JavaScript and Ext JS framework

JAVASCRIPT

Part 1

3

LANGUAGE STANDARD

Part 1 - JavaScript

- ECMAScript language features are defined in the ECMA-262 standard
- JavaScript is a superset of this standard (it contains additional objects and functions, but the syntax and core language features are the same)

4

VAR DECLARATIONS AND HOISTING

- variable declarations with var keyword are treated as if they were at the top of the function or global scope (if declared outside of a function)
- this mechanism is called hoisting and also relates to functions and function expressions

VAR DECLARATIONS AND HOISTING

```
function getColor(condition) {
    if (condition) {
        var color = 'blue';
        // other code
        return color;
    } else {
        // color exists with a value of undefined
        return null;
    }
    // color exists with a value of undefined
}
```

VAR DECLARATIONS AND HOISTING

Part 1- JavaScript

```
function getColor(condition) {
    var color;
    if (condition) {
        color = 'blue';
        // other code
        return color;
    } else {
        // color exists with a value of undefined
        return null;
    }
    // color exists with a value of undefined
}
```

7

BLOCK-LEVEL DECLARATIONS WITH LET

Part 1- JavaScript

- variable declarations with let keyword are inaccessible outside a given scope
- block/lexical scope is created inside a function or a block defined by curly braces

BLOCK-LEVEL DECLARATIONS WITH LET

Part 1- JavaScript

```
function getColor(condition) {
    if (condition) {
        let color = 'blue';
        // other code
        return color;
    } else {
        // color doesn't exist here
        return null;
    }
    // color doesn't exist here
}
```

9

VARIABLE REDECLARATION WITH LET

Part 1- JavaScript

```
var age = 30;

if (condition) {
    let age = 40;
    // other code
}

// Syntax error
let age = 40;
```

10

CONSTANTS DECLARATION

- variables declared with const keyword are considered as constants - their value cannot be changed once set and they must be initialized during declaration
- constants have a block-level scope

11

CONSTANTS DECLARATION

Part 1 - JavaScript

```
const user = {
    name: 'John'
};

// works
user.name = 'Mike';

// throws an error
user = {
    name: 'Adrew'
};
```

THE TEMPORAL DEAD ZONE (TDZ)

Part 1- JavaScript

```
console.log(typeof color); // 'undefined'

if (condition) {
    console.log(typeof color); // ReferenceError!
    let color = 'blue';
}
```

13

BLOCK BINDING IN LOOPS

Part 1- JavaScript

```
for (var i = 0; i < 100; i++) { // one should use let
    // do someting
}

// i is still accessible here
console.log(i);
```

BINDINGS IN GLOBAL SCOPE

```
var RegExp = 'Test!';
console.log(window.RegExp); // 'Test!'

let RegExp = 'Test!';
console.log(RegExp);  // 'Test!'
console.log(window.RegExp === RegExp); // false
```

15

GLOBAL VARIABLES BY MISTAKE

Part 1- JavaScript

```
function add(x, y) {
    sum = x + y;
    return sum;
}
add(2,3);
console.log(sum);  // 5


function test() {
    var a = b = 5;  // var a = (b = 5);
}
```

16

VARIABLES DECLARATION BEST PRACTICES

Part 1 - JavaScript

- use const by default and let only when you know a variable will change
- avoid global variables
- declare variables at the beginning of the function or block
- always initialize variables

OBJECTS IN JAVASCRIPT

- can be treated as an associative array or a map (properties or functions are stored under specified keys)
- can be arbitrarily modified (on both type and instance level)

OBJECT LITERAL

Part 1 - JavaScript

- definition of an object is enclosed in curly braces
- each property/function declaration is separated from each other by a comma
- the key name and the value assigned to it are separated by a colon
- the definition should be terminated with a semicolon

19

OBJECT LITERAL

Part 1- JavaScript

```
const user = {
    name : 'John',
    sayHello : function() {
        console.log('Hi there!');
    }
};
```

20

CONSTRUCTOR FUNCTIONS

Part 1 - JavaScript

```
const User = function(name) {
    this.name = name;
    this.sayHello = function() {
        console.log('Hi there!');
    }
};

const user = new User('John');
user.sayHello();
```

21 CALLING CONSTRUCTOR WITHOUT NEW OPERATOR

Part 1- JavaScript

```
const Message = function () {
    this.value = 'Hello';
};

const message = Message(); // new operator is missing
console.log(window.value); // 'Hello'
```

22

SAFE CONSTRUCTOR

Part 1 - JavaScript

```
function Message() {
    if (!(this instanceof Message)) {
        return new Message();
    }
    this.value = 'Hello';
}
```

PROTOTYPE

Part 1 - JavaScript

- every function in JavaScript has a special prototype property which defines common elements for all instances created by this function
- allows sharing of functions and properties between object instances

PROTOTYPE USAGE

Part 1- JavaScript

```
function Account(number) {
    const balanceInfo = 'Balance: ';

    this.number = number;
    this.balance = 0;
    this.printBalance = function () {
        console.log(balanceInfo + this.balance);
    };
}

Account.prototype.deposit = function (funds) {
    this.balance = funds;
};

const account = new Account('00000000000000000000000001');
account.printBalance();
account.deposit(1000);
```

PROTOTYPE INHERITANCE

```
function PremiumAccount() {
}

PremiumAccount.prototype = new Account();

const premiumAccount = new PremiumAccount();
premiumAccount.deposit(1000);

console.log(premiumAccount instanceof PremiumAccount); // true
console.log(premiumAccount instanceof Account);  // true
```

CONFIGURATION OBJECTS

- allow the replacing of many method parameters by one object literal
- benefits:
  - no need to memorize individual parameters and their order
  - improved code readability
  - optional parameters
  - simple addition / deletion of parameters

CONFIGURATION OBJECTS

```
const User = function(config) {
    this.name  = config.name || 'not set';
    this.age = config.age || 0;
    this.printInfo = function() {
        console.log(this.name + ' ' + this.age);
    }
}

const user = new User({age: 14 , name : 'Jan'});
user.printInfo();
```

FUNCTIONS

Part 1- JavaScript

- play a very important role in the language, so understanding the mechanisms involved in creating and using them should be an essential part of a good programmer's workshop

FUNCTIONS AS OBJECTS

- are full-fledged objects, so the following actions are possible:
  - dynamic creation and modification of functions
  - assignment of functions to variables
  - setting/getting properties on a function
  - pass function as an argument to another function
  - return functions as a result of another function

FUNCTIONS USAGE

Part 1 - JavaScript

```
const numbers = [1, 4, 6, 64, 2];
function forEach(arr, callback) {
    for (let index = 0; index < arr.length; index++) {
        callback(arr[index], index);
    }
}
function filter(arr, predicate) {
    const result = [];
    forEach(arr, function (value) {
        if (predicate(value)) {
            result.push(value);
        }
    });
    return result;
}
function isEven(value) {
    return value % 2 === 0;
}
function print(number) {
    console.log(number);
}
forEach(filter(numbers, isEven), print);
```

31

CLOSURE

Part 1- JavaScript

```
function test(val) {
    return function () {
        console.log(++val);
    };
}

const test1 = test(0);
test1();
test1();
test1();

const test2 = test(0);
test2();
test2();
test1();
```

IMMEDIATELY INVOKED FUNCTION EXPRESSION (IIFE)

```javascript
const lib = (function (moduleName) {
    const privateVariable = 'someValue';

    function privateFunction() {
    }

    // initialization
    console.log(moduleName);

    return {
        publicFunction: privateFunction
    };
})('module');

lib.publicFunction();
```

FAT ARROW FUNCTIONS

- delivered in ECMAScript6
- defined with short arrow syntax (=>)
- behave differently than traditional JavaScript functions
  - the value of this, super, arguments inside of the function is defined by the closest containing non-arrow function
  - cannot be called with new
  - the prototype property of an arrow function doesn’t exist
  - the value of this inside of the function can’t be changed

FAT ARROW FUNCTION SYNTAX

Part 1 - JavaScript

```
const reflect = value => value;

// equivalent to:
const reflect = function(value) {
    return value;
};
```

```
const sum = (num1, num2) => num1 + num2;
const sum = (num1, num2) => { return num1 + num2; };

// equivalent to:
const sum = function(num1, num2) {
    return num1 + num2;
};
```

THIS REFERENCE

- depending on the context, it can refer to different objects

THIS IN THE GLOBAL CONTEXT

- in the global execution context (outside of any function), refers to the global object whether in strict mode or not

```
console.log(this === window); // true

a = 10;
console.log(window.a); // 10

this.b = "Hello";
console.log(window.b)  // 'Hello'
console.log(b) //  'Hello'
```

THIS IN FUNCTION CONTEXT

```
function f1() {
  return this;
}

f1() === window; // true



function f2() {
  'use strict';
  return this;
}

f2() === undefined; // true
```

THIS IN OBJECT METHOD CONTEXT

```
const user = {
    name : ‚John',
    sayHello : function() {
        console.log('Hi my name is' + this.name);
    }
};
```

THIS IN CONSTRUCTOR METHOD CONTEXT

Part 1 - JavaScript

```
const User = function(name) {
    this.name = name;
    this.sayHello = function() {
        console.log('Hi my name is' +  this.name);
    }
};

const myUser = new User('John');
```

THIS WITH CALL / APPLY FUNCTIONS

Part 1- JavaScript

```
const color = 'yellow',
      production = {
        color: 'red'
      },
      development = {
        color: 'blue'
      };

function printColor(defaultColor) {
    console.log(this.color || defaultColor);
}

printColor('blue');  // bule
printColor.apply(production, ['orange']);  // red
printColor.call(development, 'orange');   //  blue
```

BIND FUNCTION

```
function f() {
    return this.a;
}

const g = f.bind({a: 'someObject'});
console.log(g()); // someObject

const h = g.bind({a: 'otherObject'}); // bind only works once!
console.log(h()); // someObject

const o = {a: 37, f: f, g: g, h: h};
console.log(o.f(), o.g(), o.h()); // 37, someObject, someObject
```

42

SELF MADE BIND FUNCTION

Part 1 - JavaScript

```
 function bind(context, func) {
     return function () {
         return func.apply(context, arguments);
     }
 }

function f() {
    return this.a;
}

const g = bind({a: 'someObject'}, f);
console.log(g()); // someObject
```

ASYNCHRONOUS PROGRAMMING IN JAVASCRIPT

- JavaScript is single-threated language (it can only execute one piece of code at a time)
- to service asynchronous code it uses a so-called event loop
- the event loop is a process inside the JavaScript engine that monitors code execution and manages the job queue
- whenever some code is ready to run it is added to job queue and when the JavaScript engine is finished executing the current code, the event loop executes the next job in the queue

BROWSER EVENT MODEL

Part 1- JavaScript

```
const button = document.getElementById('ok-btn');
button.addEventListener('click', function(event) {
    console.log('Clicked');
});
```

CALLBACK HELL

Part 1- JavaScript

```
method1(function (err, result) {
    if (err) {
        throw err;
    }
    method2(function (err, result) {
        if (err) {
            throw err;
        }
        method3(function (err, result) {
           // some code to run
        });
    });
});
```

46

PROMISES

Part 1- JavaScript

- a promise specifies code to be executed later in time (is a placeholder for the result of an asynchronous operation) and explicitly indicates whether the code succeeded or failed at its job
- one can chain promises together based on success or failure in ways that make your code easier to understand and debug
- are implemented by all modern browsers

PROMISE LIFECYCLE

- each promise goes through a short lifecycle starting in the pending state, which indicates that the asynchronous operation hasn’t completed yet
- once the asynchronous operation completes, the promise is considered settled and enters one of two possible states:
  - fulfilled - the asynchronous operation has completed successfully
  - rejected - asynchronous operation hasn’t completed successfully due to either an error or some other cause
- one can take a specific action when a promise changes its state by using the then() method
- a fulfillment or rejection handler will still be executed even if it is added to the job queue after the promise is already settled

PROMISE USAGE

```
const promise = getData('http://host/api/resource');

promise.then(function (contents) { // fulfillment
    console.log(contents);
}, function (err) { // rejection
    console.error(err.message);
});

promise.then(function (contents) { // fulfillment
    console.log(contents);
});
```

PROMISE ERROR HANDLING

```
promise.catch(function(err) { // rejection
    console.error(err.message);
});

// is the same as:
promise.then(null, function(err) { // rejection
    console.error(err.message);
});
```

CREATING A PROMISE

```
function getData(url) {
    return new Promise(function (resolve, reject) { // trigger the asynchronous operation
        const xhr = new XMLHttpRequest();

        xhr.open('get', url);
        xhr.onreadystatechange = function () {
            if (xhr.readyState === 4) {
                if (xhr.status > 199 && xhr.status < 300) {
                    resolve(xhr.response);
                } else {
                    reject();
                }
            }
        }
        xhr.send();
    });
}
```

CREATING SETTLED PROMISE

```
const promise = Promise.resolve(100);

promise.then(function (value) {
    console.log(value);
});
```

```
const promise = Promise.reject(100);

promise.catch(function (value) {
    console.log(value);
});
```

PROMISES ERROR HANDLING

- if an error is thrown inside an executor, then the promise's rejection handler is called (only when a rejection handler is present otherwise, the error is suppressed)
- one should always handle a rejection case

CHAINING PROMISES

- each call to then() or catch() creates and returns another promise. This second promise is resolved only once the first has been fulfilled or rejected
- there are a number of ways to chain promises together to accomplish more complex asynchronous behavior

CHAINING PROMISES

```
const promise = new Promise(function (resolve, reject) {
    resolve(100);
});

promise.then(function (value) {
    console.log(value);
}).then(function () {
    console.log('Finished');
});
```

55

PASSING VALUE IN PROMISES

Part 1- JavaScript

```
let promise = new Promise(function (resolve, reject) {
    resolve(100);
});

promise.then(function (value) {
    console.log(value);
    return value + 1;
}).then(function (value) {
    console.log(value);
});
```

EXT JS FRAMEWORK

Part 2

SENCHA EXT JS

- commercial JavaScript framework
- allows to create complex web applications designed primarily for business solutions

ADVANTAGES

Part 2 - Ext JS

- modern architecture
- large number of high-quality UI controls
- support for mobile devices
- theming
- good documentation, examples and tools

DISADVANTAGES

- high entry threshold (complexity)
- relatively difficult customization
- price

EXT JS 6 - IMPORTANT FEATURES

- unified way to create apps for different platforms
- improved performance
- better SASS compiler
- support for Promises
- new components and plugins

DEVELOPER TOOLS

Part 2 - Ext JS

- sencha Ext JS SDK
- sencha CMD
- IntelliJ + Sencha plugin
- Git

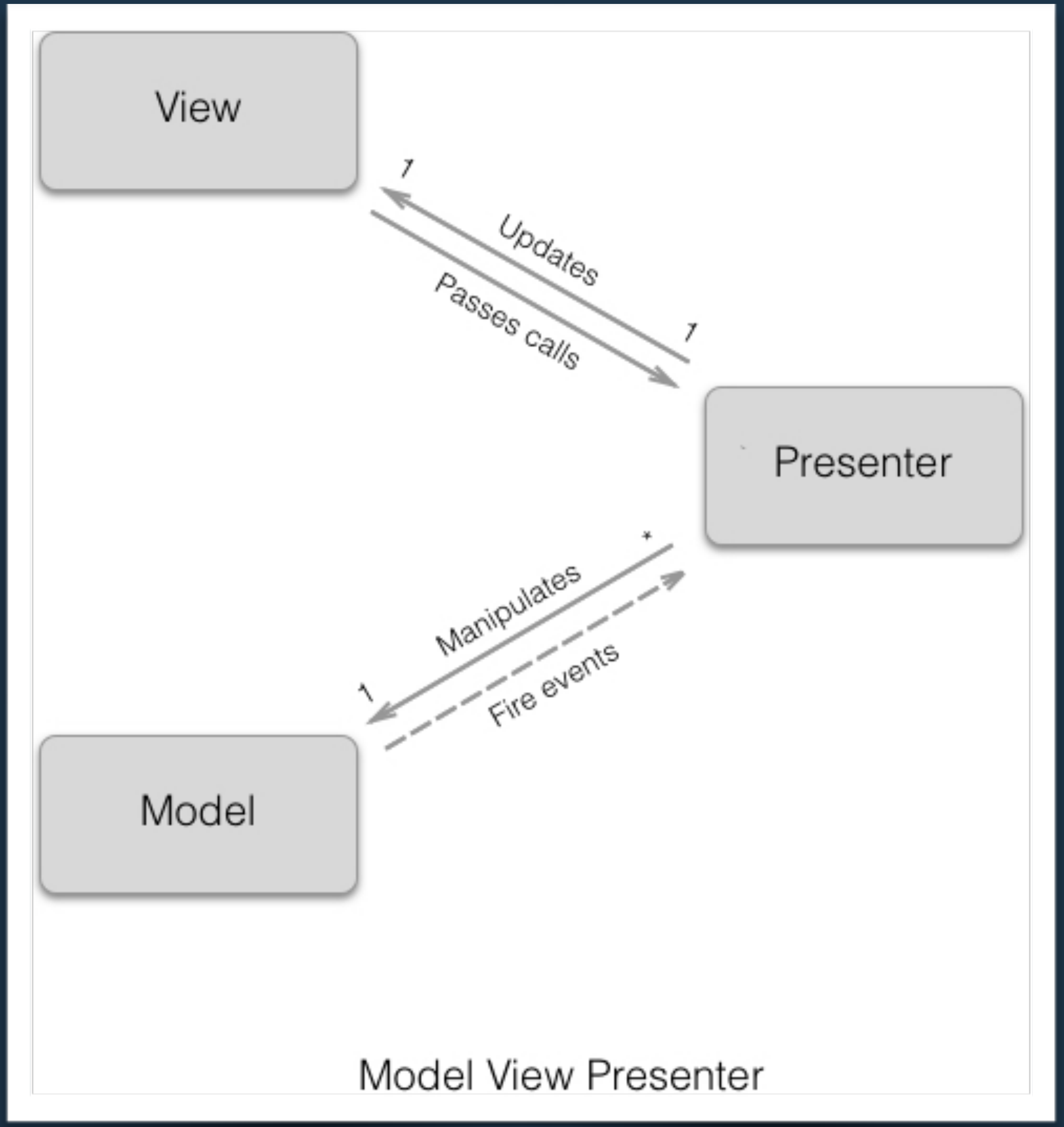
APPLICATION ARCHITECTURE

Part 2 - Ext JS

- defines types of application components, their roles and connections between them
- ensures clear separation of concerns and low coupling of elements resulting in their maintainability, scalability and reusability

MODEL VIEW CONTROLLER (MVC)

Part 2 - Ext JS



Model View Presenter

- divides an application into three layers:
  - Model - manages the data and business logic
  - View - presents information to the user
  - Controller - accepts input and converts it to commands for the model or view

HOW DOES THE MVC WORK?

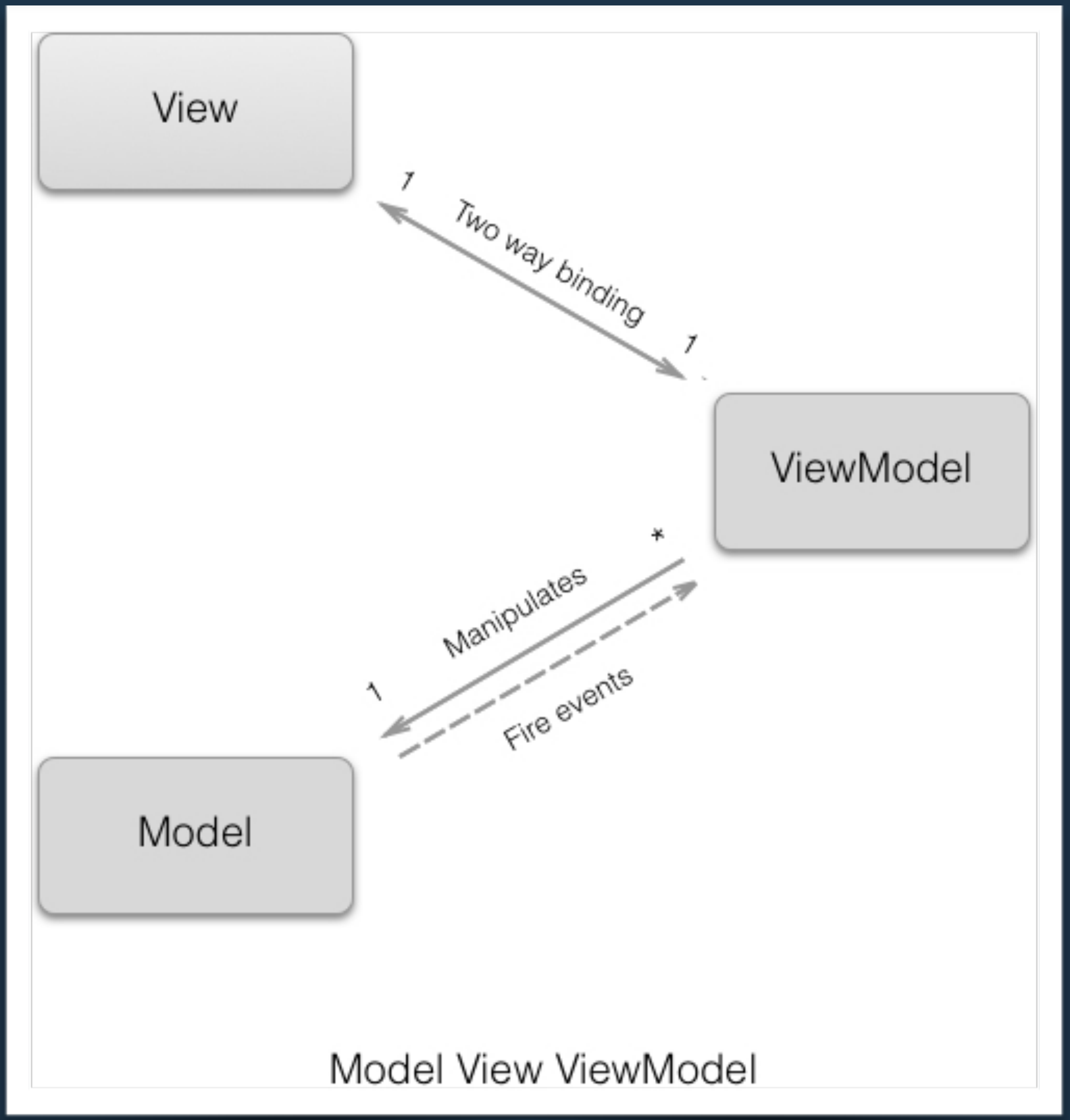
- the user interacts with Views, which displays data from Models
- Controller responds to the interactions by updating the View and Model

ADVANTAGES OF MVC

- clear separation of responsibilities which makes the app easier to test and maintain
- code reusability

MODEL VIEW VIEWMODEL (MVVM)

Part 2 - Ext JS



Model View ViewModel

- divides an application into three layers:
  - Model - manages the data and business logic
  - ViewModel - an abstraction of the view exposing public properties and commands
  - View - defines the structure and layout of the UI

MVC VS. MVVM

Part 2 - Ext JS

- the main difference between those two is ViewModel abstraction which coordinates the changes between Model's data and the Views (usually accomplished with data binding)

EXT JS ARCHITECTURE

- Ext JS supports MVC and MVVM patterns (the latter is preferred)
- both of these approaches share certain concepts and components and focus on dividing application to separate layers

EXT JS COMPONENTS - MODEL

- represents application domain (data and logic)
- built with classes called Model
  - have fields, share relationships and know how to persist themselves through the data package
  - usually used with Stores which provide data for grids and other components

EXT JS COMPONENTS - MODEL

Part 2 - Ext JS

- extends Ext.data.Model
- represents model (entities) of the application e.g. User, Invoice, Account
- the structure of the Model is defined by an array of fields
- each field is an object that contains a name and possible type (auto, string, int, float, boolean, date)
- Model class can have a relationship with each other e.g. Post has many Comments

EXT JS COMPONENTS - STORE

- extends Ext.data.Store
- represents an array of model elements
- standardizes the way of data access
- can inform view about data changes
- enables sorting, grouping and filtering data
- behaves like a local cache

EXT JS COMPONENTS - PROXY

- extends Ext.data.proxy.Proxy
- performs physical reading / writing of the data e.g. from the browser's memory or server
- most often configured at the Model level
  - usually all stores use the same model in the same way
  - there is a possibility to use a model without the Store
- Example implementations: LocalStorage, Memory, Rest

EXT JS COMPONENTS - VIEW

- components representing user interface elements
- can be composed into larger views and reused
- most often created from existing controls

EXT JS COMPONENTS - VIEW COMPONENT

- extends Ext.Component
- basic element of user interface e.g. Button, Img

EXT JS COMPONENTS - VIEW CONTAINER

- a special case of the component, which is responsible for rendering and layout of other components e.g. Panel, Window
- a typical application consists of multiple nested containers / components
- often ViewPort is the main container of the whole application

LAYOUT MANAGEMENT

- each container has assigned a default layout, which is responsible for managing the size and position of its nested elements
- available layouts: Absolute, Accordion, Border, HBox, VBox, Card, Fit, Center, Column, Table

EXT JS COMPONENTS - CONTROLLER

- manages the logic associated with the presentation e.g. rendering, model initialization, events handling, routing

EXT JS COMPONENTS - APPLICATION CONTROLLER

- extends Ext.app.Controller
- created at application launch, exists for the entire lifetime of the application
- can manage multiple instances of views
- uses selectors (component queries) or refs to match components and respond to their events

APPLICATION CONTROLLERS DISADVANTAGES

- eager initialization on startup
- no discharge mechanism
- the possibility of accidental interaction with different views
- a large number of repetitive logic e.g. search for components, attaching listeners

EXT JS COMPONENTS - VIEW CONTROLLER

- extends Ext.app.ViewController
- lifecycle associated with managed View
- reduced complexity (one-to-one relationship with the View, associated life cycle)
- scoping - selection and interaction with elements at any level below the associated view

EXT JS COMPONENTS - VIEW MODEL

- manages data specific to the View
- should be independent of the presentation layer
- by using data binding, it is possible to automatically synchronize the state of the ViewModel and the View

EXT JS COMPONENTS - VIEW MODEL

- extends Ext.app.ViewModel
- represents a model of the associated view
- enables data binding

DATA BINDING

- allows automatic state synchronization (in one or both directions, on different view hierarchy levels)
- configured declaratively at the view level

APPLICATION

Part 2 - Ext JS

- an instance of Ext.application.Application
- contains startup configuration
- specifies the namespace
- defines main components

85

ROUTING

Part 2 - Ext JS

- based on tokens and browser history
- performs navigation within the application and encapsulates its logic
- allows conditional processing

PROJECT STRUCTURE

- every Ext JS application should have unified directory structure
- all Store, Model, ViewModel and ViewController classes should be placed in app directory (each in suitable subfolder)
- ViewModels and ViewControllers should be grouped together in subfolders of app/view

PREFERRED PROJECT STRUCTURE

```
app
    model
    store
    view
classic
    src
    sass
    resources
modern
    src
    sass
    resources
resources
sass
```

CROSS PLATFORM APPLICATIONS

- from Ext JS 6, you can create applications that run on both traditional computers and mobile devices
- common base components have been standardized and the elements specific to the platform (mostly associated with the view) have been placed in separate packages (Toolkits)

CLASSIC TOOLKIT

- designed to create applications running in a desktop environment
- it is based on components / views from Ext JS 5

MODERN TOOLKIT

Part 2 - Ext JS

- designed to create applications that run in modern browsers (desktop and mobile)
- based on the component / views from Sencha Touch

EXT JS CLASS SYSTEM

- framework Ext JS provides a layer of abstraction to standardize the creation and use of custom data types and the use of mechanisms such as constructors and inheritance

Ext.define() - to define new types, also by extension, alias for Ext.ClassManager.create()

Ext.create() - to create an instance on the basis of existing class, alias for Ext.ClassManager.instantiate()

Ext.widget() - to create an instance based on the xtype name, alias for Ext.ClassManager.instantiateByAlias()

EXT OBJECT

- global singleton
- creates a base namespace
- contains other framework elements - classes, singletons, aliases, utility methods and others

EXT.DEFINE(NAME, CONFIG, [CALLBACK])

- allows to define and extend existing types

```
Ext.define('Car', {
     name: 'car',
     constructor: function(name)  {
         if (name) {
             this.name = name;
         }
     },
     start: function()  {
         alert('Car started');
     }
});
```

EXT.DEFINE(NAME, CONFIG, [CALLBACK])

```
Ext.define('ElectricCar', {
    extend: 'Car',
    start: function() {
        alert("Electric car started");
    }
});
Ext.define('Logger', {
     singleton: true,
     log: function(msg) {
       console.log(msg);
     }
});
```

EXT.CREATE([NAME], [CONFIG])

- allows to create new instances of the class
- may result in loading the class definition (synchronously)

```
var myCar = Ext.create('ElectricCar', {
    name: 'MyElectricCar'
});
```

EXT.ONREADY(CALLBACK, [SCOPE], [CONFIG])

- enables some operations to be performed in response to the page load event

```
Ext.onReady(function() {
    new Ext.Component({
        renderTo: document.body,
        html: 'DOM is ready!'
    });
});
```

EXT.WIDGET([NAME], [CONFIG])

- allows to create a widget by its xtype name

```
Ext.widget('panel', {
       renderTo: Ext.getBody(),
       title: 'Panel'
});
```

EXT.GETCLASS(OBJECT)

- returns the type of the specified instance, may return null

```
var component = new Ext.Component();

Ext.getClass(component); // returns Ext.Component
```

99

EXT.GETCLASSNAME(OBJECT)

Part 2 - Ext JS

- alias Ext.ClassManager.getName(object)
- returns type of the passed instance

```
Ext.getClassName(Ext.Action); // returns "Ext.Action"
```

OTHER IMPORTANT TYPES

- Ext.Base - base type for all framework classes
- Ext.Loader - enables asynchronous dependency loading, determined using require
- Ext.Class - stores information about the type

PRE AND POST PROCESSORS

Part 2 - Ext JS

- they participate in the preparation of the class definition

```
var pre = Ext.Class.getDefaultPreprocessors(),
post = Ext.ClassManager.defaultPostprocessors;
console.log(pre);
console.log(post);
```

PREPROCESSORS

Part 2 - Ext JS

- className - defines the namespace and class name
- loader - searches and when needed loads missing dependencies
- extend - implements inheritance of methods and properties from the parent class
- statics - defines static members of a class
- config - creates get / set methods for configuration properties
- mixins - copies the methods and properties of the indicated classes
- xtype - defines a new class of xtype
- alias - sets the alias class

103

POSTPROCESSORS

Part 2 - Ext JS

- alias - registers the class and its alias in classes manager
- singleton - creates one instance of the class
- uses - imports dependent class

MIX-INS

- the mechanism of mixing multiple classes into one
- it is based on copying the properties and / or methods of several classes to another one
- it is an alternative for inheritance (allows code reuse)
- examples Ext.util.Observable, Ext.util.Floating

MIX-INS

```
Ext.define('MyApp.Worker', {
    work: function ()  {
        console.log('Working...');
    }
});
Ext.define('MyApp.Person', {
    mixins: {
        worker: 'MyApp.Worker'
    },
    constructor: function (options) {
        Ext.apply(this, options);
    },
    compete: function ()  {
        console.log(this.work()); // logs 'Working...'
        console.log(this.mixins.worker.work()); // logs 'Working...'
    }
});
```

CONFIG

- represents a customizable set of properties
- for each configuration item postprocessor automatically creates accessor methods

CONFIG

```
Ext.define('MyApp.Invoice', {
    config: {
        tax: 0.21,
        total: 0
    },
    constructor: function (config)  {
        this.initConfig(config); // config initialization
    }
});

var invoice = Ext.create('MyApp.Invoice', {
    total: 100
});

invoice.setTax(0.23)
```

CUSTOM GET / SET LOGIC

- should be placed in the methods apply[propertyName]

```
Ext.define('MyApp.Invoice', {
    applyTotal: function (value) {
        return value + value * this.getTax(); // will be set as new value of total
    }
});
```

STATIC ELEMENTS

```
Ext.define('MyApp.Client', {
    statics: {
        SEQUENCE: 0,
        nextId: function ()  {
            return ++this.SEQUENCE;
        }
    }
});
console.log(MyApp.Client.nextId());
```

SINGLETON

- it comes in the form of a single instance
- frequently used to store permanent global application configuration like

```
Ext.define('MyApp.Constants', {
    singleton: true,
    BASE_PATH: "/myapp"
});

console.log(MyApp.Constants.BASE_PATH); // logs '/myapp'
Ext.create('MyApp.Constants'); // throws an error
```

DEPENDENCIES

Part 2 - Ext JS

- the require method creates a script tag and loads asynchronously indicated dependency, after the loading process is finished the onReady event is fired
- using Ext.create without prior indication of the dependency may result in its synchronous loading

DEFINING DEPENDENCIES

- requires - determines dependencies required during construction (preprocessor)
- uses - determines dependencies required after the instance is created (postprocessor)

```
Ext.define(‚MyApp.Bookshelf', {
    requires: ['Ext.util.MixedCollection'],
    uses: [‚MyApp.models.Book'],
});
```