BlackJack v2.0

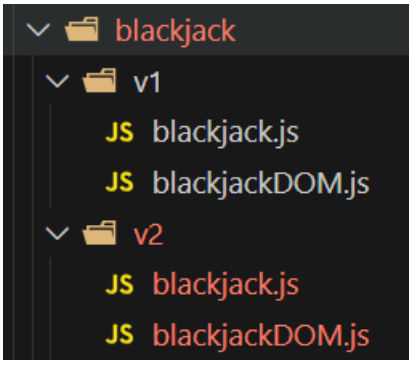
Contenido

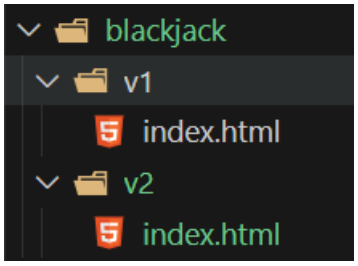
1. Introducción 2
2. Declaración de objetos 2
2.1. Excepciones *BaseException* 3
2.2. Objeto *Card* 4
2.3. Función de testeo. Objeto *Card* 6
2.4. Objeto *Deck* 7
2.5. Testeo objeto *Deck* 9
2.6. Objeto *TablePlayer* 9
2.6.1. Objeto *Player* 12
2.6.2. Objeto *Dealer* 13
2.7. Testeo objeto *TablePlayer* 13
3. Implementación del juego 14
3.1. Objeto *Game* 14
3.1.1. Método *playGame()* 15
3.1.2. Método *playPlayer()* 15
3.1.3. Método *playDealer()* 15
3.1.4. Métodos *checkPlayerWinner()* y *checkDealerWinner()* 15
3.1.5. Instanciar el objeto *Game* 16
4. Interfaz gráfica del juego 16
4.1. Manejador de eventos *newGameButton* 16
4.2. Manejador de eventos *hitButton* 17
4.3. Manejador de eventos *stayButton* 17
4.4. Funciones para mostrar el estado de los jugadores 17
4.5. Función *showWinners()* 18
5. Ejercicios 18
5.1. Contador de partidas ganadas 18
5.2. Jugada del dealer 18
6. Conclusiones 18

1. Introducción

Vamos a transformar nuestra versión de BlackJack para que utilice objetos en lugar de tipos primitivos, lo que permitirá mejorar el mantenimiento de la aplicación. Partimos de la versión 1.0 desarrollada en la unidad anterior.

Para no perder los cambios de la versión anterior, el código ha quedado estructurado en versiones, tanto a nivel de HTML como a de JS. Hemos movido el contenido a una carpeta *blackjack/v1* y hemos creado una carpeta *blackjack/v2* tanto para el HTML como para el JS para comenzar a trabajar.





El elemento `base` de **index.html** ha sido modificado para subir dos niveles y soportar con ello la nueva estructura de directorios.

```
<base href="../..">
```

2. Declaración de objetos

Vamos a crear un fichero **objects.js** para que contenga los objetos que necesitamos en nuestra aplicación. Además, necesitamos un fichero **exceptions.js** que contendrá los objetos de nuestras excepciones.

Debemos integrar ambos ficheros en **index.html**.

```
<script src="./js/blackjack/v2/objects.js"></script>
<script src="./js/blackjack/v2/exceptions.js"></script>
<script src="./js/blackjack/v2/blackjack.js"></script>
<script src="./js/blackjack/v2/blackjackDOM.js"></script>
```

2.1. Excepciones *BaseException*

Creamos la clase de excepciones base que hemos utilizado en los documentos previos. Las clases *InvalidAccessConstructorException*, *EmptyValueException* y *InvalidValueException* son muy reutilizables entre aplicaciones.

```
function BaseException(message = "Default Message", fileName,
lineNumber) {
  let instance = new Error(message, fileName, lineNumber);
  instance.name = "MyError";
  Object.setPrototypeOf(instance, Object.getPrototypeOf(this));
  if (Error.captureStackTrace) {
    Error.captureStackTrace(instance, BaseException);
  }
  return instance;
}
BaseException.prototype = Object.create(Error.prototype, {
  constructor: {
    value: BaseException,
    enumerable: false,
    writable: true,
    configurable: true
  }
});

//Excepción acceso inválido a constructor
function InvalidAccessConstructorException() {
  let instance = BaseException.call(this, "Constructor can't be called as
 a function.");
  instance.name = "InvalidAccessConstructorException";
  return instance;
}
InvalidAccessConstructorException.prototype = Object.create(BaseException
.prototype);
InvalidAccessConstructorException.prototype.constructor = InvalidAccessCo
nstructorException;

//Excepción personalizada para indicar valores vacios.
function EmptyValueException(param) {
  let instance = BaseException.call(this, "Error: The parameter " + param
 + " can't be empty.");
  instance.name = "EmptyValueException";
  instance.param = param;
  return instance;
}
EmptyValueException.prototype = Object.create(BaseException.prototype);
EmptyValueException.prototype.constructor = EmptyValueException;
```

```
//Excepción de valor inválido
function InvalidValueException(param, value) {
  let instance = BaseException.call(this, "Error: The paramenter " + para
m + " has an invalid value. (" + param + ": " + value + ")");
  instance.name = "InvalidValueException";
  instance.param = param;
  instance.value = value;
  return instance;
}
InvalidValueException.prototype = Object.create(BaseException.prototype);
InvalidValueException.prototype.constructor = InvalidValueException;
```

2.2. Objeto *Card*

Creamos un objeto *Card* para definir las cartas que podemos utilizar en el juego. Tenemos dos propiedades *suit*y *value* con el tipo de carta y su valor. Estas propiedades las haremos de solo lectura para que no puedan ser modificadas una vez construido el objeto. Antes de la asignación, debemos comprobar si el valor introducido es válido, en caso contrario lanzaremos una excepción.

```
function Card(suit, value){
  //La función se invoca con el operador new
  if (!(this instanceof Card))
    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
  });
}
Card.prototype.constructor = Card;
```

Como hemos podido observar, la validación de argumentos se hace en función de propiedades estáticas en la función constructora con los tipos y los valores de carta que podemos utilizar.

```
Card.suits = ["Corazón", "Trébol", "Diamante", "Pica"];
Card.values = ["A","K","Q","J","10","9","8","7","6","5","4","3","2"];
```

Vamos a añadir una propiedad *score* que nos devuelva la puntuación de cada tipo de carta.

```
let _score = 0;
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
});
```

Por último, tenemos que un método *toString()* para transformar el objeto en un *string*. Lo hacemos desde el prototipo para que todas las instancias puedan heredar el método.

```
Card.prototype.toString = function(){
  return this.suit + "-" + this.value;
}
```

2.3. Función de testeo. Objeto *Card*

Añadimos una función de testeo para ir comprobando cada uno de los objetos creados. La función *testCard* instancia objetos *Card* y chequea las posible las excepciones que puede lanzar.

```
function test(){
  function cardTest(){
    let c1 = new Card(Card.suits[0],Card.values[0]);
    console.log(c1.toString()); // Corazón-A
    try {
      c1.suit = Card.suits[1];
    } catch (error){
      // TypeError: Cannot assign to read only property 'suit' of object
'[object Object]'
      console.log(error.toString());
    }
    try {
      c1.suit = Card.values[1];
    } catch (error){
      // TypeError: Cannot assign to read only property 'suit' of object
'[object Object]'
      console.log(error.toString());
    }
    try {
      let c2 = new Card("AAA",Card.values[0]);
    } catch (error){
      // InvalidValueException: Error: The paramenter AAA has an invalid
value. (AAA: suit)
      console.log(error.toString());
    }
    try {
      let c3 = new Card(Card.suits[0],"AAA");
    } catch (error){
      // InvalidValueException: Error: The paramenter AAA has an invalid
value. (AAA: value)
      console.log(error.toString());
    }
  }
  cardTest();
}
test();
```

Si todo va correctamente eliminamos los arrays *suits* y *values* en **blackjack.js**, así como la función *getCardNumericValue(card)*.

2.4. Objeto *Deck*

Este objeto nos permite crear un mazo de cartas. Recibe como argumento el número de barajas que vamos a utilizar en el mazo. Los campos privados son *_num* para contener el número de barajas que vamos a utilizar y *_deck* para contener un array, aunque en primera instancia no está asignado, para contener todas las cartas. Los campos deben ser privados porque no queremos que puedan ser accedidos desde fuera del objeto.

A continuación debemos invocar las funciones *createDeck()* para crear el array con las cartas, y *shuffleDeck()* para mezclarlas de forma aleatoria como hicimos en la primera versión. Estas funciones, al alterar los campos privados, obligatoriamente deben estar en el cuerpo del constructor como veremos a continuación.

```
function Deck(num = 1){
  //La función se invoca con el operador new
  if (!(this instanceof Deck))
    throw new InvalidAccessConstructorException();
  // Campos privados
  let _num = num;
  let _deck;
  createDeck();
  shuffleDeck();
}
Deck.prototype.constructor = Deck;
```

La función *createDeck()* crea en el array las cartas necesarias, pero en lugar de un objetos literal utilizaremos los objetos *Card*. Tenemos tres bucles anidados. El primero para recorrer el número de barajas utilizadas, el segundo para recorrer los tipos de cartas en base al array estático de *Card* y el tercero con los valores en base al array estático de *Card*.

```
function createDeck() {
  _deck = [];
  for (let i=0; i<_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]);
        _deck.push(card);
      }
    }
  }
}
```

En la función *shuffleDeck()* tan solo debemos cambiar el array con el campo privado *_deck*.

```
function shuffleDeck() {
  for (let i = 0; i < _deck.length; i++) {
    let swapIdx = Math.trunc(Math.random() * _deck.length);
    let tmp = _deck[swapIdx];
    _deck[swapIdx] = _deck[i];
    _deck[i] = tmp;
  }
}
```

Tenemos una limitación en el juego ya que no podemos quedarnos sin cartas en el mazo en mitad de una partida. Vamos declarar una propiedad booleana que nos diga si solo queda el 20% de las cartas en el mazo. Declaramos dos campos privados, *_upperlimit* que contendrá si hemos alcanzado el 20%, en principio queda declarado como *false* y *_limit* que calcula la cantidad de cartas que suponen el 20% del total.

```
let _upperLimit = false;
const _limit = Math.trunc(Card.suits.length * Card.values.length * _num *
0.2);
```

Para hacer accesible este valor creamos una propiedad de acceso al campo privado. No es necesario un *setter*.

```
Object.defineProperty(this, 'upperLimit', {
  get() {
    return _upperLimit;
  },
});
```

En esta versión vamos a implementar un método *getNextCard()* que sacará de una en una las cartas del mazo para poder repartirlas durante el juego. El método queda definido en el cuerpo del constructor porque debe acceder al campo privado *_deck*. Cada vez que sacamos una carta comprobamos si hemos excedido del 20% del total.

```
this.getNextCard = function () {
  if (_deck.length <= _limit) {
    _upperLimit = true;
  }
  return _deck.pop();
};
```

Necesitamos un método que nos permita regenerar un mazo de cartas nuevo y lo mezcle.

```
this.renewDeck = function () {
  createDeck();
  shuffleDeck();
};
```

Por último, en el constructor vamos a crear un método que consuma todas las cartas del mazo y invocando una función *callback* por cada consumición para que se pueda personalizar la forma de hacer el vaciado. La función recibirá como argumentos un objeto *Card*, el índice con el número de carta que estamos consumiendo, y un booleando indicar si hemos superado el límite del 20%.

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

2.5. Testeo objeto *Deck*

Para hacer el testeo creamos una función que cree un objeto *Deck* y lo vacíe mostrando el contenido de cada carta.

```
function deckTest(){
  let deck = new Deck(4);
  deck.emptyDeck(function(card, index, upperLimit){
    console.log(index + ": " + card.toString() + " (" + card.score + ") "
 + upperLimit);
  });
}
```

Si todo funciona correctamente podemos eliminar de **blackjack.js** las funciones *createDeck()* y *shuffleDeck()* que han sido implementadas en el cuerpo de *Deck* y *getNextCard()* como método.

2.6. Objeto *TablePlayer*

El objeto *TablePlayer* lo implementamos como clase padre que permita crear una jerarquía para objetos del tipo *Player* y *Dealer*, ya que ambos comparten propiedades y métodos.

En primera instancia implementamos en el constructor con la propiedad *name*, pública y de solo lectura, la cual tendremos que validar para verificar que el nombre no es vacío.

Como campos privados tendremos dos. En primer lugar, *_cards* contiene las cartas que vayan recibiendo cada jugador. Aunque no conviene dar acceso directo a un campo privado que sea un array, ya que nos permitiría modificar el contenido del array sin seguir la lógica de negocio de nuestra aplicación, de momento habilitaremos una propiedad de acceso con el fin de corregirla en posteriores versiones.

El segundo campo privado es *_score* utilizado para almacenar la puntuación de las cartas del jugador. Los campos cuyo valor es calculado no conviene almacenarlo, pero en esta ocasión hacemos una excepción, primero porque el valor del campos solo se modifica en un único punto, al recoger una carta nueva, y en segundo lugar porque es un valor calculado en bucle y que tiene que ser consultado frecuentemente.

```
function TablePlayer(name){
  //La función se invoca con el operador new
  if (!(this instanceof TablePlayer))
    throw new InvalidAccessConstructorException();

  //Validación de argumentos de entrada
  if (!name) throw new InvalidValueException(name, "name");

  // Campos privados
  const _cards = [];
  let _score = 0;

  // Propiedades públicas
  Object.defineProperty(this, "name", {
    value: name,
    enumerable: true,
    writable: false,
    configurable: false
  });

  Object.defineProperty(this, "cards", {
    get: function(){
      return _cards;
    }
  });

  Object.defineProperty(this, "score", {
    get: function(){
      return _score;
    }
  });
}
TablePlayer.prototype.constructor = TablePlayer;
```

Creamos un método para añadir una nueva carta para el jugador. Recibe un argumento *Card* y lo añadimos al campo privado.

```
this.addCard = function (card) {
  if (!card instanceof Card) throw new NullCardException();
  _cards.push(card);
  return _cards.length;
};
```

Como vemos, hemos creado una excepción para identificar que un argumento *Card* es válido. Lo añadimos en el fichero **exceptions.js**.

```
//Excepciones BlackJackException
function BlackJackException(message) {
  message = message || "Error: BlackJack Exception";
  let instance = BaseException.call(this, message);
  instance.name = "BlackJackException";
  return instance;
}
BlackJackException.prototype = Object.create(BaseException.prototype);
BlackJackException.prototype.constructor = BlackJackException;

function NullCardException() {
  let instance = BlackJackException.call(this, "Error: Card Argument is n
ull or invalid.");
  instance.name = "NullCardException";
  return instance;
}
NullCardException.prototype = Object.create(BlackJackException.prototype)
;
NullCardException.prototype.constructor = NullCardException;
```

Dentro del cuerpo del constructor añadimos el método *initPlay()* que reinicia el array *_cards* para recoger las cartas de la siguiente partida.

```
this.initPlay = function(){
  _cards.length = 0;
}
```

Creamos un método en el prototipo del constructor *getScore()* para calcular el valor de las cartas del jugador. Este método hace uso de la propiedad pública *cards*.

```
TablePlayer.prototype.getScore = function (){
  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;
}
```

El siguiente método que implementamos genera el contenido del campo privado a *string*. Nuevamente lo implementamos en el prototipo y haremos uso de la propiedad *cards*.

```
TablePlayer.prototype.cardsToString = function (){
  return this.cards.join(";");
}
```

Vamos a definir una excepción que identifique cuándo hemos rebasado el límite de 21.

```
function Upper21Exception() {
  let instance = BlackJackException.call(this, "The score is upper 21.");
  instance.name = "Upper21Exception";
  return instance;
}
Upper21Exception.prototype = Object.create(BlackJackException.prototype);
Upper21Exception.prototype.constructor = Upper21Exception;
```

Por si sola la excepción no hace nada. Vamos a chequear la puntuación del jugador cada vez añadimos una nueva carta al jugador.

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

El siguiente método por implementar nos dice si el jugador tiene una mano BlackJack, es decir, tiene una puntuación de 21.

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

Por último, implementamos el método *showFirstRound()* para mostrar la primera ronda de cartas.

```
TablePlayer.prototype.showFirstRound = function () {
  return this.cardsToString();
};
```

2.6.1. Objeto *Player*

Este objeto hereda sus propiedades de *TablePlayer*.

```
function Player(name){
  //La función se invoca con el operador new
  if (!(this instanceof Player))
    throw new InvalidAccessConstructorException();

  TablePlayer.call(this, name);
}
Player.prototype.constructor = Player;
Player.prototype = Object.create(TablePlayer.prototype);
```

2.6.2. Objeto *Dealer*

Para el objeto *Dealer* hereda también de *TablePlayer* pero sobrescribe el método *showFirstRound()* para solamente mostrar la primera carta recibida, lo que hace más real el juego al desconocer el marcador del jugador dealer.

```
function Dealer(name){
  //La función se invoca con el operador new
  if (!(this instanceof Dealer))
    throw new InvalidAccessConstructorException();

    TablePlayer.call(this, name);
}
Dealer.prototype.constructor = Dealer;
Dealer.prototype = Object.create(TablePlayer.prototype);
Dealer.prototype.showFirstRound = function (){
  return this.cards[0].toString();
}
```

2.7. Testeo objeto *TablePlayer*

Para hacer el testeo implementamos un bucle infinito que recoja cartas de un mazo dentro de un bloque try. El bucle finaliza cuando salta la excepción de haber sobrepasado el límite de 21.

```
function playerTest(){
  let deck = new Deck(4);
  let player = new Player("Player1");
  player.addCard(deck.getNextCard());
  player.addCard(deck.getNextCard());
  console.log("Primera mano " + player.name + ": " + player.showFirstRoun
d());
  try {
    while(true){
      player.addCard(deck.getNextCard());
      console.log(player.cardsToString() + ": " + player.score);
    }
  } catch (exception){
    console.log (exception instanceof Upper21Exception); //true
    // Upper21Exception: The score is upper 21.
    console.log (exception.toString());
  }
  console.log(player.cardsToString() + ": " + player.score);
  let dealer = new Dealer("Dealer");
  dealer.addCard(deck.getNextCard());
  dealer.addCard(deck.getNextCard());
  console.log("Primera mano " + dealer.name + ": " + dealer.showFirstRoun
d());
}
```

Si está implementado correctamente podemos borrar las siguientes funciones:

- *getScore()*
- *cardsToString()*
- *checkBlackJack()*
- *isPlayerScoreUpperLimit()*

3. Implementación del juego

Vamos a modificar el fichero **blackjack.js** para rediseñar las funciones respecto a los objetos que hemos creado. En este momento, el fichero solo contiene las funciones que implementan los controladores con la lógica de nuestro juego.

3.1. Objeto *Game*

En el objeto *Game* vamos a implementar los métodos con la lógica de negocio de nuestro juego. Creamos un constructor con las siguientes propiedades:

- *player*: creamos un objeto *Player* de solo lectura.
- *dealer*: creamos un objeto *Dealer* de solo lectura.
- *deck*: Contiene el mazo de cartas sobre el que estamos jugando. También es de solo lectura.

Si lo pensamos, con estos tres objetos tenemos lo necesario para poder jugar.

```
function Game(){
  //La función se invoca con el operador new
  if (!(this instanceof Game))
    throw new InvalidAccessConstructorException();

  // Definición de propiedades no configurables y de solo lectura.
  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
  });
}
Game.prototype.constructor = Game;
```

3.1.1. Método *playGame()*

Este método nos permite iniciar una partida. En la versión anterior, el método creaba el mazo de cartas, lo mezclaba y ejecutaba el turno inicial. Ahora ya tenemos el mazo creado al instanciar *Game*, por lo que vamos a implementar directamente en este método la partida inicial.

Primero comprobamos que el total de cartas en el mazo no ha superado el límite, si lo ha hecho mezclamos las cartas nuevamente.

Al crear la propiedad *deck* ya tenemos creado el mazo de forma aleatoria. Inicializamos el array de cartas para cada jugador de la partida, y le repartimos dos cartas a cada uno de ellos.

```
Game.prototype.playGame = function () {
  // Renovamos el mazo si estamos por encima del límite de cartas
  if (this.deck.upperLimit) this.deck.renewDeck();
  this.player.initPlay();
  this.dealer.initPlay();
  for (let i = 0; i < 2; i++) {
    this.player.addCard(this.deck.getNextCard());
    this.dealer.addCard(this.deck.getNextCard());
  }
};
```

La función *initialTurn()* puede ser eliminada.

3.1.2. Método *playPlayer()*

El método consiste en pedir una nueva carta para el jugador, y guardarla en su array.

```
Game.prototype.playPlayer = function () {
  this.player.addCard(this.deck.getNextCard());
};
```

3.1.3. Método *playDealer()*

La jugada del dealer consiste en solicitar cartas mientras la puntuación del jugador es menor de 21, es decir, no se ha pasado y su puntuación del dealer es menor que la del jugador. En el caso de pasarnos, el método *addCard()* de *Dealer* generará una excepción rompiendo el bucle.

```
Game.prototype.playDealer = function () {
  try {
    while (this.player.score <= 21
      && this.dealer.score < this.player.score) {
      this.dealer.addCard(this.deck.getNextCard());
    }
  } catch (upper21Exception) {
  }
};
```

3.1.4. Métodos *checkPlayerWinner()* y *checkDealerWinner()*

Los dos métodos comparan sin han ganado la partida en función de las puntuaciones de cada jugador en la partida, y siempre teniendo en cuenta que no se hayan pasado.

```
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));
};
```

Hemos añadido una corrección para que en caso de empate no gane ningún jugador.

3.1.5. Instanciar el objeto *Game*

Necesitamos una instancia del objeto *Game* para poder jugar y podemos eliminar

```
let game = new Game();
```

Podemos eliminar el resto de funciones de la versión anterior.

4. Interfaz gráfica del juego

El fichero **blackjackDOM.js** contiene las funciones que interactúan con la página HTML, es decir, tenemos implementada la vista de la página. Los cambios en este fichero son modificar los manejadores de eventos para que utilicen el objeto *game*.

4.1. Manejador de eventos *newGameButton*

Se encarga de limpiar la página para la nueva partida e inicializar el primer turno. Una vez repartidas las primeras cartas, comprueba si un jugador tiene BlackJack en la primera mano para desencadenar las acciones de finalizar la partida y mostrar los resultados.

```
newGameButton.addEventListener("click", function() {
  cleanGame();
  game.playGame();
  //Ningún jugador en la mesa tiene BlackJack
  if (!game.dealer.checkBlackJack() && !game.player.checkBlackJack()){
    showGameButtons();
    showInitStatus();
  } else { //Un jugador tiene BlackJack
    //Jugador tiene BlackJack dejamos que juegue el dealer
    if (game.player.checkBlackJack()){
      game.playDealer();
    }
    //La partida finaliza independientemente del tipo del BlackJack
    showWinners();
    hideGameButtons();
  }
});
```

4.2. Manejador de eventos *hitButton*

El jugador pide una nueva carta y se muestra el resultado. En el caso de que la carta haga pasarse al jugador se genera una excepción para que juegue el dealer.

```
hitButton.addEventListener("click", function() {
  try {
    game.playPlayer();
    showStatusPlayer();
  } catch(upper21Exception){
    game.playDealer();
    showWinners();
    hideGameButtons();
  }
});
```

4.3. Manejador de eventos *stayButton*

Si nos plantamos el dealer realiza su jugada y se muestran los resultados.

```
stayButton.addEventListener("click", function() {
  game.playDealer();
  showWinners();
  hideGameButtons();
});
```

4.4. Funciones para mostrar el estado de los jugadores

Estas funciones muestran cómo evoluciona el juego a lo largo de la partida, teniendo en cuenta que inicialmente, el dealer no puede mostrar todas sus cartas, solamente una.

```
function showStatusPlayer(){
  playerDOM.cards.innerHTML = game.player.cardsToString();
  playerDOM.score.innerHTML = game.player.score;
}

function showStatusDealer(){
  dealerDOM.cards.innerHTML = game.dealer.cardsToString();
  dealerDOM.score.innerHTML = game.dealer.score;
}

function showInitStatus(){
  playerDOM.cards.innerHTML = game.player.showFirstRound();
  playerDOM.score.innerHTML = game.player.score;
  dealerDOM.cards.innerHTML = game.dealer.showFirstRound();
  dealerDOM.score.innerHTML = "-";
}
```

4.5. Función *showWinners()*

Esta función comprueba quién ha ganado la partida y lo muestra en la página.

```
function showWinners(){
  showStatusPlayer();
  showStatusDealer();
  playerDOM.result.innerHTML = game.checkPlayerWinner()? "Ganador":"Perde
dor";
  dealerDOM.result.innerHTML = game.checkDealerWinner()? "Ganador":"Perde
dor";
}
```

5. Ejercicios

5.1. Contador de partidas

Mantén un contador de partidas jugadas por el jugador y muestra el resultado por pantalla.

5.2. Contador de partidas ganadas y perdidas

Mantén un contador para saber cuántas partidas ha ganado el jugador y cuantas ha perdido, muestra el resultado en pantalla. En caso de empate ningún contador debe incrementarse.

Obtener 21 en el turno inicial y el dealer no implica que el jugador obtiene 1.5 en su puntuación del contador.

5.3. Jugada del dealer

Actualmente es muy difícil ganar partidas al dealer. Para hacer más real vamos a cambiar la estrategia del dealer para que esté obligado a pedir carta cuando tiene una mano de 16 o de un valor inferior. Sin importar el valor de la mano del jugador, el dealer debe sacar automáticamente una carta tras otra hasta que llega a un total de 17 o se pase.

6. Conclusiones

Al transformar el código en objetos, hacemos que nuestro código sea mucho **más fácil de mantener**, pudiendo reutilizar parte de nuestro código en otros desarrollos ya que **no hay acoplamiento** entre los objetos que hemos definidos.

El diseño que hemos hecho sienta las bases de la estructura **MVC**, típica de una aplicación web. Tenemos los tres componentes principales:

- **Modelo**: que contiene la estructura de objetos sobre los que vamos a almacenar los datos de nuestra aplicación.
- **Vista**: con las funciones que generar el interfaz en la página HTML.
- **Controlador**: que implementan la lógica de negocio de nuestra aplicación. Da acceso al Modelo, recibiendo las peticiones de la Vista.