Události a asynchronní zpracování

Události: opáčko

- alternativní způsob řízení toku programu
- event-based programming
- „Až nastane X, udělej Y.“

```
1. var func = function(e) {
2.     alert("...");
3. }
4. document.body.addEventListener("click", func, false);
```

Události: objekt události

- liší se dle typu události
- vždy obsahuje **target**, **currentTarget**, **timeStamp**, **type**
- vždy obsahuje **stopPropagation** a **preventDefault**

Události klávesnice

- ***keydown, keypress, keyup***
- modifikátory ctrlKey, altKey, shiftKey, metaKey

- stisk a uvolnění obsahuje **keyCode**
    - identifikátor klávesy na klávesnici (int)

- tištitelný keypress obsahuje **charCode**
    - unicode code point znaku (int)
    - String.fromCharCode převede na znak

- dělení na keydown/keypress = nepraktické, dělení na tištitelné/netištitelné = nepraktické
- aktuálně probíhá implementace upraveného API – jen keydown/keyup

- vlastnost **key** = identifikátor stisklé logické klávesy (string)
- vlastnost **code** = identifikátor stisklé fyzické (HW) klávesy

Události myši

- ***mousedown, mouseup, click, mouseover, mouseout, mousemove***
- **clientX** a **clientY** = souřadnice kurzoru vůči průhledu
- **button** je tlačítko (0 = levé, 1 = prostřední, 2 = pravé)

Události dotykové

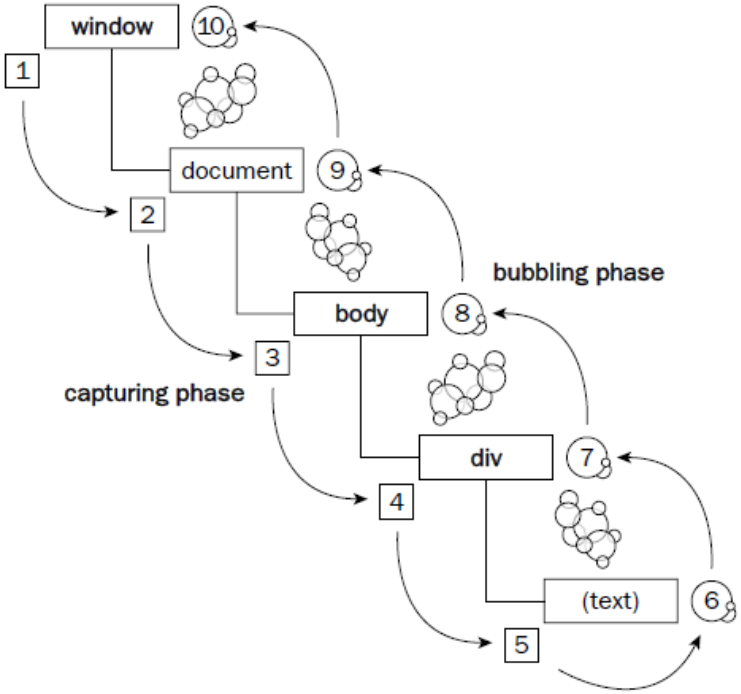
- ***touchstart, touchmove, touchend, (gesturechange)***
- **touches** = pole všech dotyků
- **changedTouches** = pole změněných dotyků
- **targetTouches** = pole dotyků na cílovém prvku
- dotyk je miniudálost, obsahuje pozici a doplňk. údaje (síla stisku, velikost prstu, natočení,...)

Události ostatní

- focus, blur
- input, change, submit
- scroll, resize
- DOMContentLoaded
- …

Události: capture a bubble

- třetí parametr pro **addEventListener** je ***useCapture***
- posluchače události jsou volány nejprve ve fázi capture, poté ve fázi bubble



- více posluchačů na stejném uzlu (ve stejné fázi) je voláno v definovaném pořadí
- zpracování události (mezi uzly) lze nastavit voláním **stopPropagation**()
- zpracování události (v rámci uzlu) lze zastavit voláním **stopImmediatePropagation**()
- metoda preventDefault() s tím nijak nesouvisí

- některé události nebublají…
  - … ale i tyto procházejí capture
  - load, unload
  - focus, blur

Události: posluchače

- **varianta 1:** posluchač je funkce

```
1. window.addEventListener("load", function(e) {
2.     alert(e.currentTarget == this);
3. });
```

    - **this** je prvek, na kterém je posluchač zavěšen (nikoliv ten, kde událost vznikla)
    - chceme-li předat parametry nebo změnit this, použijeme **bind** (nebo arrow function)

- **varianta 2:** posluchač je objekt

```
var obj = {
    handleEvent: function(e) {
        alert(this == obj);
    }
}
window.addEventListener("load", obj);
```

    - **obj** musí mít metodu **handleEvent**
    - **this** je posluchač

Asynchronní zpracování

- JS je vykonáván v jednom vlákně
- není nutné řešit přerušení a synchronizaci vykonávání

Event loop

```
var scheduledJS = "";
var listeners = [];

while (1) {
    eval(scheduledJS); /* TADY se vykoná JS */

    if (!listeners.length) break;

    /* počkat, než bude čas na nejbližší posluchač */
    var currentListener = waitFor(listeners);

    /* naplánovat jej */
    scheduledJS = listeners[currentListener];
    delete listeners[currentListener];
}
```

Zpožděné vykonávání

- je řada způsobů, jak „naplánovat“ zpožděné vykonávání kódu
- XMLHttpRequest, addEventListener
- pořadí určuje prohlížeč, ale vždy nejprve v příští iteraci event loopu

```
setTimeout(  function() { /* ... */ }, 1000);
setInterval( function() { /* ... */ }, 100);
```

Zpožděné vykonávání: this v callbacku

- Pokud někam předávám funkci, s jakým „this“ bude volána?

```
var Animal = function() {
    setTimeout(this.eat, 3000);
}

Animal.prototype.eat = function() {
    this.food += 3;
}
```

- **bind** pomůže:

```
var Animal = function() {
    setTimeout(this.eat.bind(this), 3000);
}

Animal.prototype.eat = function() {
    this.food += 3;
}
```

- **arrow function** pomůže:

```
var Animal = function() {
    setTimeout(() => this.eat(), 3000);
}

Animal.prototype.eat = function() {
    this.food += 3;
}
```

Zpožděné vykonávání: requestAnimationFrame

- **setTimeout** zní jako rozumné řešení pro JS animace
- **requestAnimationFrame** je výrazně vhodnější alternativa
- prohlížeč sám volí vhodnout délku časového kroku (zpravidla okolo 60 fps)

```
requestAnimationFrame(function() {
    /* animujeme... */
});
```

Promises

- při návrhu vlastního API narážíme na asynchronní funkce
  - takové funkce vyžadují **callback**
  - Kolikáty parametr? Co návratová hodnota? Výjimky? Co podmíněné asynchr. funkce?

- návrhový vzor **Promise** nabízí výrazně přehlednější řízení asynchronního kódu
  - je to „krabička na časem získanou hodnotu“
  - (podmíněně) asynchronní funkce vrací Promise
  - zájemce může na promise navěsit posluchače (dva různé)

```
1. var getData = function() {
2.     var promise = new Promise();
3.     /* ... */
4.     return promise;
5. }
6.
7. getData().then(
8.     function(data) { alert(data); },
9.     function(error) { alert(error); }
10. );
```

- Promise se může nacházet ve stavech pending/fulfilled/rejected
  - fulfilled/rejected == „resolved“

- tvůrce promise ji mění, konzument jen poslouchá (**then**)
- vyrobit lze již naplněnou promise: Promise.resolve(123)
- volání then() vrací novou promise → řetězení
- některé prohlížeče Promise nabízejí, pro jiné existují knihovny

```
1. getData().catch(console.error); // jako .then(null, console.error)
2.
3. var p1 = getData();
4. var p2 = getData();
5.
6. Promise.all([p1, p2]).then( ... );  // parametr callbacku je pole hodnot
7. Promise.race([p1, p2]).then( ... ); // první s hodnotou
```

Promises: tvorba a změna stavu

- je to složité – měnit stav smí jen producent
- tzn. nic jako „Promise.prototype.fulfill = …“
- API konstruktoru **new Promise** vyžaduje funkci (tzv. exekutor), které budou řídící nástroje předány

```
var promise = new Promise(function(resolve, reject) {
    /* funkce dodaná tvůrcem Promise */
    if (...) {
        resolve(value);
    } else {
        reject(error);
    }
});
return promise;
```

- nacházíme se v období přechodu z callbacků na Promises
- stará API (**setTimeout**) požadují callbacky, nová (**fetch**) vrací Promise

Async/await

- novinka z ES2017
- nadstavba nad Promises
- asynchronní funkce stále vracejí Promise
- konzument může na hodnotu čekat blokujícím způsobem

```
async function getData() {
    try {
        let data = await fetch(...); // vrací Promise
        let processed = process(data);
        return processed;            // implicitně obaleno do Promise
    } catch (e) {
        // Promise rejection
    }
}
```

- „**async**“ před deklarací funkce garantuje, že bude vždy vracet Promise

- „**await**“ před voláním funkce pozdrží vykonávání kódu, dokud nedojde k naplnění Promise, jež tato funkce vrací