Rivisitazione gioco (ver. 3)

- Il codice è stato riprogettato e implementato ad oggetti.
- Funzionalità aggiuntive:
  - salvataggio del miglior punteggio
  - pulsante per resettare il miglior punteggio.

Laboratorio 5

Struttura del progetto:

- **root**:
  - index.html
  - **js**:
    - ball.js
    - game.js
    - gameStateFlag.js
    - player.js
    - playground.js
    - popup.js
    - prey.js
    - scoreStat.js
    - sketcher.js
    - util.js
  - CSS:
    - game.css
    - **img:**
      - contiene le immagini utilizzate dal CSS

LocalStorage

- Il salvataggio del miglior punteggio è ottenuto attraverso l'utilizzo della LocalStorage:
  - al caricamento della pagina viene controllato se è presente un punteggio all'interno della localStorage.
  - se presente, il punteggio viene utilizzato per impostare il valore Max-Score, altrimenti Max-Score viene inizializzato a 0.
  - ogni volta che il giocatore effettua un nuovo miglior punteggio, tale punteggio viene memorizzato all'interno della localStorage.
  - il pulsante permette di reimpostare a 0 il miglior punteggio.
  - Il miglior punteggio viene memorizzato nella local storage con la chiave '*game*'.


```
var GAME_ID = 'game'

function ScoreStat() {
        this.playCount = 1;
        this.bestScore = 0;
        this.currentScore = 0;

        this.load();
}

ScoreStat.prototype.updateScore =
        function(){
                if (this.bestScore < this.currentScore){
                        this.bestScore = this.currentScore;
                        this.save();
                }
                this.currentScore = 0;
                this.playCount++;
        }

ScoreStat.prototype.incrementScore =
        function(valueToAdd){
                this.currentScore += valueToAdd;
        }

ScoreStat.prototype.save =
        function(){
                if (!checkLocalStorageSupport())
                        return;

                window.localStorage.setItem(GAME_ID, this.bestScore);
        }

ScoreStat.prototype.load =
        function(){
                if (!checkLocalStorageSupport())
                        return;

                var bestScore = window.localStorage.getItem(GAME_ID);
                if (bestScore !== null)
                        this.bestScore = bestScore;

        }

function resetMaxScore(){
        if (!checkLocalStorageSupport())
                        return;

        window.localStorage.setItem(GAME_ID, 0);
        location.reload();
}
```


Progettazione

- Prima di iniziare a scrivere il codice, è necessaria una fase di progettazione, in cui si devono:
  - identificare e definire delle *entità* (oggetti)
  - solitamente le *entità* sono dotate di una propria identità, nel senso che rappresentano un'astrazione di un determinato oggetto nel mondo reale o dello specifico problema da modellare
  - una volta identificate le *entità* è necessario individuare per ogni oggetto quali possano essere le caratteristiche o attributi (*property*) che si vogliono modellare, i *metodi* per poter interagire con l'oggetto stesso e le *relazioni* tra i vari oggetti.
- La definizione degli oggetti determina la flessibilità della nostra applicazione, in termini di modifiche da apportare al codice nel caso in cui si vogliano introdurre nuove funzionalità.

Oggetti

- *Player:* identifica l'entità giocatore, mantenendo durante il gioco le informazioni relative alla sua posizione (*point*) e alla sua dimensione (*radius*)
- *Prey:* identifica l'entità preda, mantenendo durante il gioco le informazioni relative alla sua posizione (*point*) e alla sua dimensione *(halfLength)*
- *Ball*: identifica l'entità nemico/malus, mantenendo durante il gioco le informazioni relative alla sua posizione (*point)*, alla sua dimensione (*radius*), alla sua velocità di movimento (*stepX* e *stepY*) e al tipo (*type*).
- *Playground*: identifica il campo di gioco, mantenendo durante il gioco le informazioni relative alla sua larghezza (*width*) e altezza (*height*) e alla posizione all'interno della pagina (*offsetLeft*, *offsetTop*)

Oggetti

- *Game* definisce l'entità gioco e mantiene tutte le informazioni relative ad una determinata partita, ossia mantiene le informazioni relative allo stato di gioco. I metodi dell'oggetto modificano tale stato (punteggio, trasparenza degli elementi, timer, ecc) e determinano *quando* l'interfaccia grafica deve essere aggiornata. L'oggetto è una composizione di altri oggetti.
- *GameStateFlag* mantiene le informazioni relative allo stato di gioco. Più in particolare, sono presenti delle property che ci permettono di capire in che stato si trova la partita di gioco in corso (in pausa, in modalità trasparenza, ecc).
- *ScoreStat* mantiene le informazioni relative al punteggio di gioco (*currentScore, bestScore, playCount*).

Oggetti

- *Sketcher* definisce un'entità 'disegnatore' che si occupa di gestire l'interfaccia grafica del gioco.
- *Point* mantiene lo stato/coordinate di un determinato punto:
  - *x:* coordinata x
  - *y:* coordinata y
- Inoltre è presente un oggetto, *MathUtil*, che in modo simile all'oggetto globale *Math*, definisce dei metodi di utilità per il calcolo della distanza tra due punti qualsiasi:
  - *distance(point1, point2):* metodo che restituisce la distanza euclidea tra due punti.
  - *squareDistance(point1, point2):* metodo che restituisce il quadrato della distanza euclidea tra due punti.

Vantaggi e Svantaggi

- I vantaggi e gli svantaggi sono quelli tipici della programmazione ad oggetti
- Vantaggi:
  - fornisce un supporto naturale per la modellazione software, definendo delle corrispondenze tra gli oggetti software e gli oggetti del mondo reale o del problema astratto da modellare
  - permette una più facile gestione e manutenzione del codice, soprattutto per progetti di grandi dimensioni
  - favorisce la modularità e il riuso del codice
- Svantaggi:
  - overhead in termini di
    - tempo di esecuzione
    - memoria

Modello architetturale

- Analizzando la struttura o architettura del nostro codice possiamo identificare tre componenti principali:
  - Lo stato o **model** che mantiene le informazioni relative allo stato del gioco. Si pensi agli oggetti *GameStateFlag, ScoreStat, Player, Prey, Ball, Point* e *Playground*
  - La **view** che si occupa della gestione dell'interfaccia grafica e di *come* debba essere mostrata all'utente in base allo stato del gioco (modello) . Si pensi all'oggetto *Sketcher*.
  - Il **controller** che riceve i comandi dall'utente, modifica lo stato di gioco e determina *quando* l'interfaccia grafica deve essere aggiornata. Si pensi all'oggetto *Game*. In particolare, i suoi metodi definiscono la logica dell'applicazione.
- Questo modello architetturale prende il nome di **MVC** (**M**odel-**V**iew-**C**ontroller).

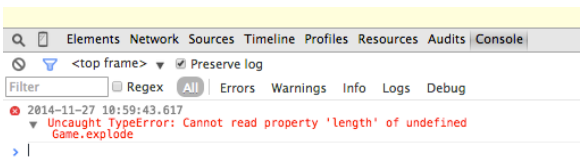
Function#bind

- Javascript non ha classi, ma oggetti e a differenza di altri linguaggi come C++, Java o C#, la keyword **this** è determinata interamente da *come la funzione è richiamata* e non da *dove la funzione è stata definita.*
- Questo non garantisce che quando la funzione sarà richiamata, la keyword **this** abbia effettivamente il valore che ci aspettiamo.
- Facciamo un esempio:
  - Prendiamo la linea di codice 14 del file game.js senza considerare il metodo *bind*

    playground.addEventListener('click', this.explode, false);
  - In questo caso stiamo associando all'evento di click dell'elemento playground (nel nostro caso l'elemento div, *HtmlDivElement*, con class='playground' che contiene graficamente il campo di gioco) la funzione *explode* dell'oggetto *Game*.

Function#bind

- Se eseguiamo il codice e proviamo a cliccare sul nostro campo di gioco, noteremo degli errori nella console Javascript del browser.



- In particolare, noteremo che tutte le property riferite tramite la keyoword **this** all'interno del metodo *explode* non sono definite.
- Infatti l'oggetto riferito da **this** è l'oggetto *HtmlDivElement* che ha generato l'evento di click.

Function#bind

- Per ovviare a questo problema è necessario utilizzare il metodo *bind* dell'oggetto *Function*:
  playground.addEventListener('click', this.explode.bind(this), false);
- Il metodo *bind* prende in ingresso un parametro permettendo di definire quale debba essere l'oggetto che deve essere riferito dalla property **this** quando la funzione verrà richiamata.
- Nel nostro caso passiamo come argomento della funzione proprio l'istanza dell'oggetto *Game*, ossia *this.*

- Si noti che il metodo *bind* è stato introdotto con le specifiche ECMA5 e quindi il suo corretto funzionamento non è garantito nei browser più datati o non conformi a tale standard.