# Exercises: Advanced Data Types

## Optional Multiplier

Write a function that receives **3 optional parameters** of type **string | number**, the function **should multiply the 3** **parameters** and **return the result**, using the following logic:

* If a **string is passed** as a parameter, **parse it** to a number and **use it in the multiplication**.
  + Any passed string will always **represent a valid integer** number.
* If one or more parameters are passed, **return** **their multiplication**
* If no parameters are passed **return 1**

The **output** should be a single number.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| '3', 5, '10' | 150 |
| '2','2' | 4 |
| undefined, 2, 3 | 6 |
| 7, undefined, '2' | 14 |
|  | 1 |

## Operator

You need to write a function that can perform one of 3 operations

Write a function that receives **3 parameters**:

* param: **string | number | string[]**
* operation: **'Index' | 'Length' | 'Add'**
* operand: **number**

If the operation is:

* **Index** and the param is string or string[] - return the **value** at **index equal to** **operand** of the string or string[]
* **Length** and the param is string or string[] – return the **length** of the string or string[] **mod the operand**
* **Add** and the param is string or number – return the **param parsed to Number** and **added** **to the** **operand**
* **Note:** Try to solve the problem with minimal type narrowing.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| ['First', 'Second', 'Third'], 'Index', 1 | Second |
| 'string', 'Index', 1 | t |
| ['Just', 'Two'], 'Length', 5 | 2 |
| 'short string1', 'Length', 5 | 3 |
| '7', 'Add', 3 | 10 |
| 11, 'Add', 3 | 14 |

## Car Diagnostics

You were tasked to modify a function that takes in car body parts to monitor their status, the function currently takes in **3 parameters**:

* **carBody: { material: string, state: string }**
* **tires: { airPressure: number, condition: string }**
* **engine: { horsepower: number, oilDensity: number }**

You need to add 2 more properties to each of the components:

* **partName: string**
* **runDiagnostics(): string -** returns the **partName**
* **Note:** Instead of modifying each object type, try to simplify changing the function signature using advanced types

### Examples

|  |
| --- |
| **Input** |
| { material: 'aluminum', state: 'scratched', partName: 'Car Body', runDiagnostics },  { airPressure: 30, condition: 'needs change', partName: 'Tires', runDiagnostics },  { horsepower: 300, oilDensity: 780, partName: 'Engine', runDiagnostics } |

## Http Codes

Write a function that takes in objects representing HTTP Responses and prints their text information. Here are all the **HttpResponse** objects your function needs to be able to process:

|  |
| --- |
| **HTTP Response objects** |
| **{ code: 200, text: 'OK'}**  **{ code: 201, text: 'Created'}**  **{ code: 301, text: 'Moved Permanently'}**  **{ code: 400, text: 'Bad Request', printChars?: number}**  **{ code: 404, text: 'Not Found', printChars?: number}**  **{ code: 500, text: 'Internal Server Error', printChars?: number}** |

Your function **should take in** an **HttpResponse** object and **console.log** their text value, if the **HttpResponse** object has a **printChars** value, instead print only the first **printChars** number of characters.

**Try to simplify the function signature** to improve type inference, by extracting common values **using advanced types**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| { code: 200, text: 'OK'} | OK |
| { code: 201, text: 'Created'} | Created |
| { code: 400, text: 'Bad Request', printChars: 4} | Bad |
| { code: 404, text: 'Not Found'} | Not Found |
| { code: 404, text: 'Not Found', printChars: 3} | Not |
| { code: 500, text: 'Internal Server Error', printChars: 1} | I |

## Hotel Visit

You are given the following function:

|  |
| --- |
| **Function** |
| **function visitFloor(floor:**  **{ number: 1, hallway: 'A', train:() => void, pass: 'Guest' } |**  **{ number: 2, hallway: 'A', dine: () => void, } |**  **{ number: 3, hallway: 'C', sleep:() => void} |**  **{ number: 1, hallway: 'C', train:() => void } |**  **{ number: 1, hallway: 'A', train:() => void } |**  **{ number: 2, hallway: 'A', dine: () => void, pass: 'Guest' } |**  **{ number: 3, hallway: 'A', sleep:() => void } |**  **{ number: 2, hallway: 'C', dine: () => void, }**  **) {**  **switch (floor.number) {**  **case 1: floor.train(); return;**  **case 2: floor.dine(); return;**  **case 3: floor.sleep(); return;**  **}**  **}** |

Clearly it could use some simplification, your task is to **use advanced types** and **simplify the parameter (but keeping the same signature compatibility)**, then **extract a new type alias for the simplified version**.

|  |
| --- |
| **Expected function** |
| **function visitFloor(floor: simplified) {**  **switch (floor.number) {**  **case 1: floor.train(); return;**  **case 2: floor.dine(); return;**  **case 3: floor.sleep(); return;**  **}**  **}** |

### Examples

You can check if your simplified type is compatible with the original signature by using the following code, all of **these calls should work:**

|  |
| --- |
| **Working calls** |
| **visitFloor({ train() { }, number: 1, hallway: 'A', pass: 'Guest' });**  **visitFloor({ dine() { }, number: 2, hallway: 'A' });**  **visitFloor({ sleep() { }, number: 3, hallway: 'C' });**  **visitFloor({ train() { }, number: 1, hallway: 'C' });**  **visitFloor({ train() { }, number: 1, hallway: 'A' });**  **visitFloor({ dine() { }, number: 2, hallway: 'A', pass: 'Guest' });**  **visitFloor({ sleep() { }, number: 3, hallway: 'A' });**  **visitFloor({ dine() { }, number: 2, hallway: 'C' });** |

And all of these calls should be **marked as TypeScript errors**:

|  |
| --- |
| **TS Errors** |
| **visitFloor({ train() { }, number: 4, hallway: 'A' });**  **visitFloor({ train() { }, number: 1, hallway: 'C', pass: 'Guest' });**  **visitFloor({ train() { }, number: 2, hallway: 'A' });**  **visitFloor({ train() { }, number: 3, hallway: 'C' });**  **visitFloor({ train() { }, number: 3, hallway: 'C', pass: 'Guest' });**  **visitFloor({ dine() { }, number: 1, hallway: 'A' });**  **visitFloor({ dine() { }, number: 1, hallway: 'B' });**  **visitFloor({ dine() { }, number: 1, hallway: 'C' });**  **visitFloor({ dine() { }, number: 3, hallway: 'C' });**  **visitFloor({ dine() { }, number: 2, hallway: 'C', pass: 'Guest' });**  **visitFloor({ dine() { }, number: 1, hallway: 'A', pass: 'Guest' });**  **visitFloor({ sleep() { }, number: 3, hallway: 'D' });**  **visitFloor({ sleep() { }, number: 4, hallway: 'C' });**  **visitFloor({ sleep() { }, number: 1, hallway: 'C' });**  **visitFloor({ sleep() { }, number: 1, hallway: 'A' });**  **visitFloor({ sleep() { }, number: 2, hallway: 'A' });**  **visitFloor({ sleep() { }, number: 2, hallway: 'C' });** |

## Type Extraction

Your task is to write a function that accepts 2 parameters and returns a function that accepts a combined version of the objects and then prints their combined information. Your function should **accept 2 parameters**:

* **names** object – contains name information and a function **getPersonInfo** to print the names information
* **location** object – contains address information and a function **getAddressInfo** to print the location information

After returning the result function, call it with the combined object to print its info to the console in the format '**Hello, <person.getPersonInfo()> from <person.getAddressInfo()>**'

**Note:** The main purpose of the task is to leverage type extraction and advanced types to save you a lot of writing, extract and use type aliases as much as possible, if you use them properly, you’ll only need to work with simple short type aliases, instead of the expansive object type declarations.

### Examples

|  |
| --- |
| **Input code** |
| **let names = { fName: 'John', lName: 'Doe', age: 22, getPersonInfo() { return `${this.fName} ${this.lName}, age ${this.age}` } };**  **let location = { city:'Boston', street: 'Nowhere street', number: 13, postalCode: 51225, getAddressInfo() { return `${this.street} ${this.number}, ${this.city} ${this.postalCode}`} };**  **let combinedFunction = createCombinedFunction(names, location);**  **let combinedPerson = Object.assign({}, names, location);**  **combinedFunction(combinedPerson);** |
| **Output** |
| **Hello, John Doe, age 22 from Nowhere street 13, Boston 51225** |

## Validate User

Your task is to create **a type predicate function** that **receives a single object parameter** and **validates** whether it is a valid **User**.

You are provided with the **User** type:

|  |
| --- |
| **User** |
| **type User = {**  **id: number | string,**  **username: string,**  **passwordHash: string | string[],**  **status: 'Locked' | 'Unlocked' | 'Deleted',**  **email?: string**  **}** |

An object is considered a **valid user** if it:

* Has an **id** of either:
  + type **number**, that has a **value over 100**
  + type **string** and **length exactly 14 characters**
* Has a **username** of type **string,** that has a **length between [5…10] characters**
* Has a **passwordHash** of either:
  + type **string** and has a length of **exactly 20 characters**
  + type **string[]** with **exactly 4 elements** each of which has **length of exactly 8 characters**
* Has a status of either:
  + **'Locked'**
  + **'Unlocked'**
* May optionally have an **email** **(not required)**

**Hint**: you can use **in** keyword as both a type guard and to check if a property exists on an object

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| { id: 120, username: 'testing', passwordHash: '123456-123456-123456', status: 'Deleted', email: 'something' } | false |
| { id: '1234-abcd-5678', username: 'testing', passwordHash: '123456-123456-123456', status: 'Unlocked' } | true |
| { id: '20', username: 'testing', passwordHash: '123456-123456-123456', status:'Deleted', email: 'something' } | false |
| { id: 255, username: 'Pesho', passwordHash: ['asdf1245', 'qrqweggw', '123-4567','98765432'], status: 'Locked', email: 'something' } | true |
| { id: 'qwwe-azfg-ey38', username: 'Someone', passwordHash: ['qwezz8jg', 'asdg-444','12-34-56'], status: 'Unlocked' } | false |
| { id: 1344, username: 'wow123', passwordHash: '123456-123456-1234567', status: 'Locked', email: 'something@abv.bg' } | false |

## Extract Interfaces

You are provided with the following function:

|  |
| --- |
| **Function** |
| **function assignTask(**  **user: { username: string, signupDate: Date },**  **task: {**  **status: 'Logged' | 'Started' | 'InProgress' | 'Done',**  **title: string,**  **daysRequired: number,**  **assignedTo: {**  **username: string,**  **signupDate: Date**  **} | undefined,**  **changeStatus(newStatus: 'Logged' | 'Started' | 'InProgress' | 'Done'): void**  **}**  **) {**  **if (task.assignedTo == undefined) {**  **task.assignedTo = user;**  **console.log(`User ${user.username} assigned to task '${task.title}'`);**  **}**  **}** |

Your task is to **extract types and interfaces** **for the function parameters**, try to reuse interfaces and types for type definitions that appear in multiple places.

Here is some sample code to test your changes, you can simplify these task definitions as well, after you extract common types and interfaces :

|  |
| --- |
| **Sample code** |
| **let user = {**  **username: 'Margaret',**  **signupDate: new Date(2022, 1, 13),**  **passwordHash: 'random'**  **}**  **let task1 = {**  **status: <'Logged' | 'Started' | 'InProgress' | 'Done'>'Logged',**  **title: 'Need assistance',**  **daysRequired: 1,**  **assignedTo: undefined,**  **changeStatus(newStatus: 'Logged' | 'Started' | 'InProgress' | 'Done') { this.status = newStatus; }**  **}**  **let task2 = {**  **status: <'Logged' | 'Started' | 'InProgress' | 'Done'>'Done',**  **title: 'Test',**  **daysRequired: 12,**  **assignedTo: undefined,**  **changeStatus(newStatus: 'Logged' | 'Started' | 'InProgress' | 'Done') { this.status = newStatus; },**  **moreProps: 300,**  **evenMore: 'wow'**  **}**  **assignTask(user, task1);**  **assignTask(user, task2);** |

## Restaurant

You are provided with the fully implemented class **CustomerOrder**:

|  |
| --- |
| **Class** |
| **class CustomerOrder implements Dish, OrderDetails {**  **dishName: string;**  **price: number;**  **isVegan: boolean;**  **orderId: number;**  **quantity: number;**  **orderStatus = "Pending";**  **constructor(dishName: string, price: number, isVegan: boolean, orderId: number, quantity: number) {**  **this.dishName = dishName;**  **this.price = price;**  **this.isVegan = isVegan;**  **this.orderId = orderId;**  **this.quantity = quantity;**  **}**  **getDishInfo() {**  **return `${this.dishName} - Price: $${this.price}, Vegan: ${this.isVegan ? 'Yes' : 'No'}`;**  **}**  **getOrderSummary() {**  **return `Order ID: ${this.orderId} - Dish: ${this.dishName}, Quantity: ${this.quantity}, Total Price: $${this.price \* this.quantity}, Status: ${this.orderStatus}`;**  **}**  **updateOrderStatus() {**  **if (this.orderStatus === "Pending") {**  **this.orderStatus = "Shipped";**  **} else if (this.orderStatus === "Shipped") {**  **this.orderStatus = "Delivered";**  **}**  **}**  **getOrderStatus() {**  **return `Order Status: ${this.orderStatus}`;**  **}**  **}** |

And the following object:

|  |
| --- |
| **Sample Code** |
| **let order: FullOrder = {**  **dishName: 'Cheese Burger',**  **price: 12,**  **isVegan: false,**  **orderId: 101,**  **quantity: 2,**  **discount: 10,**  **deliveryAddress: '456 Burger Lane, Food City',**  **orderStatus: 'Pending',**  **getDishInfo(): string {**  **return `${this.dishName} - Price: $${this.price}, Vegan: ${this.isVegan ? 'Yes' : 'No'}`;**  **},**  **getOrderSummary(): string {**  **return `Order ID: ${this.orderId} - Dish: ${this.dishName}, Quantity: ${this.quantity}, Total Price: $${this.price \* this.quantity}`;**  **},**  **updateOrderStatus(): void {**  **if (this.orderStatus === "Pending") {**  **this.orderStatus = "Shipped";**  **} else if (this.orderStatus === "Shipped") {**  **this.orderStatus = "Delivered";**  **}**  **},**  **getOrderStatus(): string {**  **return `Order Status: ${this.orderStatus}`;**  **},**  **getFinalPrice(): string {**  **const totalPrice = this.price \* this.quantity;**  **const finalPrice = totalPrice - (totalPrice \* (this.discount / 100));**  **return `Final Price after ${this.discount}% discount: $${finalPrice}`;**  **}**  **}**  **console.log(order2.getDishInfo());**  **console.log(order2.getOrderSummary());**  **console.log(order2.getFinalPrice());**  **console.log(order2.getOrderStatus());**  **order2.updateOrderStatus();**  **console.log(order2.getOrderStatus());** |
| **Sample Output** |
| **Cheese Burger - Price: $12, Vegan: No**  **Order ID: 101 - Dish: Cheese Burger, Quantity: 2, Total Price: $24**  **Final Price after 10% discount: $21.6**  **Order Status: Pending**  **Order Status: Shipped** |

Your task is to implement the **Dish**, **OrderDetails** and **FullOrder** interfaces:

|  |
| --- |
| **Interfaces** |
| **type OrderStatus = "Pending" | "Shipped" | "Delivered";**  **interface Dish {}**  **interface OrderDetails {}**  **interface FullOrder extends CustomerOrder {}** |

Using the following conditions:

* **All** the properties and functionality in **CustomerOrder** should be described in the **Dish** and **OrderDetails** interfaces
  + You can use the method implementations and sample output to determine how to distribute the properties and functionality between the 2 interfaces
* The **order** object from the sample has to be compatible with the **FullOrder** interface