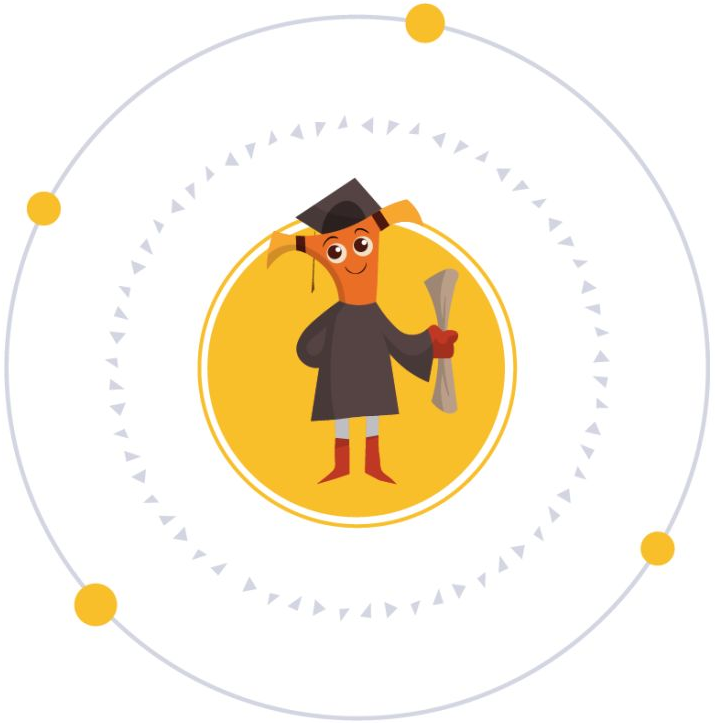


SPARK. SCHOOL

Tips & tricks

Design patterns

Javascript tips and tricks

- Falsy vrijednosti ?
- Typeof ? Kakve vrijednosti vraća ?
- toFixed() toPercision() ?
- Ternary operator ?
- console.log(Boolean(undefined));
- console.log(Boolean([]));
- var x = ‘5’; console.log(x + 1);
- Brisanje vrijednosti iz niza ?

Design patterns?

A pattern is a reusable solution that can be applied to commonly occurring problems in software design

- **Patterns are proven solutions**
- **Patterns can be easily reused**
- **Patterns can be expressive**
- **Combined experience of many developers**
- **It’s not an exact solution, it provides solution scheme**
- **Creational design patterns, Structural design patterns, Behavioral design patterns**

Constructor pattern

- Used to initialize a newly created object
- In JavaScript, almost everything is an object

Three common ways to create new objects:

- `var newObject = {};`
- `var newObject = Object.create( Object.prototype )`
- `var newObject = new Object();`

Module pattern

- Private and public properties and methods are separated
- Based in part on object literals
- Provide both private and public encapsulation
- Encapsulates “privacy”, state and organization using closures
- Only public API is returned, keeping everything else within the closure private

primjer: CODE

```
var myModule = (function () {
  var privateName = '',
    privateNumber = 0;

  function letsGetPrivate() {
    return privateName;
  }

  function getPrivateNumber() {
    return privateNumber;
  }

  return {
    increase: function () {
      return ++privateNumber;
    },
    showPrivateNumber: function () {
      return privateNumber;
    }
  };
})();

console.log(myModule.increase());
console.log(myModule.increase());
```

Revealing module

- Improved version of module pattern
- Author: Christian Heilmann
- Returns anonymous object with pointers to private functionalities
- Can be used to reveal private functions and properties with specific naming
- Syntax of our script is more consistent
- Makes it more clear

```
var revealed = function () {
  var a = [1, 2, 3];

  function abc() {
    return (a[0] * a[1]) + a[2];
  }

  return {
    name: 'revealed',
    abcfn: abc
  }
}();

console.log(revealed.name);    //=> 'revealed'
console.log(revealed.abcfn()); //=> 5 (1*2+3)
```

Singleton pattern

- Restricts instantiation of a class to single object
- It’s not class or object, it’s structure
- Provide single point of access for functions
- Rules for singleton pattern:
  - *There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.*
  - *When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.*

```
var Singleton = (function () {
  var instance;

  function createInstance() {
    var object = new Object("I am the instance");
    return object;
  }

  return {
    getInstance: function () {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

function run() {
  var instance1 = Singleton.getInstance();
  var instance2 = Singleton.getInstance();

  console.log("Same instance? " + (instance1 === instance2));
}

run();
```

Observer pattern

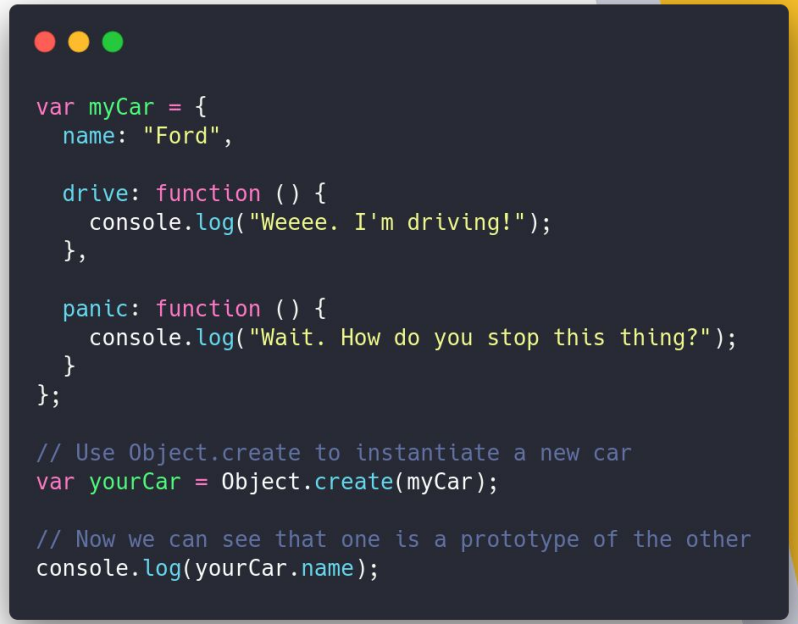
- Object maintains list of objects depending on it
- Automatically notifying objects of any changes to state
- Broadcasting notification to the observers
- One-to-many dependency between objects; pub-sub pattern

Components to implement Observer pattern

- **Subject**: maintains a list of observers, facilitates adding or removing observers
- **Observer**: provides a update interface for objects that need to be notified of a Subject's changes of state
- **ConcreteSubject**: broadcasts notifications to observers on changes of state, stores the state of ConcreteObservers
- **ConcreteObserver**: stores a reference to the ConcreteSubject, implements an update interface for the Observer to ensure state is consistent with the Subject's

Prototype pattern

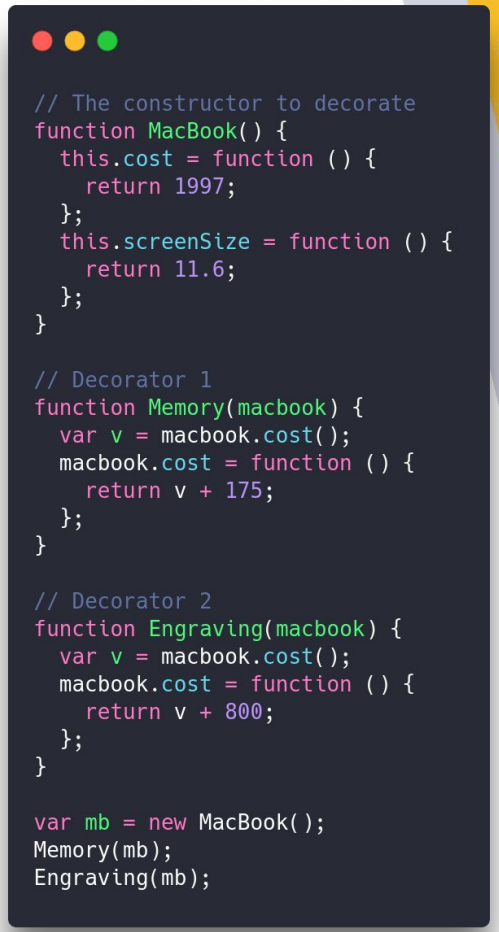
- Creates object based on template of an existing object
- Based on prototypal inheritance
- Comes with performance boost
- Functions created by reference
  (all child objects points to the same function)



```
var myCar = {
  name: "Ford",

  drive: function () {
    console.log("Weeee. I'm driving!");
  },

  panic: function () {
    console.log("Wait. How do you stop this thing?");
  }
};

// Use Object.create to instantiate a new car
var yourCar = Object.create(myCar);

// Now we can see that one is a prototype of the other
console.log(yourCar.name);
```

Decorator pattern

- Aim to promote code reuse
- Can be considered another viable alternative to object sub-classing
- Add behaviour to existing classes in a system dynamically



```
// The constructor to decorate
function MacBook() {
  this.cost = function () {
    return 1997;
  };
  this.screenSize = function () {
    return 11.6;
  };
}

// Decorator 1
function Memory(macbook) {
  var v = macbook.cost();
  macbook.cost = function () {
    return v + 175;
  };
}

// Decorator 2
function Engraving(macbook) {
  var v = macbook.cost();
  macbook.cost = function () {
    return v + 800;
  };
}

var mb = new MacBook();
Memory(mb);
Engraving(mb);
```

Command pattern

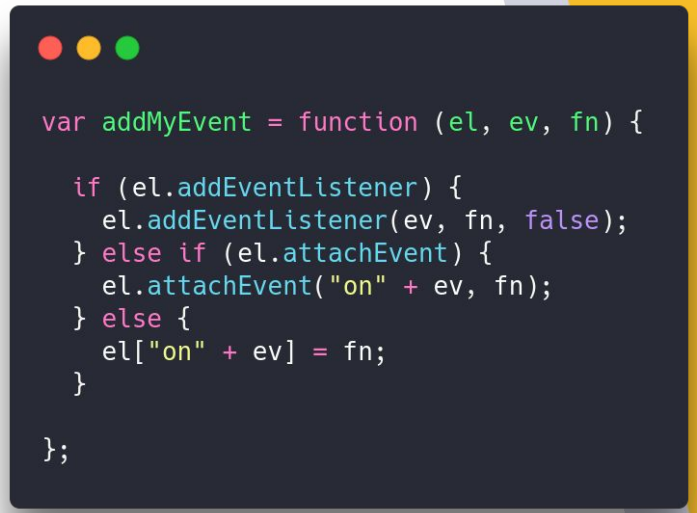
- Encapsulate method invocations, requests or operations into a single object
- Enables decouple objects
- Giving us greater degree of overall flexibility in swapping out concrete classes
- Separate the responsibilities of issuing commands from anything executing commands

```
var carManager = {
  // request information
  requestInfo: function (model, id) {
    return 'The information for ' + model + ' with ID ' + id + 'is great!';
  },
  // purchase the car
  buyVehicle: function (model, id) {
    return 'You have successfully purchased Item ' + id + ', a ' + model;
  },
};

carManager.execute = function(name) {
  return carManager[name]
    && carManager[name].apply(carManager, [].slice.call(arguments, 1));
}

var action = carManager.execute('buyVehicle', 'BMW', '435671');
```

Facade pattern

- Provides convenient higher-level interface
- Structural pattern

Typical example is
jQuery support for older browsers



```
var addMyEvent = function (el, ev, fn) {

  if (el.addEventListener) {
    el.addEventListener(ev, fn, false);
  } else if (el.attachEvent) {
    el.attachEvent("on" + ev, fn);
  } else {
    el["on" + ev] = fn;
  }

};
```

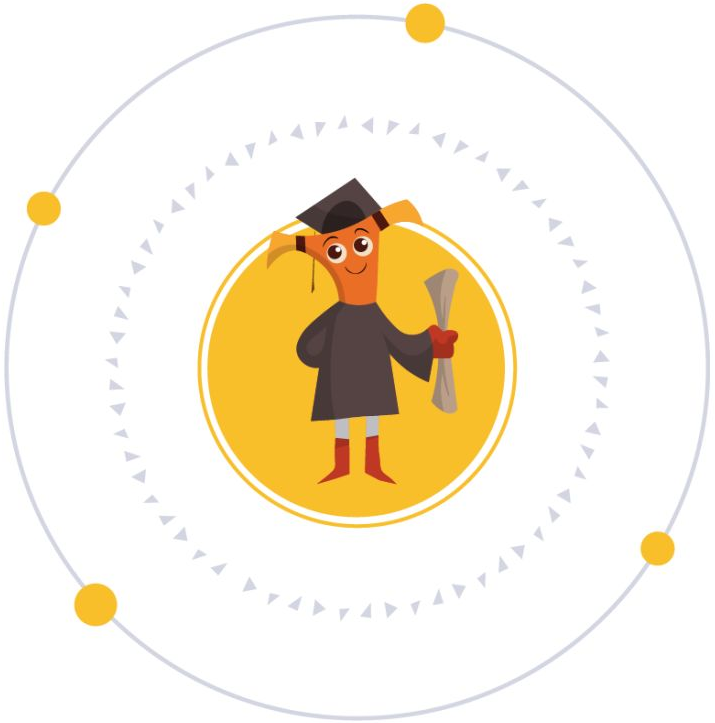
Za vježbu

Napravite klasu Person koja će imati javno svojstvo
name i privatno svojstvo age, te pripadajuće
javne/privatne metode za dohvat tih svojstava i izmjenu istih.

Koristeći odgovarajući pattern, omogućit dinamičko
dodavanje javnih svojstava u Person klasu.
Koristeći facade pattern, prikažite ime samo punoljetnih osoba.

Dodatni materijal (ebook):

Learning JavaScript Design Patterns by Addy Osmani - https://www.oreilly.com/library/view/learning-javascript-design/9781449334840/



That‘s all folks!