# Exercises: Sample TypeScript Tasks

## Data Class

Write a JS class using TypeScript that holds data about an HTTP request. It has the following properties:

* method (String)
* uri (String)
* version (String)
* message (String)
* response (String)
* fulfilled (Boolean)

The first four properties (**method**, **uri**, **version**, **message**) are set trough the **constructor**, in the listed order. The **response** property is initialized to undefined and the **fulfilled** property is initially set to false..

### Examples

|  |  |
| --- | --- |
| Sample Input | Resulting object |
| let myData = new Request('GET', 'http://google.com', 'HTTP/1.1', '') | { method: 'GET',  uri: 'http://google.com',  version: 'HTTP/1.1',  message: '',  response: undefined,  fulfilled: false } |

### Hints

Using ES6 syntax, a class can be defined similar to a function, using the class keyword:



At this point, the class can already be **instantiated**, but it won’t hold anything useful, since it doesn’t have a constructor. A **constructor** is a function that initializes the object’s context and attaches values to it. It is defined with the keyword constructor inside the body of the class definition and it follows the syntax of regular JS functions – it can take arguments and execute logic. Any variables we want to be attached to the **instance** must be prefixed with the this identifier:



The description mentions some of the properties need to be set via the constructor – this means the constructor must receive them as parameters. We modify it to take four named parameters that we then assign to the local variables:



Note the input parameters have the same names as the instance variables – this isn’t necessary, but it’s easier to read. There will be no name collision, because the this identifier tells the interpreter to look for a variable in a different context, so this.method is not the same as method.

Our class is complete, now try to add all necessery TS logick

## Tickets

Write a JS program, using TS, that manages a database of tickets. A ticket has a **destination,** a **price** and a **status**. Your program will receive **two arguments** – the first is an **array of strings** for ticket descriptions and the second is a **string**, representing **sorting criteria**. The ticket descriptions have the following format:

<destinationName>|<price>|<status>

Store each ticket and at the end of execution **return** a sorted summary of all tickets, sorted by either **destination**, **price** or **status**, depending on the **second parameter** that your program received. Always sort in ascending order (default behavior for **alphabetical** sort). If two tickets compare the same, use order of appearance. See the examples for more information.

### Input

Your program will receive two parameters – an array of strings and a single string.

### Output

**Return** a **sorted array** of all the tickets that where registered.

### Examples

|  |  |
| --- | --- |
| Sample Input | Output Array |
| ['Philadelphia|94.20|available',  'New York City|95.99|available',  'New York City|95.99|sold',  'Boston|126.20|departed'],  'destination' | [ Ticket { destination: 'Boston',  price: 126.20,  status: 'departed' },  Ticket { destination: 'New York City',  price: 95.99,  status: 'available' },  Ticket { destination: 'New York City',  price: 95.99,  status: 'sold' },  Ticket { destination: 'Philadelphia',  price: 94.20,  status: 'available' } ] |
| ['Philadelphia|94.20|available',  'New York City|95.99|available',  'New York City|95.99|sold',  'Boston|126.20|departed'],  'status' | [ Ticket { destination: 'Philadelphia',  price: 94.20,  status: 'available' },  Ticket { destination: 'New York City',  price: 95.99,  status: 'available' },  Ticket { destination: 'Boston',  price: 126.20,  status: 'departed' },  Ticket { destination: 'New York City',  price: 95.99,  status: 'sold' } ] |

## People

Define several JS classes, that represent a company’s employee records. Every employee has a **name** and **age**, a **salary** and a list of **tasks**, while every position has specific properties not present in the others. Place all common functionality in a **parent** **abstract** class. Follow the diagram bellow:



Every position has different tasks. In addition to all common properties, the manager position has a **dividend** he can collect along with his salary.

All employees have a **work** function that when called cycles trough the list responsibilities for that position and prints the current one. When all tasks have been printed, the list starts over from the beginning. Employees can also **collect salary**, which outputs the amount, plus any **bonuses**.

Your program needs to expose a module, containing the three classes Junior, Senior and Manager. The properties name and age are set trough the constructor, while the salary and a manager’s dividend are initially set to zero and can be changed later. The list of tasks is filled by each position. The resulting objects also expose the functions work() and collectSalary(). When work() is called, one of the following lines is printed on the console, depending on the current task in the list:

{employee name} is working on a simple task.

{employee name} is working on a complicated task.

{employee name} is taking time off work.

{employee name} is supervising junior workers.

{employee name} scheduled a meeting.

{employee name} is preparing a quarterly report.

And when collectSalary() is called, print the following:

{employee name} received {salary + bonuses} this month.

### Input / Output

Any input will be passed as valid arguments, where applicable. Print any output that is required to the console as a string.

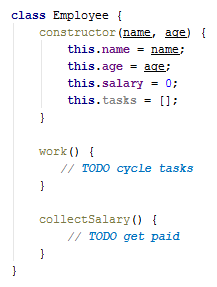
Submit your code as a revealing module, containing the three classes. Any definitions need to be named exactly as described above.

### Hints

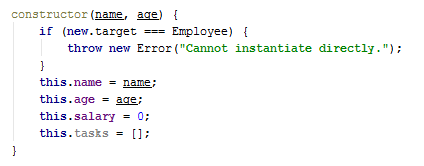
We should begin by creating a parent class, that will hold all properties, shared among the different positions. Looking at the problem description, we see the following structure for out parent object:

|  |
| --- |
| JavaScript |
| {  age: Number,  name: String,  salary: Number,  tasks: [],  work: Function,  collectSalary: Function  } |

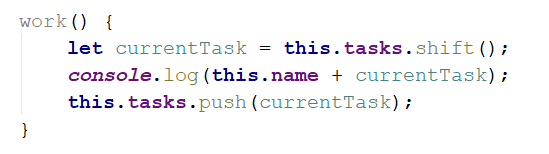
Data variables will be part of the object attached to its local context with this inside the **constructor**. Any properties that need to be initialized at instantiation time are defined as function parameters. Functions are defined inside the class body.



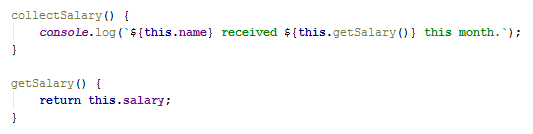
The problem description requires that the parent class is abstract. To achieve this, we have to add a condition in the constructor which prevents its direct instantiation. Using the new.target keyword we can check whether the object was created from the abstract constructor or through a child class.



The work() function has to cycle trough the list of tasks and print the current one. The easiest way to do this is to shift the first element from the array and push it at the end.



Printing the salary is pretty straightforward. However, since the manager has an additional bonus to his salary, it’s best to get the whole sum with an internal function, that the manager can **override**.

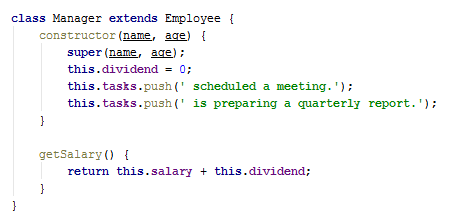


Now any objects that inherit from Employee will have all of its properties as well as anything new that’s defined in their declaration. To inherit (extend) a class, a new class is defined with the extends keyword after its name. They also have to call the parent constructor from their own constructor, so the prototype chain is established. For **Junior** and **Senior**, the only difference from the parent **Employee** is the elements inside the tasks array, since they can use the functions directly from the base class. Child classes will call the parent with any parameters that are needed and push their tasks directly to the array.

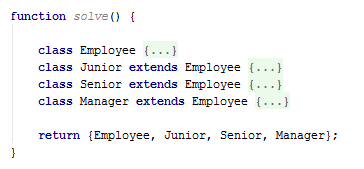




The **Manager** is not much different, with the exception that his constructor has to attach a dividend property that is initially set to zero. His definition also needs to override the getSalary() function we added to the base class earlier, so it includes the bonus.



After we’re done with the definitions of all object constructors, we need to wrap them in a revealing module for use by other parts of our program without polluting the global namespace.



## The Elemelons

If Watermelons exist, Firemelons, Earthmelons and Airmelons should also exist. Create **classes** for **The** **4 Elemelons**.

Create an **abstract class** for the Elemelons. Name it Melon.  
The Melon class should be initialized with weight (Number), and melonSort (String). The 2 arguments should be **public members**.

Create classes Watermelon, Firemelon, Earthmelon, Airmelon. Each of them should **inherit** the **abstract class** Melon and its functionality. Aside from the abstract functionality, **each** of the **Elemelons** should have property elementIndex (Number), which is **equal** to its weight \* the **string length** of its melonSort. The property should have only a **getter**.

All of the classes should hold a toString() function, which returns the following result for them:

“Element: {Water/Fire/Earth/Air}”  
“Sort: {elemelonSort}”  
“Element Index: {elemelonElementIndex}”

Create one more class which is called Melolemonmelon, which inherits **one** of the **4 elemelons**, **regardless of which**.   
The Melolemonmelon **has no element**, but it can **morph** into any of the others. Implement a function **morph()**, which **changes the current element** of the Melolemonmelon, **each time** it is called.   
Upon initialization, the **initial element** is **Water**. From then it should go in the following order: **Fire, Earth, Air, Water, Fire…** and so on.  
The toString() function should remain the same as its parent class.

### Example

|  |
| --- |
| scripts.js |
| **let** test = **new** Melon(100, **"Test"**); *//Throws error* **let** watermelon = **new** Watermelon(12.5, **"Kingsize"**); ***console***.log(watermelon.toString());  *// Element: Water // Sort: Kingsize // Element Index: 100* |