03) Object-Oriented Design

→ OO languages help you to create apps that are flexible, maintainable & extensible.

→ This process is focused on S/W design so we won't be writing code, instead we will use basic components of Unified Model Language (UML) to visualize object-oriented systems.

→ UML is how we can articulate our ideas to collaborate with others.

→ Procedural languages have code written as a long series of operations to execute.
Straightforward approach!

→ New programmers have tendency to write code in procedural manner, as it's easy to think of a program in simple steps.

→ OO code is split into several self-contained objects, with each object having its own methods or functions.

→ It's almost like several mini programs, where each object contains its own data & logic, to describe how it behaves and interacts with other objects.

- Code-reusability is one of the great advantages of Object-Oriented approach.

→ OOP isn't a language, it is rather a programming paradigm. A set of ideas to write efficient, reusable code.

→ Other programming paradigms beyond OO & Procedural e.

(i) Logic Programming Languages e.g. Prolog

(ii) Functional - " —— e.g. Haskell

→ For the practical world of creating web & mobile apps, desktop apps or game development you'll almost certainly be using OOP languages.

e.g. C#, C++, Go, JAVA, JavaScript, Perl, PHP, Python, R, Ruby, Swift, VB.NET & many others

→ All of the top high demand languages today are OO.

---

* Attributes, Properties, Characteristics, State, fields, variables are used interchangably.

* All Objects have :

(i) Identity : Coffee_Mug

(ii) Attributes : color, size, fullness

(iii) Behaviors : fill(), empty(), clean()

↳ These things describe objects in OOP language.

Object : Bank Acct. :

number :
balance :
deposit() :
withdraw() :
query_balance() :

Objects = Nouns

- Things
- People
- Places
- Ideas
- Concepts

→ If you can put the word "The" in front of it in a conversation, it can be an object.

Behaviors = Verbs

- Ringing
- Texting
- Calling

1.3) Classes :

→ Objects don't appear magically in program, we have to create objects & classes acts as blueprint of the objects we make.

→ Class is the descriptive template of what an object will be.

→ Once we write a class & define it, we may use the class to create as many objects based on that class we want.

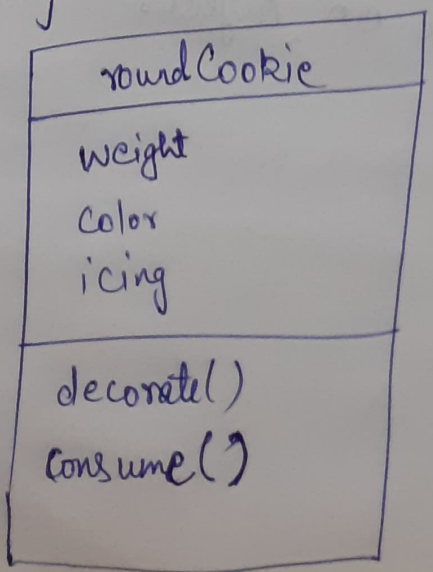
→ Different classes allow us to create different kinds of objects.

→ Class always comes first.

→ Components of a class : (i) ID (ii) Attributes (iii) Behaviors.
Type Properties Operations
Methods

Method : A program procedure that can return a value. Methods are basically functions, but methods are defined as part of a class.

→ Methods can only access data known to the object to which they belong.



Frameworks & libraries

- Java Class Library
- C++ STL
- .NET Framework BCL
- Ruby Standard Library
- Python Standard Library

Abstraction : One of the four fundamental ideas of OOP.

→ Abstraction

→ Polymorphism

→ Inheritance

→ Encapsulation

- Abstraction : focusing on essential qualities of something rather than one specific instance of it.

- Encapsulation : Containing attributes of an object We bundle an object's attributes along with methods that operate on that data within same unit or class.
- One reason to do so is to restrict access to some of the Object's components.
- We can add a method which allows us to modify private variables only through the method.
- One of the principles about encapsulation is that an object should not make anything about itself available, except when absolutely necessary for other parts of the app.

→ This concept is refered to as black boxing. We are closing off the inner workings of the class & only revealing specific I/O.

→ App component calling the method needn't know how `requestCookie()` method is implemented under the hood to call & use it.

→ Benefits of Object Orientation being it allows us to safely change the way methods/objects work, without changing the entire application.

Q) If we are the ones writing classes, why hide your own code?

A: It's not about being secretive, it' has to do with reducing dependencies between different parts of the app.

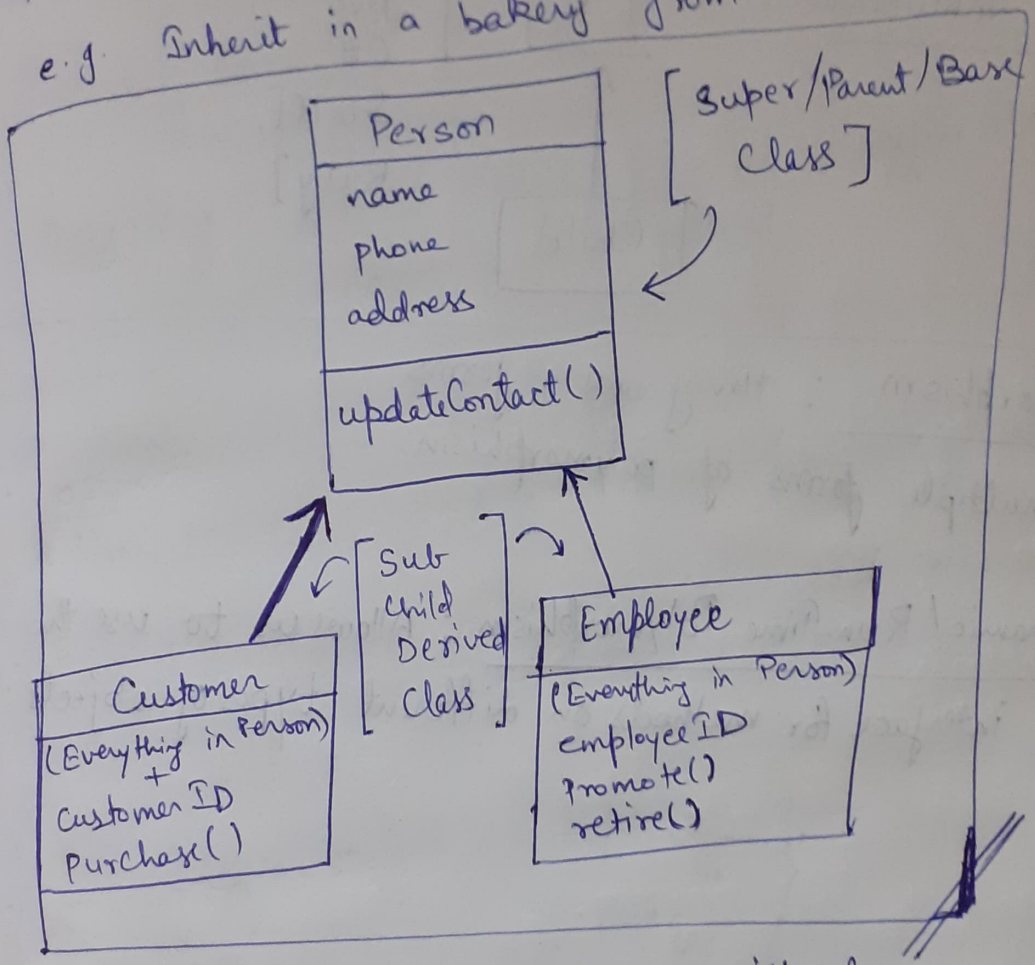
→ It prevents change in one place causing a domino effect which requires changes at multiple places in your code base.

RULE: Encapsulate as much as possible. give away only as much as ~~you may~~ other components need.

Inheritance

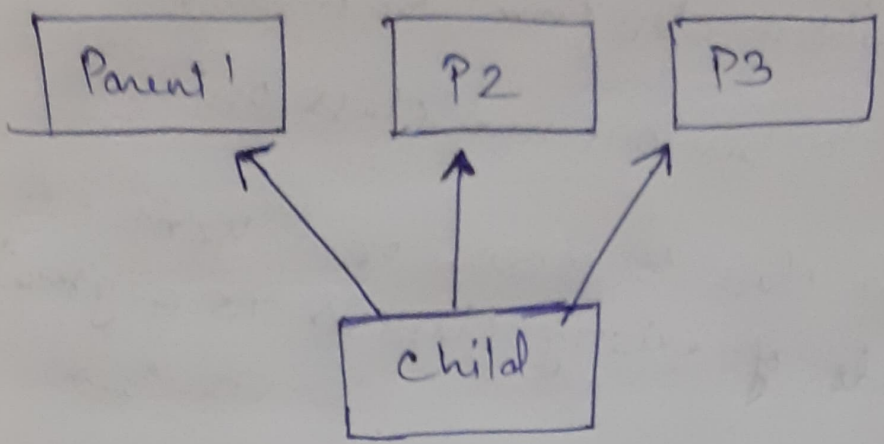
* Not always required to build a class from Scratch!
* we may use inheritance, to base our new class on existing class.
* Inheritance enables a new class to receive or inherit the attributes & methods of existing classes & is a great form of code re-use.

e.g. Inherit in a bakery from Person Class

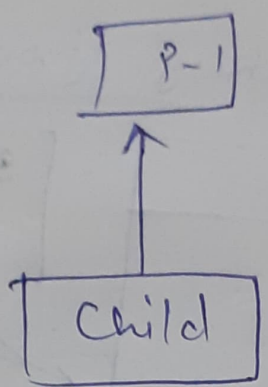


→ Adding a method to Superclass will also make that method available to all child/sub classes.

- Python, C++ : Allow multiple inheritance.



=> Single Inheritance

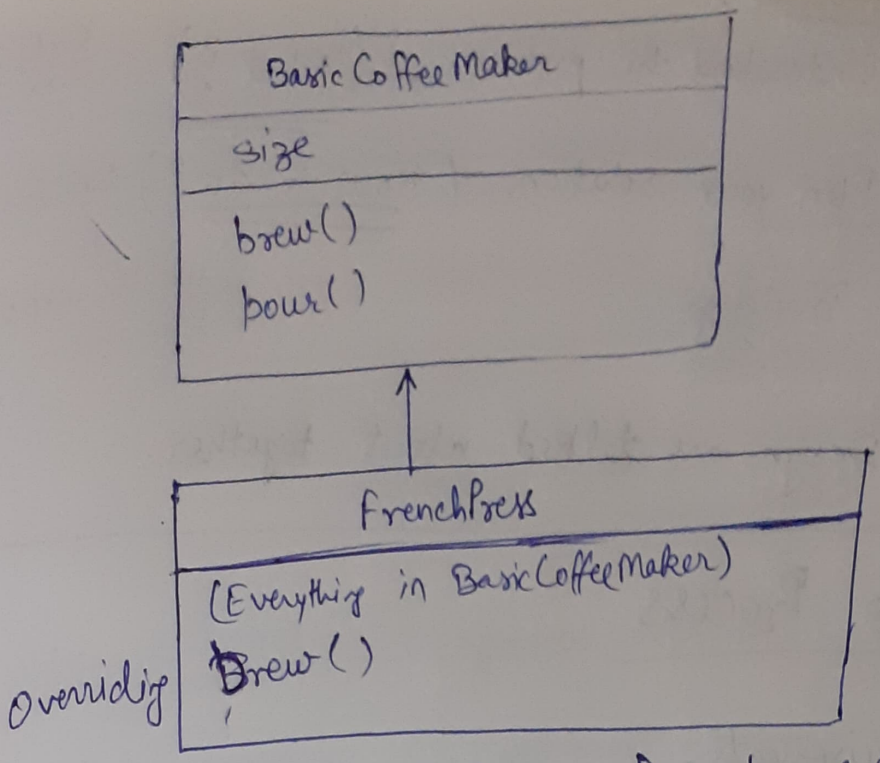


[Java, C#, Swift, Ruby]

Polymorphism : Having many forms.

→ ∃ multiple forms of polymorphism.

(i) Dynamic/Run-Time Polymorphism : Allows us to use the same interface for methods on different types of objects.



Overriding : Allowed French Press to re-define its own unique version of brew method.

→ Static (Compile-Time) Polymorphism

* Uses method overloading (Different from method overriding)

Method Overloading : Implements multiple methods with same name, but different parameters.

- brew(coffee, water) → cupOfCoffee
- brew(tea, water) → cupOfTea

→ 2 different methods, with 2 different signatures.

→ Overloaded methods provide different but very similar functionality.

☆ Object-Oriented:

Analysis: Understand the problem. (What?)

Design: Plan your solution. (How to do?)

Program: Building

* Analysis & Design are talked about together.

1.8) # 5 Step Process

① Gather Requirements.

② Describe the application: How people will use it?

③ Identify main objects.

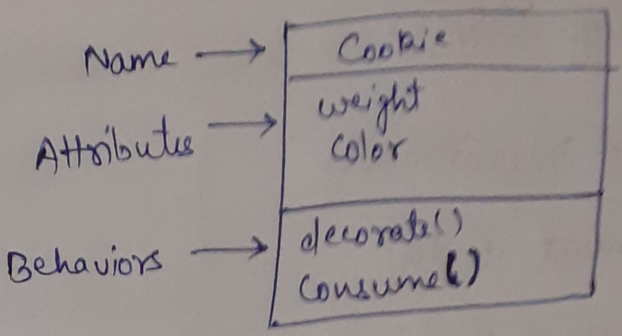
④ Describe the interactions

⑤ Create a class diagram.

1.9 Unified Modelling Language UML

-> Standardized notation for diagrams to visualize OO systems

Class Diagram



UML - Structural Diagrams

- Class Diagram
- Component Diagram
- Deployment - " -
- Object - " -
- Package - " -
- Profile - " -

Class & Use Case Diagram will use.

Wikipedia: List of Unified Modeling Language Tools.

Book: UML - Distilled by Martin Fowler

(2) Requirements

#2.1) Defining Req.

Functional Requirements :

- What must it do?

Non-Functional Req :

- Legal - Performance - Security - Support

→ Writing requirements is a general skill that applies to all kinds of projects.

→ If you're developing an app for a customer or a client you'll want to get as much info you want to understand the problem to solve.

→ Client may or may not have full idea of what they need, as a developer, always take time to understand why client wants to do something.

* TRAP : Dozen of semi-formed ideas about cool features app could have v/s nailing down what app really needs to do.

for functional requirements use phrase:

→ The system must do ... :

→ The application must do ... :

Space Microwave :

The system must ...

(i) heat meals in a space-packaging

(ii) Allow user to set a time for food to be ready

(iii) Notify when food is ready

(iv) change cook time based on type of meal

X ~~(v) Inherit meal types from an abstract superclass~~

(vi) continue to function w/o network connection.

Non-fx^ req: The system should be ... :

Describes ilities → Maintainability, Reliability, Scalability & Usability, Availability, Compatibility

- Available → 24/7
- Usable → With working gloves
- ~~Compatibility : Windows, iOS, Android~~

→ Focus only on nailing absolute minimum set of requirements, not the optional, nice to have things. Minimum Viable Product.

#2.2) FURPS+ Requirements

→ One commonly used model for classifying S/W quality attributes is FURPS

- F → Functionality
- U → Usability
- R → Reliability
- P → Performance
- S → Supportability

} FURPS

Functionality: Encapsulates Capability, Reusability, Security. (Core of What Customer Wants)

Usability: Human factors, Aesthetics, Consistency, Documentation

Reliability: Availability, Failure Rate, & Duration & Predictability

Performance: Speed, Efficiency, Resource Consumption, Scalability.

Supportability: Testability, Extensibility, Servicability, Configurability.

FURPS (+)

(+) → Design : Constraints on how S/W must be built.

→ Implementation : Does it have to be written in a certain language? Standards & methodologies to follow.

→ Interface : Not the UI, external system that needs to be interfaced with.

→ Physical Requirements : Constraints related to H/W on which app will be deployed on.

References :

1) Software Requirements – Karl Wiegers

2) Mastering the Requirements Process – Suzanne & James Robertson

Jukebox :

System must : (Functional)

- maintain library of albums & songs
- allow users to browse albums & songs
- Select individual songs.
- maintain queue of songs to play
- play music
- allow users sort by artist.
- identify individual users.

Non fxn. Things System should be :

- Intuitive to use while floating in space
- Available 24/7
- Low Power
- Updatable

3 Use cases & User Stories :

#3.1 Use Cases

-> focus on user & how they accomplish a specific goal.
-> Use case to capture that
-> ∄ single correct way to write use cases, & these could be written at several levels of formality
-> Atleast a user case needs 3 things :

Title : What's the goal?

Primary Actor : Who desires it?

Success Scenario : How is it accomplished?

Title → Short Phrase with an active verb that describes a goal.
e.g. Heat Meal ; Heat Delayed Meal.

* Keep titles short & simple.

Actors → All humans that could use the program.
Any entity (including computers) that act on system are actors.

Success Scenario : One paragraph. Short & Succinct. Everyday non-technical language, so it can be understood by the user of the application.

Use cases are written typically to describe ideal transaction by the actor on system

Use Case : Additional Details

Postconditions :

Secondary Actors :

Stakeholders :

Scope :

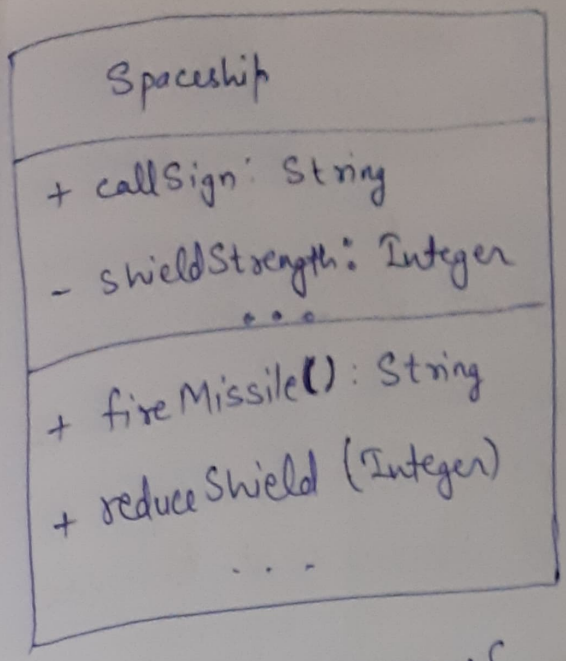
Priority :

Owner :

Rule of thumb : Don't spend more than a few days working on use cases

Refrence :

1) Writing Effective Use Cases
— Alistair Cockburn.




```
public class Spaceship{
//instance variables
public String callSign;
private int shieldStrength;
//methods
public String fireMissile(){
return "Pew!";
}
public void reduceShield(int value){
shieldStrength -= value;
}
}
```

* Same for Java & C#.

```
class Spaceship():
    def __init__(self):
        self.callSign = ''
        self.__shieldStr = 100

    def fireMissile(self):
        return "Pew!"

    def reduceShield(self, val):
        self.shieldStr -= val
```

Python

Ruby

```
class Spaceship

  # instance variables
  @call_sign
  @shield_strength

  # methods
  def fire_missile
    return "Pew!"
  end

  def reduce_shield(amount)
    shield_strength -= amount
  end
end
```

Instantiating Classes :

→ We may use keyword **new** .

Java : Spaceship myShip = new Spaceship();
& C#

C++ : Spaceship *myShip = new Spaceship();

Ruby : myShip = Spaceship.new

Python : myShip = Spaceship()

Swift : let myShip: Spaceship = Spaceship()

-> When we instantiate an object of a class in background, computer allocates a section of memory to hold new objects, including space for each variable in object.

-> To initialize those variables to some value, returns a reference to that object in memory & that gets assigned to a variable needed myShip.

-> Always consider what the internal state of an object will be immediately after you, instantiate it.

-> Spaceship in our case has 2 attr. callSign -> String & shieldStrength -> Integer.

-> In Java by default String is NULL & int = 0.

-> Constructor : A special method that sets default values of properties & gets called automatically whenever new class is created!

-> Constructor has same name as that of class.

```
public Spaceship(){
    name = "The nameless ship";
    ShieldStrength = 100;
}
```

-> Most of the languages allow us to create multiple Constructor methods via Constructor overloading.

-> One constructor method of same name but different values assigned based on inputs.

```
Public Spaceship(String name){
    callSign = name;
    shieldStrength = 200;
}
```

-> When class made with a string.

-> Destructor

-> A special method that gets called when an object is no longer needed.

-> Languages using garbage collection make use of finalizer rather than a destructor, but concept remains same.

-> Destructors are used if you've got an object that's holding a resource, say object has document open on file system.

5.6] Static Attributes & Methods

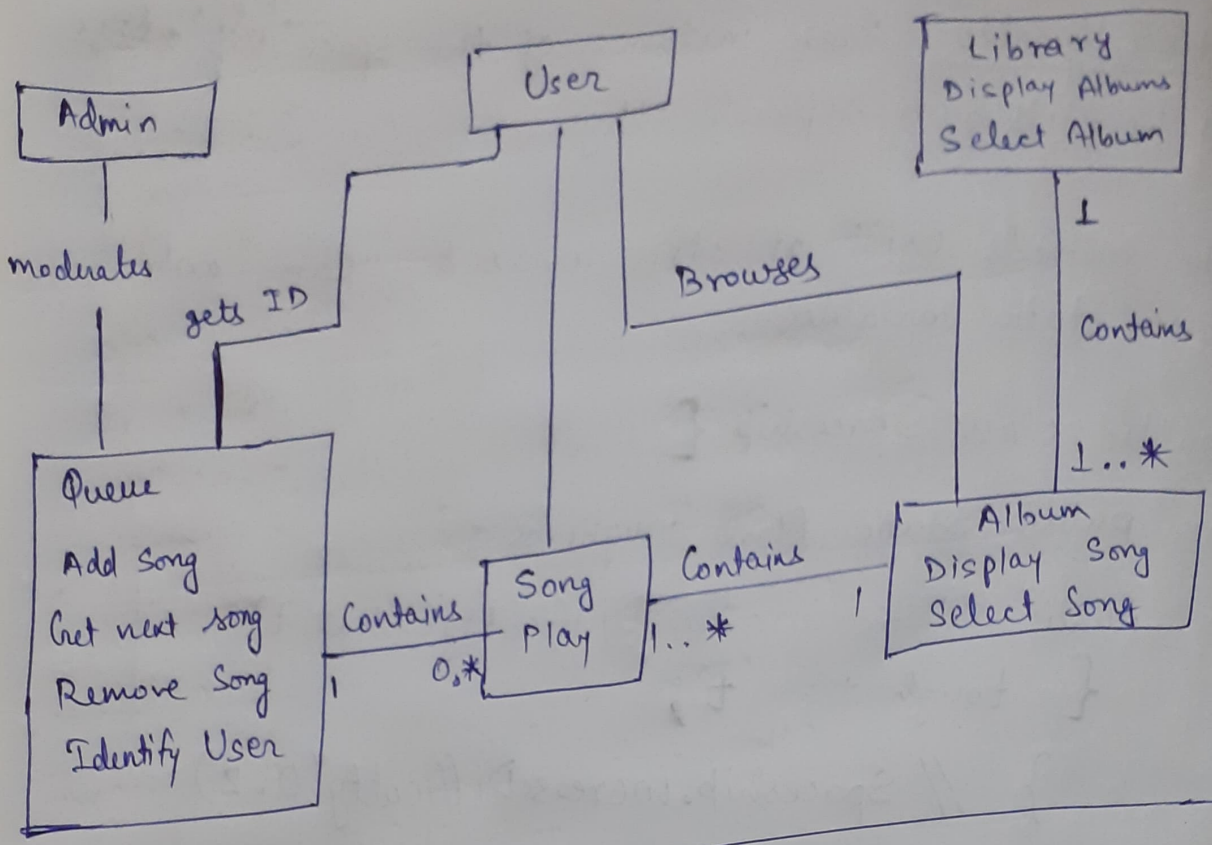
→ Instance Variable : Each instance of the spaceShip class will have its own separate copy.

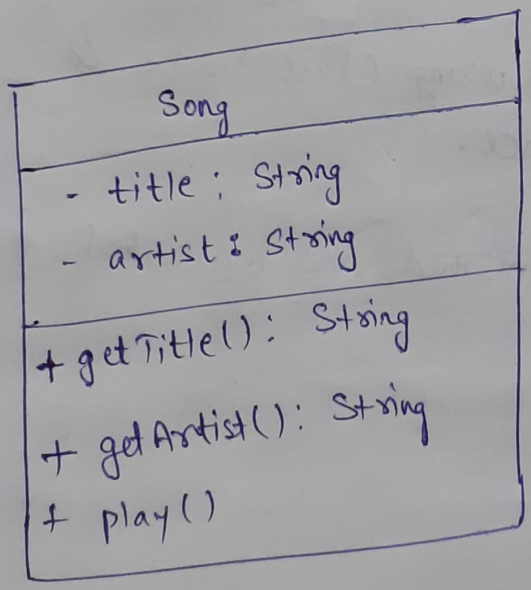
→ Static methods exist at class levels & can only access static variables.

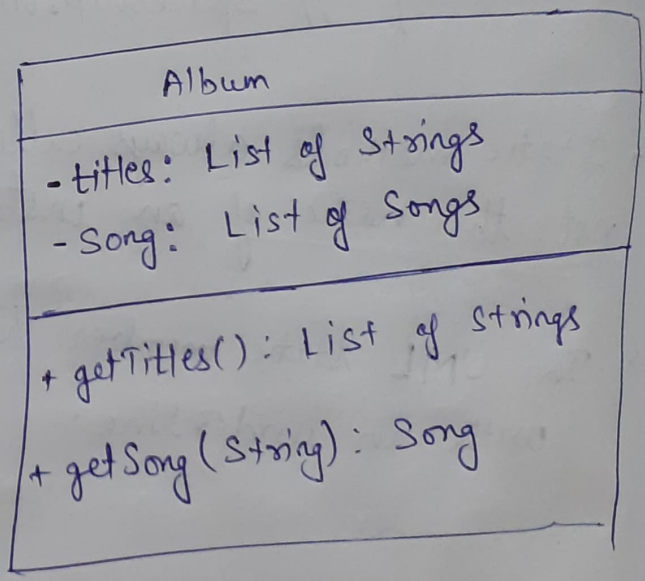
```
public class Spaceship {
    public static float toughness;
    public static increaseDifficulty(float t)
    { tough += t;
    } // Spaceship.increaseDifficulty(0.2)
```

→ Static methods always called using class name & not the name of an instance.

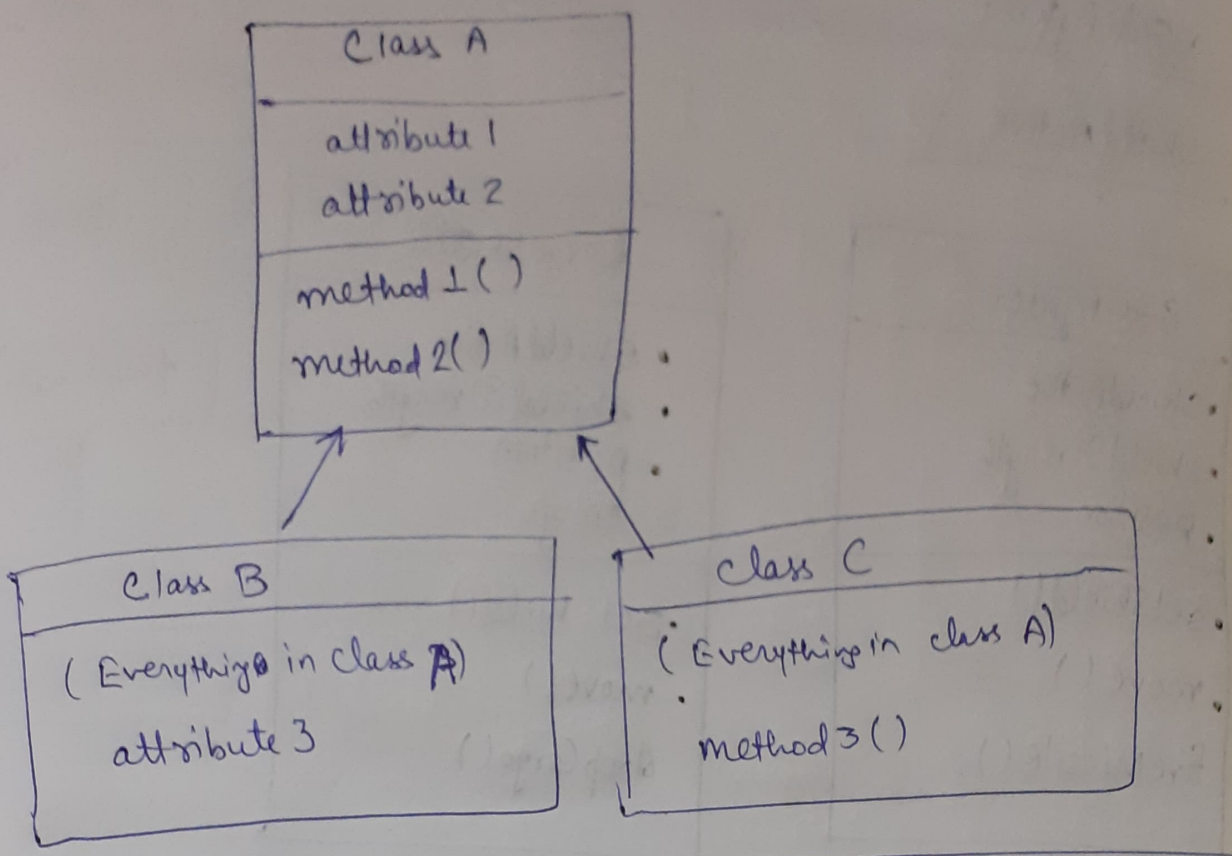
→ In UML static member & methods are indicated with an underline.


Admin
User
Library
Display Albums
Select Album
1
moderates
gets ID
Browses
Contains
1..*
Queue
Add Song
Get next song
Remove Song
Identify User
Contains
1
0,*
Song
Play
Contains
1..*
1
Album
Display Song
Select Song


Song
- title : String
- artist : String
+ getTitle() : String
+ getArtist() : String
+ play()


Album
- titles: List of Strings
- Song: List of Songs
+ getTitles() : List of Strings
+ getSong(String) : Song

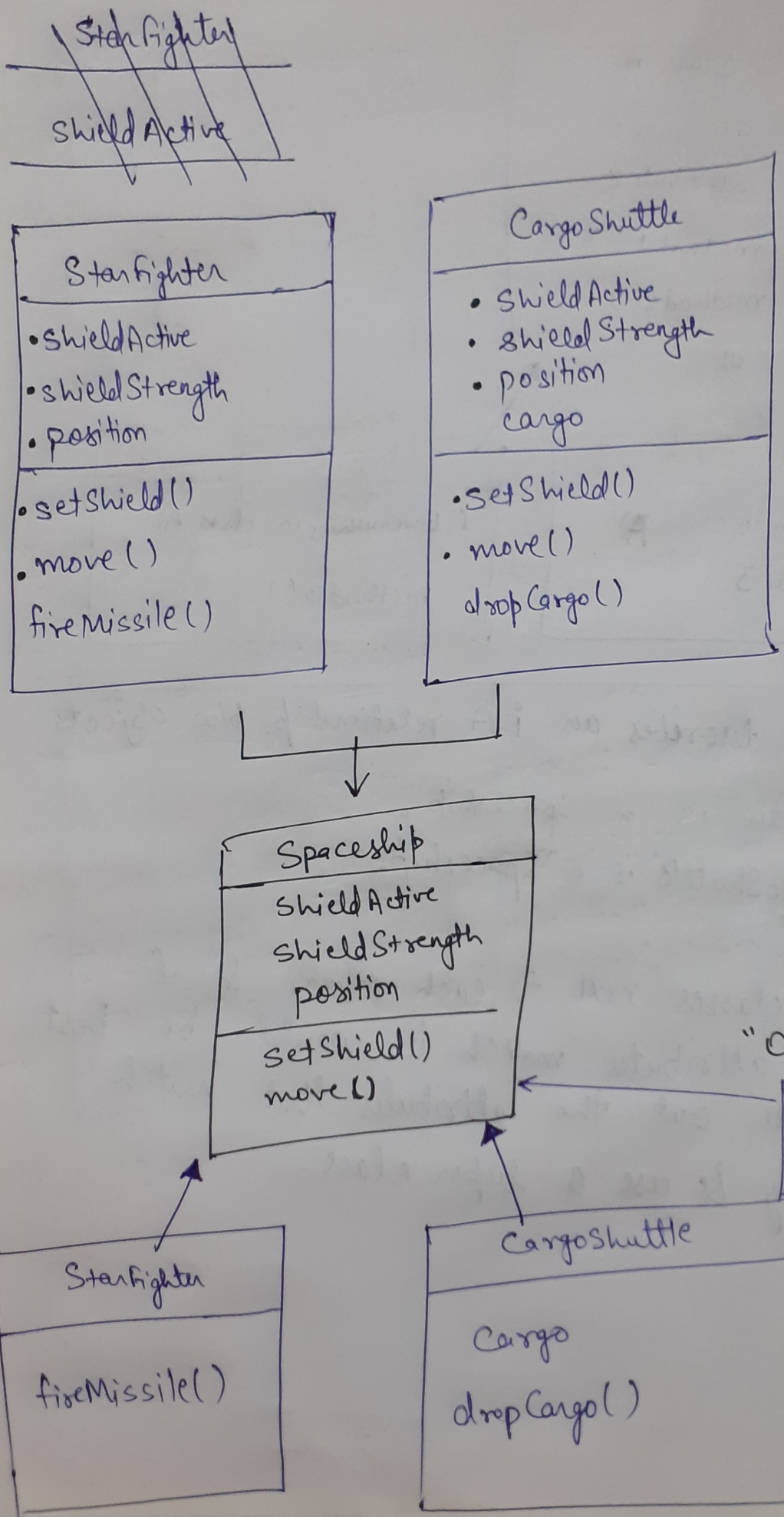
Inheritance



→ Inheritance describes an is A relationship. b/w objects.

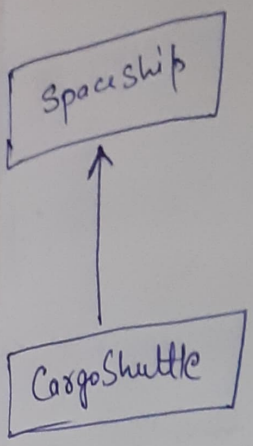
- Starfighter is a Spaceship.
- A CargoShuttle is a Spaceship.

→ Put 2 classes next to each other, & if enough attributes match it would be best to take out the attributes that match separately & use a super class.


StarFighter
shieldActive
StarFighter
•shieldActive
•shieldStrength
•position
•setShield()
•move()
fireMissile()
CargoShuttle
• shieldActive
• shieldStrength
• position
cargo
•setShield()
• move()
dropCargo()
Spaceship
shieldActive
shieldStrength
position
setShield()
move()
"Overriding"
WarpCruiser
move()
StarFighter
fireMissile()
CargoShuttle
Cargo
dropCargo()

→ Class usage overrides more method of super class.
→ Do not over-emphasize inheritance & don't come up with class diagrams of 5 level depth.

6.2] Identifying Inheritance



Java: public class CargoShuttle extends Spaceship { ...

C#: public class CargoShuttle : Spaceship { ...

C++: class CargoShuttle : public Spaceship { ...

Swift: class CargoShuttle : Spaceship { ...

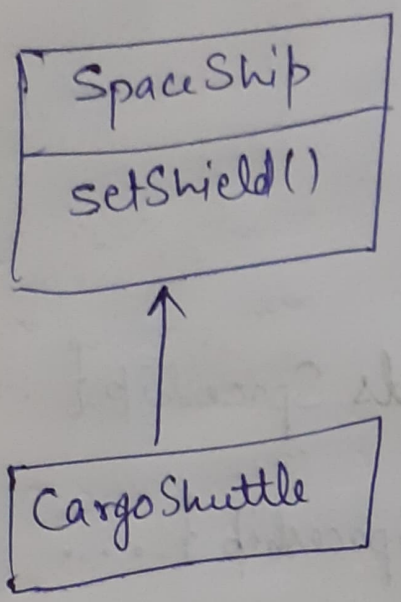
Python: class CargoShuttle(spaceship):

Ruby: class CargoShuttle < Spaceship

Overriding

-> Allowing a subclass to replace the implementation of a method from superclass.

Calling a method in Super/Parent/Base Class while inheriting



Java : super.setShield()

C# : base.setShield()

Swift + Python : super.setShield()

Ruby : super set_shield

C++ : SpaceShip::setShield()

{∵ C++ allows for multiple inheritance, ∴ we have to specify class name & not just super.}

#6.3) Abstract & Concrete Classes

- Exists for other classes to inherit - Abstract Class
- Cannot be instantiated.
- Contains atleast one abstract method.

* Italicized font for Abstract Classes in UML.

$>abstract class SpaceShip{ ...

-> Some languages like Java or C# lets us explicitly mark a class as abstract when declaring it, which prevents the language from allowing that class to be instantiated.

-> We must inherit from abstract class with another subclass, implement abstract methods in it & then we can use that subclass to actually instantiate an object.

* Java also allows us to mark classes as final which has opposite meaning of abstract.
* A final class is meant to be instantiated & cannot be inherited from. AKA Concrete Class.

```
final class WrapCruiser{ ...
```

* Benefit of keywords abstract & final is to communicate our intentions for a class to other programmers. It's to let them know whether or not the class was designed with inheritance in mind.

#6.4] Interfaces

→ List of methods for a class to implement. Doesn't contains any actual behavior.

e.g. JAVA

```
interface Moveable(){
// method signatures
    void move(int x, int y);
}
```

→ Not allowed to put any functionality inside interface.

→ When we defined a new class & choose to implement a specific interface it's like signing a contract, promising that new class we're defining will implement all of the methods of an interface.

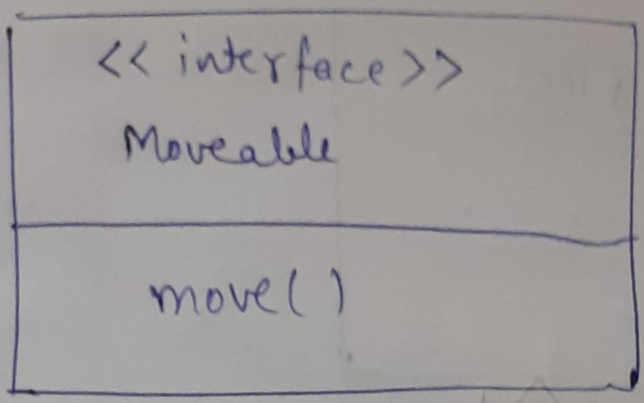
→ We are free to implement different inner workings of those methods, & however we want as long as method names, inputs & outputs matches the interface.

e.g.
```
Class Astesoid implements Moveable {
    public void move(int x, int y) {
        ...
    }
}
```

→ Interfaces & abstract classes might seem familiar at first, but they serve different purposes.

→ Interfaces are used to represent a capability that a class implements.

→ Abstract Class represent a type that another class may inherit.



UML Representation : Interface

Class which implements interface shown by - - - - ->

* A class can implement multiple interfaces.

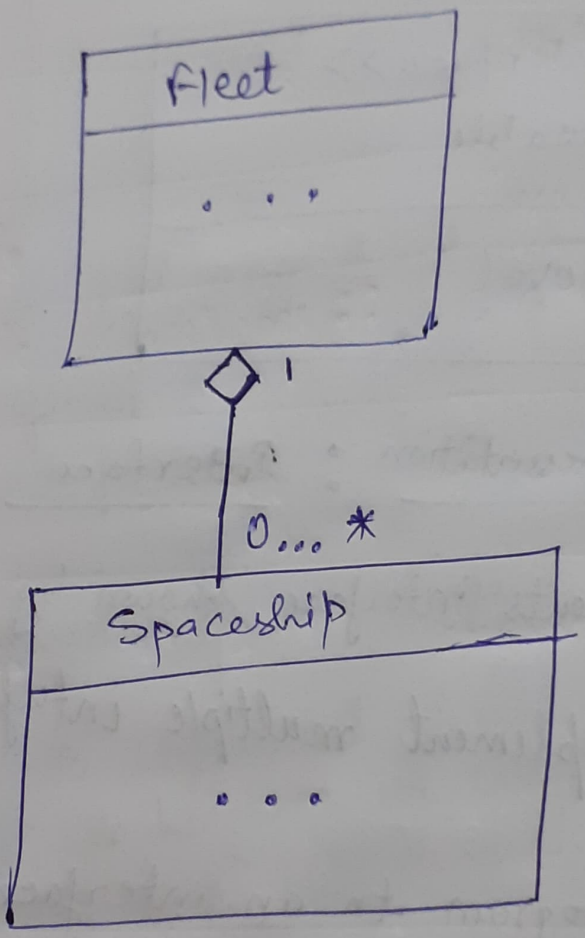
"Good developers program to an interface, not to an implementation because it's a developer's choice how to implement those methods."

#6.5] Aggregation – "has a" relationship / Uses many

→ A fleet "has a" spaceship.

Aggregation in UML

Represents Fleet Object that has a collection of Spaceship Object(s).



→ A fleet "Uses many" Spaceship(s).

#6.6] Composition : Implies Ownership.

Key Difference b/w Composition & Aggregation:

↳ If the owner object is destroyed is composition, contained objects are destroyed too.

→ Shown with filled diamond head ◆——

→ Important to show as "If lifetime of an object is dependent on another object's existence, that's actually worth showing."

→ May need to write constructor & destructor to take care of creating & deleting internal objects.

OOP Support Across Different Languages

Language	Inheritance	Call to Super	Typing	Interfaces	Abstract Classes
Java	Single	super	static	Yes	Yes
C#	Single	base	Static	Yes	Yes
Swift	Single	super	static	Protocols	No
Python	Multiple	Super	dynamic	Abst. Class	Yes
C++	Multiple	ClassName::	static	Abst. Class	Yes
Ruby	Mixins	super	dynamic	NA	NA
JS	Prototypes	n/a	dynamic	NA	NA

General Development Principles - SOLID

- **S**ingle Responsibility Principle : A class should have only a single responsibility primarily.
- **O**pen / Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

- DRY - Don't repeat yoursely !
- YAGNI - You ain't gonna need it ! Don't ~~write~~ fall to extensible code trap.
- KISS : Keep It Simple Silly !