JavaScript II & Einführung in Phaser
Softwareentwicklungspraktikum „Spieleentwicklung mit Javascript“

Martin Bogner

Lehr- und Forschungseinheit für Programmier- und Modellierungssprachen
Institut für Informatik
Ludwig-Maximilians-Universität München

15. Mai 2017
Objects in JavaScript

JavaScript is object-oriented

- roughly: Objects in JS are maps (dictionaries) from String to values
- (key, value) entry in an object is called an property of the object
- functions may be properties, too
- dynamic objects: add / remove properties at runtime
- prototype-based inheritance
Creating an Object

### Object Literals

```javascript
let counter = {
    counter: 0,
    changes: 0
};

counter.anotherProperty = 5
counter['anotherProperty'] // => 5
counter['anotherProperty'] = 6
counter.anotherProperty // => 6
```
Creating an Object

Object Literals

```javascript
let counter = {
    counter: 0,
    changes: 0,

    // functions are proper values, too
    increment: function() {
        counter += 1;
        changes += 1;
    }
};
```
Creating an Object

Object Literals

```javascript
let counter = {
    counter: 0,
    changes: 0,

    // ES6 shorthand for method definition:
    decrement() {
        counter -= 1;
        changes += 1;
    }
};
```
Creating an Object

Constructors

```javascript
function AdventureHero (name) {
    this.name = name;
}

const larry = new AdventureHero('Larry Laffer');
const guybrush = new AdventureHero('Guybrush Treepwood');
larry.name;    // => 'Larry Laffer'
guybrush.name; // => 'Guybrush Treepwood'
larry instanceof AdventureHero; // => true
larry.constructor.name; // => 'AdventureHero'
```

Constructor characteristics

- function with no return value
- declares object properties with `this` keyword
- `this` is bound to new object instance
- by convention: capitalize first letter
Creating an Object

Constructors

```javascript
function AdventureHero (name) {
    this.name = name;
}

const larry = new AdventureHero('Larry Laffer');
const guybrush = new AdventureHero('Guybrush Treepwood');
larry.name; // => 'Larry Laffer'
guybrush.name; // => 'Guybrush Treepwood'
larry instanceof AdventureHero; // => true
larry.constructor.name; // => 'AdventureHero'
```

Creating an Object with an constructor

- `new` keyword creates new object
- `larry` has the prototype `Person`
Inheritance

Prototypal Inheritance

- JS does not have classes
- JS implements inheritance via **prototypes**
  ⇒ objects inherit from other objects
- every constructor has a property **prototype**:  
  ⇒ same “parent object“ of all objects created by a given constructor
- setting properties of prototype ⇒ accessible in all “child“ objects

```javascript
const larry = new AdventureHero('Larry Laffer');
larry.age = 44;
larry.age;  // => 44
larry.occupation;  // => undefined

const guybrush = new AdventureHero('Guybrush Treepwood');

AdventureHero.prototype.occupation = 'adventurist';
larry.occupation;  // => 'adventurist'
guybrush.occupation;  // => 'adventurist'
```
Inheritance

Adding properties to prototype

- set function as property of prototype
  ⇒ one function instance for all objects of prototype

```javascript
// ...
AdventureHero.prototype.occupation = 'adventurist';

AdventureHero.prototype.sayGreeting = function () {
  console.log("Meet " + this.name + ", an " + this.occupation + "!");
}
guybrush.sayGreeting();
// prints "Meet Guybrush Treepwood, an adventurist!"
```
Inheritance

Overwriting prototype properties

- prototype-properties can be overwritten in single object
- all other objects keep the value of the prototype

```javascript
// guybrush occupation is modified
guybrush.occupation = 'pirate';
guybrush.sayGreeting = function(){
    console.log("Arrr, I’m " + this.name + " and I want to be a " +
                   this.occupation);
}

// => 'pirate'
guybrush.occupation;
guybrush.sayGreeting();
// prints "Arrr, I’m Guybrush Treepwood and I want to be a pirate"

// larry occupation is not modified
larry.occupation;  // => 'adventurist'
larry.sayGreeting();
// prints "Meet Larry Laffer, an adventurist!"
```
ES6 Classes

- ES6 introduces a new syntax, which resembles Java-classes
  ⇒ **internally still prototype-inheritance, just a cleaner notation**
  ⇒ **recommendation: use class-syntax for your projects**

```javascript
class AdventureHero {
  constructor(name) {
    this.name = name;
    this.occupation = "adventurist";
  }

  sayGreeting() {
    console.log("Meet " + this.name + ", an " + this.occupation);
  }
}
```
ES6 Classes - Inheritance

- ES6 introduces a new syntax, which resembles Java-classes
  ⇒ internally still prototype-inheritance, just a cleaner notation
  ⇒ recommendation: use class-syntax for your projects
- inheritance via `extends`, addressing parent-functions via `super`

```javascript
class Pirate extends AdventureHero {
  constructor(name) {
    // calls parent-constructor
    super(name);
    this.occupation = "pirate";
  }

  sayGreeting() {
    console.log("Arrr, I’m " + this.name + " and I want to be a " + this.occupation);
  }

  sayCasualGreeting() {
    super.sayGreeting();
  }
}
```
ES6 Classes - Inheritance II

```javascript
let guybrushTreepwood = new Pirate('Guybrush Treepwood');

guybrushTreepwood instanceof Pirate; // => true
guybrushTreepwood instanceof AdventureHero; // => true

guybrushTreepwood.sayGreeting();
// prints "Arrr, I’m Guybrush Treepwood and I want to be a pirate"

guybrushTreepwood.sayCasualGreeting();
// prints "Meet, Guybrush Treepwood, an pirate"
```
Gotcha: Object Equality

- equality testing with `==`/`===` only works for primitive types
- only true for objects if variables actually reference the same object
  (*dasselbe vs. das gleiche*)

```javascript
const objA = {color: 'green'};
const objB = {color: 'green'};
const objC = objA;
objA === objB; // => false
objA == objB;  // => false
objA === objC; // => true
```

⇒ You would have to iterate over all properties one by one and check for equality!
Gotcha: Object Equality

Minimal Example (doesn’t capture all cases, e.g., NaN, nested objects)

```javascript
function ObjectsAreEquivalent(a, b) {
    // Get all properties of both objects
    const aPropNames = Object.getOwnPropertyNames(a);
    const bPropNames = Object.getOwnPropertyNames(b);

    // Number of properties different => objects are different
    if (aProps.length != bProps.length) { return false; }

    for (let i = 0; i < aProps.length; i++) {
        let propertyName = aPropNames[i];
        // If values of same property are not equal,
        // objects are not equivalent
        if (a[propertyName] !== b[propertyName]) { return false; }
    }

    // objects have equal properties and property-values
    return true;
}
```

⇒ Don’t reinvent the wheel: Use utility-libraries like Lo-Dash\(^1\)

\(^1\)https://lodash.com/\(_\).isEqual(objA, objB)
Gotcha: Self-reference in JS with this

The **this** keyword

- one of the most common mistakes when learning JS
- **this** intuitive: references the object which called the function
  - **this** changes according to the context / binding a function was called
  - context can be manually set with `bind()`

```javascript
let obj = {
    name: "someObject",
    f: function () {
        return this + "":" + this.name;
    }
};

console.log(obj.f()); // => [object Object]:someObject

let f = obj.f
console.log(f()); // => [object global]:undefined
let g = obj.f.bind(obj)
console.log(g()); // => [object Object]:someObject
```
Gotcha: Self-reference in JS with this

Issues with this

let obj = {
  name: "someObject",
  f: function () {
    console.log(this.name); // this.name = "someObject"
    setTimeout(function () {
      console.log(this.name);
      // this == global Object => this.name = undefined
    }, 1000);
  }
};
obj.f(); // prints "someObject", afterwards prints "undefined"
Gotcha: Self-reference in JS with this

Issues with this - workarounds

- use bind():
- use local helper variable: let that = this;

```javascript
let obj = {
    name: "someObject",
    f: function () {
        let that = this;
        console.log(this.name); // this.name == "someObject"

        setTimeout(function () {
            console.log(this.name); // this == obj
            }.bind(this), 1000);

        setTimeout(function () {
            console.log(that.name); // that == obj
            // that == obj
            }, 1000);
    }
};

obj.f(); // prints "someObject", afterwards prints "undefined"
```
# Phaser

## Getting started

- Download and include `phaser.js` inside your html

```html
<script type="text/javascript" src="phaser.js"></script>
```

- `phaser.min.js` is the minified version of `phaser.js`, but does not include any documentation

- Load your project source code afterwards
Basic working example - rotating player sprite

```javascript
const mainState = {
    preload: function () {
        // Load and register image
        game.load.image('player', 'player.png');
    },
    create: function () {
        // Actually display the image
        this.player = game.add.sprite(200, 150, 'player');
    },
    update: function () {
        // Turns player by 1 degree, 60 times per second
        this.player.angle += 1;
    }
};

// Phaser.Game(width, height, renderer, element_id_of_game_in_html)
const game = new Phaser.Game(640, 480, Phaser.AUTO, 'gameDiv');
game.state.add('main', mainState);
game.state.start('main');
```
Phaser-States

States are the life cycle stages of a game

Most games are divided into different scenes or "states"
A game can have as many states as you want, e.g.:

- **Boot**  set general settings, load assets for preloading screen
- **Preload** load game assets, show loading bar to user
- **MainMenu** welcome screen, settings, start game, ...
- **SaveMenu** save or load a game
- **Game** actual game
- **GameOver** highscore, restart, credits, back to main menu, ...

Loading a new state destroys local variables of old state
Phaser-State

**Hook Methods**

*Every state has hook methods that get called by Phaser*

- **init**: access parameters passed into state
- **preload**: load assets
- **create**: initialise game objects, *set the stage*
- **update**: called on every game tick
- **render**: debug information
const mainState = {
  preload: function () {
    // Load and register image
    game.load.image('player', 'player.png');
  },
  create: function () {
    // Actually display the image
    this.player = game.add.sprite(200, 150, 'player');
  },
  update: function () {
    // Turns player by 1 degree, 60 times per second
    this.player.angle += 1;
  }
};

// Phaser.Game(width, height, renderer, element_id_of_game_in_html)
const game = new Phaser.Game(640, 480, Phaser.AUTO, 'gameDiv');
game.state.add('main', mainState);
game.state.start('main');
Switching States

Minimal Example

```javascript
const mainState = {
    /* see last slide */
};

const bootState = {
    create: function(){
        // immediately switch to main-state
        game.state.start('main');
    }
};

const game = new Phaser.Game(640, 480, Phaser.AUTO, 'gameDiv');
game.state.add('boot', bootState);
game.state.add('main', mainState);
//start with boot-state
```
```
## Coordinates in the Game World

Game world expands indefinitely into every direction

- Phaser uses Cartesian coordinate system
- At start time: (0,0) is top left corner of canvas

![Cartesian coordinate system](image)

![Game canvas with (0,0) origin](image)
Animations vs. Tweens

Animations

Animations show different pictures for the same sprite
- show the character “walking”
- frames usually sourced from a spritesheet
- sprite stays in same place (unless moved otherwise)

```javascript
// in preload-method
game.load.spritesheet('player_spritesheet', 'mummy37x45.png', 37, 45, 18);

// in create-method
this.player = game.add.sprite(100, 100, 'player_spritesheet')
//animations.add(name, frames, frameRate, loop)
this.player.animations.add('right', [0, 17], 8, true);
```
Animations vs. Tweens

**Tweens**

Tweens alter a property of your sprite
- e.g. position, size, or similar
- calculate requisite animation automatically
- displayed picture of a sprite stays the same

```javascript
let player = game.add.sprite(100, 100, 'player_spritesheet')

let tween = game.add.tween(this.player);

// move to yPos 200 in 1000ms
tween.to({y: 200}, 1000);

// start tween
tween.start();
```
Physics

- Phaser has several integrated physics systems.
- `Phaser.Physics.ARCADE` is the most basic.

```javascript
var mainState = {
  // ...
  create: function() {
    this.player = game.add.sprite(200, 150, 'player');
    this.floor = game.add.sprite(0, 160, 'floor');
    // activate physics system
    game.physics.startSystem(Phaser.Physics.ARCADE);
    // enable physics for specific objects
    game.physics.arcade.enable(this.player);
    game.physics.arcade.enable(this.floor);
    // properties of the objects
    this.player.body.gravity.y = 500;
    this.floor.body.immovable = true;
  },
  update: function() {
    // check for collision on every update
    game.physics.arcade.collide(this.player, this.floor);
  }
};
```
Further Reading

Phaser

- http://www.pms.ifi.lmu.de/lehre/praktikum/progprakt/17ss/#Literatur ;-)
  - http://phaser.io/tutorials/making-your-first-phaser-game
  - http://phaser.io/examples
  - https://gamedevacademy.org/how-to-debug-phaser-games/
  - ...

Resources (Sprites, Sounds, ...)

Only use resources under Creative Commons or similar licenses!
E.g.:

- http://opengameart.org/
- http://kenney.nl/assets
- http://www.freesound.org/
- ...
