Passing arguments(value vs reference)

const flight = 'LH234';

const jonas = {

  name: 'Jonas schmedtman',

  passport: 3509,

};

const checkIn = function (*flightNum*, *passenger*) {

  flightNum = 'LH99';

  passenger.name = 'MR. ' + passenger.name;

  if (passenger.passport === 3509) {

    alert('Checked In');

  } else {

    alert('Wronf passport');

  }

};

**checkIn(flight, jonas);**

**console.log(flight);**

**console.log(jonas);**

**reference to object jonas is passed as an argument**

**so any changes made in that reference will reflect in jonas as well**

**but for primitive datatype value wont be changed**

**//JS does not have passing by reference**

**//in case of object we pass the value as refernce to heap i.e not considered as passing by reference where as in c++ we have reference for primitives as well**

**//hence only pass by value is there**

**Default parametrs**

const bookings = [];

const creatBooking = function (

*flightNum*,

*numPassengers* = 1,

*price* = 199 \* numPassengers

) {

  //ES5

  //numPassenger||=1;

  //price=price||199\*numPassengers

  //just by using the variable name propert and value will be defined in an object

  const booking = {

    flightNum,

    numPassengers,

    price,

  };

  console.log(booking);

  bookings.push(booking);

};

**FIRST CLASS VS HIGHER ORDER FNS**

1) JAVACRIPT TREATS FNS AS FIRST CLASS CITIZENS

2)THS MEANS THAT FNS ARE SIMPLY VALUES

3)FUNCTIONS ARE JUST ANOTHER 'TYPE' OF OBJECT

**WHY WE SAY THAT JS PROVIDES FC FNS**

* USE CASE OF FC FNS EARLIER EXAMPLES
* WE HAVE ALREADY STORED FNS IN VARIABLES
* ARGUMENTS TO OTHER FNS
* RETURN FUNCTION FROM FUNCTION
* SINCE FNS ARE OBJECT THERE CAN BE METHODS ON IT AS WELL
* SINCE JS PROVIDE FIRST CLASS FUNCTION WE ARE ABLE TO WRITE HIGHER ORDER FNS
* HIGHER ORDER FN IS FN THAT RECEIVES ANOTHER FN AS AN ARGUMENT, THAT RETURNS A NEW FN ,OR BOTH
* THIS IS ONLY POSSIBLE BECAUSE OF FIR
* building higher order fn

**higher order fns**

this fn operates at higher level of abstraction leaving these lower level details to lower level fns

**Callback function**

In JavaScript, a callback function (often referred to simply as a "callback") is a function that is passed as an argument to another function and is intended to be executed at a later time

**abstraction** means we hide the details of some code implementation

we dont really care about those details this allows us to think about problems at an abstract level

**higher order fns** create abstraction

**Call,Apply,Bind**

book.call(swiss, 583, 'Mary Cooper');

//call function helps the book to have this keyword point to swiss and also pass arguments and call book function

**Apply method**(similar to call method but we need to pass in array instead of parameters of the fn)

**BIND METHOD**

* IT also allows us to manually set the this keyword for any function call
* the difference is that bind does not immediately call the function
* instead it returns a new fn where the this keyword is bound(i.e it is set to a different objec)

const bookEW23 = book.bind(eurowings, 23); //in the book fn the first argument is flightNum which is set to 23(this is not default this is constant setting it cannot be changed)

//now in the bookEW23 fn there will be one paramaeter the first argument of book is omiited as it is already set to 23

//so the only parameter in bookEW23 is just the name

//specifying parts of the parameter before hand just like we specified 23 in the above fn is actually a common pattern called partial application

//partial application means that part of the arguments of the orginal function are already applied which means alreadySet

//ex:bookEW23

**Bind method useful in event listeners:**

document

  .querySelector('.buy')

  .addEventListener('click', lufthansa.buyPlane.bind(lufthansa));

**summary**

* **call method call the function with this keyword set to a specific object**
* **bind method returns a new fn with this keyword set to a specific object**

**IMMEDIATELY INVOKED FUNCTIONS EXPRESSION**

* **FN THAT IS EXEUCTED INLY ONCE**
* **AFTER IT IS CALLED ONCE IT DISAPPEARS**

Code:

//IIFE

(function () {

  console.log('this will never run again');

  const isPrivate = 23;

})();

**all data defined inside a scope is private**

**we also say that this datA IS ENCAPSULATED**

**FOR EG  isPrivate IS ENCAPSULATED INSIDE THIS FN SCOPE WHICH IS CREATED**

**DATA ENCAPSULATION AND DATA PRIVACY IS EXTREMELY IMPORTANT**

//AFTER ES6//JUST A BLOCK INSIDE PARENTHESIS CREATES A NEW SCOPE

{

  const isPrivate = 23;

}

// Closures

const secureBooking = function () {

  let passengerCount = 0;

  return function () {

    passengerCount++;

    console.log(`${passengerCount} passengers`);

  };

};

const booker = secureBooking();

booker();

// secure booking has already finished execution .ie it is take out of the call stack

// but still the inner function i.e the booker fn has access to variables of the secureBooking fn

// what makes this possible is a closure

**beauty of closure:**any function always has access to the variable environment of the execution context in which fn was created