Exercises: Classes

Classes

1. Rectangle

Write a JS **class** for a rectangle object. It needs to have a **width** (Number), **height** (Number) and **color** (String) properties, which are set from the constructor and a **`calcArea()`** method, that calculates and **returns** the rectangle's area.

Input

The constructor function will receive valid parameters.

Output

The **`calcArea()`** method should **return** a number.

Submit the class definition as is, **without** wrapping it in any function.

Examples

Sample Input	Output

`let rect = new Rectangle(4, 5, 'red');`
`console.log(rect.width);`
`console.log(rect.height);`
`console.log(rect.color);`
`console.log(rect.calcArea());`
	
	`4`
	`5`
	`Red`
	`20`

2. Person

Write a JS **class** that represents a personal record. It has the following properties, all set from the constructor:

- **`firstName`**
- **`lastName`**
- **`age`**
- **`email`**

And a method **`toString()`**, which prints a summary of the information. See the example for formatting details.

Input

The constructor function will receive valid parameters.

Output

The **`toString()`**method should **return** a string.

Submit the class definition as is, **without** wrapping it in any function.

Examples

Sample Input
`let person = new Person('Maria', 'Petrova', 22, 'mp@yahoo.com');` `console.log(person.toString());`
Output
`Maria Petrova (age: 22, email: mp@yahoo.com)`

3. Get Persons

Write a JS function that returns an array of Person objects. Use the class from the previous task, create the following instances, and return them in an array:

First Name	Last Name	Age	Email
Maria	Petrova	22	mp@yahoo.com
SoftUni			
Stephan	Nikolov	25	
Peter	Kolev	24	ptr@gmail.com

For any empty cells, do not supply a parameter (call the constructor with less parameters).

Input / Output

There will be **no input**, the data is static and matches the table above. As **output**, **return an array** with Person **instances**.

Submit a function that returns the required output.

4. Circle

Write a JS **class** that represents a **Circle**. It has only one data property – it's **radius**, and it is set trough the **constructor**. The class needs to have **getter** and **setter** methods for its **diameter** – the setter needs to calculate the radius and change it and the getter needs to use the radius to calculate the diameter and return it.

The circle also has a getter **`area()`**, which calculates and **returns** its area.

Input

The constructor function and diameter setter will receive valid parameters.

Output

The **`diameter()`** and **`area()`** getters should **return** numbers.

Submit the class definition as is, **without** wrapping it in any function.

Examples

Sample Input	Output

`let c = new Circle(2);`
`console.log(`Radius: ${c.radius}`);`
`console.log(`Diameter: ${c.diameter}`);`
`console.log(`Area: ${c.area}`);`
`c.diameter = 1.6;`
`console.log(`Radius: ${c.radius}`);`
`console.log(`Diameter: ${c.diameter}`);`
`console.log(`Area: ${c.area}`);`
	
	`2`
	`4`
	`12.566370614359172`
	
	`0.8`
	`1.6`
	`2.0106192982974678`

5. Point Distance

Write a JS **class** that represents a **Point**. It has **x** and **y** coordinates as properties, that are set through the constructor, and a **static method** for finding the distance between two points, called **`distance()`**.

Input

The **`distance()`** method should receive two **Point** objects as parameters.

Output

The **`distance()`** method should **return** a number, the distance between the two point parameters.

Submit the class definition as is, **without** wrapping it in any function.

Examples

Sample Input	Output

`let p1 = new Point(5, 5);`
`let p2 = new Point(9, 8);`
`console.log(Point.distance(p1, p2));`
	
	`5`

6. Cards

You need to write an **IIFE** that results in an object containing two properties **`Card`** which is a class and **`Suits`** which is an object that will hold the possible suits for the cards.

The **`Suits`** object should have exactly these 4 properties:

- **SPADES**: ♠
- **HEARTS**: ♥
- **DIAMONDS**: ♦
- **CLUBS**: ♣

Where the key is **SPADES**, **HEARTS** e.t.c. and the value is the actual symbol ♠, ♥ and so on.

The **Card** class should allow for creating cards, each card has 2 properties **face** and **suit**. The **valid** faces are the following **`["2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"]`** any other are considered invalid.

The **Card** class should have **`setters`** and **`getters`** for the **face** and **suit** properties, when creating a card or setting a property validations should be performed, if an invalid face or a suit not in the **`Suits`** object is passed an **`Error`** should be **thrown**.

Code Template

You are required to write and submit an **IIFE** which results in an object containing the above-mentioned **`Card`** and **`Suits`** as properties. Here is an example template you can use:

cards.js

```
(function(){

    // TODO

    return {
        Suits:Suits,
        Card:Card
    }
}())
```

Screenshot

An example usage should look like this:

```
let result = (function(){...}());
let Card = result.Card;
let Suits = result.Suits;

let card = new Card("Q",Suits.CLUBS);
card.face = "A";
card.suit = Suits.DIAMONDS;
let card2 = new Card("1",Suits.DIAMONDS);   //Should throw Error
```

Unit testing on Classes

7. String Builder

You are given the following **JavaScript class**:

string-builder.js

```
class StringBuilder {
  constructor(string) {
    if (string !== undefined) {
```

```
      StringBuilder._vrfyParam(string);
      this._stringArray = Array.from(string);
    } else {
      this._stringArray = [];
    }
  }

  append(string) {
    StringBuilder._vrfyParam(string);
    for(let i = 0; i < string.length; i++) {
      this._stringArray.push(string[i]);
    }
  }

  prepend(string) {
    StringBuilder._vrfyParam(string);
    for(let i = string.length - 1; i >= 0; i--) {
      this._stringArray.unshift(string[i]);
    }
  }

  insertAt(string, startIndex) {
    StringBuilder._vrfyParam(string);
    this._stringArray.splice(startIndex, 0, ...string);
  }

  remove(startIndex, length) {
    this._stringArray.splice(startIndex, length);
  }

  static _vrfyParam(param) {
    if (typeof param !== 'string') throw new TypeError('Argument must be string');
  }

  toString() {
    return this._stringArray.join('');
  }
}
```

Functionality

The above code defines a **class** that holds **characters** (strings with length 1) in an array. An **instance** of the class should support the following operations:

- Can be **instantiated** with a passed in **string** argument or **without** anything
- Function **`append(string)`** – **converts** the passed in **string** argument to an **array** and adds it to the **end** of the storage
- Function **`prepend(string)`** – **converts** the passed in **string** argument to an **array** and adds it to the **beginning** of the storage
- Function **`insertAt(string, index)`** – **converts** the passed in **string** argument to an **array** and adds it at the **given** index (there is **no** need to check if the index is in range)

- Function **`remove(startIndex, length)`** – **removes** elements from the storage, starting at the given index (**inclusive**), **length** number of characters (there is **no** need to check if the index is in range)
- Function **`toString()`** – **returns** a string with **all** elements joined by an **empty** string
- All passed in **arguments** should be **strings.** If any of them are **not**, **throws** a type **error** with the following message: "**Argument must be a string**"

Examples

This is an example how this code is **intended to be used**:

Sample code usage

```
let str = new StringBuilder('hello');
str.append(', there');
str.prepend('User, ');
str.insertAt('woop',5 );
console.log(str.toString());
str.remove(6, 3);
console.log(str.toString());
```

Corresponding output

```
User,woop hello, there
User,w hello, there
```

Your Task

Using **Mocha** and **Chai** write **JS unit tests** to test the entire functionality of the **`StringBuilder`** class. Make sure it is **correctly defined as a class** and instances of it have all the required functionality. You may use the following code as a template:

```
describe("TODO …", function() {
    it("TODO …", function() {
        // TODO: …
    });
    // TODO: …
});
```

8. Payment Package

You are given the following **JavaScript class**:

PaymentPackage.js

```
class PaymentPackage {
  constructor(name, value) {
    this.name = name;
    this.value = value;
    this.VAT = 20;      // Default value
    this.active = true; // Default value
  }

  get name() {
    return this._name;
  }

  set name(newValue) {
```

```
    if (typeof newValue !== 'string') {
      throw new Error('Name must be a non-empty string');
    }
    if (newValue.length === 0) {
      throw new Error('Name must be a non-empty string');
    }
    this._name = newValue;
  }

  get value() {
    return this._value;
  }

  set value(newValue) {
    if (typeof newValue !== 'number') {
      throw new Error('Value must be a non-negative number');
    }
    if (newValue < 0) {
      throw new Error('Value must be a non-negative number');
    }
    this._value = newValue;
  }

  get VAT() {
    return this._VAT;
  }

  set VAT(newValue) {
    if (typeof newValue !== 'number') {
      throw new Error('VAT must be a non-negative number');
    }
    if (newValue < 0) {
      throw new Error('VAT must be a non-negative number');
    }
    this._VAT = newValue;
  }

  get active() {
    return this._active;
  }

  set active(newValue) {
    if (typeof newValue !== 'boolean') {
      throw new Error('Active status must be a boolean');
    }
    this._active = newValue;
  }

  toString() {
    const output = [
      `Package: ${this.name}` + (this.active === false ? ' (inactive)' : ''),
      `- Value (excl. VAT): ${this.value}`,
      `- Value (VAT ${this.VAT}%): ${this.value * (1 + this.VAT / 100)}`
```

```
    ];
    return output.join('\n');
  }
}
```

Functionality

The above code defines a **class** that contains information about a **payment package**. An **instance** of the class should support the following operations:

- Can be **instantiated** with two parameters – a string name and number value
- Accessor **`name`** – used to get and set the value of name
- Accessor **`value`** – used to get and set the value of value
- Accessor **`VAT`** – used to get and set the value of VAT
- Accessor **`active`** – used to get and set the value of active
- Function **`toString()`** – return a string, containing an overview of the instance; if the package is **not active**, append the label "**(inactive)**" to the printed **name**

When creating an instance, or changing any of the property values, the parameters are validated. They must follow these rules:

- **`name`** – non-empty string
- **`value`** – non-negative number
- **`VAT`** – non-negative number
- **`active`** – Boolean

If any of the requirements aren't met, the operation must throw an error.

Scroll down for examples and details about submitting to Judge.

Examples

This is an example how this code is **intended to be used**:

Sample code usage

```
// Should throw an error
try {
    const hrPack = new PaymentPackage('HR Services');
} catch(err) {
    console.log('Error: ' + err.message);
}
const packages = [
    new PaymentPackage('HR Services', 1500),
    new PaymentPackage('Consultation', 800),
    new PaymentPackage('Partnership Fee', 7000),
];
console.log(packages.join('\n'));

const wrongPack = new PaymentPackage('Transfer Fee', 100);
// Should throw an error
try {
    wrongPack.active = null;
} catch(err) {
    console.log('Error: ' + err.message);
}
```

Corresponding output

```
Error: Value must be a non-negative number
Package: HR Services
- Value (excl. VAT): 1500
- Value (VAT 20%): 1800
Package: Consultation
- Value (excl. VAT): 800
- Value (VAT 20%): 960
Package: Partnership Fee
- Value (excl. VAT): 7000
- Value (VAT 20%): 8400
Error: Active status must be a boolean
```

Your Task

Using **Mocha** and **Chai** write **JS unit tests** to test the entire functionality of the **PaymentPackage** class. Make sure instances of it have all the required functionality and validation. You may use the following code as a template:

```
describe("TODO …", function() {

    it("TODO …", function() {
        // TODO: …
    });

    // TODO: …
```

```
});
```