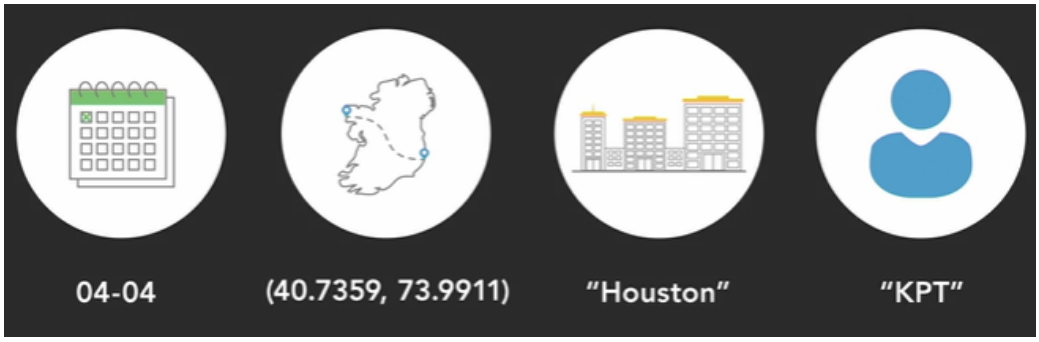
- 05 - Programming Foundations: Data Structures

Certificate	Abschlusszertifikat_Programming Foundations Data Structures.pdf
Completed Sections	8
Course Links	https://www.linkedin.com/learning/programming-foundations-data-structures-2?resume=false
Goal Sections	8
Quick note	
Start and Finish Date	
Thumbnail	
Time	2h 20m 56s
Understanding %	
نسبة الإنتهاء	✅Done

Introduction to Data Structures

▼ **data types**

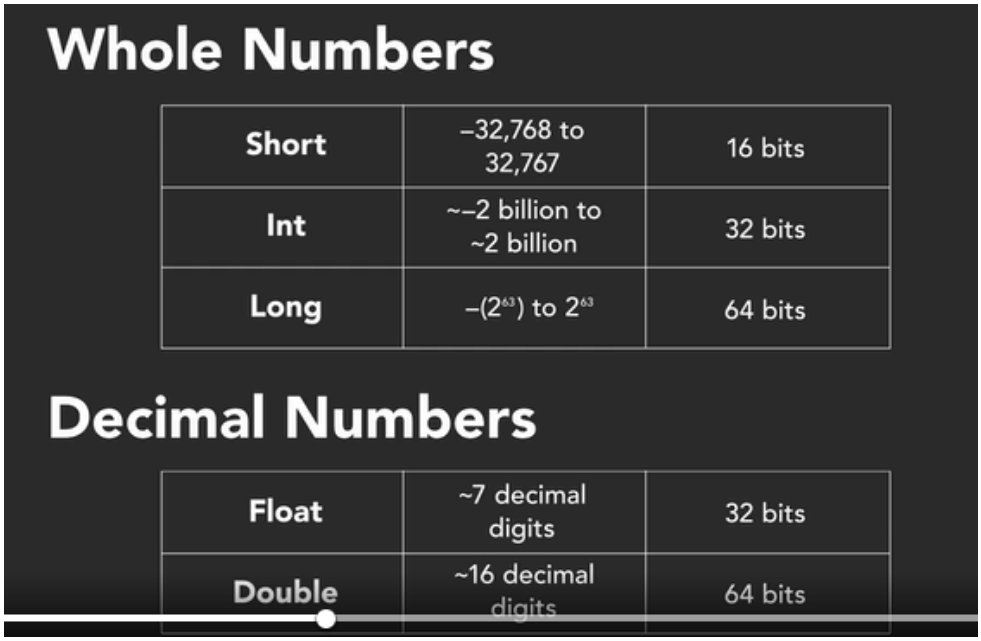
- An attribute of data that describes the values it can have and how the data can be used
  - Numbers
  - letters
  - True (1) or False (0)
  - ▼ Example



▼ Numerical data Types

They can be Signed or Unsigned

- Signed : Positive or negativ
  - Example -32.7 to 32.7
  - Whole Numbers
    - Int
    - long
  - Decimal Numbers
    - float
  - The difference between them ist the storage what they need
    - ▼ Example



Whole Numbers

Short	–32,768 to 32,767	16 bits
Int	~–2 billion to ~2 billion	32 bits
Long	$-(2^{63})$ to 2^{63}	64 bits

Decimal Numbers

Float	~7 decimal digits	32 bits
Double	~16 decimal digits	64 bits

- Or Unsigned
    - Only positive 0 to 65

▼ Booleans and Characters

- A Boolean is a true or false value
    - eg. The light
        - Is the light on ? True
        - Is the light of ? False
- A Characters
    - we build a variable and we give it a value.

▼ Primitive types in memory

- A string describes a group of characters
- ▼ No matter what the value of a string is it have the same bits value :

int	32 bit value
15	00000000000000000000000000001111
2191	00000000000000000000100010001111

-

▼ **Introduction to data structures**

- Data structures are containers.
    - Data structures are made up of pieces of data
    - They help us to connect different data in a group of data to handle with
    - They give us organization, storage and access

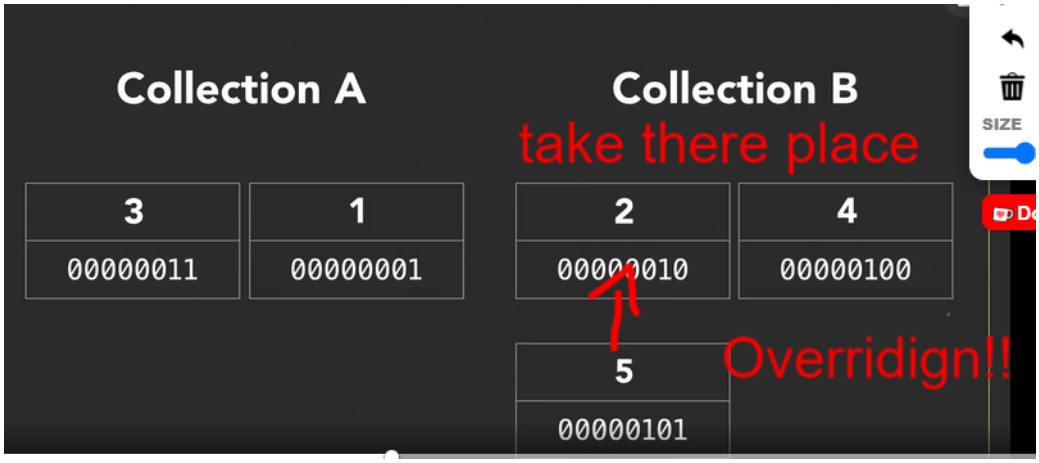
▼ String

- A string describes a group of characters

▼ primitive vs. reference types in memory

- data structures we do not know how much space they will take up until we know how many items they

  ▼ There are build in primitive type like int (49) what we can access directly , but that can be dangerous , it can führt to overriding eg:



  - that's why we need an extra layer which will called references
    - Again, it's important to remember that with reference types, we are adding this extra address layer whereas with primitive types, we directly access the data.
    - Referenced types use pointers to addresses.

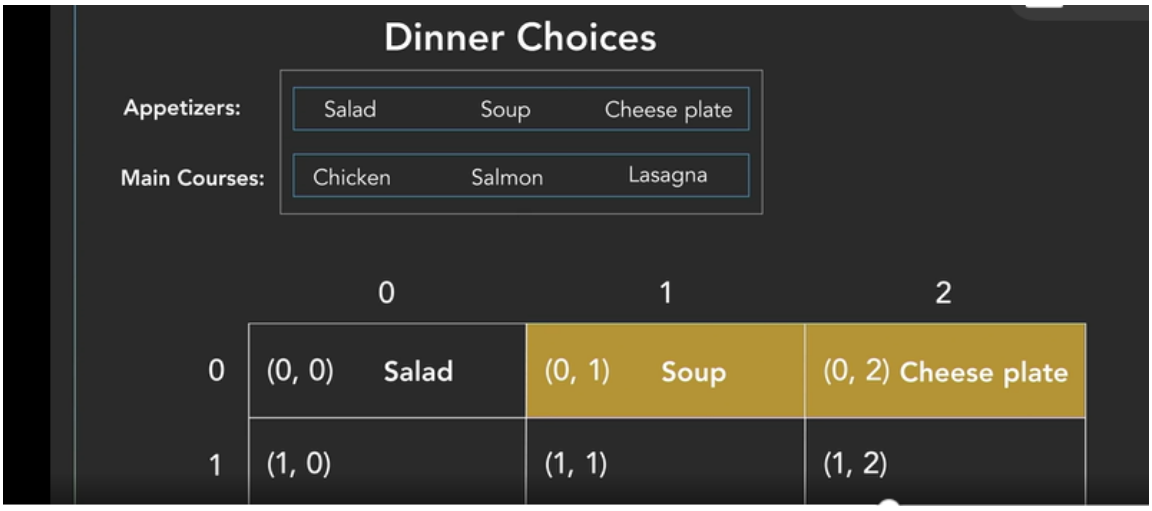
Arrays

▼ What are arrays

- is a collection of elements , where each item is identified by an index or key
  - Collection
  - Data structure is a collection with a defined way of accessing and storing items
  - Arrays provide
    - Organization

- Storage
- Access

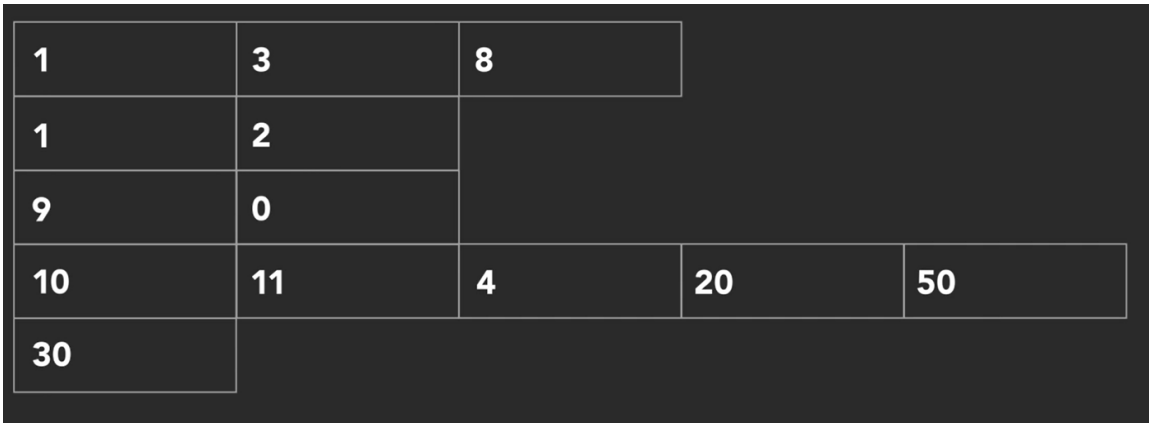
▼ Multidimensional arrays




```
dinnerChoices= [["salat","Suppe","Käse"],["Chicken","Salamon","lasagna"]]
appIndex=0
mainIndex=1

firstapp=dinnerChoices[appIndex][0]
secondapp=dinnerChoices[appIndex][1]
thirdapp=dinnerChoices[mainIndex][2]
print(firstapp)
print(secondapp)
print(thirdapp)
```

▼ **Jagged arrays**

- A jagged array can have elements of different dimensions and sizes
- With a jagged array, the number of columns is not fixed, meaning the inner arrays can be any length we'd like
-



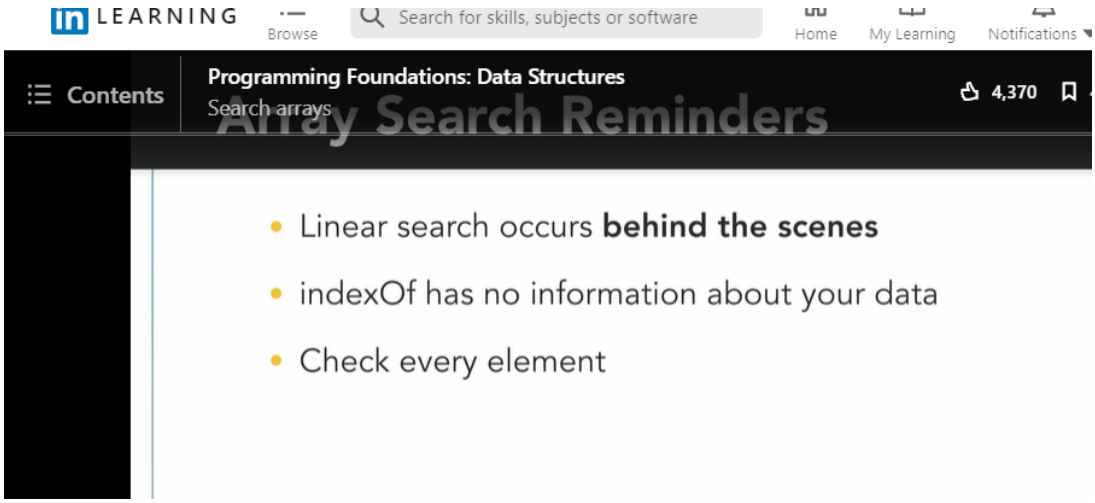
▼ Resizable arrays and language support

- Resizable means I can add new elements to this array
  - Java , C++ XXX
    - Basic arrays cannot be resized
    - Immutable : bacis array data
    - Mutable: java classes give us resizable versions
    - Array list: comes with extra functionality.
  - Ruby, java script
    - Are resizable or mutable
    - Add, push ———→ Adding to the back of the array
    - Remove, pop —→ Removing from the back of the array

▼ Search arrays

- In searching we check every item in the array and we look if it's match's it returns true or false
- But it takes a linear big O , because as much items we have it's take longer

▼ pic

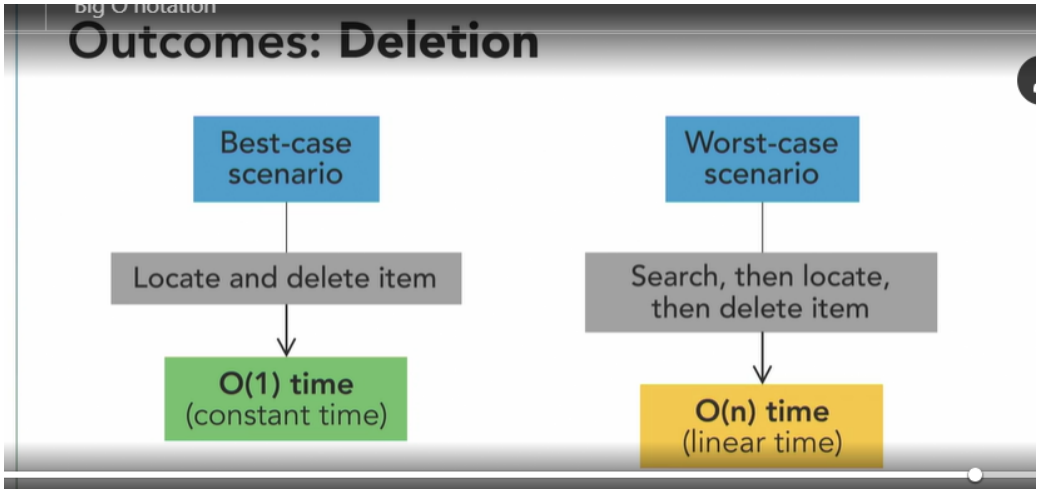


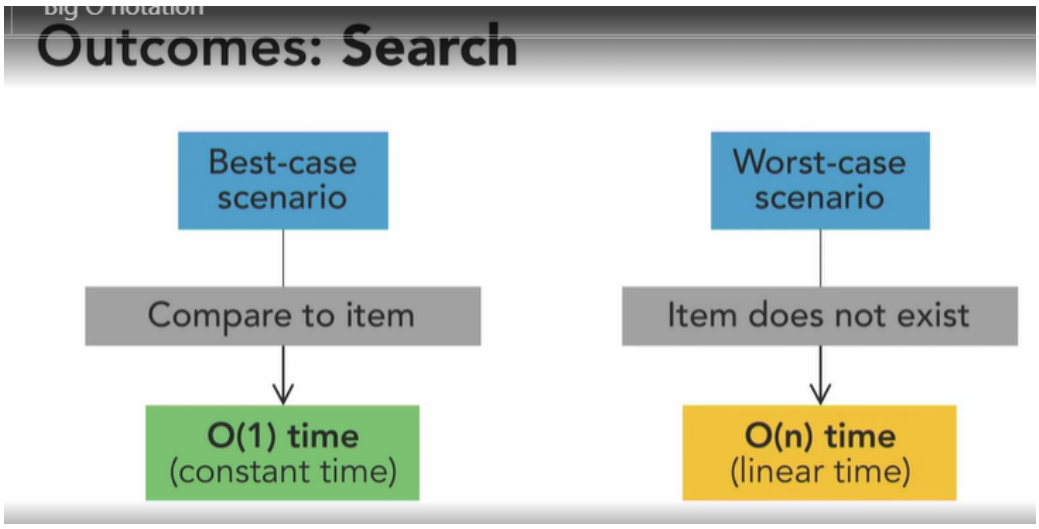
▼ Sort arrays

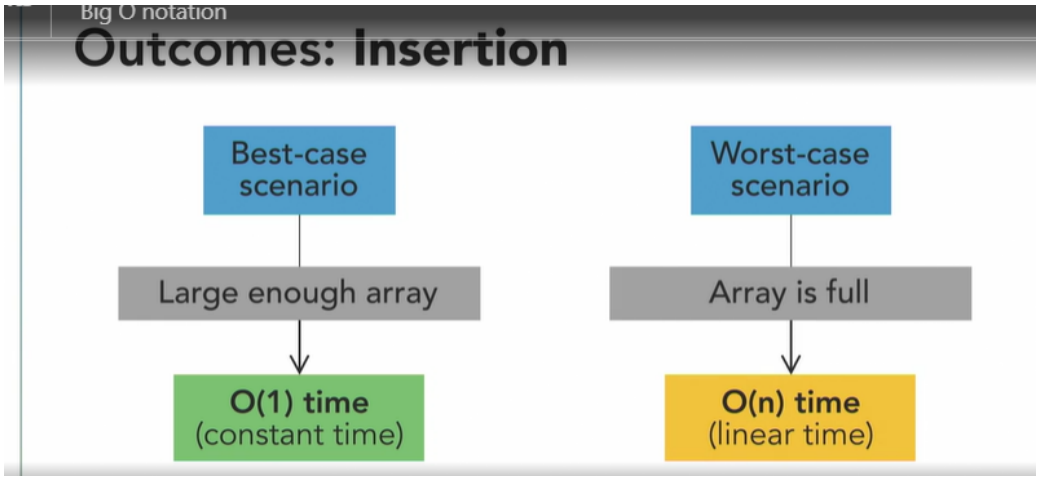
- if we have an array of numbers, a sorted version of this array could have the numbers in increasing or decreasing order , 1 , 2, 5, 6, 7,
- With characters and strings, we can sort in alphabetical order. Apple , banane , cucamber
- How
    - sometimes it's build in
- Objects
    - sometimes you will need to sort custom objects, and to do that, we have **to define how we compare each object to each other.**
      We define how we should order the objects.

▼ Big O Notation

- Notation used to describe the performance or complexity of an algorithm
- O(1) Time
    - Consistent duration of algorithm execution in same time (or space) regardless of the size of the input /constant time
        - adding a item to an array
        -

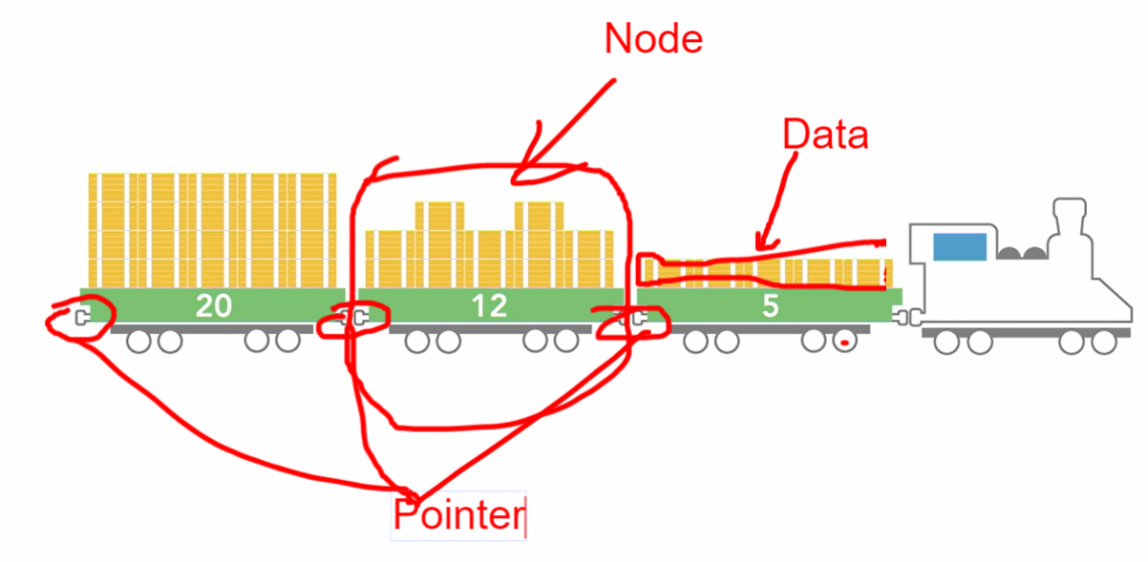

Big O notation
Outcomes: Deletion
Best-case scenario
Locate and delete item
O(1) time (constant time)
Worst-case scenario
Search, then locate, then delete item
O(n) time (linear time)


Big O notation
Outcomes: Search
Best-case scenario
Compare to item
O(1) time (constant time)
Worst-case scenario
Item does not exist
O(n) time (linear time)


Big O notation
Outcomes: Insertion
Best-case scenario
Large enough array
O(1) time (constant time)
Worst-case scenario
Array is full
O(n) time (linear time)

Lists

▼ Linked Lists



▼ **Singly vs. doubly linked lists**

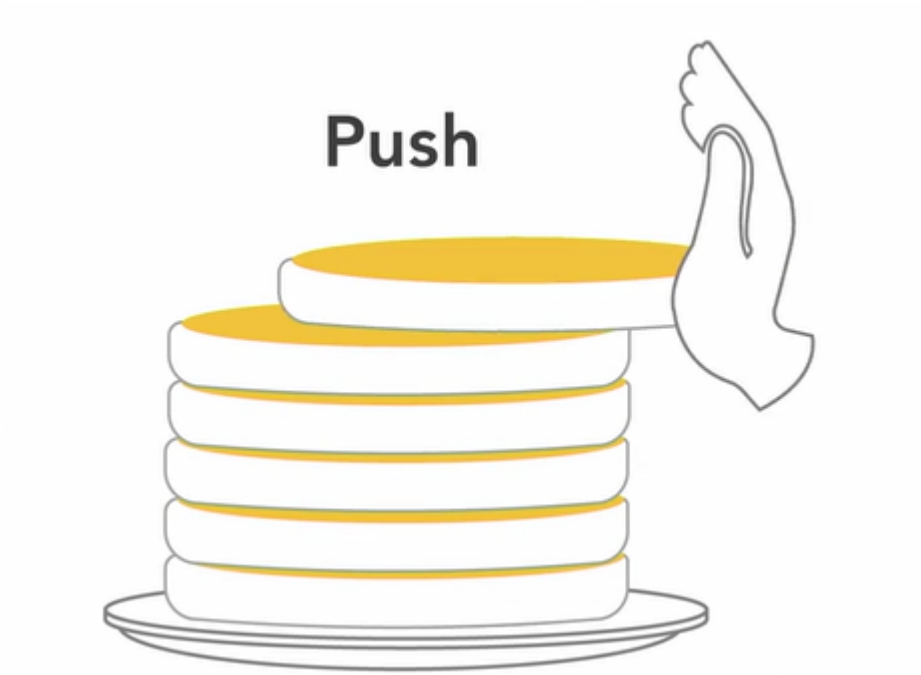
- by adding a previous pointer to each of our nodes. Now, each node of the linked list will contain a piece of data, a next pointer, as well as a previous pointer. Than we have a doubly linked lists

▼ Pros and cons of lists

Stacks and Queues

▼ What are stacks ?

- are containers type we can use to store our data
- Analogy
  - It's like a pancake , if we want to add an new layer we have to put (push) that on the top of the pancake. And if we want to access a layer we have to keeping pushing up layers until we achieves the layer what we want , and that is way it's a very good containers type for undo or go back method.




```
class Stack {
    var stackArray = [String]()
    //push
    func push(item:String){
        self.stackArray.append(item)

    }
    //pop
    func pop ()->String?{
        if self.stackArray.last !=nil {
            let lastString = self.Stackarray.last
            self.stackArray.removeLast()
            return LastString!

        } else {
            return nil
        } //peek
        func peek()-> String? {
            if self.stackArray.las !=nil {
                return self.stackArray.last

            }else{
                return nil
            }
        }
    }
}
```

▼ What are queues?

- The first element we add (enqueue) to the list is the last element out of the list.

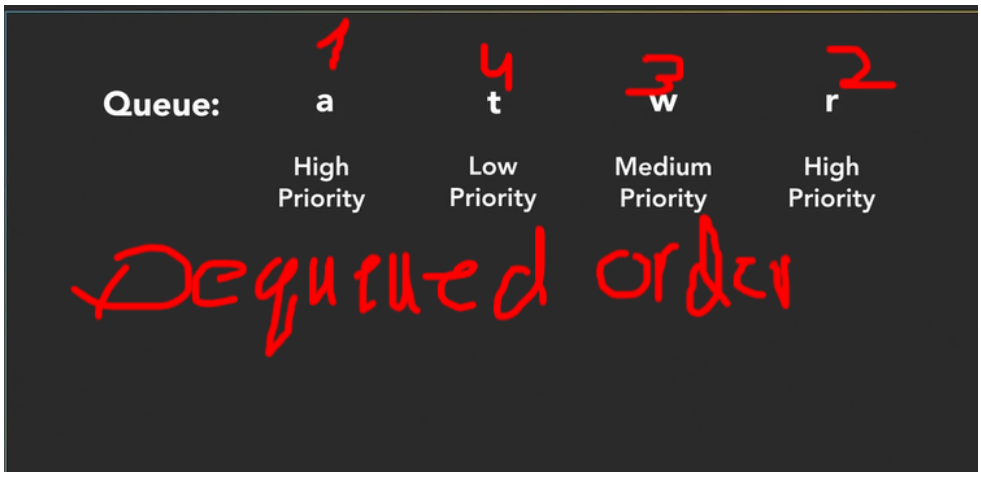
- Enqueue
  - is when an item is added to a list
- dequeue
  - is when an item is removed from the list
- peek()
  - see the first item in the queue without removing it.
- There is no indexing in the queue

```
class Queue {
    var queueArray = [String]()

    func enqueue(item:String) {
        self.queueArray.append(item)
    }

    func dequeue() -> String?{
        if self.queuekArray.first != nil {
            let firstString = self.queueArray.first
            self.queueArray.removeFirst()
            return firstString!
        } else {
            return nil
        }
    }

    func peek() -> String? {
        if self.stackArray.first != nil {
            return self.stackArray.first
        } else {
            return nil
        }
    }

var myQueue = Queue()
myQueue.enqueue(item: "Fouad")
myQueue.enqueue(item: "Alaa")
myQueue.enqueue(item: "Ahmad")

print(myQueue.peek()!)
var firstToleave = myQueue.dequeue()
print(myQueue.peek()!)
```

▼ **Specialized queues**

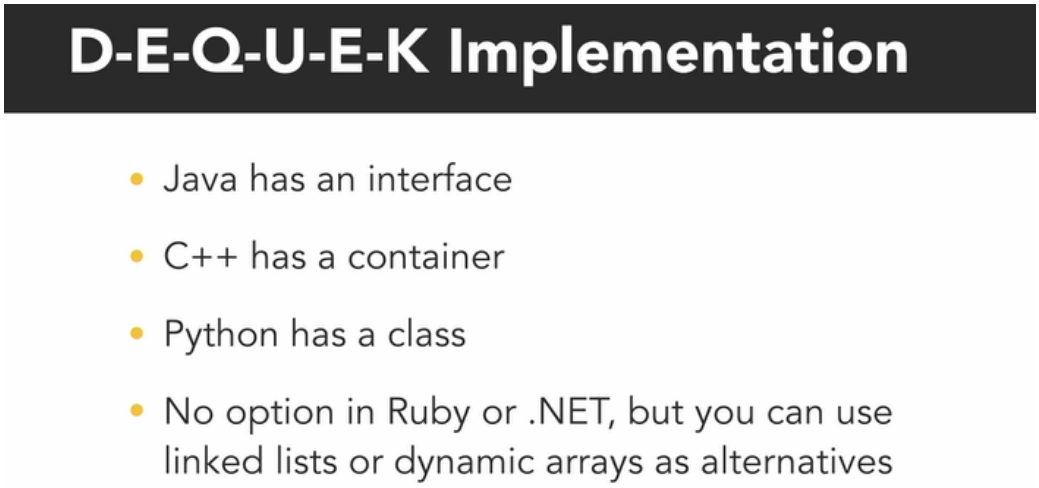
- Priority Queue
  - Each element has a priority associated with it

- it is good to help dequeuing some items form the middel of the list
- The first element will dequeued is the element with the highest priority



- in languages
  - java has priority queues
  - c++ has a priority container

- DEQUEK (Double-ended queue)
  - Items can be added or removed form either end.
  - In languages



-

▼ Pros and cons f stacks and queues

- Stack
  - Pros
    - Reversing things
    - keeping track of state
    - Add/remove form back of a structure
    - Stacks are best to help keep state (Stacks are advantageous for las in , first out)
  - Cons
    - If you find yourself needing to index your data structure and get a specific item in the middle of your use case, stacks are not the solution.
- Queue
  - are advategeous for first in frist out (FIFO)

Hash based data structure

▼ Associative Array

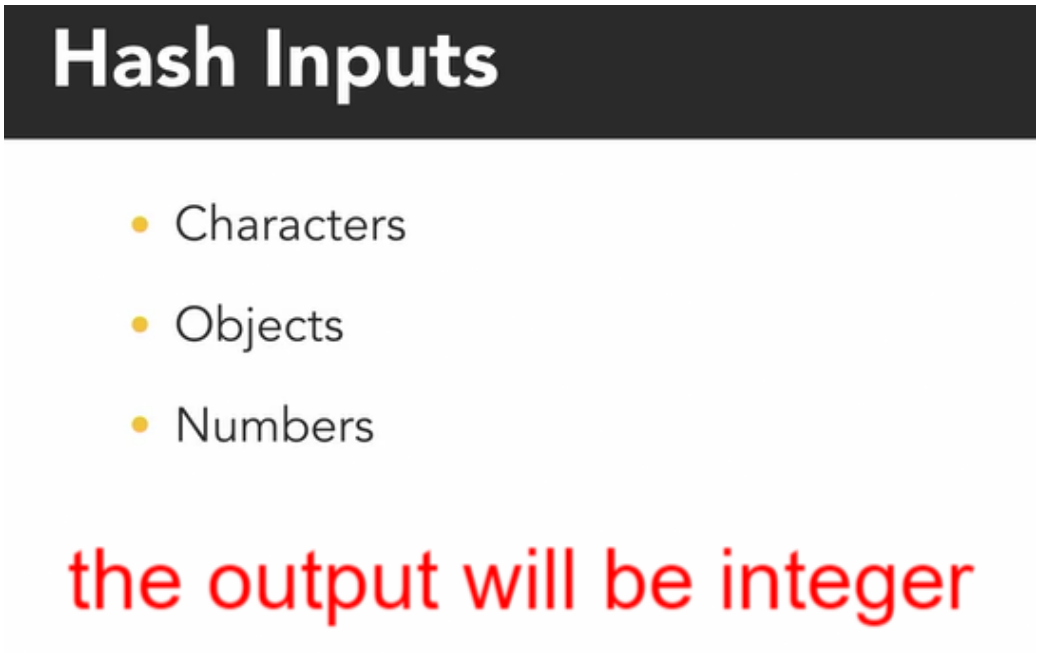
- Collection of key -value pairs
  - key: Value
  - California: Scramento
    - There no relationship with the index and value assocatied with it, we just need a way we store and access it through a key
- Rules
  - Key- value paris are bound together
  - Each key must be unique
  - Order isn’t important

- Values are accessed with the key
- Values do not need to be unique

▼ **Understanding hash functions**

- Hashing is a data convertsion process , where we take a row data and convert them to form a single piece of data

- Hash function
  - is the Inputting the row data ingredients to turn them to a product
  - example

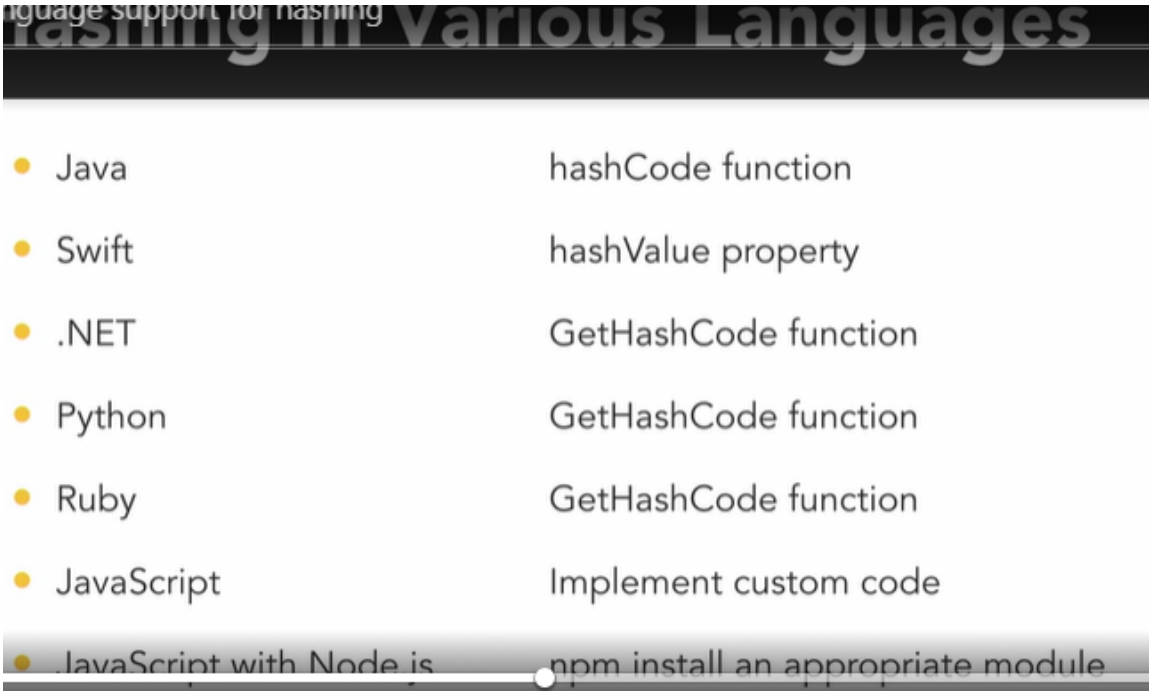


  - There are not reversible
  - ASC ||
    - Numerical representation of text characters
  - Collision
    - Anytime two inputs produce the same hash value

▼ Understanding hash tables

- A hash table is an implementation of the associative array abstract data structure
  - Adding Key-Value Pairs
    - Always added as a set
    - Keywords vary by language
      - put
      - add
      - Insert

▼ Language support for hashing

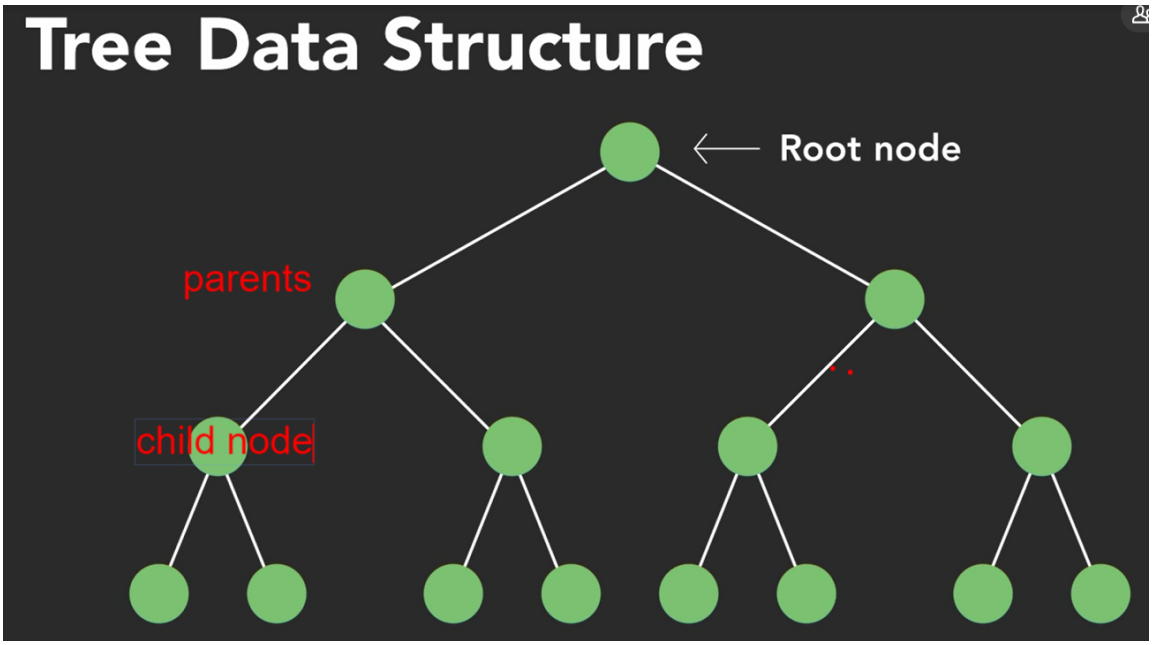


▼ Pros and cons

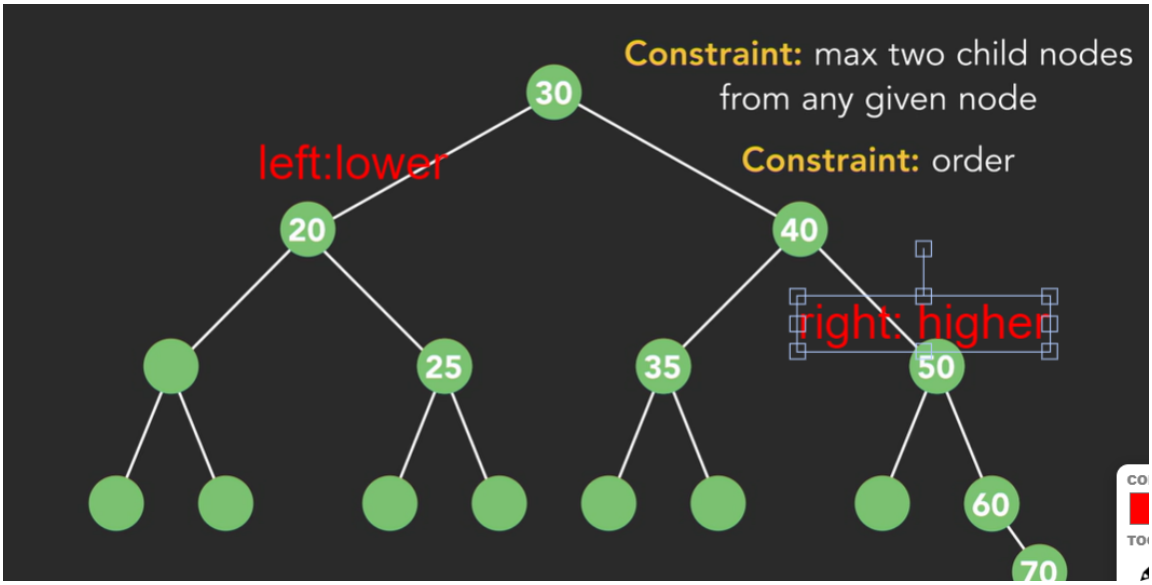
- pro
  - Hash map operations always take the same amount of time , regardless of the size of the hash table
    - Search: o(1)
    - add : o(1)
    - deletion: = o(1)

Trees and Graphs

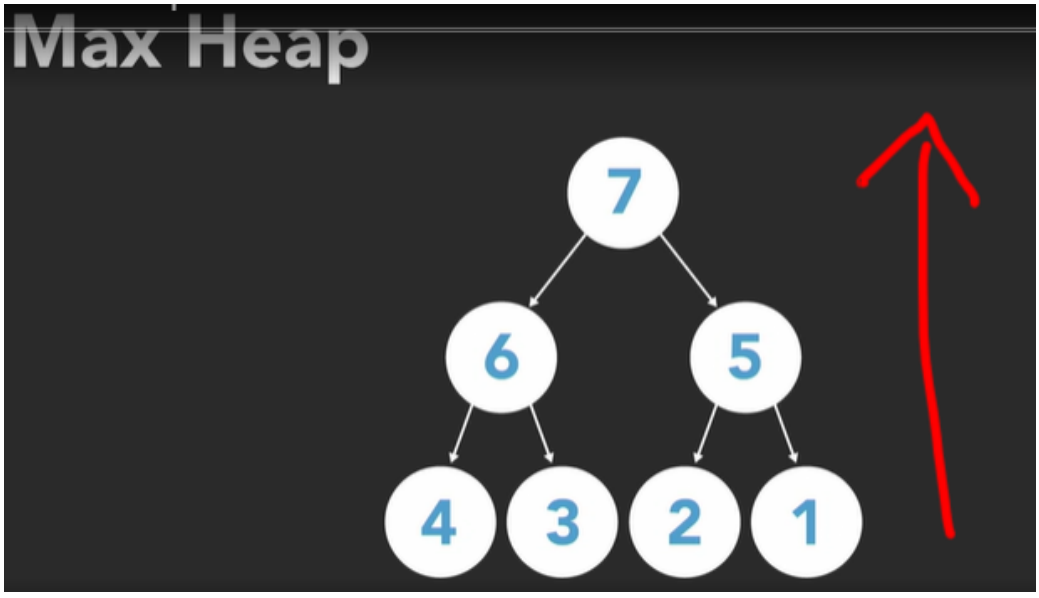
- ▼ What are sets
  - a collection of unique items
  - Order doesn't matter
  - None of the elements are duplicated
  - Membership
    - is gupping things with a common property
  - Implementation
    - Array(linear search)
    - Linked list (traverse)
    - Associative array (specific key)
- ▼ Introduction to tree data structures

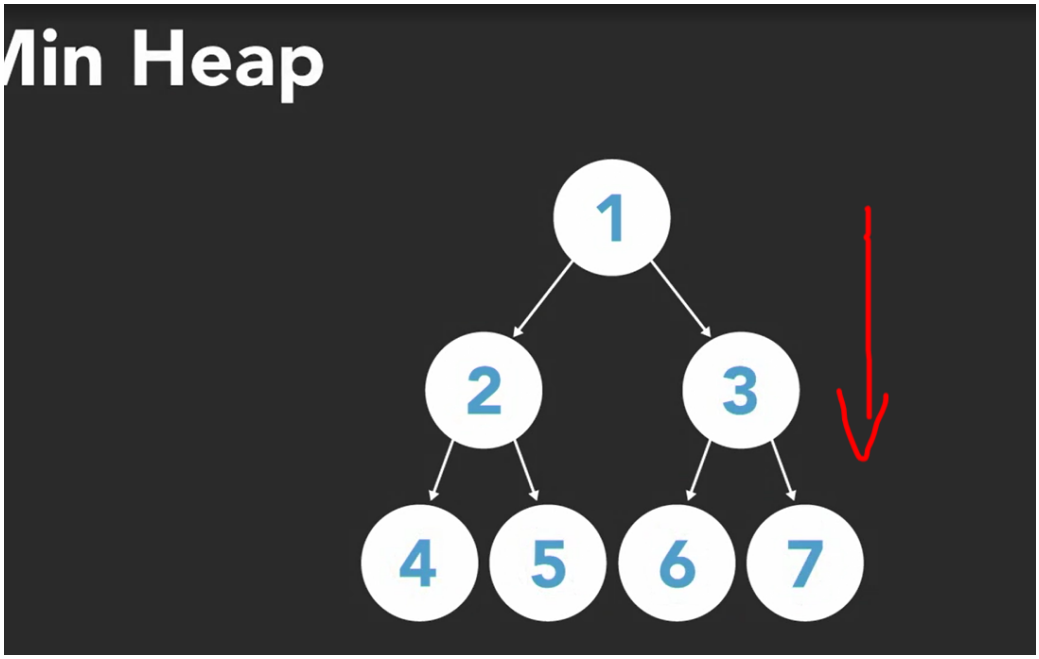


- ▼ Understand binary search trees



▼ Understanding heap

- a data structure implemented as a binary tree
- Priority queue
  - 



- Sets pro and cons
  - pro
    - if you need to constantly check if a certain value exists in the set and do not need duplicate values
    - Binary search Tree: maintain sorted order while staying fas for insertion, deletion , and accessing

▼ Challenge and solution

```java
public interface Drone {
        public void beep();
        public void spin_rotors();
        public void take_off();
}

public class SuperDrone implements Drone {
        public void beep() {
                System.out.println("Beep beep beep");
        }
        public void spin_rotors() {
                System.out.println("Rotors are spinning");
        }
        public void take_off() {
                System.out.println("Taking off");
        }
}
```

Solution

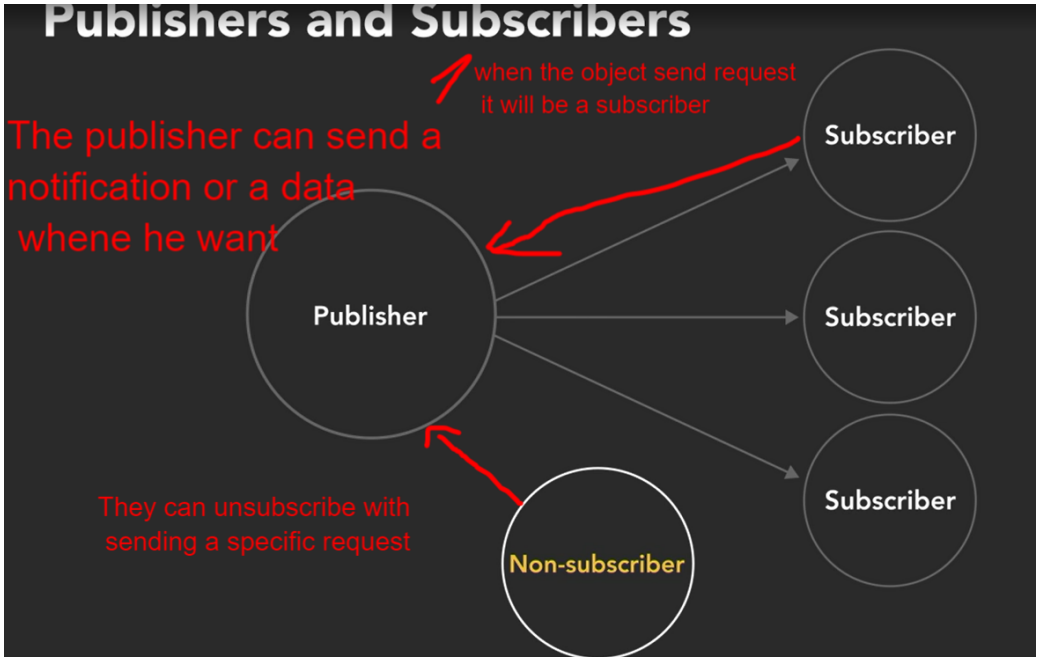
```java
blic class DroneAdapter implements Duck {
    Drone drone;

    public DroneAdapter(Drone drone) {
        this.drone = drone;
    }

    public void quack() {
        drone.beep();
    }

    public void fly() {
        drone.spin_rotors();
        drone.take_off();
    }
```

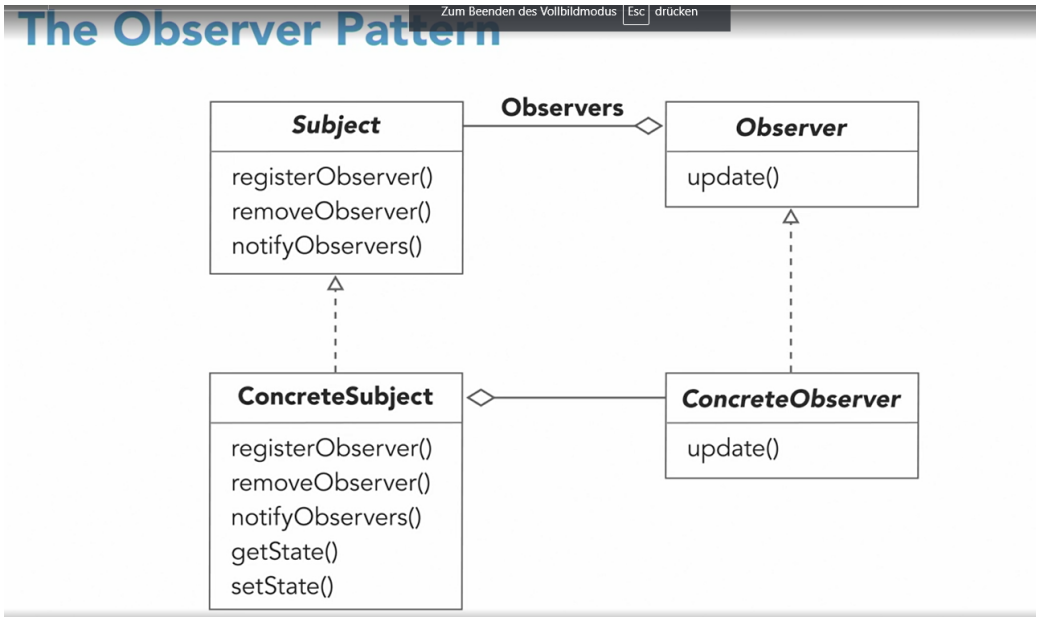
```java
public interface Drone {
        public void beep();
        public void spin_rotors();
        public void take_off();
}
public class SuperDrone implements Drone {
        public void beep() {
                System.out.println("Beep beep beep");
        }
        public void spin_rotors() {
                System.out.println("Rotors are spinning");
        }
        public void take_off() {
                System.out.println("Taking off");
        }
}
```

The Observer pattern

▼ What is the observer pattern

- Loose Coupling
  - Strive for loosely coupled designs between objects that interact
  - youtube channel analogy





- DEF
  - This pattern defines a one-to-many dependency between objects so that when one object changes state , all of its dependents are notified and updated automatically