BlackJack v3.0 Patrones ES5

Contenido

1. Introducción

En este documento vamos a trabajar la transformación de la aplicación para el uso de patrones de diseño versión ES5. Las primeras modificaciones que vamos a hacer es la resolución de los ejercicios planteados en la versión anterior. Vamos a aplicar el patrón *Singleton* al objeto *Game* y el iterador para la propiedad *cards* de *TablePlayer*.

Las instrucciones en este documento no van a ser muy detalladas por haber sido tratadas en documentos anteriores. En esta ocasión nos limitaremos a enumerar los pasos realizados.

2. Resolución de ejercicios

2.1. Victorias del jugador

Vamos a mejorar el juego no solo contando las victorias del jugador sino también las del crupier, además vamos a crear un contador de todas las partidas creadas para que se vaya mostrando por pantalla.

Comenzamos creando un campo privado en *Game* para contar el número de partidas que hemos jugado.

```
let _gameCounter = 0;
```

Debemos habilitar una propiedad de acceso para mostrar en la interfaz el número de partidas.

```
Object.defineProperty(this, 'gameCounter', {
  get() {
    return _gameCounter;
  },
});
```

Las victorias están relacionadas con el objeto *TablePlayer*, por tanto, añadimos un campo privado en el constructor que sirva de contador de partidas ganadoras así como las propiedades de acceso para que se puedan actualizar.

```
let _winCounter = 0;
```

Aquí tenemos la propiedad de acceso.

```
Object.defineProperty(this, 'winCounter', {
  get() {
    return _winCounter;
  },
  set(value) {
    _winCounter = value;
  },
});
```

El método *playGame()* del objeto *Game* que inicia una nueva partida debe actualizar el contador de partidas. El método lo pasamos al cuerpo de la función constructora por necesitar acceso al contador y que se encuentra como campo privado. Incrementamos el contador después de iniciar la partida de los jugadores.

```
_gameCounter += 1;
```

Creamos un nuevo método en *Game* que vamos a utilizar para actualizar los contadores de los jugadores después de acabar la partida. El método lo dejamos en la función constructora pensando evolucionar la aplicación un número de jugadores mayor de uno.

```
this.endGame = function () {
  // Actualización de contadores
  if (this.checkPlayerWinner()) {
    this.player.winCounter += 1;
    if (this.player.checkBlackJack()) this.player.winCounter += 0.5;
  }
  if (this.checkDealerWinner()) {
    this.dealer.winCounter += 1;
  }
};
```

Tenemos que invocar a este método al final de *playDealer()* ya que es cuando acaba la partida, y estamos en condiciones de actualizar los contadores.

```
this.endGame();
```

Añadimos una capa en *index.html* para mostrar las victorias de cada jugador.

```
<div class="py-4">
    <h2>Partidas</h2>
    Número de partidas: <span class="bg-primary text-white" id="game-
counter">0</span>. <br>
    Victorias jugador: <span class="bg-success text-white" id="player-
counter">0</span>.
    Victorias crupier: <span class="bg-danger text-white" id="dealer-
counter">0</span>.
</div>
```

En el fichero *blackjackDOM.js* declaramos los objetos DOM con los contadores para actualizar la página.

```
const counters = {
  gameCounter: document.getElementById('game-counter'),
  playerCounter: document.getElementById('player-counter'),
  dealerCounter: document.getElementById('dealer-counter'),
};
```

Reescribimos la función *showWinners()* para actualizar los contadores en la página. Antes de actualizar los contadores debemos marcar la partida como finalizada.

```
function showWinners() {
  showStatusPlayer();
  showStatusDealer();
  playerDOM.result.innerHTML = game.checkPlayerWinner() ? 'Ganador' :
'Perdedor';
  dealerDOM.result.innerHTML = game.checkDealerWinner() ? 'Ganador' :
'Perdedor';
  counters.gameCounter.innerHTML = game.gameCounter;
  counters.playerCounter.innerHTML = game.player.winCounter;
  counters.dealerCounter.innerHTML = game.dealer.winCounter;
}
```

Ya solo falta actualizar el chequeo de victorias de los jugadores para que en caso de empate no se considere una victoria.

```
Game.prototype.checkPlayerWinner = function () {
  return !!((this.player.score <= 21
    && (this.dealer.score > 21
      || this.player.score >= this.dealer.score)
    && this.dealer.score !== this.player.score));
};
Game.prototype.checkDealerWinner = function () {
  return !!((this.dealer.score <= 21
    && (this.player.score > 21
      || this.dealer.score >= this.player.score)
    && this.dealer.score !== this.player.score));
};
```

2.2. Puntuación del dealer

La restricción para la jugada del dealer es que debe jugar siempre y cuando su puntuación sea menor de 17, independientemente de la jugada del player. Como vemos en el código tenemos que simplificar la condición del bucle while. No siempre tendremos que complicar el código, algunas soluciones pueden ser muy sencillas como en este caso.

```
Game.prototype.playDealer = function () {
  try {
    while (this.dealer.score < 17) {
      this.dealer.addCard(this.deck.getNextCard());
    }
  } catch (upper21Exception) {
  }
};
```

3. Singleton

Creamos la estructura del patrón.

```
const Game = (function (){
  let instantiated = null;

  function init(){
  }

  return {
    getInstance: function () {
      if (!instantiated) {
        instantiated = init();
      }
      return instantiated;
    }
  };
})();
```

En la función anónima añadimos la clase *Game*, y en *init()* creamos una instancia congelada suya.

```
function init() {
  const game = new Game();
  Object.freeze(game);
  return game;
}
```

Por último, creamos una instancia a partir del patrón Singleton para poder utilizarla en la Vista que genera la interfaz en la página.

```
let game = Game.getInstance();
```

4. Clase abstracta *TablePlayer*

Comenzamos definiendo la expresión IIFE.

```
(function(){
  let abstractCreateLock = true;

})();
```

Añadimos la clase *TablePlayer* con el chequeo del seguro al comienzo de la función.

```
if (abstractCreateLock) { throw new Error("You can't instantiate
TablePlayer!"); }
abstractCreateLock = true;
```

Añadimos las subclases con la desactivación del cerrojo previa la llamada al superconstructor. Lo hacemos tanto en *Player* como en *Dealer*.

```
abstractCreateLock = false;
TablePlayer.call(this, name);
```

Por último, declaramos las clases como globales.

```
window.TablePlayer = TablePlayer;
window.Player = Player;
window.Dealer = Dealer;
```

5. Iterador de cartas

Creamos un iterador de cartas en *TablePlayer*. Sustituimos la propiedad *cards* por el iterador.

```
Object.defineProperty(this, 'cards', {
  get() {
    let nextIndex = 0;
    return {
      next() {
        return nextIndex < _cards.length
          ? { value: _cards[nextIndex++], done: false }
          : { done: true };
      },
    };
  },
});
```

Los métodos que utilizan la propiedad de *cards* los vamos a añadir en el cuerpo del constructor para que sigan teniendo acceso al array. Los métodos son *getScore()* y *cardsToString()*.

```
this.getScore = function (){
  let score = 0;
  let hasAce = 0;
  for (let i = 0; i < _cards.length; i++) {
    score += _cards[i].score;
    if (_cards[i].value === "A") {
      hasAce++;
    }
  }
  while (hasAce > 0 && score > 21){
    score -= 10;
    hasAce--;
  }
  return score;
}

this.cardsToString = function (){
  return _cards.join(";");
}
```

Por último, el método *showFirstRound()* de *Dealer* debe hacer uso del iterador por no tener acceso al array para devolver la primera carta del jugador *dealer*.

```
Dealer.prototype.showFirstRound = function (){
  return this.cards.next().value;
}
```