BlackJack v4.0 Objetos ES6

Contenido

1. Introducción

En este documento vamos a trabajar la transformación de la aplicación para el uso de objetos versión ES6.

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.

Para la declaración de los objetos seguiremos los pasos básicos:

1. Declaración de campos privados.
2. Declaración de constructor. Validamos argumentos de entrada para su asignación con campos privados o propiedades públicas.
3. Declaramos getter/setter de campos privados.
4. Declaramos métodos públicos.
5. Declaramos propiedades y métodos estáticos.

2. Excepciones

Transformamos las excepciones para que hereden de *BaseException*, especialmente modificamos *Upper21Exception* para informar que hemos sobrepasado el límite de 21.

```
class BaseException extends Error {
  constructor (message = "", fileName, lineNumber){
    super(message, fileName, lineNumber);
    this.name = "BaseException";
    if (Error.captureStackTrace) {
      Error.captureStackTrace(this, BaseException)
    }
  }
}
class InvalidAccessConstructorException extends BaseException {
  constructor (fileName, lineNumber){
    super("Constructor can’t be called as a function.", fileName, lineNum
ber);
    this.name = "InvalidAccessConstructorException";
  }
}

//Excepción personalizada para indicar valores vacios.
class EmptyValueException extends BaseException {
  constructor (param, fileName, lineNumber){
    super("Error: The parameter " + param + " can't be empty.", fileName,
 lineNumber);
    this.param = param;
    this.name = "EmptyValueException";
  }
}
```

```
//Excepción de valor inválido
class InvalidValueException extends BaseException {
  constructor (param, value, fileName, lineNumber){
    super(`Error: The paramenter ${param} has an invalid value. (${param}
: ${value})`, fileName, lineNumber);
    this.param = param;
    this.name = "EmptyValueException";
  }
}

//Excepción personalizada para clases abstractas.
class AbstractClassException extends BaseException {
  constructor (className, fileName, lineNumber){
    super(`Error: The class  ${className} is abstract.`, fileName, lineNu
mber);
    this.className = className;
    this.name = "AbstractClassException";
  }
}

class Upper21Exception extends BaseException {
  constructor (fileName, lineNumber){
    super(`The score is upper 21.`, fileName, lineNumber);
    this.name = "Upper21Exception";
  }
}
```

3. Clase *Card*

No tenemos campos privados en la clase, y seguimos utilizando las propiedades configuradas de solo lectura. Movemos las propiedades asignadas al constructor como estáticas en la clase.

```
class Card {
  constructor(suit, value){
    if (!new.target) throw new InvalidAccessConstructorException();
    //Validación de argumentos de entrada
    if (Card.suits.indexOf(suit) === -1)
      throw new InvalidValueException(suit, "suit");
    if (Card.values.indexOf(value) === -1)
      throw new InvalidValueException(value, "value");

    // Definición de propiedades no configurables y de solo lectura.
    Object.defineProperty(this, "suit", {
      value: suit,
      enumerable: true,
      writable: false,
      configurable: false
    });
```

```
    Object.defineProperty(this, "value", {
      value: value,
      enumerable: true,
      writable: false,
      configurable: false
    });

    let score;
    switch (value){
      case "A":
        score = 11;
        break;
      case "K": case "Q": case "J":
        score = 10;
        break;
      default: score = +value;
    }

    Object.defineProperty(this, "score", {
      value: score,
      enumerable: true,
      writable: false,
      configurable: false
    });

  }

  toString(){
    return this.suit + "-" + this.value;
  }

  static get suits(){
    return ["Corazón", "Trébol", "Diamante", "Pica"];
  }
  static get values(){
    return ["A","K","Q","J","10","9","8","7","6","5","4","3","2"];
  }
}
```

4. Clase *Deck*

En la clase *Deck* estamos utilizando métodos privados, por lo que debemos declararlos antes de ser utilizados, justo después de la declaración de campos privados. La asignación de valores para campos privados la hacemos en el constructor, y no en su declaración.

```
class Deck {
  #num;
  #deck;
  #upperLimit = false;
  #limit;

  #createDeck() {
    this.#deck = [];
    for (let i=0; i<this.#num; i++){
      for (let suitIdx = 0; suitIdx < Card.suits.length; suitIdx++) {
        for (let valueIdx = 0; valueIdx < Card.values.length; valueIdx++)
 {
          let card = new Card (Card.suits[suitIdx], Card.values[valueIdx]
);
          this.#deck.push(card);
        }
      }
    }
  }

  #shuffleDeck() {
    for (let i = 0; i < this.#deck.length; i++) {
      let swapIdx = Math.trunc(Math.random() * this.#deck.length);
      let tmp = this.#deck[swapIdx];
      this.#deck[swapIdx] = this.#deck[i];
      this.#deck[i] = tmp;
    }
  }

  constructor(num = 1){
    if (!new.target) throw new InvalidAccessConstructorException();
    this.#num = num;
    this.#limit = Math.trunc(Card.suits.length * Card.values.length * thi
s.#num * 0.2);
    this.#createDeck();
    this.#shuffleDeck();
  }

  get upperLimit(){
    return this.#upperLimit;
  }
```

```
  getNextCard(){
    if (this.#deck.length <= this.#limit){
      this.#upperLimit = true;
    }
    return this.#deck.pop();
  }

  renewDeck (){
    this.#createDeck();
    this.#shuffleDeck();
  }

  emptyDeck (callback){
    let index = 0;
    while (this.#deck.length > 0){
      callback(this.getNextCard(), ++index, this.upperLimit);
    }
  }
}
```

5. Estructura TablePlayer

Transformamos las 3 clases indicando que las clases tienen que extender de *TablePlayer*.

5.1. Clase TablePlayer

No debemos olvidar en el iterador *cards* declarar una variable que referencie el array, para que no se pierda en el *closure*. El getter debe devolver el iterador directamente, por esa razón hemos implementado una función generadora local para que, a partir de ella, obtengamos el iterador.

```
class TablePlayer {
  #cards = [];
  #score = 0;
  #winCounter = 0;

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

```
  constructor(name) {
    if (!new.target) throw new InvalidAccessConstructorException();
    if (new.target === TablePlayer) throw new Error("You can't
instantiate TablePlayer!");
    if (!name) throw new InvalidValueException(name, 'name');
    Object.defineProperty(this, 'name', {
      value: name,
      enumerable: true,
      writable: false,
      configurable: false,
    });
  }

  get cards() {
    const array = this.#cards;
    function* cardsGenerator() {
      for (const card of array) {
        yield card;
      }
    }
    return cardsGenerator();
  }

  get score() {
    return this.#score;
  }

  get winCounter() {
    return this.#winCounter;
  }

  set winCounter(value) {
    this.#winCounter = value;
  }

  initPlay() {
    this.#cards.length = 0;
  }

  addCard(card) {
    if (!card instanceof Card) throw new NullCardException();
    this.#cards.push(card);
    this.#score = this.#getScore();
    // Chequeo de puntuación.
    if (this.score > 21) throw new Upper21Exception();
    return this.#cards.length;
  }
```

```
  checkBlackJack() {
    return (this.#cards.length === 2 && this.score === 21);
  }

  cardsToString() {
    return this.#cards.join(';');
  }
}
```

5.2. Clase *Player*

```
class Player extends TablePlayer {
  #winnigPlays = 0;

  constructor(name) {
    if (!new.target) throw new InvalidAccessConstructorException();
    super(name);
  }
}
```

5.3. Clase Dealer

```
class Dealer extends TablePlayer {
  constructor(name) {
    if (!new.target) throw new InvalidAccessConstructorException();
    super(name);
  }

  showFirstRound() {
    return this.cards.next().value;
  }
}
```

6. Clase *Game*

La última clase es *Game*, la cual es modificada dentro del patrón *Singleton*.

```
class Game {
  #gameCounter = 0;

  constructor() {
    if (!new.target) throw new InvalidAccessConstructorException();

    Object.defineProperty(this, 'player', {
      value: new Player('player1'),
      enumerable: true,
      writable: false,
      configurable: false,
    });
    Object.defineProperty(this, 'dealer', {
      value: new Dealer('dealer'),
      enumerable: true,
      writable: false,
      configurable: false,
    });
    Object.defineProperty(this, 'deck', {
      value: new Deck(4),
      enumerable: true,
      writable: false,
      configurable: false,
    });
    Object.defineProperty(this, 'gameCounter', {
      get() {
        return this.#gameCounter;
      },
    });
  }

  playGame() {
    if (this.deck.upperLimit) this.deck.renewDeck();
    this.player.initPlay();
    this.dealer.initPlay();
    this.#gameCounter += 1;
    for (let i = 0; i < 2; i++) {
      this.player.addCard(this.deck.getNextCard());
      this.dealer.addCard(this.deck.getNextCard());
    }
  }
```

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

  playPlayer() {
    this.player.addCard(this.deck.getNextCard());
  }

  playDealer() {
    try {
      while (this.dealer.score < 17) {
        this.dealer.addCard(this.deck.getNextCard());
      }
    } catch (upper21Exception) {
    }
    this.endGame();
  }

  checkPlayerWinner() {
    return !!((this.player.score <= 21
      && (this.dealer.score > 21
        || this.player.score >= this.dealer.score)
      && this.dealer.score !== this.player.score));
  }

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

7. Implementación modular

Comenzamos por las excepciones. Exportamos todas las excepciones disponibles.

```
export {
  BaseException,
  InvalidAccessConstructorException,
  EmptyValueException,
  InvalidValueException,
  AbstractClassException,
  Upper21Exception,
};
```

Las excepciones son utilizadas tanto en el juego como en los objetos, por lo que las importamos en ambos ficheros.

```
import {
  BaseException,
  InvalidAccessConstructorException,
  EmptyValueException,
  InvalidValueException,
  AbstractClassException,
  Upper21Exception,
} from './exceptions.js';
```

Exportamos las clases de los objetos que hemos creado.

```
export {
  Card, Deck, TablePlayer, Player, Dealer,
};
```

Las importamos en el juego.

```
import {
  Card, Deck, TablePlayer, Player, Dealer,
} from './objects.js';
```

Según la implementación que tenemos, para no cambiar nada, exportamos la instancia del juego que hemos creado. En este caso incluso no haría falta tener implementado un Singleton, ya que que los módulos tienen esa misma naturaleza.

```
export default game;
```

En el fichero donde implementamos la interfaz importamos el juego.

```
import game from './blackjack.js';
```

Por último, importamos en la página HTML el fichero DOM como un módulo.

```
<script src="./js/blackjack/v4/blackjackDOM.js" type="module"></script>
```