Mutabilità e Immutabilità

PERCHÉ L'IMMUTABILITÀ

- Il concetto di (im)mutabilità è particolarmente importante, prima di tutto perché ha degli impatti notevoli sulla qualità del codice:
  - **manutenibilità**, ovvero «quanta fatica dovrò spendere in futuro per risolvere bug? quanta per evolvere il codice?»
  - **stabilità**, ovvero «quanto alta è la probabilità che modifiche al sorgente introducano bug inattesi?»
  - **leggibilità**, ovvero «quanta fatica dovranno spendere i miei colleghi per comprendere il comportamento del mio codice?»

Mutabilità e Immutabilità

- In JavaScript gli oggetti sono “mutabili”, ovvero la lista delle proprietà che li compongono può essere mutata dopo la loro creazione.

  Ipotizziamo di avere un oggetto `obj` inizialmente privo di alcuna proprietà:

```
const obj = {}; // a volte detto “oggetto vuoto”
```

MUTABILITÀ E IMMUTABILITÀ

Possiamo **aggiungere** proprietà:

```
obj.someNewProperty = 12;
console.log(obj); // { someNewProperty: 12 }
```

Possiamo **cambiare il valore** delle proprietà:

```
obj.someNewProperty = 345;
console.log(obj); // { someNewProperty: 345 }
```

Possiamo **rimuovere** delle proprietà:

```
delete obj.someNewProperty;
console.log(obj); // { }
```

Ciascuna di queste è una **mutazione**.

MUTABILITÀ E IMMUTABILITÀ

- La mutabilità è ormai riconosciuta universalmente come la causa di numerosi bug e instabilità del software.

  Linguaggi di programmazione più moderni introducono l'immutabilità di default — come Rust.

  Altri ancora lo rendono addirittura l'unica opzione — come Haskell.

MUTABILITÀ E IMMUTABILITÀ

- Perché la mutabilità è considerata “deprecata”?

  Prendiamo il seguente codice:

  ```
  const obj = { someProperty: 12 };
  someFunction(obj);
  ```

  Dopo questo codice `obj.someProperty` varrà ancora `12`? L'unica risposta possibile è “non lo sappiamo” a meno che di non conoscere il codice di `someFunction`.

  In poche parole, se un oggetto viene mutato da un’altra porzione di codice, per poter fare delle assunzioni corrette su tale oggetto dobbiamo conoscerne il comportamento.

MUTABILITÀ E IMMUTABILITÀ

- L'immutabilità non è gratis: nuovi oggetti significa un uso più estensivo della memoria (anche se quelli vecchi vengono “garbage collected”).

  I vantaggi superano però di gran lunga i costi.

- JavaScript, dovendo rimanere retro-compatibile, non si può permettere di cambiare strada, per cui gli oggetti rimangono mutabili di default e possiamo:

  1. essere disciplinati e trattarli come immutabili (evitando le “mutazioni” di cui sopra), oppure

  2. utilizzare alcune costose (dal punto di vista delle performance) API per rendere immutabile un oggetto — `Object.freeze` e `Object.seal`.

> NOTA
> sebbene non possa cambiare il comportamento di default, o forzare un nuovo comportamento, ci sono alcune proposte per introdurre una nuova categoria di oggetti immutabile di default: i Record. Non è ancora stata standardizzata.

MUTABILITÀ E IMMUTABILITÀ

Come si “modifica” un oggetto senza poterlo “mutare”? La risposta è semplice: è necessario creare ogni volta nuovi oggetti.

> NOTA
> Poiché è un approccio tedioso, vedremo tra poco che sono state introdotte sintassi specifiche che ci aiutano molto: **Object e Array Spread**.

```
const original = {
  a: 1,
  b: 2,
  c: 3
};

const modified = {
  a: original.a,
  b: original.b,
  c: 345
};
```

ARRAY

- Gli array in JavaScript sono la struttura dati per gestire "liste di valori".

```
const arr = [1, 2, 3, 4];
console.log(arr[2]); // 3
```

> NOTA
> Gli array di JavaScript sono più simili alle `java.util.List` di Java che ai agli array di Java: come per le `List`, può cambiare il numero di elementi, possono essere rimossi ed aggiungi elementi.

- Particolare attenzione va posta agli array, perché alcuni metodi mutano l'array su cui vengono chiamati. Non c'è altra strada che imparare a memoria il comportamento di ciascun metodo, oppure preoccuparsene ogni volta.

  Di seguito elenchiamo questi metodi “malevoli” cui dobbiamo stare attenti.

ARRAY

- `.copyWithin()` (ES2015)

> Copies a sequence of array elements within the array.

- `.fill()` (ES2015)

> Fills all the elements of an array from a start index to an end index with a static value.

```
new Array(4).fill(9); // [9, 9, 9, 9]
```

- `.pop()`

> Removes the last element from an array and returns that element.

```
const arr = [1, 2, 3, 4];
arr.pop(); // 4
arr; // [1, 2, 3]
```

ARRAY

- `.push()`

> Adds one or more elements to the end of an array and returns the new length of the array.

In assoluto quello più comunemente usato tra questi metodi. **Non bisogna mai dimenticarsi che muta l'array!**

```
const arr = [1, 2, 3, 4];
arr.push(5); // 5
arr; // [1, 2, 3, 4, 5]
```

ARRAY

- `.reverse()`

> Reverses the order of the elements of an array in place — the first becomes the last, and the last becomes the first.

Molto comune. **Non bisogna mai dimenticarsi che muta l'array!**

```
const arr = [1, 2, 3, 4];
arr.reverse(); // [4, 3, 2, 1]
arr; // [4, 3, 2, 1]
```

ARRAY

- `.shift()`

> Removes the first element from an array and returns that element.

```
const arr = [1, 2, 3, 4];
arr.shift(); // 1;
arr; // [2, 3, 4]
```

- `.sort()`

> Sorts the elements of an array in place and returns the array.

Molto comune. **Non bisogna mai dimenticarsi che muta l'array!**

```
const arr = [1, 4, 3, 2];
arr.sort(); // [1, 2, 3, 4]
arr; // [1, 2, 3, 4]
```

ARRAY

- `.splice()`

> Adds and/or removes elements from an array.

Evitate di usarlo, è confusionario, e nessuno ci azzecca mai.

- `.unshift()`

> Adds one or more elements to the front of an array and returns the new length of the array.

```
const arr = [1, 2, 3, 4];
arr.unshift(0); // 5
arr; // [0, 1, 2, 3, 4]
```