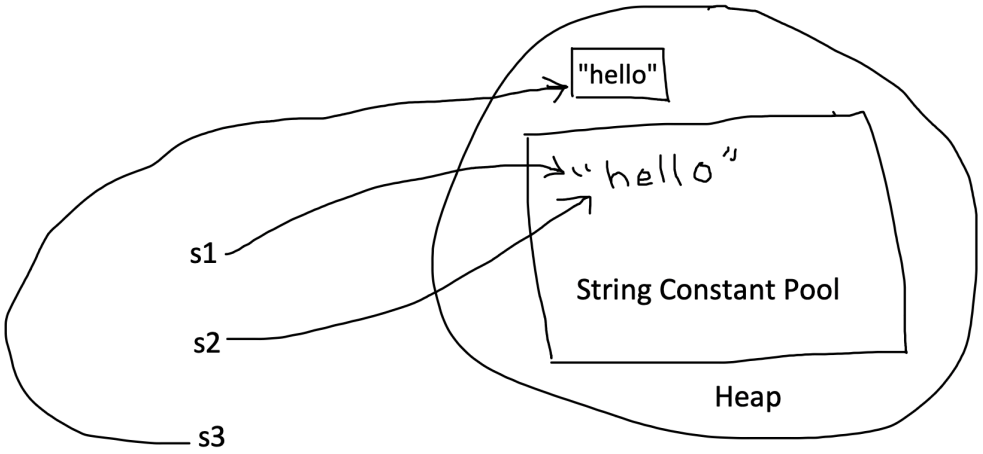
Consider the following instruction,
String s1 = "hello"
Here "hello" is known as a string literal. String literals are stored in the String Constant Pool in the heap, and s1 will point to this literal in the string pool.

Now if we try to create another string s2, like
String s2 = "hello"
Before putting a literal in the string constant pool, Java will check if the string literal is already present in the String Constant Pool and if it is then instead of creating another instance of "hello" literal in the string pool, the existing one will be reused and s2 will refer to this literal as well.

In brief:
S2 => Is the string literal "hello" already present in the string constant pool => If yes, then s2 will point to the existing "hello" literal.

If we create a string using the "new" keyword, then it won't be considered as a string literal, instead a new String object will be created and stored inside the heap.

String s3 = new String("hello")
Here s3 will not be stored in the String constant pool, as it was created using the new keyword.



Why are Strings immutable

This is because whatever value we have created through string literals will be stored inside the string constant pool. So for example when we run String s = “hello”, the literal “hello” is stored in the String Constant pool, now assume we change s to “apple”, in this case first Java will check if “apple” is present in the string constant pool, if not it will create the “apple” literal inside the string constant pool, and s will refer to this literal. However the old literal “hello” will not be deleted; it'll remain in the string constant pool. The literals stored in the string constant pool can’t be modified, they are constants.

s1.equals(s2): Matches content
s1 == s2: Checks whether they are pointing to the same address or not.

Interface reference data type

Similar to class reference type, however we cannot instantiate an object of the interface itself, however the parent interface can be used as a reference to the child concrete classes, which implement the interface.

```
public class Bike implements Vehicle {

    @Override
    public void drive() {
        System.out.println("Bike");
    }

}

public class Truck implements Vehicle {

    @Override
    public void drive() {
        System.out.println("Truck");
    }

}

public static void main(String[] args) {
    Bike bike =  new Bike();
    Truck truck = new Truck();
```

```
    Vehicle bikeVehicle = new Bike();
    Vehicle truckVehicle = new Truck();
}
```

Array reference data type

An array is basically a sequence of contiguous memory locations which hold the same data type. Array is a reference type because it holds the reference to the actual memory location

Wrapper Class

Wrapper classes in Java are used to encapsulate primitive data types as objects. This is necessary because many Java libraries and frameworks, such as the Collections framework, only work with objects or Non-primitive types.

So for example if we want to create a list of ints, then we can't do List<int>, instead we need to use List<Integer>

Primitive data type	Wrapper Class / Reference type
int	Integer
short	Short
byte	Byte
float	Float
double	Double
char	Character
boolean	Boolean
long	Long

Autoboxing: Java 5 introduced Autoboxing, which automatically converts primitive types to their corresponding Wrapper class. For example int is converted to Integer. (Primitive Type -> Wrapper Class).

Integer x = 20;

Unboxing: Reverse process of Autoboxing, which automatically converts Wrapper class objects to their corresponding primitive type. For example Integer is converted to int. (Wrapper Class -> Primitive Type).

int x1 = x;

SINGLETON CLASS

The objective of the singleton class is to create one and only one object (or instance) of the class. For example a DBConnection which handles the connection with the database, we don't want to have multiple objects of such a class.

How to design a singleton class:

1. Eager Initialization
2. Lazy Initialization
3. Synchronized Method
4. Double Checked Locking
5. Bill Pugh Solution
6. Enum

Eager Initialization

Here we initialize the object as soon as the program is run. When a client wants an object of the class they can call a static method to get the object we created. However the client can't create an object itself, as the constructor is declared as private

Why do we call it eager initialization: All the static variables are preloaded into the memory, as soon as the program runs the object will be created which is also a disadvantage of the Eager Initialization technique, as it can lead to wasteful creation of objects.

```
public class DBConnection {
    private static DBConnection connObject = new DBConnection();
    private DBConnection() {}

    public static DBConnection getInstance() {
        return connObject;
    }
}
```

We declare getInstance() as static, so that the method can be accessed

without creating an object of the class, which is our goal here, preventing the clients from creating an object of the class and instead using the one created and handled by the Singleton class.
Do note the private constructor as well.

Lazy Initialization

Since the object is created in an eager manner in the eager initialization technique, it can lead to wastage of resources if the object is not actually needed or used. Hence as an alternative we can use Lazy Initialization, where we create the object only when it is needed. The code is similar to Eager Initialization, the only change being that we'll create the object (or instance) in the getInstance method, and not as part of declaration.

```
public class DBConnection {
    private static DBConnection connObject;

    private DBConnection() {}

    public static DBConnection getInstance() {
        if(connObject == null) {
            connObject = new DBConnection();
        }
        return connObject;
    }
}
```

Synchronized Method

However one of the problems with the Lazy Initialization method (which is not present in case of Eager Initialization) is that Lazy Initialization is not thread safe, what if 2 threads T1, T2 concurrently access getInstance method and end up creating 2 objects of DBConneciton.

To fix this, we can use a synchronized method by adding the synchronized keyword to the getInstance method. When we add the synchronized keyword to the method it'll do locking and unlocking in the background, and hence only one thread can enter the getInstance method (critical section) at a time.

```
public class DBConnection {
    private static DBConnection connObject;
```

```
    private DBConnection() {}

    synchronized public static DBConnection getInstance() {
        if(connObject == null) {
            connObject = new DBConnection();
        }
        return connObject;
    }
}
```

A disadvantage with this method, is that we are adding the synchronized block at a method level, which makes the method execution slow.

Double Checked Locking

Putting a synchronized block on the entire getInstance method is expensive, a better approach would be to put the synchronized block over the object creation logic only.

```
public class DBConnection {
    private static volatile DBConnection connObject;

    private DBConnection() {}

    public static DBConnection getInstance() {
        if(connObject == null) { => FIRST CHECK
            synchronized(DBConnection.class) {
                if(connObject == null) { => SECOND CHECK
                    connObject = new DBConnection();
                }
            }
        }

        return connObject;
    }
}
```

However double checked locking can lead to memory related issues in some conditions. For example thread T1 running on core 1 calls the getInstance method, and a new object is created. Core 1 updates it’s L1

cache however it hasn't updated the memory yet (memory writes are generally done in batches for performance reasons), in the meantime thread T2 running on core 2 calls the getInstance method, since the updated value of connObject hasn't been written to memory yet, T2 will determine that connObject is null, and another object will be created via the getInstance method and core 2 will update it's L1 cache, hence 2 objects of DBConnection got created.

To fix this we use the volatile keyword. Read and Writes performed on a variable declared volatile, always happen from the memory and never from the cache.
However the use of volatile makes the execution slower, as volatile prohibits the use of cache, and we are using the synchronized keyword as well.

Bill Pugh Solution

Bill Pugh Solution improves on the Eager Initialization technique, it uses a nested class which initializes the object. The advantage of using a nested class is that nested classes aren't preloaded hence there is no wasteful creation of objects.
Nested classes aren't loaded when the application is started, they are loaded only when they are referred.

```
public class DBConnection {
    private DBConnection() {}

    private static class DBConnectionHelper {
        private static final DBConnection connObject = new
DBConnection();
    }

    public static DBConnection getInstance() {
        return DBConnectionHelper.connObject;
    }
}
```

ENUM

In case of ENUM all constructors are private by default, and per JVM only one object of ENUM is present hence by default Enum is singleton only.

```
public enum DBConnManagerEnum {
```

```
    INSTANCE;

    public void getConnectionDetails() {
        System.out.println("DB Connection Manager - 3 active
connections");
    }
}
```

IMMUTABLE CLASS

Immutable class is a class whose state cannot be changed
Once the object of an immutable class is created it cannot be modified.
=> The class is declared final so that it cannot be extended
=> All class members are private, to prohibit direct access and final, to prohibit modification.
=> The class members can be initialized only once, using the constructor.
=> An immutable class cannot have any setters.
=> The getter methods should return a copy of the member variables.

POJO

Stands for Plain Old Java Object.
=> A POJO can contain variables, and their getters and setters
=> The class should be public.
=> It should only use a Public default constructor
=> The class should not have any annotations, like @Table, @Id, @Entity etc.
=> The class should not extend any other class or implement any interface.
=> The variables and methods themselves can have any access modifier.
=> POJO is just a simple java class.

ENUM Class

An Enum is a collection of constants (variables whose value cannot be changed).
=> Constants declared in an enum are static and final by default.
=> It cannot extend any class as it internally extends the Java.lang.Enum class (and multiple inheritance is not possible in Java).
=> Like other classes though it can implement interfaces, have variables, constructors and methods.

=> It cannot be instantiated (as in case of Enum the default constructor is private)
=> No other class can extend the Enum class.
=> Finally, it can have abstract methods, and all constants must implement the abstract methods.

```
public enum EnumSample {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY;
}
```

Internally, constants defined in an enum are assigned values (beginning from 0, incrementing by 1).

As the Enum class extends from the java.lang.Enum class, and gets its methods from the parent.

```
public class Driver {
    public static void main(String[] args) {
        // Iterating over an enum
        for(EnumSample enumSample: EnumSample.values()) {
            System.out.println(enumSample + " " +
                               enumSample.ordinal());
        }

        // Getting an enum variable, case sensitive
        EnumSample enumSample = EnumSample.valueOf("FRIDAY");
        System.out.println(enumSample.name() + " " +
                           enumSample.ordinal());
    }
}
```

We can customize the enum with self defined values

```
public enum CustomEnum {
    MONDAY(101, "Today is Monday"),
    TUESDAY(102, "Today is Tuesday"),
    WEDNESDAY(103, "Today is Wednesday"),
    THURSDAY(104, "Today is Thursday"),
    FRIDAY(105, "Today is Friday"),
    SATURDAY(106, "Today is Saturday"),
    SUNDAY(107, "Today is Sunday");

    private int val;
    private String comment;

    CustomEnum(int val, String comment) {
        this.val = val;
        this.comment = comment;
    }

    public int getVal() {
        return val;
    }

    public String getComment() {
        return comment;
    }

    public static CustomEnum findByValue(int value) {
        for(CustomEnum customEnum: CustomEnum.values()) {
            if(customEnum.val == value) {
                return customEnum;
            }
        }
        return null;
    }

}
```

The abstract methods defined in the enum class must be implemented by all the constants.

Method Overriding by Constants

As can be seen above the enum class can contain methods, these methods can be static (associated with the class) or instance (specific to a constant), i.e. each constant will have its own copy of the instance method [To call this method we will need access to the constant, while static methods can be called by using the enum class name, as the static methods have a single copy, which is associated with the class].

A constant can override methods as well.

```
public enum EnumMethods {
    MONDAY {
        @Override
        public void dummyMethod() {
            System.out.println("This is the overriden method");
        }
    },
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY;

    public void dummyMethod() {
        System.out.println("This is the original dummy method");
    }
}

Invocation:
for(EnumMethods enumMethods: EnumMethods.values()) {
     enumMethods.dummyMethod();
}

OR

EnumMethods enumFriday = EnumMethods.valueOf("FRIDAY");
EnumMethods enumMonday = EnumMethods.valueOf("MONDAY");

enumFriday.dummyMethod();
enumMonday.dummyMethod();
```

```
Output:
This is the overriden method
This is the original dummy method
This is the original dummy method
This is the original dummy method
This is the original dummy method
This is the original dummy method
This is the original dummy method

This is the original dummy method
This is the overriden method
```

Enum class can have abstract methods as well

We can define an abstract method in an enum just like in a normal class, however all of the constants defined in the enum will need to implement (override) the abstract method.
Each constant can override any of the non-static methods (These are the methods declared post the enum definitions, which are done at the top / start).

```
public enum EnumAbstractMethod {
    SATURDAY {
        @Override
        public void dummyAbstractMethod() {
            System.out.println("Saturday !!!");
        }
    },
    SUNDAY {
        @Override
        public void dummyAbstractMethod() {
            System.out.println("Sunday !!x");
        }
    };

    public abstract void dummyAbstractMethod();
}
```

Enums can implement interface

```
public enum EnumImplementIntf implements Intf {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY;

    @Override
    public String toLowerCase() {
        return this.name().toLowerCase();
    }
}

for(EnumImplementIntf enumImplementIntf:
                      EnumImplementIntf.values()) {
   System.out.println(enumImplementIntf.toLowerCase());
}
```

Final Class

A final class cannot be inherited.

```
final class felina {

}
```

Methods in Java

Access Specifiers:

public int getSum(int a, int b) {}

Consider the above method, Here public is the access specifier. What are the possible options for Access specifier?
=> public, private, protected, default

Public Access Specifier: A public method can be accessed by any class in any package.

What is a package
A Package is a collection of similar classes (Java source files)

Private Access Specifier: A method defined as private, can only be accessed (or invoked) by the methods in the same class. Not even a subclass can access a private method.

Protected Access Specifier: A protected method can be accessed by other classes in the same package, as well as by subclasses in different packages.

Default Access Specifier: Method can be accessed by other classes in the same package.

Types of Methods:

System Defined Methods: Ready to use methods provided by Java libraries, like Math.sqrt().

User Defined Methods: Created by programmer.

Overloaded Methods: More than one method with the same name exists in the class. Overloaded methods are differentiated based on the arguments (type of arguments, number of arguments).

=> Methods cannot be overloaded solely based on return type.
For example void method() cannot be overloaded with int method().
This is incorrect, as overloaded methods are differentiated on the basis of arguments and not return type.

Overridden Methods: Subclass has the same method as the parent class, and it overrides the method of the parent class.

Static Methods: These methods are associated with the class (not with the object), and can be called by just using the Class Name.

For a non-static method, each object of the class will have its own copy of this method. However there is only a single copy of the static method which is associated with the class, i.e. the objects don't have their separate copies of the static method (the static method is common for all the objects, it is not present inside each object), which also explains why static methods are called by using the class name directly.

=> Static methods cannot access Non-static variables and methods, i.e. static methods cannot access instance variables.

Observation: Static variables and methods can actually be accessed in a non-static way as well.

Why can't static methods access instance variables?
Static methods are associated with the class, on the other hand instance variables are associated with each of the objects, so each object has its own copy of the instance variable. So if in a static method which is associated with the class and common for all the objects, we try to access an instance variable then Java will not be able to understand which version of the instance variable we are referring to, because each object has its own copy of the instance variable.

For example we have an instance variable laptopCount, and 3 objects of the class: obj1, obj2, obj3. Each of these objects has its own copy of instance variable laptopCount, i.e. obj1.laptopCount, obj2.laptopCount and obj3.laptopCount. When we try to access laptopCount from within the static method (which is associated with the class), it will not be able to resolve which version of laptopCount we are referring to? Is it obj1.laptopCount, obj2.laptopCount or obj3.laptopCount.

A similar explanation follows for why static methods can't access non-static methods in a class
=> However if we specify the object, then we can call its non-static methods from within the static method (think of the main function).

Static methods cannot be overridden.
If we try to override a static method of the parent in the child class, then the method in the child class just hides the method in the parent class. This is because with static methods dynamic binding is not possible (since, static methods are not associated with objects).

Final Method: As seen above a subclass can override the methods of the parent class, if we want to ensure that the child class cannot override a method of the parent class, then we can declare the method as final. The

implementation of a method declared final cannot be altered by the child class.

Abstract Methods: Abstract methods can only be declared in abstract classes, an abstract method has a signature but no implementation, it is the responsibility of the child class to implement the abstract method.

Constructors

We will get to the definitions, but it's important to note that a constructor needs to have the same name as the class. So if a class is named Employee, then so should be the constructor. However important to note is the fact that other methods in the class can also use the same name as the class (thereby the constructor), so we can define a method with the name Employee. Consider the following example:

```
public class Employee {
    public Employee() {
        System.out.println("Constructor called");
    }

    public int Employee() {
        System.out.println("Employee 1");
        return 9;
    }

    public int Employee(int a) {
        System.out.println("Employee 2");
        return a;
    }
}
```

In summary: The constructor has the same name as the class, however it is possible for other methods to have the same name as the class name as well.

Unrelated but do note: We were able to define methods with the same name as the constructor, this is not an example of method overloading though.

We can define multiple versions of the constructor, each accepting a different set of inputs meaning we can overload the constructor.

The new keyword tells Java at runtime that it needs to call the constructor (and not any other method by the same name in the class).

Constructor is used to create and initialize an instance.

We mentioned before that the constructor name must be the same as the class name. Why is that?
The constructor name is the same as class name, as it makes identifying the constructor easier.

Why doesn't the constructor have a return type?
As mentioned before it is possible to declare methods in the class having the same name as the constructor (Employee() and method: int Employee()). So if such methods are defined then how will Java identify the constructor? This is why the constructor doesn't have a return type, so that it can be easily identified as such.

Implicitly, return type of Class will be added to the constructor method, however since we do not explicitly define a return type for the constructor (not even void), hence it can be easily identified as the Constructor by Java.

Together the 2 constraints that the
=> Name of the constructor must be the same as the name of the class
=> The constructor must not have a return type
help to distinguish the constructor from other methods.

Constructors cannot be static, final, abstract or synchronized.

Why not final?
It's important to note that constructors cannot be inherited by the child class, the use of final keyword is to prevent a method from being overridden by the child class, however if the method is not inherited in the first place then there is no question of it being overridden. So what's the point of the final keyword?

> Why are constructors not inherited by the child class?
> Suppose the parent class is called Employee, with a constructor called Employee and a class Manager which extends the Employee class. As the constructor has the name as child class hence the constructor of Manager must be called Manager. If the parent class constructor were inherited in the Manager, then it won't be treated as a constructor (because it has a different name than the Manager class) instead it'll be treated as a method, a Method must have a return type and since the constructor has no return type, hence a

compile time error for the return type will be generated. Hence constructors are not inherited.

Why not abstract?

An abstract method is a method with no implementation and hence it needs to be implemented by the child class (or subclass), however since constructors are not inherited by the child class in the first place hence how can the child class provide an implementation for this constructor.

Why not static?

As described before static methods cannot access instance variables and non-static methods of the class, in a constructor call we usually initialize member variables (instance variables) of the class, however if the constructor is not able to access these variables then how can it initialize them? (1) A static constructor can't initialize instance variables.

Why do interfaces not have constructors?

As mentioned before we cannot create an object of an interface, i.e. we cannot instantiate an interface. Constructor is used to create an instance / object. For interface we cannot create objects, and if we cannot create objects then what is the point of Constructor?

Types of Constructors

Default Constructor: If we do not specify any constructor, then the default constructor is automatically added. (Can be checked in the .class bytecode). Default constructor will set all the members to default values.

Default Constructor:

```
Employee() {} // automatically added behind the scenes
```

Note: Once we explicitly declare any constructor, the default constructor will no longer be added.

No argument Constructors

Parameterized Constructors

Overloaded Constructors: Constructors can be overloaded, as in the case of method overloading the constructors will be differentiated by the arguments they take as input, i.e. count of arguments and types of arguments.

=>==> Obviously constructors cannot be overridden

Private Constructors: We can declare a constructor private, as is the case with private methods no other class will be able to call the constructor to create an object. Only the methods in the class which has the constructor can access it and create objects.

Assume:

```
public class PrivateConstructor {
    private PrivateConstructor() {

    }
    public PrivateConstructor getInstance() {
        PrivateConstructor privateConstructor = new
PrivateConstructor();
        return privateConstructor;
    }
}
```

Great, but how can another class access this one altogether now!!, it won't be able to create an object of it.

To solve this problem, we declare getInstance as static so that it can be accessed directly by using the class name.

The primary use of a private constructor is to ensure that no other class can create an object of this class, the class itself should have full control over object creation.

Constructor Chaining
=> Using "this" keyword:
Chaining within the same class

this(...) is the constructor.

```
public class ConstructorChaining {
    public ConstructorChaining() {
        System.out.println("Base constructor");
    }

    public ConstructorChaining(int a) {
        this();
        System.out.println("Single arg constructor");
    }

    public ConstructorChaining(int a, int b) {
        this(a);
        System.out.println("Two arg constructor");
    }
}
```

Constructor call must be the first statement in a constructor.

=> Using “super” keyword:

The first thing the child constructor does is call the constructor of the parent class.
The child constructor internally first invokes the parent constructor, using the super keyword. We do not need to add the call to super ourselves, Java will automatically do that.

=> This means first the parent object needs to be created then only the child object can be created.

```
public class Child extends Parent {
    public Child() {
        System.out.println("Child Class");
    }
}

public class Parent {
    public Parent() {
        System.out.println("Parent class");
    }
}
public class Driver {
```

```
    public static void main(String[] args) {
        new Child();
    }
}
```

Output:
Parent class
Child Class

Interfaces

An Interface helps two systems to interact with each other without any system having to know the details of the other, in other words Interfaces help to achieve abstraction and hide complexity from the clients.

Interface access modifiers: Only default (package-private) and public access modifiers are allowed (private and protected modifiers are not allowed)

=> An interface can extend multiple interfaces, so multiple inheritance is possible in case of Interfaces, Interfaces cannot extend from classes.

Why use Interfaces?

Abstraction

Using interfaces we can achieve 100% abstraction i.e. full abstraction. Interfaces define what a class must do, not how it should do that. Goes without saying, all the methods defined in an interface are abstract.

Polymorphism

Interface can be used as a data type. The parent type can hold the objects of the child, i.e. act as a reference to the child object. This means the interface type can hold the reference to the child object, hence the interface can be used as a reference variable.

```
public interface Bird {
    public void fly();
}

public class Eagle implements Bird {

    @Override
    public void fly() {
        System.out.println("Fly Eagle");
    }

}

public class Hen implements Bird {

    @Override
    public void fly() {
```

```
        System.out.println("Fly Hen");
    }

}

public class Driver {
    public static void main(String[] args) {
        Bird bird = new Eagle();
        bird.fly();

        bird = new Hen();
        bird.fly();
    }
}
```

Here we are dynamically calling the fly() method, and at run time it is determined which fly() method to call based on the object.

In summary:
=> We cannot create an object of an interface, but the interface can hold references to all the child classes which implement it.
=> At Runtime it is determined which method needs to be invoked (well consider a method called fly() exposed by the interface, implemented by class C1, C2, C3. Depending on the object we might be calling C1.fly(), C2.fly() or C3.fly() and the interface type can hold references to all of C1, C2 and C3).

Multiple Inheritance

In Java, multiple inheritance is possible only through Interfaces. Why is multiple inheritance not possible in case of classes: Diamond Problem.

There are 2 parts to this multiple inheritance:

1. An interface can extend from multiple interfaces
2. A concrete class can implement multiple interfaces (class c implements i1, i2, i….)

Methods in Interfaces

=> All methods in an interface are implicitly public only.
=> Methods cannot be declared as final

Variables in Interfaces

=> All variables in an interface are implicitly public static final (CONSTANTS).

=> i.e. all the variables are public (we cannot make them private or protected) and they are constants.

Notes on implementing an Interface

=> The Overriding method cannot have a more restrictive access specifier than the interface.
=> Concrete class needs to implement (override) all the methods declared in the interface.

SIDE NOTE: In case of an abstract class, the concrete class might not need to provide implementation for all the methods (as abstract classes provide 0 - 100% abstraction, and in an abstract class, methods can have implementations).

=> An abstract class implementing the interface is not forced to override all the methods.

Nested Interfaces

=> Interface within an Interface
=> Interface within a class

=> We can use Nested Interfaces to group logically related interfaces together.

Interface within interface.

- A nested interface declared within another interface, must be public.

```
public interface SuperInterface {
    public void tortoise();

    public interface NestedInterface {
        public void rabbit();
    }
}
```

Concrete classes can implement the outer Interface and the nested interface separately, i.e. concrete class C1 can implement SuperInterface only, while concrete class C2 can implement NestedInterface only. In other words if a class is implementing the outer interface, then it doesn't necessarily need to implement the inner interface as well and vice versa.

Interface within class.

- A nested interface within a class can be private / protected / public or default (no explicit declaration).

Differences b/w Interfaces and Abstract class

Interface	Abstract Class
Keyword: “interface”	Keyword: “abstract”
Child class: “implements”	Child class: “extends”
It can only have abstract methods	It can have both abstract and non-abstract methods
It can only extend from other interfaces.	It can extend from another class and multiple interfaces.
Variables are public static final (i.e. by default variables are constants)	Variables can be static, non-static, final, non-final
Variables and methods are by default public.	Variables and methods can be public / private / protected and default.
Support multiple inheritance	Do not support multiple inheritance.
It cannot have constructor	It can have constructor
The methods are by default abstract, and public	To declare a method abstract, use the “abstract” keyword. The abstract methods can be public / protected and default.

Interfaces extension: New Features, Functional and Lambda

Interfaces and Java 8 features

Default Methods: Introduced in Java 8, before Java 8 all the methods in an interface were abstract.

Need for default Methods: We need to consider the shortcomings of the traditional interface design, where only abstract methods are allowed. Suppose we have an interface with multiple concrete classes implementing it. If we add an additional method to the interface, then all the child classes will need to override it (provide an implementation for it).
Hence, if an interface can only have abstract methods then all the child classes which implement the interface will need to be changed, and implement the newly added method.

This might lead to code duplication (the method is common for most of the classes), also it's possible that some methods are not applicable for some of the child classes, if the method is abstract then the child class will still need to override it.

To overcome this shortcoming of interfaces, the concept of default Methods was introduced in Java 8.

```
public interface Bird {
    public void canFly();
    default int getAge() {
        return 30;
    }
}

public class Duck implements Bird {

    @Override
    public void canFly() {
        System.out.println("Yes");
    }

}

public class Eagle implements Bird {

    @Override
```

```
    public void canFly() {
        System.out.println("NO");
    }
}
```

So, in summary, default methods allow us to provide implementation of methods in the interface itself, i.e. the classes which implement the interface don't need to override those methods.

Notes about inheritance with default methods,
In case of interfaces multiple inheritance is possible, however if the interfaces have conflicting default methods then the multiple inheritance will not be possible. As java will not be able to figure, which version of the method to call. However if the child overrides that method and provides its own implementation then Multiple inheritance is possible.

A class implementing an interface with a default method, can override the default method.

Error:
Duplicate default methods named cobra with the parameters () and () are inherited from the types Parent and Parent1.

=> Observation: This indicates obviously that if intf1 has default method m1() and intf2 has default method m1(), then multiple inheritance won't be possible.
=> However if intf1 has m1(int a) and intf2 has m1() then multiple inheritance is possible.
=> If intf1 has m1() and intf2 has m1() then multiple inheritance is still possible if the concrete class implementing intf1 and intf2 provides its own implementation for m1 (thus removing any ambiguity).

Notes on interface inheritance, if the parent interface has default methods:

The child interface can override the default method either with its own version of the default method (i.e. change implementation, but keep method default) or it can even override the default method with an abstract method and force the concrete class to implement it.

Another important note: Suppose an interface P has default method dm1(), if the interface C extends P, then C can override dm1() and provide its totally unique implementation for it. The interesting part is how C can internally call P's dm1()? In case of classes we can use super keyword. However we can't directly do that in case of interfaces instead we need to call dm1 from C as follows.

How can a child interface call the parent's default methods?
P.super.dm1(); [InterfaceName.super.methodName]

```
public interface Parent {
    default int cobra() {
        return 90;
    }
}

public interface ExtendParent extends Parent {
    default int cobra() {
        int getValueFromParent = Parent.super.cobra();
        return getValueFromParent + 23;
    }
}
```

Static Methods: Like Default Methods, static methods (added in Java 8) can have implementations in the interface itself. Similar to static methods in classes, static methods in interfaces can be accessed by using the Interface name alone. Intf.staticMethodName(), and are by default public.

=> Static methods cannot be overridden by the classes which implement the interface.

```
public interface Parent {

    static String staticMethodInAnInterface() {
        return "Static Method";
    }
}

public interface ExtendParent extends Parent {

    static String staticMethodInAnInterface() {
        return "Override";
    }
}
```

Private Methods: Introduced in Java 9, similar to private methods in classes, private methods in interfaces can only be accessed within the

interface itself, i.e. by other methods within the interface.

=> Private methods can be either static or non-static.

Abstract methods don’t have an implementation, so the only methods which can access private methods are static methods and default methods.

=> Static methods in an interface cannot access non-static methods, as static methods cannot access non-static variables or methods in general. So static methods can only access static methods and static variables.
=> Non-static methods can access both static methods as well as non-static methods

```
public interface AInterface {
    private void privateMethod() {
        System.out.println("Complex BST-AVL CODE");
    }
    private static void staticPrivateMethod() {
        System.out.println("Static Complex BST-AVL CODE");
    }
    default void method1() {
        privateMethod();
    }
    static void method2() {
        staticPrivateMethod();
    }
}
```

Private methods in an interface or Abstract class cannot be abstract.

=> One more important point: Private variables and methods can’t be accessed anywhere outside the class, not even in the child class.

Functional Interfaces and Lambda Expressions

A Functional interface is an interface which contains only one abstract method. Note: The interface needs to contain strictly 1 abstract method however it is free to have other types of methods like default, static etc. But the important point is that there should be one and only one abstract method.

Functional interfaces are also known as SAM Interfaces (Single Abstract Method)

We can add the annotation @FunctionalInterface to indicate that the interface is Functional Interface, however that is not necessary any interface with a single abstract method will be treated by Java as a functional interface.

=> The annotation @FunctionalInterface will force you to have only one abstract method in the interface, i.e it will restrict from adding more abstract methods to the interface. Once again, the annotation is not mandatory.

Functional Interfaces = One Abstract methods (Rest can be default, static methods etc. but only one abstract method).

Note: In Interface if we define methods of the Object class, then the concrete class does not need to provide implementation for these methods, as by default all classes are subclasses of the Object class, and Object class provides the implementation for its methods.

So what does the Functional Interface contain (Summary):
=> A single abstract method (one and only)
=> 0 or more Default methods
=> 0 or more static methods
=> 0 or more Methods from the Object class.

How can we implement a functional interface?
We can implement it using these techniques: (1) Creating a child class and using the implement keyword (2) Using anonymous class (3) Use Lambda Expressions

Lambda Expression (introduced in Java 8) is a way to implement the functional interface.

Consider the following example:

```java
@FunctionalInterface
public interface Bird {
    public void canFly(String val);
}

public class Eagle {
    public static void main(String[] args) {
        Bird bird = (String value) -> System.out.println(value);

        bird.canFly("Kelloggs");
    }
}
```

Types of Functional Interfaces

Consumer - Takes one input argument, and returns nothing
Supplier - Takes no input, and returns a result
Function - Takes one input argument and returns a result
Predicate - Takes one input argument and returns a boolean

These Functional interfaces are present in java.util.function

```java
public class Driver {
    public static void main(String[] args) {
        Consumer<String> consumer = (String input) -> {
            System.out.println("You entered: " + input);
        };
        consumer.accept("Kellogs");

        Supplier<String> supplier = () -> {
            String dateString = "Today is Thursday";
            return dateString;
        };
        System.out.println(supplier.get());

        Function<Integer, String> function = (Integer intVal) -> {
            String greetString = "Hippo ID: " + intVal;
            return greetString;
        };
        System.out.println(function.apply(23));

        Predicate<Integer> predicate = (Integer testVal) -> {
```

```
            if(testVal < 40) {
                return true;
            }
            return false;
        };
        System.out.println(predicate.test(40));
    }
}
```

=> Functional Interface extending Non Functional Interface

The following inheritance is valid.

```
public interface Animal {
    public void dataDog();
}


@FunctionalInterface
public interface Cat extends Animal {

}
```

However if the Animal interface has multiple abstract methods then our Functional interface can't extend from it. Or in the above example if we try adding an abstract method to Cat, it will generate a compile time error. The key is the Functional interface can have one and only one abstract method (whether this method is defined in the functional interface itself, or defined in it's parent interface it doesn't matter).

=> Functional Interface extending Functional Interface

Say we are extending from functional interface Intf1, since it is functional it'll have one abstract method. Now suppose Intf2 is the interface extending it, to satisfy the SAM constraint, Intf2 cannot define any abstract methods of its own. Otherwise the inheritance is invalid.

What if Intf2 overrides the abstract method of Intf1
This is valid, since there is only one abstract method in Intf2

Memory Management and Garbage Collection

Types of Memory:
Stack
Heap

Both stack and heap memories are created by the JVM in the RAM.

Stack Memory

The Stack will store local variables, return addresses and function parameters; it has a separate memory block or frame for each method in the call chain, inside each frame the local variables, function parameters and references created as part of that method will be stored.

=> The Stack stores temporary variables, Primitive data types
=> It will store references to heap objects.

Each thread has its own stack memory, however all the threads share the same heap memory.

As soon as a temporary variable goes out of scope it is deleted from the stack.

=> When the stack is full, we get the Stack Overflow error.

When the scope is finished, all the variables and references created within that scope will be deleted from the stack.

One thing to note is that references to heap objects are stored in the stack, if the reference variables goes out of the scope, then the variable is deleted from the stack, however the actual object allocated on the heap is unaffected, i.e. the object itself is not deleted from the heap only a reference to it is deleted.

The Garbage Collector is used to delete unreferenced objects from the heap. Garbage collector runs periodically, JVM controls when to run the garbage controller. The Garbage collector will scan the heap and check for unreferenced objects and delete them.

To explicitly run the garbage collector we can use the method System.gc(), however there is no guarantee that even if we use the above method the Garbage Collector will run, it totally depends on the JVM which has full control over when to run the Garbage collector.

Hence Garbage Collector provides automatic memory management. So we don’t have to worry about freeing the memory, the Garbage collector

takes care of it, the JVM runs the garbage collector periodically.

Types of References:

=> Strong Reference:

Consider the following reference
Person personRef = new Person();

This is known as a strong reference, as long as the strong reference is present the Garbage collector cannot delete the object from the heap.

When the garbage collector runs it will check if the object has any references, and it will detect that the object has a strong reference so it won't delete it.

=> WeakReference

WeakReference<Person> personWeakRef = new WeakReference<>(new Person())

If an object has a weak reference, then whenever the Garbage collector runs it will delete the object from the heap. This means the object can only be accessed via the Weak Reference until the garbage collector has not run, post which if we try to access the object it will return null.

=> Soft Reference

Soft Reference is a type of weak reference, however the Garbage Collector will only deallocate the object from the heap only if it is very urgent, this happens when there is no space left in the heap and we need to create more objects. The decision whether to delete the item from heap or not is taken by the GC.

Heap Memory

The objects are allocated on the heap.
The heap is divided into parts

Young Generation	Old Generation

Heap Memory

Additionally there is a Non-heap Metaspace (MetaSpace is not a part of the heap memory).

So the heap memory is the Young Generation + Old Generation

The Young Generation is further divided into 3 parts.

Eden	S0	S1

Young Generation

S0 and S1 are also survivors.

The basic idea is that in the young Generation part of the heap the Garbage collector will run very frequently, these will be the objects which are quickly created and deallocated from the heap. However there will be some objects which won't be deleted even after the Garbage collector has run multiple times (this is because these objects still have references, and are being actively used). Such objects will be transitioned to the Old Generation part of the heap. In the Old Generation memory part, the Garbage collector runs less frequently.

On what basis do we transition the object from Young Generation to Old Generation?

We use the concept of age, when the object is created it is stored in the Eden part of young Generation and it has an age of 0, when the garbage collector runs either the object will be deleted (if no references) else it will be moved to one of S0 or S1 and its age will be incremented by 1, again after some time later the garbage collector will run and if the object still has a reference it can't be deleted, however the object will be moved to the other survivor part (S0 or S1), and it's age increases by 1, in other words the object will keep moving alternatively b/w S0 and S1 and it's age keeps increasing. Once the age has crossed a predefined threshold we transition the object from Young Generation to Old Generation, this is called promotion.

=> When we create an object it is first stored in the Eden part of the Young generation.

Terminologies:

Minor GC: Describes garbage collection in the Young Generation, GC happens very frequently in the Young Generation space, Objects are created / deleted quickly in the Young Generation, for example

```
method() {
    Person person = new Person();
```

}

Method goes out of scope, so the person reference goes out of Scope (deleted from stack) => There is no reference to the Person object in the heap, so it will be quickly cleaned up.

Major GC: Describes garbage collection in the Old Generation, GC happens less frequently in the Old Generation space. The objects present in the Old Generation are used frequently, these objects are alive for a long time and can have multiple references.

One thing to note, in the case of Young Generation there are 3 parts: Eden, S0 and S1. When the garbage collector does run it will run across all the three partitions, in other words Mark and Sweep will be performed on all of S0, S1 and Eden.

Mark: Scan the objects and mark (or identify) the unreferenced objects.
Sweep: Delete all the marked objects and sweep the other ones into one of S0 or S1 (alternatively selected).

So after every time the GC runs in the Young Generation, the Eden part will be empty, and one of S0 and S1 will contain all the objects while its counterpart will be empty.

Note:
-> Promotion involves moving the object to the Older Generation.
-> Even in the Old Generation Garbage collection will happen (even though less frequently) and it involves the same Mark and sweep cycle, however there is no transitioning from one part to another within the Old Generation as it is not further subdivided like the Young generation.
-> Old Generation is also called Tenured Generation
-> Metaspace is also called Permanent Generation (permgen) [However premgen was a part of the heap, and was not expendable, it was used in earlier versions of Java, before 7).

What about the Metaspace?

The Metaspace is some space outside the stack which stores class variables (Variables created using static keyword), class metadata (Information about the class, using which objects can be created), constants (static final). When the JVM needs to load some class it will load it in this Non-Heap Metaspace, and when it doesn't need it anymore it'll remove that class from the Metaspace.

Garbage Collector Algorithm

1. Mark and Sweep Algorithm: First phase mark, second phase sweep

discussed above.
2. Mark and Sweep with Compact Memory: In addition to traditional Mark and Sweep it will also perform Compaction of free memory.

Versions of GC

1. Serial GC: Uses only one thread, hence it is slower. One thing to note: GC is an expensive operation, this is because when GC starts all application threads will be paused [World stops when the Garbage collector runs]. Hence if GC is slow then the applications will be slower.

2. Parallel GC [default in Java 8] - Multiple threads run in parallel (depending on the number of cores in the machine), hence GC will be faster and less application pause time. However do note the application threads are still being paused. Parallel Garbage Collector does perform Compaction.

3. Concurrent Mark and Sweep - In case of Concurrent Mark and Sweep, While the Garbage Collector is running, the application threads can run concurrently (they are not paused), the application threads and GC threads are running concurrently. However it offers best effort performance, as it is not guaranteed that the application threads will not be paused, that decision is taken by the JVM. There is no compaction.

4. G1 GC - Similar to Concurrent Mark and Sweep, however it performs compaction of free space as well.

Streams

We can consider stream as a pipeline through which our collection elements pass through.
While the elements pass through the pipeline, Stream can perform various operations like sorting, filtering etc.
- Useful when dealing with large amounts of data (bulk processing) as it can do parallel processing.

How do streams work?
Step 1: Create a stream from the collection of data, could be a list for example, or even an array.

Step 2: Apply the intermediate operations like filter(), sorted(), map(), distinct() etc, we can apply multiple intermediate operations.
=> An intermediate operation transforms the stream into another stream and more intermediate operations can be done on top of it.
=> Intermediate operations are lazy in nature, i.e. these operations will only get executed when the terminal operation is invoked.
=> We can have zero intermediate operations as well, so it is not mandatory to have intermediate operations.

Step 3: Apply the terminal operation, like count(), reduce(), collect() etc. These operations trigger stream processing, i.e. they trigger the execution of all the intermediate operations and produce an output. After the terminal operation is used, no more operations can be performed on the stream, as the terminal operation will close the stream as well.

=> Original data is never modified, each operation will produce a new modified stream, hence the original collection is never changed.
=> We need to have at least one terminal operation.

Stream demo

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();

list.addAll(Arrays.asList(1234,255,647,36,8,342,657,8,242,678));
    long output = list.stream().filter(
                          (Integer num) -> (num < 500)).count();
    System.out.println(output);
}
```

Methods to create a stream

1. From Collection
2. From Arrays
3. Using the static 'of' method (variable args)
4. From Stream Builder

```
public class Intro {
    public static void main(String[] args) {
        // Stream from collection
        List<Integer> list = Arrays.asList(1, 23, 56, 23, 25, 18);
        Stream<Integer> stream = list.stream();

        // Stream from arrays
        Integer [] values = {1, 5, 2, 67, 10, 90};
        Stream<Integer> streamArrays = Arrays.stream(values);

        // Stream from static method
        Stream<Integer> streamFromStaticMethod = Stream.of(1, 23);

        // Stream from stream builder
        Stream.Builder<Integer> streamBuilder = Stream.builder();
        streamBuilder.add(2);
        streamBuilder.add(17);
        streamBuilder.add(35);
        streamBuilder.add(27);

        Stream<Integer> streamFromBuilder = streamBuilder.build();

    }
}
```

Different Intermediate operations

We can chain multiple intermediate operations together to perform complex processing before getting the result by using a terminal operation.

Do remember that the terminal operation closes the string, so we cannot perform multiple sets of operations on the same stream, i.e. for example this is not valid:

List<String> names = <>
Stream<String> streamNames = names.stream();
System.out.println(streamNames.filter((String val) -> (val.length() <= 5)).count());
System.out.println(streamNames.filter((String val) -> (val.length() <= 3)).count());

Output:

2
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
at java.base/java.util.stream.AbstractPipeline.<init>(AbstractPipeline.java:203)
at java.base/java.util.stream.ReferencePipeline.<init>(ReferencePipeline.java:96)
at java.base/java.util.stream.ReferencePipeline$StatelessOp.<init>(ReferencePipeline.java:800)
at java.base/java.util.stream.ReferencePipeline$2.<init>(ReferencePipeline.java:167)
at java.base/java.util.stream.ReferencePipeline.filter(ReferencePipeline.java:166)

Also, since any intermediate operation will return a stream as well, hence the following set of operations is valid:

Filter

```
List<String> names = Arrays.asList(<>);
Stream<String> streamNames = names.stream();

streamNames = streamNames.filter(
                            (String val) -> (val.length() <= 5));
System.out.println(streamNames.count());

/*
// stream already closed by above method
List<String> elementsInStream =
             streamNames.collect(Collectors.toList());
*/
```

Map

```
List<String> names = Arrays.asList(<>);
Stream<String> namesStream  = names.stream();
namesStream = namesStream.map((String val) -> "hello: " + val);

List<String> newList = namesStream.collect(Collectors.toList());
System.out.println(newList);
```

flatMap

```
List<List<String>> dll = Arrays.asList(
    Arrays.asList("Ship", "Fox", "Cat"),
    Arrays.asList("Horse", "Dog", "Eagle"),
    Arrays.asList("Fish", "Sloth", "Moth")
);

Stream<String> dllStream = dll.stream().
                                flatMap((List<String> sentence) ->
                                            sentence.stream());
System.out.println(dllStream.collect(Collectors.toList()));

Stream<String> dllStream2 = dll.stream().
                                flatMap((List<String> sentence) ->

sentence.stream().map((String val) ->

val.toUpperCase()));

System.out.println(dllStream2.collect(Collectors.toList()));
```

Distinct

```
Stream<Integer> streamInts = list.stream().distinct();
System.out.println(streamInts.collect(Collectors.toList()));
```

Sorted

```
Integer [] values = {1, 5, 2, 67, 10, 90};
Stream<Integer> streamArr = Arrays.stream(values);
System.out.println(streamArr.sorted().collect(Collectors.toList())
);
```

```
// Sort in desc
List<Integer> sortedList = streamArr.sorted((Integer val1,
                                            Integer val2) ->
                                            val2 - val1).
```

```
                                        collect(Collectors.toList());
System.out.println(sortedList);
```

Peek

```
Stream<Integer> streamValues = Arrays.stream(values).
                                map((Integer val) -> (val * 10)).
                                peek((Integer val) ->
                                      System.out.println(val));

streamValues.collect(Collectors.toList());
```

Limit

```
List<Integer> cacil = Arrays.asList(1, 56,4, 46, 12, 16);
Stream<Integer> streamWithLimit = cacil.stream();

List<Integer> newCacil =
streamWithLimit.limit(4).collect(Collectors.toList());
System.out.println(newCacil);
```

Working with primitives

All the above methods work with non-primitives or reference types. How do we work with primitives like int?

We use IntStream for dealing with ints instead of using the usual Stream<Integer>

```
int [] primitiveValues = {23, 56, 13, 67, 45, 17};
IntStream streamPrimitives = Arrays.stream(primitiveValues);

streamPrimitives =
         streamPrimitives.filter((int val) -> (val > 20));
int [] modifiedValsArray = streamPrimitives.toArray();
```

=> mapToInt method

```
Stream<String> nsNames = names.stream();
```

```
int [] modArray =
        nsNames.mapToInt((String val) -> val.length()).toArray();
```

It should be noted that IntStream provides the terminal operation toArray.

Similar to IntStream, LongStream and DoubleStream are also supported.

General Notes on Intermediate operations

=> As mentioned before the intermediate operations are executed in a lazy manner, until and unless we call a terminal operation none of the intermediate operations will run.

=> On the order of processing of the intermediate operations.
If we chain multiple intermediate operations A1, A2, A3 … And together, the expectation would be that A2 only gets executed once A1 has been executed completely, and that A3 will only be executed once A2 is complete and so on. However this is not always true.

There are some intermediate operations which can work with partial streams like peek, filter, map etc. However certain operations can only work with the complete stream, for example sorted.

The elements of the collection are processed sequentially, if the operation can work with partial streams then the element will be passed to the next operation in the chain, and so on. However, as soon as we encounter an operation (like sorted) which can only work with complete streams (i.e. this operation cannot be started until all the stream elements are present). then that element cannot be passed to the next operation. Instead the operation will wait for the complete stream (i.e. all the elements) and once it receives the complete stream then this operation can execute.

This technique of execution helps stream to process the tasks faster.

Different Terminal operations

forEach

```
nums.stream()
    .filter((Integer val) -> (val >= 5))
    .forEach((Integer val) -> System.out.println(val));
```

toArray

Return Object[]

```
Stream<Integer> kStream = list.stream()
                              .filter((Integer val) -> (val <=
30));
Object [] modValues = kStream.toArray();
```

Getting an array of a particular Object.

```
Integer [] intValues = list.stream()
   .filter((Integer val) -> (val <= 30))
   .toArray((int size) -> new Integer[size]);
```

reduce

Performs Associative aggregation

```
Optional<Integer> sum = list.stream()
                         .reduce((Integer a, Integer b) -> a + b);

System.out.println("sum is: " + sum.get());
```

collect

```
Stream<String> namesStream  = names.stream();
namesStream = namesStream.map((String val) -> val.toUpperCase());
List<String> newList = namesStream.collect(Collectors.toList());
System.out.println(newList);
```

min / max

```
// Using Stream
System.out.println(list.stream().max(Comparator.naturalOrder()).ge
t());

// Using IntStream
```

```
int maxElement = list.stream()
                                .mapToInt((Integer val) -> val)
                                .max().getAsInt();
System.out.println(maxElement);
```

count

Returns number of elements in the stream

```
System.out.println("Count: " + list.stream().count());
```

anyMatch

Check if any value matches the given predicate. Returns boolean

```
System.out.println(list.stream().anyMatch((Integer val) -> (val <=
30)));
```

allMatch / noneMatch

```
System.out.println(list.stream().allMatch((Integer val) -> (val <=
30)));

System.out.println(list.stream().noneMatch((Integer val) -> (val
>= 3000)));
```

findFirst / findAny

firstFirst: Return first element from the stream
findAny: Return any random element from the stream

Parallel Streams

.stream() creates a sequential stream, to create a parallel stream use the parallelStream method.

Using a Parallel Stream we can perform operations on the stream

concurrently, taking advantage of the multicore CPU. This helps in performing the tasks much faster.
Parallel stream achieves this by dividing the task into multiple subtasks, each of which can be further subdivided. These subtasks are run parallelly and eventually the results from each of these subtasks is joined back to get the complete result.

Components of parallel stream:
Task Splitting: Parallel streams use the SplitIterator interface to split the data into multiple chunks. SplitIterator recursively splits the task / input, by using the trySplit() method [Split in half].

Task submission and parallel processing: After data has been splitted, Parallel streams internally use fork join pool technique for parallel processing, i.e. it will hand over the smaller chunks to fork join pool to execute concurrently, and eventually combine the results of these smaller tasks to get the overall output.

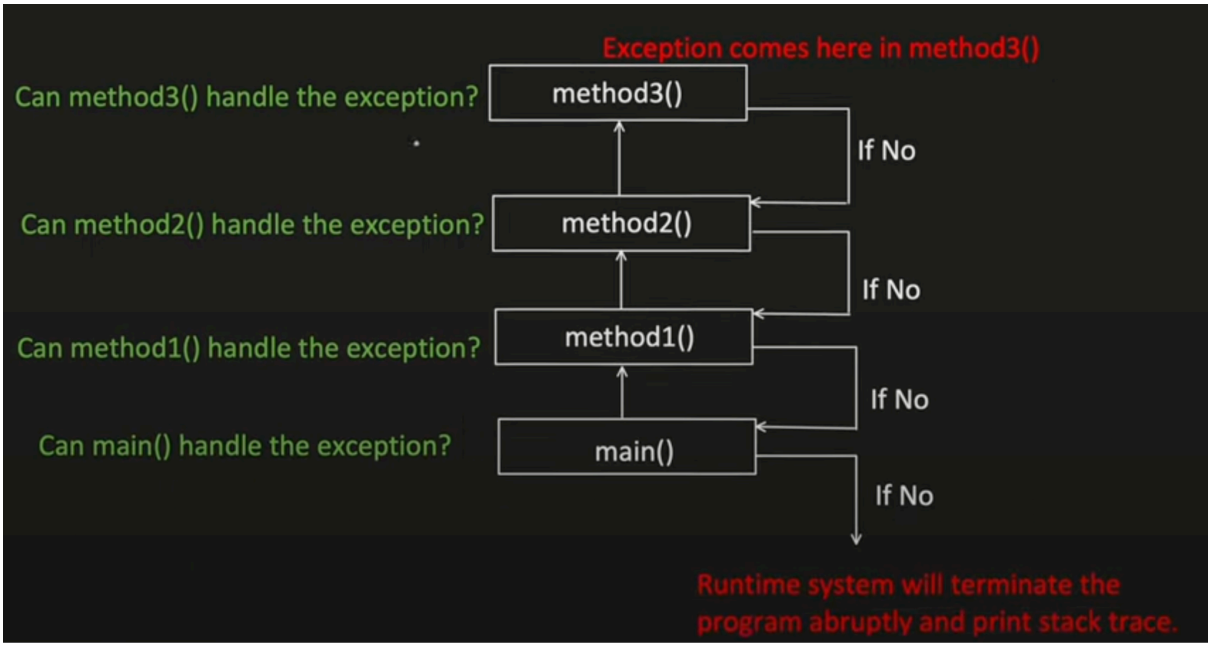
Exception Handling

Exception is an event that occurs during the execution of a program, and it disrupts the normal program flow.

When an exception occurs an Exception object will be created which will contain information about the exception like:

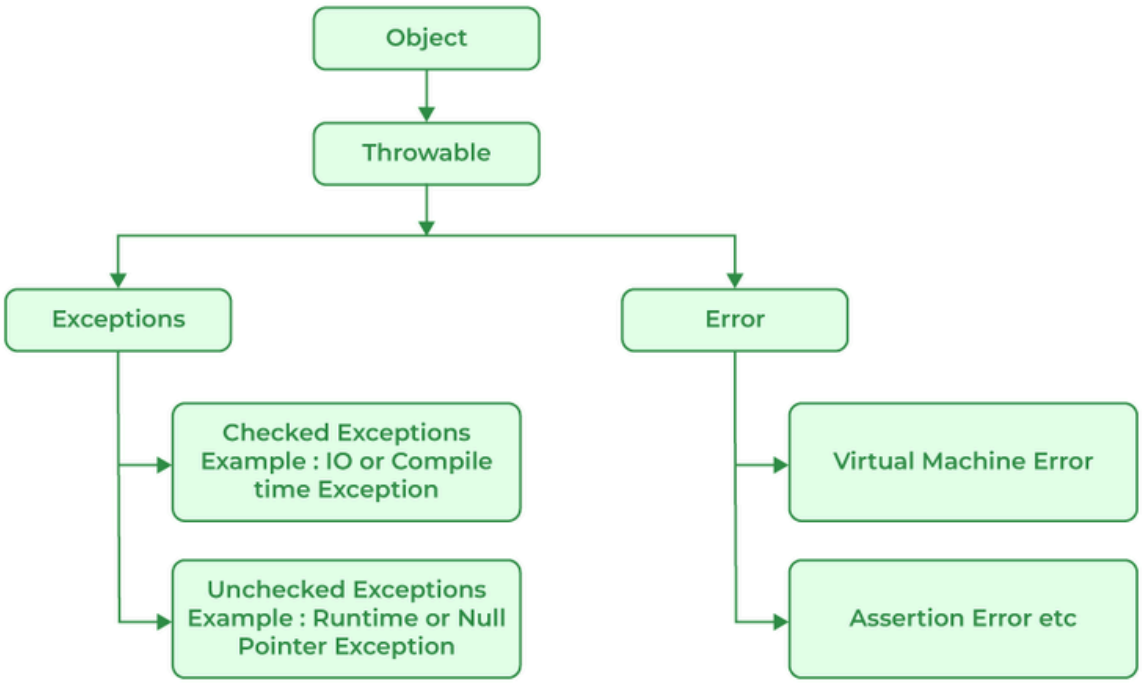
=> Type of exception and message
=> Stack trace etc.

Runtime system uses the exception object to find the class which can handle the exception.



If no method can handle the exception then the runtime system will terminate the program, and print the stack trace.

Exception Hierarchy



Errors: Errors are events which are not in our control, for example Out Of Memory Error or Stack Overflow error and hence we cannot handle them, these events are issues related to the JVM and can't be handled by the programmer.

```
public class OutOfMemoryError {
    public static void main(String[] args) {
        String [] dataDog = new String[234356 * 2356 * 245 *
23454];
    }
}
```

Exceptions: Exceptions are events which can be handled, as they are related to our code, which we have control over.

Types of Exceptions:
=> Unchecked Exceptions (Runtime Exception): If our code has unhandled unchecked exceptions it will compile successfully however when we run the program, it might fail.

=> Checked Exceptions (Compile time Exception): Code won't even compile if an unhandled checked exception is present, in other words if a checked / compile time exception is not handled then the compilation will fail.

Examples of Exceptions

Compile Time Exceptions	Run Time Exceptions
ClassNotFoundException	ArithmeticException
InterruptedException	IndexOutOfBoundException
IOException	NullPointerException
SQLException	IllegalArgumentException
	ClassCastException

Runtime (Unchecked) Exceptions: These are the exceptions which occur during runtime and the compiler doesn't force us to handle them.

Compile time (Checked) Exception - Compiler verifies during the compile time that the checked exceptions have been handled, if not handled then the program compilation will fail.

Handling Exceptions using throw

The throw keyword indicates that the method may or may not throw an exception and that the caller must handle the exception accordingly.

```
public class Driver {
    private static void accessDB() {
        throw new ClassNotFoundException();
    }
    public static void main(String[] args) {
        accessDB();
    }
}

Exception in thread "main" java.lang.Error: Unresolved compilation
problem:
        Unhandled exception type ClassNotFoundException

        at ExceptionHandling.Driver.accessDB(Driver.java:5)
        at ExceptionHandling.Driver.main(Driver.java:8)
```

So the throws keyword allows us to delegate the exception handling to

the caller.

Handling Exceptions using try / catch

```
public class Driver {
    private static void accessDB() throws ArithmeticException {
        throw new ArithmeticException("This is a major exception,
company is down");
    }

    public static void main(String[] args) {
        try {
            accessDB();
        } catch(Exception e) {
            System.out.println(e.getMessage());
        } finally {
            System.out.println("Oh no!!");
        }
    }
}

This is a major exception, company is down
Oh no!!
```

Multiple catch blocks can be used to catch all exceptions thrown by the code in the try block. If we add a catch block to catch an exception which cannot be thrown by the try block then we'll get a compilation error [Unreachable catch block for FileNotFoundException. This exception is never thrown from the try statement bodyJava(83886247)].

In addition the following code will throw a compilation error:

```
   public static void main(String[] args) {
        try {
            accessDB(2);
        } catch(Exception e) {

        } catch(ClassNotFoundException e) {
            System.out.println(e.getMessage());
        } catch(InterruptedException e) {
            System.out.println(e.getMessage());
```

```
        }
    }
```

Unreachable catch block for ClassNotFoundException. It is already handled by the catch block for ExceptionJava(553648315)

Catching multiple exceptions in a single catch block.

```
public class Driver {
    private static void accessDB(int type) throws
ClassNotFoundException, InterruptedException {
        if(type == 1) {
            throw new ClassNotFoundException("This is a major
exception, company is down");
        } else {
            throw new InterruptedException("This is a slightly
less catastrophic exception");
        }

    }

    public static void main(String[] args) {
        try {
            accessDB(1);
        } catch(ClassNotFoundException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

The finally block will always be executed whether or not the code in the try block throws an exception or not, interesting even if we return from the try or method, even in that case the finally block will run.

```
public class Driver {
    private static void accessDB(int type) throws
           ClassNotFoundException, InterruptedException {
    }

    public static void main(String[] args) {
        try {
            accessDB(1);
```

```
            return;
        } catch(ClassNotFoundException | InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println("In the finally block");
        }
    }
}

Output:
In the finally block
```

There can only be one finally block.

Creating custom Exceptions

One thing to note is whether the custom exception created will behave as a checked (compile time) or unchecked (runtime) exception. This depends on the parent class our custom exception is extending from.
If we are extending from RuntimeException the custom exception will of the un-checked kind.
However if we extend the Exception class then the custom exception will be a checked exception.

```
public class CustomException extends RuntimeException {
    public CustomException(String message) {
        super(message);
    }
}

public class Driver {
    private static void accessDB(int type) throws CustomException
{
        throw new CustomException("Hi Hello database");
    }

    public static void main(String[] args) {
        accessDB(0);
    }
}
```

Notes: A method can always directly throw the Exception object, it is a checked (compile time) exception.

Collections Framework

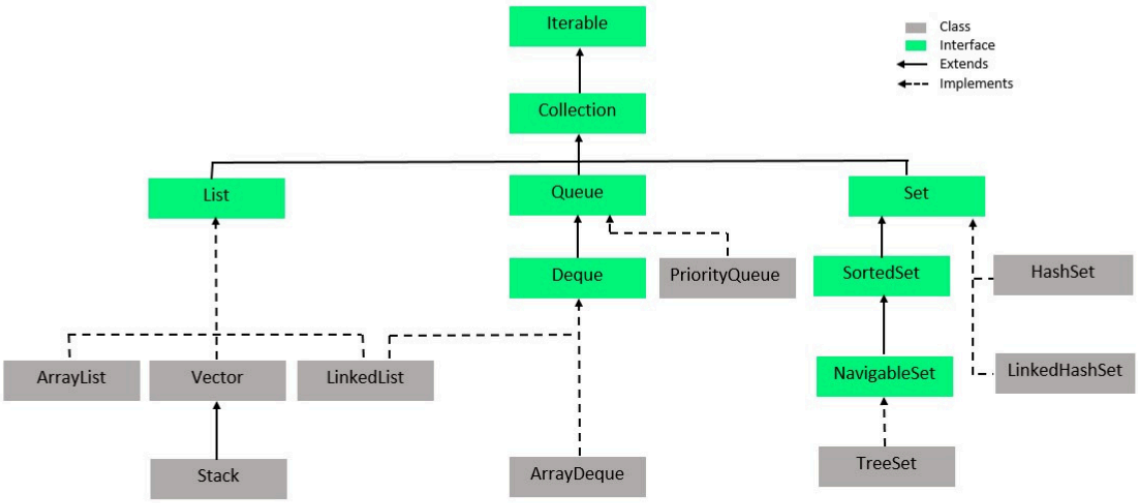
The Collections Framework was added in Java 1.2
=> A Collection is just a group of objects.
=> Present in java.util package

Framework provides the architecture to manage these "group of objects", i.e. CRUD operations, Add, Update, Delete, Search etc.

Why do we Need the Java Collections Framework (JCF)

Prior to the Java Collections Framework (JCF) Java had Array, Vector and HashTable.
=> The problem is that there is no common interface, so it is difficult to remember the methods for each of the collections.

Java Collections Framework hierarchy



Iterable Interface

The Iterable Interface was added in Java 1.5, it is used to traverse the collection.

Commonly used methods exposed by the Iterable Interface.

Method	Usage
iterator() [Introduced in Java 1.5]	It returns the iterator object, which

<table>
<tr>
<td></td>
<td>provides the following methods to iterate the collection.
<table>
<tr><th>Method</th><th>Usage</th></tr>
<tr><td>hasNext()</td><td>Returns true, if there are more elements in the collection.</td></tr>
<tr><td>next()</td><td>Returns the next element in the collection</td></tr>
<tr><td>remove()</td><td>Removes the last element returned by the iterator from the collection.</td></tr>
</table>
</td>
</tr>
<tr>
<td>forEach() [introduced in Java 1.8]</td>
<td>Iterate collection using lambda expression. The lambda expression is called for each element in the collection.</td>
</tr>
</table>

Further the iterable interface provides a for-each loop (enhanced for loop) to iterate the collection, any concrete class which implements Iterable can be iterated over by using a for-each loop.
Notice: Distinction b/w for-each loop and forEach() method.

forEach method internally uses a for-each loop. Iterate over all the elements using the for-each-loop and call the lambda expressions for each element.

Collection Interface

The Collection Interface was added in Java 1.2. The collection interface contains the iterator() method as well, before the introduction of Iterable in Java 1.5, the iteration was done by using the iterator() method exposed by the Collection Interface.

As mentioned before a Collection is just a group of objects, the Collection interface provides the methods to work with these groups of items.

Commonly used methods exposed by the Collection Interface.

Method	Since
size(), isEmpty(), contains(), add(), remove(), addAll(), removeAll(), clear(), equals(), toArray(), iterator()	Java 1.2
stream(), parallelStream() [Provide effective ways to work with collections like filtering, processing data etc.]	Java 1.8

Note: Java 1.x == Java x

Collection vs Collections

Collection: The Collection interface is a part of the Java Collections Framework, as mentioned it is an interface which exposes various methods which are implemented by the Collection classes, like ArrayList, Stack, PriorityQueue.

Collections: It is a utility class, consisting of various static methods, that can be used to perform operations on collections like Searching, swapping, reversing, sorting etc.

Queue Interface

The queue interface is a child of the Collection interface.
Generally a queue follows FIFO ordering, but there are exceptions like Priority Queue.

The queue interface supports all the methods available in the Collection interface + some additional methods like:

Method	Usage
add() / offer()	Insert an element into the queue => Cannot pass a null object as the parameter, it will throw a Null Pointer Exception.

	Return: The element was successfully inserted: True Element couldn’t be inserted: add(): Raise Exception offer(): Return false
remove() / poll()	Retrieves and removes the head of the queue. Return: If queue is not empty, return the element Queue is empty remove(): Raise Exception (NoSuchElementException) poll(): Return null
element() / peek()	Return the element at the head of the queue. Return: If queue is not empty, return the element Queue is empty element(): Raise Exception (NoSuchElementException) peek(): Return null

---

Priority Queue Concrete class

The priority queue class implements the Queue interface. There are two types of priority queues: Min Priority Queue and Max Priority Queue. Priority queue internally uses the heap data structure.

Elements are ordered according to the natural ordering by default, or by the Comparator if specified.

For example, for Integers the natural ordering is ascending, for Strings natural ordering is Lexicographical.

Comparator vs Comparable

Comparator and Comparable both provide a way to sort a collection of objects.

=> Arrays.sort internally uses the Quick Sort (dual pivoted) algorithm.

Need of Comparator / Comparable
=> Take an example of an array, we call Arrays.sort, it gets sorted in ascending order however how to sort in descending
=> If we have an array of objects and try to call Arrays.sort on it, then the method will throw a ClassCastException.

```
class Car {
    String name;
    String type;

    public Car(String name, String type) {
        this.name = name;
        this.type = type;
    }
}
```

```
Car [] cars = new Car[4];
cars[0] = new Car("hyundai", "SUV");
cars[1] = new Car("Tata", "Sedan");
cars[2] = new Car("Maruti", "HatchBack");
cars[3] = new Car("Kia", "Sedan");

Arrays.sort(cars);
```

Run:

```
java.lang.ClassCastException: class Car cannot be cast to class
java.lang.Comparable (Car is in unnamed module of loader 'app';
java.lang.Comparable is in module java.base of loader 'bootstrap')
```

Why this error?
The sort algorithm in the background will compare the elements in the collection and determine if they are in the correct order or not, if they are not they will be swapped. For example if we have 2 items a and b, where a occurs first in the collection and we wish to sort the collection in ascending order, then if a <= b then the 2 elements are already in the correct order, else swap a and b.

How does Java perform this comparison?
To compare two objects, java uses the compare method which needs to be provided by the object. Here the Car object has no compare method, hence Java can't compare the 2 objects. More generally the question is how to compare 2 objects? It is not as straightforward as comparing 2 primitives.

In summary, java needs to use a compare method to compare objects

and eventually order them. Since in our case the Car object has no compare method, hence when we try to sort the collection we get the error.

How can we add the compare method to the Car object, this is where the Comparator and Comparable come in, which are essential to sorting a collection of objects.

Comparator

The Comparator is a functional interface, which provides an abstract method compare.

int compare (T obj1, T obj2)

Where T is a generic object.

Method Returns:
1: If obj1 is greater than obj2 (in general any number greater than 0, would work)
0: If obj1 is equal to obj2
-1: If obj1 is smaller than obj2 (in general any number smaller than 0, would work)

The sorting algorithm uses the compare method provided by the Comparator interface to compare 2 objects, and subsequently decides whether to swap them or not.

If the method returns 1 (or a number > 0), then the sorting algorithm swaps obj1 and obj2.

In general if on comparison, we identify that
obj1 > obj2, swap obj1 and ob2
otherwise: carry the same order.

Example:
Arrays.sort(vals, (Integer a, Integer b) -> b - a);

Arrays.sort expects as its second argument the Comparator, which is a functional interface. We use a lambda expression and pass this function to the functional interface.

Some documentation for java.util.Arrays

public static <T> void sort(T[] a, Comparator<? super T> c)

Sorts the specified array of objects according to the order induced by the

specified comparator. All elements in the array must be mutually comparable by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the array). If no comparator is passed then c is null.

Comparable

The second argument to Arrays.sort(), i.e. c can be null, in this case how will the Sorting Algorithm be able to compare 2 objects? This is where the Comparable interface comes in, which provides an abstract method called compareTo.

int compareTo(T obj)

Just like the compare method, the compareTo method can be used to compare objects.

The Comparable interface needs to be implemented by the object itself, for example the Integer class implements the Comparable interface, and hence provides an implementation for compareTo.

If our object doesn't implement this method, and we don't specify a Constructor to Arrays.sort, then the aforementioned error is the result.

```
java.lang.ClassCastException: class Car cannot be cast to class
java.lang.Comparable (Car is in unnamed module of loader 'app';
java.lang.Comparable is in module java.base of loader 'bootstrap')
```

Comparator and Comparable are both used to sort a collection of objects, with Comparator we don't need to make any changes to the object class itself, we just need to specify the Comparator, in the call to the Arrays.sort or Collections.sort, using Comparator allows us to perform sorting in multiple ways.

On the other hand Comparable requires changes in the object class (as the class needs to implement the Comparable interface, and implement the compareTo method), and with a Comparable we can only have one kind of sorting, i.e. sorting by the Natural ordering, for example for Integers, the Natural Ordering is ascending, for Strings it is lexicographically increasing.

Deque Interface

The deque interface is a child of the Queue interface, hence it supports all the methods exposed by the Queue Interface as well as it adds some new methods exposed via the deque interface.

Hence it will support the normal Queue methods like add, remove, peak, poll, element and offer. However in addition it'll also support the following methods:

Insertion	addLast() addFirst() Throw exception if the insertion fails. offerLast() offerFirst() Return true / false
Removal	removeFirst() removeLast() Throw an exception if the dequeue is empty pollLast() pollFirst() Return null if the dequeue is empty
Examining	getFirst() getLast() Throw an exception if the dequeue is empty peekFirst() peekLast() Return null if the dequeue is empty

What about the existing add, offer, peek, element, poll, remove methods
add: Internally calls addLast
offer: Internally calls offerLast
peek: Internally calls peekFirst
element: Internal callsgetFirst
poll: Internally calls pollFirst
remove: Internally calls removeFirst

We can use deque to implement Stack as well as Queue

Dequeue provides 2 additional methods as well:

1. push(): Internally calls addFirst()
2. pop(): Internally calls removeFirst()

ArrayDeque Concrete class

The ArrayDeque class implements the Deque interface, i.e. it implements all the methods exposed by the Deque interface as well as the methods inherited by the Deque interface. ArrayDeque internally uses a resizable array, i.e. the elements of the deque are internally stored in an array, we maintain two pointers head and tail to allow insertions and deletions to take place at both ends. This is a major advantage of ArrayDeque as it allows us to insert / delete elements from beginning as well as end in O(1) time.

The array has some initial capacity, once it is filled up its size gets doubled, the size can be increased up to a certain limit beyond which Java will throw a OutOfMemory error.

Aromatized Insertion time complexity is O(1), most of the time the insertion takes place O(1), however if the array is full then it needs to be resized which takes O(n) time, however this operation is very rare.

PriorityQueue and ArrayDeque are not thread safe. In a multithreaded environment multiple threads can access simultaneously our data structure, which can potentially lead to race conditions and inconsistencies in data, a thread safe data structure will have the mechanism to guard against such inconsistencies. However, PriorityQueue and ArrayDeque are not thread safe, hence prone to race conditions. So how can we use Priority Queue and ArrayDeque in a multithreaded environment?

Instead of using a PriorityQueue, use a PriorityBlockingQueue.
And instead of ArrayDeque, use ConcurrentLinkedDeque.

All the methods are exactly the same.

In other words the PriorityBlockingQueue is the thread safe version of PriorityQueue
And ConcurrentLinkedDeque is the thread safe versions of ArrayDeque

Collection	Is Thread Safe	Allows duplicates	Maintains Insertion Order	NULL ELEMENTS ALLOWED
PriorityQueue	No	YES	NO	NO
ArrayDeque	No	YES	YES	NO

List Interface

The list interface is a child of the Collection interface. A list is itself a collection of objects, so how is it different from a queue. In a Queue the elements are inserted / removed and accessed from the start or from the end, there is no other option. A list by contrast allows data to be inserted / removed and accessed from anywhere, this is because the list makes use of index. Now the operations are no longer restricted to the 2 ends, instead they can be performed anywhere in the collection.

The List interface implements the Collection interface hence all the methods exposed by the Collection interface are also available here, in addition it adds some methods of its own.

Methods exposed by the list interface

get(index)	GET The element at the specified index
add(index, element)	Insert the element at the specified index, moving all the required elements one position to the right.
sort(comp)	Sort the collection in the order induced by the comparator
set(index, newElement)	Modify the element at index = index, to newElement
remove(index)	Remove the element at the specified index and move the subsequent elements to the left by one position.
indexOf()	Return the index of the first occurrence of the element in the

	collection. -1 if the element is not present in the list.
lastIndexOf()	Return the index of the last occurrence of the element in the collection. -1 if the element is not present in the list.
listIterator()	Similar to the method iterator() exposed by the Iterable and Collection interface. This method returns a ListIterator object which is used to iterate the list. The ListIterator extends from the Iterator. As a result it supports all the methods supported by the Iterator like hasNext(), next() and remove(). However in addition to these ListIterator also provides methods to iterate the list in reverse (backward) order. hasPrevious(), previous()
subList(start, end)	Splice the list, and get the subList from start to end - 1 This is a shallow copy, any changes in the sublist will reflect in the main list and vice versa.

ArrayList Concrete class

The ArrayList class implements the List interface, i.e. it will implement the methods exposed by the List interface.

ArrayList internally uses the array data structure with some initial capacity to store the elements, and if the array gets filled up it will be resized, and the size of the array will be doubled.
Time Complexities:

Inserting an element to the end of the list, and space is sufficient: O(1)
Inserting an element at the beginning of the list, and space is sufficient: O(n) [Shifting of values]

Inserting an element to a particular index: O(n) [As it requires shifting of elements]

Inserting an element, when array size has reached the threshold: O(n) [As array will need to be resized, which is an O(n) operation as it involves copying the elements to a bigger array].

Deletion: O(n), again involves shifting of values.

ArrayList summary box
Internal Data Structure: Arrays Maintains Insertion Order: Yes Is Thread Safe: No Allows NULL elements: Yes Insertion Complexity: O(1) end of the list, O(n) beginning. Caveats: Since it is implemented using an array it involves the complexity of having to resize the array and shifting multiple elements in case of insertion / deletion at some arbitrary index.

ArrayList is not thread safe so it is prone to race conditions, so how do we use it in a multithreaded environment? What is the thread safe version of ArrayList<>

Use CopyOnWriteArrayList instead of ArrayList, which is the thread safe variant of ArrayList.

LinkedList Concrete class

The LinkedList class implements the List interface as well as the Deque interface.

LinkedList collection internally uses the Linked List data structure.
Since it implements both Deque and List interfaces, it provides implementation for methods like addFirst, pollFirst, addLast, pollLast from the Deque interface as well as get(index) and set(index, value) etc. from the List interface.

Since it internally uses the LinkedList data structure hence there is no need for shifting

Time Complexity
Insertion / Deletion at start and end: O(1)
Insertion at arbitrary index: O(n) for lookup, O(1) for insertion.
Deletion at arbitrary index: O(n) for lookup, O(1) for deletion.
Search: O(n)

LinkedList summary box

Internal Data Structure: Linked List
Maintains Insertion Order: Yes
Is Thread Safe: No
Allows NULL elements: Yes
Duplicate Elements Allowed: yes
Insertion Complexity: O(1) end of the list, O(1) beginning.
Caveats: Since it is implemented using a Linked List It involves the complexity of having to loop through the linked list to perform any index related operation or lookups, so we are losing O(n) to get to the required index.

Thread safe version of ArrayList is the CopyOnWriteArrayList.

Vector Concrete class

Vector class implements the list interface, and is exactly the same as ArrayList, and since it implements list interface it has indexing capabilities. However Vector is thread safe, which ArrayList and LinkedList are not.

It puts a lock when an operation is performed on the vector. As a result it is thread safe, however this makes it less efficient than ArrayList, as for each operation it will do lock / unlock internally.

Vector summary box

Internal Data Structure: Array
Maintains Insertion Order: Yes
Is Thread Safe: YES
Allows NULL elements: Yes
Duplicate Elements Allowed: yes
Insertion Complexity: O(1) end of the list, O(n) beginning.
Caveats: Since it is implemented using an array it involves the complexity of having to resize the array and shifting multiple elements in case of insertion / deletion at some arbitrary index required index.

Stack class

Stack extends the vector class, since it extends the Vector class it is also thread safe.

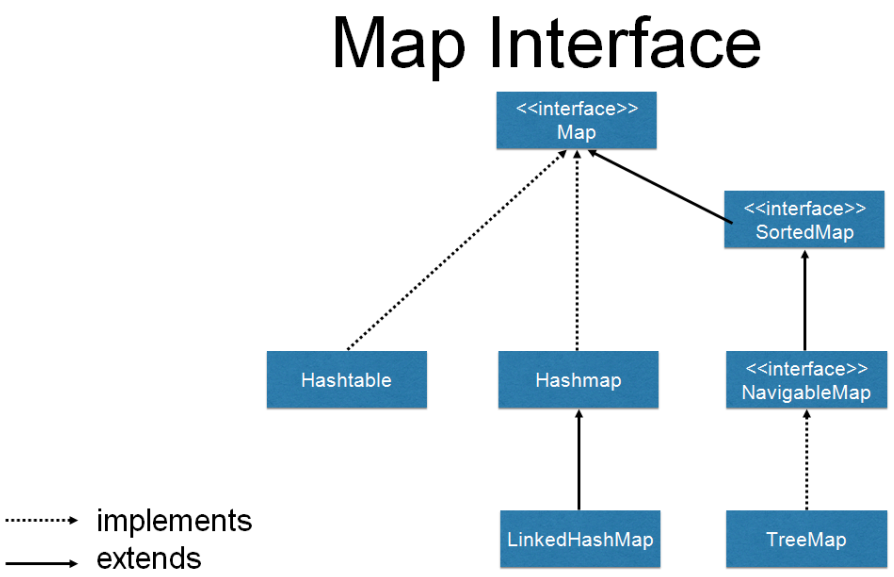
As mentioned in the Deque section, we can use a Deque to implement Stack, however Deque is not thread safe but Stack is.

Time Complexity
Insertion / Deletion: O(1) [Always done at the top]
Search: O(n)

Stack summary box Extends the Vector Class Is Thread Safe: YES Allows NULL elements: Yes Duplicate Elements Allowed: yes

Collection	Is Thread Safe	Allows duplicates	Maintains Insertion Order	NULL ELEMENTS ALLOWED
ArrayList	No	YES	YES	YES
LinkedList	No	YES	YES	YES
Vector	YES	YES	YES	YES
Stack	YES	YES	NO	YES

Map Framework



Maps are implemented using a different framework, they are not a part of the Collection framework, as Maps have very different methods than say a list, this is because maps deal with Key, Value pairs.

Map is an interface, and it has the following implementations

1. HashMap
2. TreeMap
3. LinkedHashMap
4. HashTable

Note: A map cannot contain duplicate keys.
=> Values can be duplicated.

Map Interface

The Map interface was introduced in Java 1.2, Methods exposed by the Map Interface:

size()	
isEmpty()	
containsKey(key)	RETURN: TRUE - Specified key is present in the map.

	FALSE - Key not found
get(key)	Return the value corresponding to this key.
put(key, value)	If the map has the key already present then the value corresponding to it will be overwritten with the new value. Else, if not present simply add the new key - value pair to the map.
containsValue(value)	Return true if map contains the specified value mapped to any key.
putIfAbsent(key, value)	
remove(key)	
keySet()	
values()	

The map interface has a subinterface called Entry, which can be accessed only via the Map interface (Map.Entry).

HashMap Concrete Class

HashMap class implements the Map interface, and the methods it exposes.

How are HashMaps internally implemented?

Internally HashMap stores the data (K, V pairs) in an array. In case of a hash map each array slot will contain an object of type Node<K,V>. As mentioned before, map has a subinterface called Entry (Entry<K, V>). Node<K,V> implements the Entry<K,V> sub-interface.

Note# In case of a hashmap the value can be null as well as the key. i.e. hashMap.put(null, null) is valid

So essentially in the case of a HashMap data is stored in an array of Node<K,V>'s.

What does the class Node<K,V> consist of?

It contains the following 4 attributes:
(int) hash, (K) key, (V) value,(Node <K, V>) next

What is the default size of this array (called as table in the implementation)?
It is 16 (1 << 4),
When we create a HashMap like this, without specifying any capacity
Map<Integer, String> map = new HashMap<>()
then by default, the map will have a default initial capacity of 16, which is the size of the table array.

However we can specify the capacity we want while creating the hash map, instead of using the default initial capacity.

Which index to store the entry to?

When we insert a new K, V mapping to the map, how is the data actually stored in the array and at what index.
We use a hash function, which given a key (x) will produce a hash value hash(x). This hash function generally produces a large value, next we take the mod of this value with S (The table size).

hash(x) % S, is guaranteed to be a number in the range [0, S - 1], which is used as the index, and this is where the entry data will be stored.

In other words the hash function will map the key to a specific index in the hash table.

Insert K x, V v => i= hash(x) % s => Insert <K, V> at index i in the table array.

Index identified, how is the data stored in the array?

As mentioned before, we are using an array of Node<K, V> to store the hashmap entries. Once we have determined the index where the entry has to be stored (by using the hash function), we can create a Node<K,V> object and put it in the array at the determined location.

What are the attributes of the Node<K, V> object
=> key: K(from: K, V)
=> value: V (from K, V)
=> hash: INTEGER (the output of the has function, before we performed the mod)
=> next: Node<K,V> (Chaining)

Example we wanted to insert the entry {1, "Jack"}
and say hash function produces the output 1234557

On performing hash(x) % S, say we get the index i.

Then we insert the following data in the table array at index i.

```
table[i] = {
    key : 1,
    value : “Jack”,
    hash : 1234557,
    next: null
}
```

The use of the next attribute and Collisions

A collision is said to occur when for 2 keys k1, k2 (k1 != k), hash(k1) % S == hash(k2) % S, i.e. both k1 and k2 are mapped to the same index by the function. In such a scenario the next attribute of the Node<K,V> object comes into play.

To handle the collision we can use the strategy of Chaining, where all the Entries which have been mapped to the same index by the hash function will be chained together as a Linked List. The array index will point to the head of this linked list, and subsequent elements will be chained together via the next pointer.

Array Indexes	Linked List (chain of entries)
0	-> {k1, v1} -> {k2, v2}
1	-> NULL
2	-> {k1, v3}-> {k2, v1}-> {k3, v3}
3	-> NULL
4	-> NULL

Retrieving values?

How do we retrieve the value corresponding to a key k.
Again we use the hash function to determine the index where the entry corresponding to k will be stored. As mentioned before each element of the array points to the head of the linked list. Hence once we determine the index, we’ll need to iterate over this list and then find the entry with the matching key, if found the corresponding value will be returned.

Contract b/w hashcode and equals method

The hashCode method is used to compute the hash value for a given key. equals method is used to compare 2 objects.

What is the contract?

1. If obj1 == obj2, then hash(obj1) == hash(obj2), which implies that if we pass the same object (or key) through the hash function multiple times then we will always get the same hash value.
2. If the hash of 2 objects is the same, then they may or may not be the same object.

HashMap Performance

The hashmap offers average O(1) time complexity for searching / Insertion and Deletion

However worst case time complexity for searching / Insertion / Deletion is O(n)

Load Factor - The concept of load factor has been discussed before, it is defined as the total number of elements in the hashmap divided by the size of the table (array).

By default if this value exceeds 0.75, then rehashing takes place. Rehashing involves increasing the size of the table array, more specifically doubling the size of the array, as the size of the table must always be a power of 2, and the elements will be rehashed to new indexes in the resulting table.
Load Factor doesn't let the chains grow too long.

However this doesn't protect us from non-uniform distribution, what if there is sufficient space in the table, but all the elements we are inserting are all getting mapped to the same index. In this case even though there is vacant space in the table array, the insertion / searching / deletion time complexity will be O(n). How does Java handle such a situation?

It does so by using a TREEIFY_THRESHOLD, the idea is that if the chain is getting too long, it'll be better to store it as a self balanced BST, instead of storing it simply as a linked list. Using a BST, we can get O(logn) performance instead of O(n).

The value for the TREEIFY_THRESHOLD is 8, so as long as the chain size is smaller <= 8, it'll be stored as a linked list. However if it increases any further then the linked list will be converted to a tree. This tree is a balanced BST like AVL, or Red Black Tree.

So the worst time complexity for insertion / searching / deletion will be O(logn) if we are using a tree to store the chain / nodes or bins (that is what the source calls them). In case of using a linked list the worst TC is O(n)

HashMap Summary Box
Can Contain duplicate keys: NO Is Thread Safe: NO Maintains Insertion Order: NO Allows null key: YES Allows null values: YES

HashMap is not thread safe, so in a multithreaded environment we can use its thread safe version ConcurrentHashMap or HashTable.

HashTable Concrete Class

HashTable class implements the Map interface, it is almost exactly the same as HashMap, however it is thread safe.

HashTable does not allow null keys or null values.

HashTable Summary Box
Can Contain duplicate keys: NO Is Thread Safe: YES Maintains Insertion Order: NO Allows null key: NO Allows null values: NO

LinkedHashMap Concrete Class

The LinkedHashMap class extends the HashMap class, it was introduced in Java 1.4. It has similar methods as HashMap, however it provides ordering, as it maintains the insertion order and access order.

LinkedHashMap internally uses a doubly linked list.

Insertion Order - The order in which the elements are accessed is the same as the order in which they were inserted.

Access Order - Most frequently accessed elements are present in the last, while less frequently accessed elements are present at the head.

An entry in this doubly linked list will consists of the following attributes:
=> Key k,
=> Value v,
=> hash
=> next
=> after
=> before

As mentioned, LinkedHashMap extends the HashMap class, it uses the HashMap.Node<K,V> and adds 2 additional fields: after and before.

When we insert an element to the LinkedHashMap, first similar to a HashMap a hash function will be used to compute the index where this element will be stored, however the data is organized as a doubly linked list.

next is used for chaining in case of collisions
while after and before link the doubly linked list

Time complexity and performance is the same as HashMap.

However, LinkedHashMap is not thread safe and there is no thread safe version available for it, however if we want a thread safe LinkedHashMap we can use the **Collections.synchronizedMap** method, which will take our LinkedHashMap as an argument, all the operations will be performed on the LinkedHashMap only, however synchronizedMap method will actually call the LinkedHashMap methods in a synchronized block.

From the documentation:

This class provides all of the optional Map operations, and permits null elements. Like HashMap, it provides constant-time performance for the basic operations (add, contains and remove), assuming the hash function disperses elements properly among the buckets. Performance is likely to be just slightly below that of HashMap, due to the added expense of maintaining the linked list, with one exception: Iteration over the collection-views of a LinkedHashMap requires time proportional to the size of the map, regardless of its capacity. Iteration over a HashMap is likely to be more expensive, requiring time proportional to its capacity.

LinkedHashMap Summary Box Can Contain duplicate keys: NO Is Thread Safe: NO Maintains Insertion Order: YES [Maintains Access Order: YES] Allows null key: YES Allows null values: YES

SortedMap Interface

The SortedMap Interface extends the Map interface.
It supports all the methods exposed by the Map interface as well as some additional methods:

Method	Usage
headMap(key)	Get all the entries in the map with the key smaller than the specified one
tailMap(key)	Get all the entries in the map with greater than or equal the specified one
firstKey()	Get the Smallest key in the map
lastKey()	Get the largest key in the map

NavigableMap Interface

The NavigableMap is a child of the SortedMap interface, so it supports all the methods exposed by the SortedMap interface as well some additional ones:

lowerEntry(key) - Get the entry with the greatest key smaller than the specified key
floorEntry(key) - Get the entry with key == specified key or with the greatest key smaller than the specified key.
lowerKey(), floorKey()

ceilingKey()
ceilingEntry()
higherEntry()
higherKey()
firstEntry(), lastEntry()
pollFirstEntry(), pollLastEntry()
headMap(key, inclusive: boolean), tailMail(key, inclusive: boolean)

TreeMap Concrete Class

The TreeMap class implements the NavigableMap interface.
TreeMap stores the elements in a sorted order by the key, according to the natural ordering of the key or by the Comparator provided during map creation.

Internally it uses the Red Black Tree (Self Balancing BST) data structure.

Time Complexity:
Insertion: O(logn)
Remove: O(logn)
Searching / Lookup: O(logn) - As the data is stored in BST, so the time complexity is O(logn)

As can be seen, TreeMap is slower than LinkedHashMap, HashMap and HashTable.

TreeMap Summary Box

Can Contain duplicate keys: NO
Is Thread Safe: NO
Maintains Insertion Order: NO
Allows null key: NO
Allows null values: YES

TreeMap is not thread safe, the thread safe version is ConcurrentSkipListMap.

Set Interface

The Set Interface is a child of the Collection interface.

=> A set is a collection of objects, but it does not contain duplicate values.
=> Unlike List, Set is not an ordered collection, i.e. it doesn't follow the insertion order.
=> Unlike List, Set cannot be accessed via index.

HashSet Concrete Class

The HashSet implements the Set Interface, and all methods exposed by it. Internally HashSet uses the HashMap data structure.

In the case of a set we are only concerned with the values, while in the map we have key value pairs. So how can we use a map to store the Set data? The elements in the set are internally stored as the keys of a hashMap, for the value of the entry in the hashMap we just use some dummy Object (new Object())

set.add(35) => map.put(35, new Object())

HashSet Summary Box

Can Contain duplicate keys: NO
Is Thread Safe: NO
Maintains Insertion Order: NO
Allows NULL elements: Yes, but only once.

HashSet is not thread safe, the thread safe version of HashSet is the newKeySet method present in the ConcurrentHashMap.

```
ConcurrentHashMap map = new ConcurrentHashMap();
Set<Integer> set = concurrentHashMap.newKeySet();
```

Average time complexity of Insertion / Deletion / Searching: O(1)

LinkedHashSet

Internally it uses the LinkedHashMap hence it maintains the insertion order.

LinkedHashSetSummary Box Can Contain duplicate keys: NO Is Thread Safe: NO Maintains Insertion Order: YES Allows NULL elements: Yes, but only once.

LinkedHashSet is not thread safe, and it doesn't have a thread safe version either, so instead we use the Collections.synchronizedMap() method, which will perform all the operations of the LinkedHashSet in synchronized blocks.

TreeSet

Internally it uses the TreeMap, it cannot store null values

It stores the elements in a sorted order, the order is based on the Natural Ordering of the elements in the set, or on Comparator if one is specified during set creation.

Collection	Thread Safe Version	Type of Lock Used

PriorityQueue	PriorityBlockingQueue	ReentrantLock
LinkedList	ConcurrentLinkedDeque	Compare And Swap
ArrayDeque	ConcurrentLinkedDeque	Compare And Swap
ArrayList	CopyOnWriteArrayList	ReentrantLock
HashSet	newKeySet method inside ConcurrentHashMap	Synchronized
TreeSet	Collections.synchronizedSortedSet	Synchronized
LinkedHashSet	Collections.synchrnonized Set	Synchronized
Queue Interface	ConcurrentLinkedQueue	Compare And Swap

Child Class can be stored in the parent reference.
Parent class can hold an object of the child class.
Interface can hold the reference to the object.
Child object can be cast to the Parent type.

Reflection and Annotations

Reflection is used to examine Classes, Interfaces, methods, fields at run time, and we can also change the behavior of the class (for example change the field values at runtime).

What we can examine, take the example of a Class
- What all methods are there in the class
- What all fields are there in the class
- Modifier of class
- What all interfaces class has implemented.

Further:
We can change the private / public fields of the class using Reflection
Call methods using Reflection

Notes about class Class.
For doing reflection we need an object of Class, this class 'Class' represents a class during runtime.
=> JVM creates one Class object for each class loaded during runtime.
=> JVM loads the classes as and when needed.
=> This class 'Class' has metadata about the class it represents, like the methods of the class, fields of the class etc.

So how to get the Class object?
Notice we cannot simply do a "new Class", because the class object actually represents another class (say Employee, Animal etc.) during runtime and thus is created by the JVM.

class Bird{}

Method: 1
=> Using forName() method
Example: Get the Class object for our class 'Bird'

```
Class birdClass = Class.forName("Bird")
```

Method: 2
=> Using .class
Example:

```
Class birdClass = Bird.class
```

Method: 3
=> Using getClass()
Works on objects of the class

```
Bird bird = new Bird();
Class birdClass = bird.getClass();
```

Note again, we can only get the 'Class' object, we cannot create it; that is done by JVM.

Once we get the object of Class for the class which we want to Reflect, we can invoke methods on this Class object to get the metadata it stores.

Note:
=> Reflection violates Encapsulation.
=> Reflection breaks Singleton

Quick Note on Annotations:

We can create our annotations by using @interface.
While creating an annotation we need to specify two meta annotations.
=> Meta annotations are annotations applied on top of annotations.
The meta annotations are:

- Target: This meta annotation specifies where the annotation can be applied. Class / Method / Constructor / Attributes etc.
- Retention: This meta annotation specifies how the annotation will be stored in Java, possible values:
    - SOURCE: Annotation discarded by Compiler itself, hence not recorded in .class file. Useful for readability purposes.
    - CLASS: Annotation recorded in .class file, but ignored by JVM during run time.

- RUNTIME: Annotation recorded in .class file, and also available during runtime.

How are annotations processed / used by JVM?
=> Through Reflection.

SpringBoot

Introduction and Need

Servlets provide the foundation for building web applications. A servlet is a java class which handles client requests, processes it and returns the response.
Servlet containers are used to manage a group of servlets.

=> Web.xml file is used to perform servlet mapping, i.e it stores the URL to servlet mapping, for example if the user has requested the URL "/home" then which servlet should handle this request? This information is stored in the Web.xml file.

Applications built using servlets were packaged as WAR files and deployed on servers like TomCat, which acts as the Servlet Container.

Spring framework was introduced to overcome the shortcomings of

servlets.

- Removal of Web.xml, Spring Framework introduced Annotations based configuration.
- IOC (Inversion of Control) - IOC is a flexible way to manage object dependencies and lifecycles. Dependency Injection is an implementation of IOC (note the usage of @Autowired and @Component).

With Spring Framework (Spring MVC), we still need to deploy the application as WAR on TomCat, i.e. the user needs to handle the packaging and deployment.

Flow of request / response with Spring MVC

=> User request comes in to the Servlet Container (TomCat server, where the application is deployed)
=> Next the request is sent to the Dispatcher servlet (known as the first controller). We can define several controllers, by using the @Controller annotation. Dispatcher Servlet will determine which Controller needs to handle the incoming request.
=> Once the Dispatcher Servlet determines which controller needs to handle the request it will need to create an instance of that controller, it will use IOC for this purpose to resolve the dependencies of the controller class.
=> Next Dispatcher servlet will invoke the controller method which matches the request path.
=> The controller method will process the request and return a response.

Spring Boot overcomes the challenges which exist with the Spring MVC.

- Dependency Management: No need to specify individual dependencies and manage their versions. Spring Boot basically abstracts the details of Spring MVC.
- Auto Configuration: No need to specify dispatcher servlet, App Config or Component Scan. Spring Boot handles these details internally by default. Spring Boot is opinionated.
- Embedded Server: We do not need to build a WAR and deploy it ourselves to the TomCat server (Servlet Container), this is because SpringBoot comes with an embedded Tomcat, we can just run the application.

Spring Boot in summary:
- Provides a way to build production ready applications quickly.
- It is based on the Spring framework, internally it still uses the Spring framework, but provides the developer a simplified interface.
- Convention over Configuration (Auto Configuration)

JAR: Java Archive = Standalone Java Application and it's libraries
WAR: Web Archive = Java Application + HTML / CSS / JS + JSP etc.

Layered Architecture:
The SpringBoot application is structured in three layers:

- Controller Layers - The controller layer consists of classes annotated with @Controller or @RestController. The controller layer provides the endpoints for client requests.
- Service Layer - The service layer contains the business logic of the application, classes in this layer are generally annotated with @Service.
- Repository Layer - Handles the details of Database interaction. Annotated with @Repository.

Entity Classes (@Entity) are POJO classes which are a direct representation of the database tables.

Group ID, Artifactid and version together uniquely identify our projects in the Maven Central.

Maven

Maven is a project management tool which helps developers with build generation, dependency resolution, documentation etc.

=> With Maven we only need to specify what to do, Maven takes care of how to do it.
=> Maven uses POM (Project Object Model to achieve this), the pom.xml file in the root directory of the project stores the configurations.
=> The spring boot starter parent provides the Maven defaults, the pom.xml inherits the properties from the parent's POM.

If not for Maven:
Whenever we build a project we need other dependencies like Hibernate, Commons Logging, JSON etc. One approach would be to download the JAR file from each project website and then add the JAR files to the

classpath / build path

On the other hand, Maven will take the dependencies and download them and make them available during compile / run time, i.e. it will handle the build / class path.

Annotations

@Controller, @RestController - This class is responsible for handling incoming HTTP requests

RestController = Controller + ResponseBody

Other annotations:
=> ResponseBody
=> RequestMapping / [GetMapping / PostMapping etc. in case of RestController]
=> PathVariable - /api/users/{id} -> path variable
=> RequestParam - /api/cities?country=india&season=winter -> Request Params
=> RequestBody

Beans

Bean is a Java Object, which is managed by Spring Container (IOC container). IOC Container contains all the beans created and manages them.

Creating a Bean
- @Component: When we mark a class with this annotation, we tell Spring to create a bean of this class and manage it. It uses convention (Auto Configuration).

- @Bean: We can use this annotation if we want to specify some configuration details. These details will be used by Spring when creating the bean.

How does Spring find what beans need to be created?
It does this by performing a component scan, and searching for classes annotated with @Component or classes annotated with @Configuration (as they contain Bean declarations).

When are the beans created?
Beans can be created in an eager or lazy manner
Eager - Beans are created as soon as the application starts.

Lazy - Beans are created only when needed.

Bean Scopes - Singleton OR Prototype

=> Singleton Scoped Beans are eagerly created
=> Prototype Scoped Beans are lazily created
We can use @Lazy annotation with Singleton beans to create them in a lazy manner.

Default Scope is "Singleton"

Using Dependency Injection we can decouple our classes from their dependencies.
=> Dependency Injection is used to inject dependencies into the constructed bean.
=> @Autowire tells Spring to look for a particular object / Bean in the IOC container.
=> If the bean is found in the container, it is injected, otherwise the bean is created and then injected.

Types of Dependency Injection:

- FieldInjection (Dependencies are resolved at a field level)
- ConstructorInjection (dependencies are resolved at the time of object creation itself).
- SetterInjection (dependencies are resolved via setter methods)

In case of Dependency Injection an external source (in this case Spring) injects the dependencies.

Bean Scopes

Singleton - Default Scope, only one instance of the bean is created per IOC container.
=> Eagerly initialized.

Prototype - Each time a new object is created.
=> Lazily Initialized, i.e. bean is only created when it is needed.

Request - New Bean is created for each HTTP request.
Lazily initialized.
=> For a specific request, a bean with the scope of Request will be created only once.

Session - Session scope is similar to Request scope, a new bean is created for each HTTP session.
Lazily Initialized

Bean Lifecycle

Application Starts ->
SpringBoot initialises the IOC container ->
Spring scans and finds all the classes annotated with @Configuration or @Component within the root package ->
Construct the bean (in case of eager initialisation) ->
Inject the dependencies into the constructed bean ->
@PostConstruct (perform initialization) ->
Use the bean ->
@PreDestroy (perform Cleanup) ->
Bean is Destroyed

@ConditionalOnProperty Annotation

Create beans conditionally.

@ConditionalOnPropery(prefix="<>", value="<>", havingValue="<>", matchIfMissing=<true/false>)

application.properties file defines settings in the following format:
prefix.value = havingValue

=> havingValue is a string

@Profile Annotation

@Profile helps in environment separation, we might have different configurations in different environments.
The configurations are stored in the application.properties file.

application.properties:
    application-{profile1}.properties,
    application-{profile2}.properties,
    application-{profile3}.properties

Where each profile corresponds to an environment. So instead of having a single application.properties file we can create multiple environment specific profiles (application-{profile}.properties).

application.properties file is the default configuration / profile, to change the profile, modify the spring.profiles.active key in application.properties

The value of spring.profiles.active can be dynamically set via CLI at the time of application startup.

CLIs:

Start the springboot application -
mvn spring-boot:run

Set the value of spring.profiles.active
mvn spring-boot:run -Dspring-boot.run.profiles=prod

Using the @Profile annotation we can tell spring boot to only create a bean when a particular profile is set.

Use @ConditionalOnProperty instead of @Profile for conditional creation of beans.

Aspect Oriented Programming (AOP)

Aspect Oriented Programming allows us to intercept method invocation, and we can perform some tasks before and after the method.

=> AOP handles boilerplate and repetitive code like logging, transactions etc.

=> Aspect is the module which handles this boilerplate / repetitive code.

=> AOP provides code reuse and better code maintainability.

This task which we need to perform before / after the method is known as advice.

Pointcut is an expression which tells where the advice should be applied.

Types of Advice: Before, After and Around.

Types of Pointcut:

1. Execution - Matches a particular method in a particular class.
2. Within - Matches all the methods within a particular class or package.
3. @within - Matches all the methods in a class which has the specified annotation.
4. @annotation - Matches all the methods with the specified annotation.
5. Args - Matches all the methods with the specified arguments.
6. @args - Matches all the methods where the specified arguments have the specified annotation.

7. target - Matches all the methods called on a particular instance of a class.

@Before and @After are part of advice (not pointcut)
The advice (action) runs before / after or around the method.

Note about @Around, around is basically before + after. In other words @Arround basically surrounds the method execution. If we are using Around then we need to take care of method invocation ourselves, for doing this we use JoinPoint.

JoinPoint - It's the point where the actual method invocation happens.

Flow of how interception works with AOP:

1. Scan for classes annotated with @Aspect
2. Parse the pointcut expressions in these classes
3. Store the parsed pointcut expressions in some efficient data structure or cache.
4. Look for other classes @Component, @Controller, @Servcie etc. and check if they are eligible for interception.
5. If the class is eligible for interception (based on pointcut expressions) create a proxy for that class using JDK Dynamic Proxy or CGLIB proxy. The proxy class contains code which executes the advice before method invocation, and after it if there is any. The proxy class actually overrides the methods of the original class which need to be intercepted (in case of CGLIB).

@Transactional

The Transactional annotation can be applied at class level and method level. If we apply at class level, then it will be automatically applied to all the public methods (not private methods though).

The Transactional annotation handles details of transactions like BeginTransaction, EndTransaction, Rollback, Commit and we can focus on the business logic.

@Transactional internally uses the concept of AOP, and it uses "Around" type of advice. Transactional annotation invokes an interceptor in the background which provides the advice.

Transaction Managers - The transaction managers help to talk to the database, and manage the transaction. For example JDBC TransactionManager, Hibernate TransactionManager, JPA TransactionManager, DataSource TransactionManager etc.

When we use @Transactional, spring automatically picks a TransactionManager (generally JPA), however if we want to we can change the TransactionManager by creating a bean of PlatformTransactionManager.

The Concrete transaction Manager mentioned above implements the PlatformTransactionManager interface.

Types of Transactional Management:

- Declarative: In Declarative Transaction Management, Spring handles the details of transactions, and we can just focus on the business logic.
- Programmatic: In Programmatic transaction management, we need to manage the transaction ourselves, it is more flexible but difficult to maintain.
    - Option 1: Use the TransactionManager directly
    - Option 2: Use transaction template

Transaction Propagation Options:

- REQUIRED (default): If parent transaction is present use it, else create a new transaction. If not present, create a new transaction.
    - For example, method1 annotated with @Transactional internally calls method2 also annotated with @Transactional

- REQUIRED_NEW - Suspend parent transaction, create a new transaction and once finished resume the new transaction. If the parent transaction is not present, create a new transaction.

- SUPPORTS - If parent transaction is present use it, else execute the method without any transaction.

- NOT_SUPPORTED - If parent transaction is present, suspend it and execute the method without any transaction, once finished resume the parent transaction. If the parent transaction is not present, execute the method without any transaction.

- MANDATORY - If parent transaction is present, use it else throw an exception.

- NEVER - If parent transaction is present throw an exception, else execute the method without any transaction.

Notes on Isolation Levels

Isolation Level tells how changes made by one transaction are visible to

other transactions running in parallel. Default isolation level depends on the database in use.

First we consider a few problems relating to Isolation:

1. Dirty Read Problem: A dirty read happens when a Transaction A reads the uncommitted data of another transaction B. If B is rolled back, then A will have wrong / stale data, this is known as a Dirty Read.
2. Non Repeatable Read - If suppose a transaction A reads the data from the same row multiple times as part of the transaction, and there is a chance that it gets different values then it is known as Non-Repeatable read.

   => Begin Transaction
   => Read row, value is y
   => Some operations ….
   => Read row, value is z! [some other txn updated row value to z]
   => End Transaction
3. Phantom Read Problem - If suppose transaction A executes some query multiple times as part of the transaction, and there is a chance that it gets different number of rows then it is known as Phantom Read Problem.
   => Begin Transaction
   => Execute query SELECT * from table where id > 5, 10 columns returned
   => Some operations…
   => Execute query SELECT * from table where id > 5, 15 columns returned!
   => End Transaction

Shared and Exclusive Locks:

Shared Lock (S) also called Read Lock can be acquired by multiple transactions simultaneously to read a resource (for example a database row). Shared locks are meant for reading only. If one or more transactions have acquired a shared lock on a resource, then no other transaction can get an exclusive lock on that resource at the moment.

Exclusive Lock (X) also called Write lock can be acquired by only a single thread. Once a transaction has acquired an exclusive lock on a resource, no other transaction can acquire a shared or exclusive lock on that resource.

Isolation Level	Description of Locking	Dirty Read	Non Repeatable	Phantom Read

		Problem exists	Read Problem exists	Problem exists
Read Uncommitted	Read: No lock is acquired Write: No lock is acquired Highest level of concurrency, as there is no locking but obviously suffers from race conditions and all of the problems mentioned above. Use this isolation level for read-only resources.	YES	YES	YES
Read Committed	Read: Shared lock is acquired and released as soon as read is done. Write: Exclusive lock is acquired and kept till the end of the transaction. Why is the write lock kept till the end of the transaction? A transaction consists of multiple operations, to perform these operations we need to take locks on multiple resources. For writing to a resource we need an exclusive lock. The problem with releasing the exclusive lock as	NO	YES	YES

	soon as the write is done is that the transaction is still not complete, only one of its steps writing to some resource is complete. But the transaction itself is still going on and there is a chance it might get rolled back. Hence if we don’t keep exclusive lock till the end then we’ll face the Dirty Read problem again. If the transaction in question keeps the write lock till the end, then any other txn will not be able to acquire a shared lock on the resource, hence not read it and hence no dirty reads.			
Repeatable Read	Read - Shared lock is acquired and kept till the end of the transaction. Write - Exclusive lock is acquired and kept till the end of the transaction.	NO	NO	YES
Serializable	Locking strategy is the same as Repeatable Read + apply Range lock which is only released at the end of the transaction. Range lock is a type of shared lock.	NO	NO	NO

	Obviously the level of concurrency is very low.			

Read Committed and Repeatable Read are commonly used.
This transaction code is present in the DBTransactionManager, each database provides its own transaction manager. Different databases support different isolation levels. So these details (shared locks / exclusive locks, locking / releasing etc.) are handled by the database transaction manager, the application layer is not concerned with these details.

@Async Annotation

=> The async annotation is used to mark methods that should run asynchronously.
=> Running Asynchronously means running on another thread, hence not blocking the main thread.

During application startup Spring checks if there is any default executor, if it finds that there is no ThreadPoolTaskExecutor Bean specified in the configuration classes then it will create a default ThreadPoolTaskExecutor, with the following config:
corePoolSize = 8
maxPoolSize = INT_MAX
queueSize = INT_MAX

ThreadPoolTaskExecutor is just a wrapper around the ThreadPoolExecutor.

If we specify our own ThreadPoolTaskExecutor bean, then SpringBoot will use it by default, else it will define the default ThreadPoolTaskExecutor.
=> How to define our own ThreadPoolTaskExecutor?
Create a bean for it in the configuration class.

ThreadPoolTaskExecutor is the spring version of pure java's ThreadPoolExecutor. If we create a ThreadPoolExecutor bean in the configuration class (@Configuration) then spring will default to using SimpleAsyncTaskExecutor for running the async (@Async) tasks. However we can force spring to use the ThreadPoolExecutor by specifying the bean name in the annotation - @Async, in which case the SimpleAsyncTaskExecutor will not be used.

Interceptors and building custom Interceptors

Interceptor is a mediator which gets invoked before or after our actual code.

=> Intercepting a request before it reaches the controller:
To create our own mediator, we need to implement the HandlerInterceptor interface which contains the methods preHandle, postHandle and afterCompletion.

preHandle will run before the actual method is invoked, postHandle will run after the method has been executed if there were no exceptions. In case there were exceptions, the postHandle method will not run. The method afterCompletion will run regardless every time at the end, similar to a finally block.

Use Cases: Authentication, Authorization.

=> We can create our own annotations, and intercept methods via @annotation or @within pointcuts.

Aside: Multiple Servlets:
As mentioned before Servlets are Java classes which handle client requests, process them and return a response. Applications can have multiple servlets, with each servlet responsible for handling a subset of requests, for example servlet A handles REST requests while servlet B handles SOAP requests. The servlet container (Tomcat) manages the servlets, and is responsible for routing the request to the appropriate servlet. However with spring, the pattern is different. Spring introduced the Dispatcher servlet which is configured to accept all requests matching /* (i.e. all the requests). Dispatcher servlet is responsible for routing the request to the correct controller, there is no role of other servlets with Spring.

Interceptors are specific to SpringBoot, they are used to intercept requests before they reach the appropriate controller, these are the requests which have been processed by the Dispatcher Servlet, and it will redirect the request to the appropriate controller, the interceptor will intercept the request before it reaches to the controller.

Filters

Filters are similar to interceptors in that they also intercept requests, however Filters intercept the requests before they reach the Dispatcher Servlet, i.e. they intercept requests b/w Tomcat and Dispatcher Servlet. In case of SpringBoot all the traffic is directed to the Dispatcher Servlet (/*), Filters are useful for tasks like logging, security etc. We can have a chain of filters F1, F2, F3, F4 ….. Fn.

Incoming Requests: F1 -> F2 -> F3 -> F4 -> …. Fn
Outgoing Response: Fn -> Fn-1 -> Fn-2 -> Fn-3 -> …. F1

Filters are useful in multi-servlet applications, filters intercept the requests before they reach the appropriate Servlet. These requests are redirected by the Servlet Container to the appropriate servlets, filters will intercept the request before it reaches the Servlet.

Use Filters if the interceptor code is very generic and can be applied to all servlets

HATEOAS

It tells the client what actions they can perform next, after they have performed a particular action.

Why are Hateoas links useful:
- Loose Coupling b/w client and server application
- Api Discovery

When a client sends requests, the server not only sends the required core details in the response, but also sends a list of APIs in the response, which the client can invoke next.

However do note, adding all the next set of actions to the response can make the API response bloat up. Hence it is not recommended to add all the possible next set of actions (APIs) to the response.

Problems with excessive Hateoas links:
- Too much load on the server
- Higher latency
- Increase payload size

However if we don't specify any APIs which the client can invoke next, then that results in tight coupling b/w the client and the server, and in such cases the client will need to handle a lot of business logic (thick client).

HTTP Responses, ResponseEntity, ResponseBody

HTTP Response consists of:
- Status Code
- Body
- Header (Optional)

ResponseEntity

We use ResponseEntity<T> in Spring to send HTTP responses, T is the type of the Response Body, it is generic. For example ResponseEntity<String> indicates our HTTP response body is of type String.

Header consists of key-value pairs, however it is not always required.

If we return a type of User from our controller method, then SpringBoot will internally translate it to ResponseEntity<User>, similarly if we are returning a String from a controller method, then Spring will translate it to ResponseEntity<String>. In such cases what will be the values of status and headers? By default status will be 200 (OK), and headers will be empty.

ResponseBody

Whenever we return String or POJO directly from the controller, then @ResponseBody annotation is required. This is needed so that the object returned is treated as a body and as a view.

Spring JPA (Java Persistence API)

ORM (Object Relational Mapping) Framework: ORM is a bridge between Java objects and Relational Databases.

JPA is an interface, Hibernate is an implementation of JPA.
Hibernate communicates with the JDBC. Again, JDBC is an (API) interface and is implemented by specific DB drivers. Each database like MySQL, Postgres etc. has its own DB driver.

About JDBC

JDBC (Java Database Connectivity) allows us to connect to a database, and perform queries like insertion, searching etc. As mentioned JDBC just provides an interface, the actual implementation is provided by DB specific drivers. For example MySql has the driver Connector/J, Postgres provides the driver PostgreSQL JDBC driver.

JDBC allows us to:

- Connect to a database
- Query DB
- Process the result

We can use JDBC directly to interact with databases, however there are some problems with this approach:

- Boilerplate Code (Driver loading, Connection making etc.)
- Exception Handling

- Closing of connections and other resources

Using JDBCTemplate: SpringBoot comes with the JDBCTemplate class which removes all the boilerplate. SpringBoot uses a DataSource object for Database Connection and Connection Pooling. SpringBoot by default uses Hikari DataSource, which comes with the Hikari Connection Pool.

Advantages of Spring JDBC:
- Automates Driver Loading: We don't need to perform Class.forName call ourselves.
- DB Connection Making automation: Whenever we run any query using the JDBCTemplate (execute, update, query etc.), then JDBC Template will handle the connection making, either by creating a new connection or using a connection from the JDBC pool.
- Exception Handling
- Closing the Connection - After we execute any update or query, JDBC Template will take care of either closing the connection or returning the connection to the pool.
- DB Connection Pooling: SpringBoot provides default JDBC Connection Pool called Hikari connection pool. Spring Boot uses by default the Hikari Data Source, which comes with the Hikari connection pool.

Object Relational Mapping (ORM) framework:

Previously we saw how our applications can make use of JDBC directly, ORM framework sits b/w the application logic and JDBC, and allows our application to work directly with objects and still manage the database.

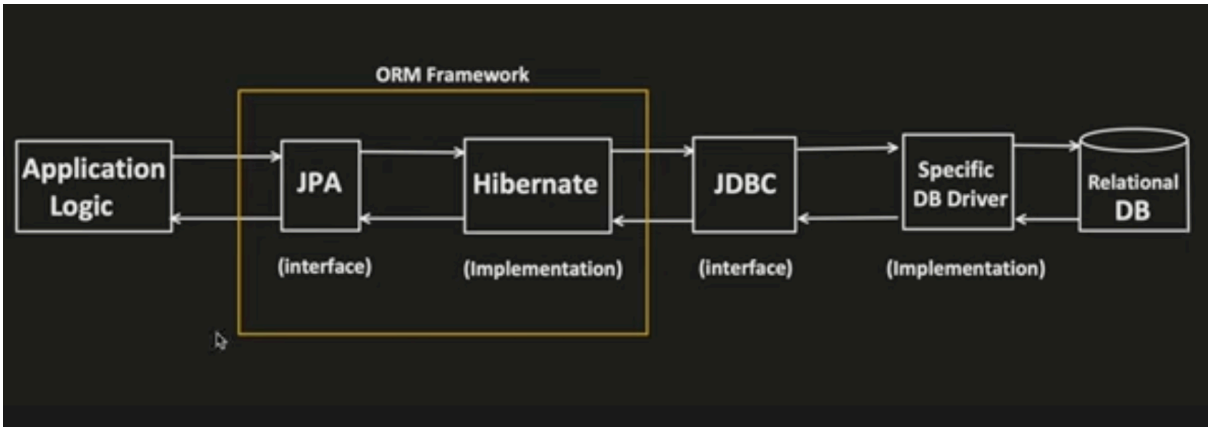
ORM Framework has 2 components
- JPA (Interface)
- Hibernate / OpenJPA / EclipseLink (Implementation of JPA)

ORM Tools - Hibernate, EclipseLink etc.

In other words JPA is a framework, and Hibernate / EclipseLink etc. implement the framework. SpringBoot by default uses Hibernate as the implementation of JPA.

ORM acts as a bridge between Java Objects and Relational Database Tables.
=> If we use JDBC, then we need to work with SQL. However with ORM we can interact with the database using Java Objects.

Entity - A class annotated with @Entity maps directly to a database table.
=> Dialect translates JQL / HQL to SQL which JDBC understands.



Ease of Use

1. JPA
2. Spring JDBC (JDBC Template)
3. Plain JDBC

Controller interacts with the service layer, which in turn interacts with the Repository layer. The Repository layer handles details of interacting with the database.

SpringBoot Exception Handling

Exception Handling flow in SpringBoot:
If our controller method throws an Exception, or the Dispatcher servlet is not able to send the request to the controller, then how will such Exceptions be handled?

Flow:
HandlerExceptionResolverComposite ->
ExceptionHandlerExceptionResolver ->
ResonseStatusExceptionResolver ->
DefaultHandlerExceptionResolver ->
DefaultErrorAttributes.

(Chain of Responsibility Design Pattern).

ExceptionHandlerExceptionResolver, handles the annotations:
@ExceptionHandler, @ControllerAdvice

Spring Security

Spring Security as the name suggests helps to make our applications secure.

Spring Security makes use of Filters. When we add spring-security to the project, we are immediately prompted with a login form, this is provided by the login filter added by spring security.
=> UsernamePasswordAuthenticationFilter is used to intercept login form submissions, it checks for username, password in the application.properties file. If not provided it creates its own.