



**University of Maryland College Park**  
**Department of Computer Science**  
**CMSC335 Spring 2022**

**Exam #2 Key**

**FIRSTNAME, LASTNAME (PRINT IN UPPERCASE):**

**STUDENT ID (e.g., 123456789):**

**Grader Use Only**

Problem #1 (Miscellaneous)	40	
Problem #2 (Array Functions)	32	
Problem #3 (Custom Type Definition)	54	
Problem #4 (Class Declaration using "class")	54	
Problem #5 (Diagram)	20	
<b>Total</b>	200	

## Problem #1 (Miscellaneous)

1. (3 pts) Which of the following will **always** work when identifying an object as an array?

- (a.) alert
- (b.) instanceof
- (c.) Array.isArray()
- (d.) null

Answer: c.

2. (3 pts) Complete the implementation of the **getAmount** function, by assigning to the **answer** variable a **template literal** that has the sum of the parameters preceded by a dollar sign. For example, calling **document.writeln(getAmount(5, 7));** will return **\$12**. You cannot add any variables; just provide what goes inside of the template literal backticks.

```
function getAmount(a, b) {  
    let answer = ` `;  
    return answer;  
}
```

Answer: `\$\$\${a + b}`;

3. (4 pts) The **order** function has the following prototype: **function order(customerName, item, howMany, when)**  
Write a function call that will use the spread operator and the array **["milk", 4]** to initialize the **item** and **howMany** parameters. You can assume the **customerName** and **when** parameters are "Laura", and "Sunday", respectively.

Answer:

```
order("Laura", ...["milk", 4], "Sunday");
```

4. (5 pts) Using the **=>** operator, initialize the variable **sum** with a function that takes two parameters and returns the sum.

**let sum =**

Answer: (a, b) => a + b;

5. (6 pts) Write the **JSON** (not a JavaScript object) representation of an object that has the following properties:

- a. **name** property with a value of "Peter"
- b. **salary** property with a value of 45.60
- c. **owsCar** property with a value of false

Answer:

```
{"name": "Peter", "salary": 45.60, "owsCar": false}
```

6. (9 pts) Define a function called **compare** that will allow us to sort the following array in increasing order of **creditScore** value by using the **sort** method (creditScores.sort(compare)).

```
let creditScores = [  
    { name: "Kelly", creditScore: 700},  
    { name: "Alan", creditScore: 600},  
    { name: "Rose", creditScore: 800}  
];
```

Answer:

```
function compare(x, y) {  
    return x.creditScore - y.creditScore;  
}
```

7. (10 pts) Define an **Error** type called **InvalidPressure**. The following is an example of using your error type.

```
try {
  let value = Number(prompt("Enter positive (or 0) value"));
  if (value < 0) {
    throw new InvalidPressure("positive value expected");
  }
} catch (error) {
  alert(error.message);
}
```

Answer:

```
function InvalidPressure(message) {
  this.message = message;
}
InvalidPressure.prototype = new Error();
```

## **Problem #2 (Array Functions)**

A **cars** array keeps track of cars in a dealership. The following is an example of some entries the array could have:

```
const cars = [{make: "Toyota", cost: 400.00}, {make: "Ford", cost: 700.00}, {make: "Honda", cost: 500.00}, {make: "Honda", cost: 200.00}, {make: "Toyota", cost: 90.00}];
```

To answer the following questions, you may not use any **for/while/do-while** loops and only the following functions (otherwise, you will not get credit): **filter**, **forEach**, **some**, **find**, **reduce**, **join**, **findIndex**.

1. (6 pts) Complete the following statement, so each car's make is printed using **document.writeln**. Your code should work with different data (not just the entries shown above).

**NOTE: Using any function other than filter, forEach, some, find, reduce, join, findIndex (after the cars.) will make the score for the particular problem 0.**

**cars.**

Answer: **cars.forEach(i => document.writeln(i.make + "<br>"));**

2. (6 pts) Complete the following statement, so **lessThan300** is initialized with an array of cars having a cost of **less than** 300.00. Your code should work with different data (not just the entries shown above).

**const lessThan300 = cars.**

Answer: **cars.filter(aCar => aCar.cost < 300);**

3. (6 pts) Complete the following statement, so **hasAtLeastAFord** is initialized to true if there is a least one car with a "Ford" make and false otherwise.

**const hasAtLeastAFord = cars.**

Answer: **cars.some(aCar => aCar.make == "Ford");**

4. (14 pts) Complete the following statement, so **costSum** is initialized with the sum of the costs of cars that have a **make** corresponding to "Toyota". For example, for the above data, **costSum** will be initialized to 490. Your code should work with different data (not just the entries shown above).

**const costSum = cars.**

Answer: **cars.reduce((total, aCar) => (aCar.make == "Toyota") ? total + aCar.cost: total, 0);**

### Problem #3 (Custom Type Definition)

Write **JavaScript** that defines two classes (**Book** and **ElectronicBook**) using the "Default Pattern for Custom Type Definition" presented in class. **If you use E6 class definitions (similar to what you have in Java where we use class, extends), you will not receive any credit for this problem.**

#### 1. Book

- Define a **Book** custom type with two instance variables named **title** and **price** (they are not private).
- Define a constructor that has two parameters: **title** and **price**.
- Define a method named **setPrice** that will update the **price** instance variable if the parameter is a number; otherwise, the price will be set to 50.
- Define a method called **details** that returns a string with the **title** and **price** (see example below for format information).
- Your implementation must be efficient (i.e., do not create unnecessary objects).**

Answer:

```
function Book(title, price) {
    this.title = title;
    this.price = price;
}

Book.prototype = {
    constructor: Book,

    setPrice: function(price) {
        this.price = !isNaN(price) ? price : 50;
    },

    details: function() {
        return `Title: ${this.title}, Price: ${this.price}`;
    }
};
```

#### 2. ElectronicBook

- Define an **ElectronicBook** custom type that "extends" the **Book** custom type. The type has an instance variable named **bytes**; this instance variable is not private.
- Define a constructor that has **title**, **price**, and **bytes** as parameters. The constructor will initialize the corresponding instance variables.
- Define a method named **getBytes** that returns the bytes.
- Your implementation must be efficient (i.e., do not create unnecessary objects).**

**If you use E6 class definitions (similar to what you have in Java where we use class, extends), you will not receive any credit for this problem.**

Answer:

```
function ElectronicBook(title, price, bytes) {
    Book.call(this, title, price);
    this.bytes = bytes;
}

ElectronicBook.prototype = new Book();
ElectronicBook.prototype.constructor = ElectronicBook;
ElectronicBook.prototype.getBytes = function() {
    return this.bytes;
}
```

## Problem #4 (Class Declaration using "class")

Write **JavaScript** that defines two classes (**Computer** and **Laptop**) using E6 class definitions (using **class**, **extends**, **super** as in Java). **If you use the "Default Pattern for Custom Type Definition" presented in class, you will not get any credit.**

### 1. Computer

Define a **Computer** class with the specifications below. A computer is associated with a **make** and several **cpus**.

- A **private** static field named **totalComputers** initialized to 0.
- Two **private** instance variables named **make** and **cpus**. You must use the approach described in the lecture to make them private.
- Define a constructor that has two parameters: **make** and **cpus**. The constructor will initialize the corresponding instance variables and increase the **totalComputers** static variable.
- Define a **non-static** method called **info()** that prints (using `document.writeln`) the **make** and **cpus**. See the sample driver for format information.
- Define the equivalent of the `toString()` Java method. The method will return a string with the **make** and **cpus** values separated by a comma. The driver we provided has an example of using this method (look for `***string:`).
- Define static method called **getTotalComputers()** that returns the total number of **Computer** objects created.

Answer:

```
class Computer {
    static #totalComputers = 0; /* Static variable */

    #make; /* private */
    #cpus; /* private */

    constructor(make, cpus) {
        this.#make = make;
        this.#cpus = cpus;
        Computer.#totalComputers++;
    }

    info() {
        document.writeln("Make: " + this.#make);
        document.writeln(", Cpus: " + this.#cpus + "<br>");
    }

    [Symbol.toPrimitive]() {
        return this.#make + ", " + this.#cpus;
    }

    static getTotalComputers() { /* Static method */
        return Computer.#totalComputers;
    }
}
```

### 2. Laptop

The **Laptop** class extends the **Computer** class, and it is associated with a battery life. Define the **Laptop** class with the specifications below.

- A **private** instance variable named **batteryLife**. You must use the approach described in the lecture to make it private.
- Define a constructor with three parameters: **make**, **cpus**, and **batteryLife**. The constructor will call the base class constructor and initialize the **batteryLife** instance variable with the corresponding parameter.
- Define a **non-static** method called **info()** that calls the base class **info()** method and then prints the **batteryLife** value using `document.writeln`. See the sample driver for format information.

Answer:

```
class Laptop extends Computer {    // Using extends
    #batteryLife; /* private */

    constructor(make, cpus, batteryLife) {
        super(make, cpus);
        this.#batteryLife = batteryLife;
    }

    info() {
        super.info();
        document.writeln("Battery Life: " + this.#batteryLife + "<br>");
    }
}
```

### Problem #5 (Diagram)

The **Door** function is defined as follows:

```
function Door(location) {
    this.location = location;
}
```

Draw a diagram that illustrates the objects and the relationships among the objects present after the following two **Door** objects are created. Please make sure you label prototype objects as such (e.g., Door.prototype). In your diagram, we expect to see the **prototype** and **\_\_proto\_\_** properties (and the objects they refer to). Add the **location** property to the appropriate objects.

```
let dOne = new Door("Lobby");
let dTwo = new Door("FirstF");
```

Answer:

