UNIVERSITÀ DEGLI STUDI DI PALERMO

dipartimento di ingegneria unipa

Moduli

Corso di Programmazione Web e Mobile

a.a. 2021/2022

Prof. Roberto Pirrone

LABORATORIO DI INTERAZIONE UOMO-MACCHINA
CHILAB

Sommario

- Modularizzazione del codice
  - Struttura base di un modulo
  - Valutazione dei dati come codice
- Moduli CommonJS
  - Node.js
- Moduli ECMAScript

Modularizzazione del codice

- In Javascript, come in altri linguaggi di programmazione, i progetti di grandi dimensioni sono articolati in *pacchetti* a loro volta contenenti uno o più *moduli*

- *Modulo*: una unità di codice che espone all'esterno una data interfaccia e nasconde completamente l'implementazione interna

  - Estendibilità
  - Manutenibilità

Modularizzazione del codice

- *Pacchetto*: un componente software dedicato ad assolvere una certa categoria di compiti, autoconsistente, che può essere distribuito ed installato autonomamente
  - È articolato in uno o più moduli
  - Può dipendere da altri pacchetti o essere una *dipendenza* (necessario per il funzionamento) per altri pacchetti
  - Viene aggiornato e manutenuto autonomamente

Modularizzazione del codice

- Un progetto software dipende, in genere, da molti pacchetti che a loro volta hanno le proprie dipendenze
- Per gestire un ecosistema di pacchetti serve una infrastruttura software capace di consentire:
  - Upload di pacchetti nuovi o aggiornati
  - Download e installazione dei pacchetti disponibili
  - Gestione delle differenti versioni di un pacchetto perché sono a loro volta dipendenze per diverse versioni dei pacchetti «dipendenti»
- Node Package Manager (NPM)

Modularizzazione del codice

- L'essenza di un modulo è la possibilità di definire tutta la sua implementazione in uno *scope* (ambito di visibilità) locale e invisibile all'esterno

- L'ambiente migliore per fornire un ambito di visibilità locale è *l'esecuzione di una funzione*

- Se la funzione restituisce un oggetto con all'interno le sole proprietà/metodi di interfaccia, il resto rimane invisibile, ma utilizzabile dall'interfaccia per via del meccanismo di closure.

Modularizzazione del codice

```
const weekDay = function() {
        const names = [
        "Sunday",
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday"
        ];
        return {
                name(number) { return names[number]; },
                number(name) { return names.indexOf(name); }
        };
}();
```

```
> .load WeekDay.js
> undefined
> weekDay.name(3)
'Wednesday'
> weekDay.number('Friday')
5
```

Esegue la funzione restituendo l'oggetto `{name,number}` nello scope globale

Modularizzazione del codice

- Il meccanismo di gestione delle dipendenze implica che un modulo debba *richiedere* l'importazione di un altro da cui dipende

- Poiché i moduli importati saranno su file, dev'essere possibile valutare un flusso di caratteri in ingresso al modulo come *codice*:

```
eval(stringaCodice)
```

- `eval` usa lo scope corrente e quindi non va bene.

Modularizzazione del codice

```
let myFun = Function(stringaParametri,
                     stringaCodice)
```

- `Function` converte un flusso di caratteri in codice ed *usa il proprio scope* che è isolato da quello generale.

Moduli CommonJS

- I moduli CommonJS sono lo standard de facto che viene usato da Node.js

- Utilizzano un oggetto *convenzionalmente* chiamato `exports` per esportare le prorietà/metodi di interfaccia

- Necessitano della definizione di una funzione chiamata `require` che carica il file sorgente `.js`, `.json` o `.node` e crea un wrapper di tipo `Function` per convertire il testo in codice e creare lo scope locale.

Moduli CommonJS

```
require.cache = Object.create(null);

function require(name) {
        if (!(name in require.cache)) {

                let code = readFile(name);
                let module = { exports: {} };

                require.cache[name] = module;

                let wrapper = Function("require, exports, module", code);
                wrapper(require, module.exports, module);
        }
        return require.cache[name].exports;
}
```

La cache serve per registrare i moduli caricati e non caricarli due volte

È necessario implementare una funzione di lettura dello stream. Node.js implementa `readFile` all'interno del modulo `fs`

Moduli CommonJS

```
require.cache = Object.create(null);

function require(name) {
        if (!(name in require.cache)) {

                let code = readFile(name);
                let module = { exports: {} };

                require.cache[name] = module;

                let wrapper = Function("require, exports, module", code);
                wrapper(require, module.exports, module);
        }
        return require.cache[name].exports;
}
```

Il wrapper crea il contesto e rende disponibilii binding a `require`, all'oggetto `exports` e all'intero modulo all'interno dello scope di quest'ultimo

Il modulo deve solo elencare le proprietà di `exports` e non più invocare il wrapper al suo interno

Moduli ECMAScript

- I moduli ES sono definiti in ES rev. 6 e successive
- Si basano principalmente sulle direttive `export` e `import`

Moduli ECMAScript

```
export [default] [class | function | let ] nomeExport
export [default] {export1, … exportn}

import {export1 [as alias1], … exportn [as aliasn]} from
      «nomeModulo»

import defaultExport from «nomeModulo»
import «nomeModulo»
```

Moduli ECMAScript

- *Non* è possibile richiamare moduli ES *al di fuori* di altri moduli ES

- L'esecuzione di una applicazione che importa moduli ES necessita esplicitamente che l'ambiente di esecuzione sia etichettato come *modulo*

  - Uso di `<script type='module'></script>` sul browser

  - Definizione esplicita di un file `package.json` in cui sia stato definito l'attributo '`type': 'module'` all'interno di Node.js