**Object Oriented Programming**

**Paradigms: Object Oriented Programming and Functional Programming (JS – multi-paradigm)**

**More Clear + Understandable**

**Easy To Extend**

**Easy To Maintain**

**Memory Efficient**

**Code Dry – Not Repeating codes**

**Object Oriented Programming**

Data and behavior combined into object

**Functional Programming**

Data and behavior are different and should be kept distinctly

**Object Oriented Programming**

**Object** = data(state) and behavior (methods /actions) into a box || organized

Two types :1) class based and 2) prototypal based

**Constructor Functions**

function Elf(name, weapon) {

this.name = name;

this.weapon = weapon;

}

Elf.prototype.attack = function() {

return 'atack with ' + this.weapon

}

const sam = new Elf('Sam', 'bow');

const peter = new Elf('Peter', 'bow');

sam.attack()

new keyword-> constructor function

use capital letter to indicate to other programmers // no arrow function

**Object.create()**

const elfFunctions = {

attack: function() {

return 'atack with ' + this.weapon

}

}

function createElf(name, weapon) {

//Object.create creates \_\_proto\_\_ link

**newElf = Object.create(elfFunctions)**

newElf.name = name;

newElf.weapon = weapon

return newElf

}

const sam = createElf('Sam', 'bow');

const peter = createElf('Peter', 'bow');

sam.attack()

**Factory functions:**

function that is used to create object

function createElf(name, weapon) {

return {

name: name,

weapon: weapon,

atack() {

return 'atack with ' + weapon

}

}

}

const sam = createElf('Sam', 'bow');

const peter = createElf('Peter', 'bow');

sam.atack()

**Constructor function:** only way to create a variable inside the function is by using this keyword

Elf.prototype was created because of new keyword

Only functions have prototype. Others does not

**Class:** blue print

**class** Elf {

constructor(name, weapon) { **// runs every time it instantiates or new keyword is used**

this.name = name; // all these instances have a separate variables of name and weapon

this.weapon = weapon;

}

attack() { // attack is out of constructor because it is shared by all the instances

return 'atack with ' + this.weapon // it is stored in a single memory with references for all the objects

}

} // an instance ->when we call the class and create an object

**const fiona = new Elf('Fiona', 'ninja stars');** // new -> instantiating a class

console.log(fiona **instanceof** Elf) // Fiona is an instance of Elf

const ben = new Elf('Ben', 'bow'); // Fiona and ben are instances

fiona.attack()

Syntactical sugar: js still uses prototypal method underneath this code. So it’s not exactly class. Syntactically it is but class keyword actually it is prototypal. It uses new keyword to create class etc. still an object in JS

**Object.create vs class :** both does the same job . one is pure prototypal and another one is oop.

**This :** 4ways

//explicit binding

const person3 = {

name: 'Karen',

age: 40,

hi: function() {

console.log('hi' + this.setTimeout)

}.bind(window)

}

person3.hi()

this points to window but still it is explicitly binded with window here . Explicit binding is done by using bind (), call () or apply()

**arrow functions**

const person4 = {

name: 'Karen',

age: 40,

hi: function() {

var inner = () => {

console.log('hi ' + this.name)

}

return inner()

}

}

person4.hi()

this 🡪 points to person4 .

if no arrow function .

this 🡪 points to window

**new binding**

function Person(name, age) {

this.name = name;

this.age =age;

console.log(this);

}

const person1 = new Person('Xavier', 55)

**implicit binding**

const person = {

name: 'Karen',

age: 40,

hi() {

console.log('hi' + this.name)

}

}

person.hi()

**Inheritance:** it simply links up the proto chain (in js objects inherits from objects; technically there is no classes) c++ java has actual class

class Character {

constructor(name, weapon) {

this.name = name;

this.weapon = weapon;

}

attack() {

return 'atack with ' + this.weapon

}

}

class Elf **extends** Character { **// subclass(Elf) and baseclass(Character)**

constructor(name, weapon, type) { // own constructor of ELF

super(name, weapon) // calls super class constructor with parameters

this.type = type;

}

}

class Ogre extends Character { **// Ogre has prototype chain upto character**

constructor(name, weapon, color) {

super(name, weapon); // **inorder to use this keyword super should be called** when extending class

this.color = color;

}

makeFort() { // this is like extending our prototype.

return 'strongest fort in the world made'

}

}

const houseElf = new Elf('Dolby', 'cloth', 'house') // new tells this belongs to ELF

const shrek = new Ogre('Shrek', 'club', 'green')

shrek.makeFort()

**Public vs private:** under consideration; check whether it is updated

**4 pillars of OOP:**

**Encapsulation**: wrapping of code into boxes. Boxes interacts with methods and properties

**Abstraction:** hiding the complexity from the users.

**Inheritance:** inherit properties of object .re-usability of code and memory efficiency

**Polymorphism:** calling same method with different obj (method overriding, the same method acts differently or method overloading , adding extra features adding on to the existing method) reuse

**Polymorphism:**

attack() { //constructor(name, weapon, kind)

console.log(super.attack());

return `I am the ${this.name} of ${this.kind}, now bow down to me!`

}

const victoria = new Queen('Victoria', 'army', 'hearts');

victoria.attack()

attack() { //super class

return 'atack with ' + this.weapon

}