# Exercises: Generics

Problems for exercises and homework for the ["TypeScript Advanced" course @ Software University](https://softuni.bg/trainings/2696/typescript-advanced-december-2019).

## Generic Box of String

Create a generic **class** **Box** that takes one property from the constructor. This property must be initialized with **generic** **type**. Create a **toString()** method that returns a message in the following format: **“{data} is of type {type}”**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| let box1 = new Box(['test']);  let box2 = new Box(20);  let box3 = new Box('Hello');  console.log(box1.toString());  console.log(box2.toString());  console.log(box3.toString()); | test is of type object  20 is of type number  Hello is of type string |

## Generic Compare Elements

Create a generic class that has as a property an array of **generic type.** Create a **compare(comparator)** method that takes an argument of any type. Iterate through the array and compare each element of the array to the comparator and return how many times the comparator is matched.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| let c = new CompareElements(['a', 'b', 'ab', 'abc', 'cba', 'b']);  console.log(c.compare('b')); | 2 |
| let c = new CompareElements([1, 2, 3, 4, 5, 1, 1]);  console.log(c.compare(1)); | 3 |

## Tuple

Create a generic class that receives **two** **generic type parameters** and make a **toString()** that returns a message in the following format: **“{parameterOne} -> {parameterTwo}”**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| let n1 = new Tuple('Svetoslav', 'Sofia');  console.log(n1.toString()); | Svetoslav -> Sofia |
| let n2 = new Tuple("Apples", 10);  console.log(n2.toString()); | Apples -> 10 |
| let n3 = new Tuple(10, 100091)  console.log(n3.toString()); | 10 -> 100091 |

## Threeuple

Create a Class **Threeuple**. You may extend the previous class so that it take three generic types parameters and return a message in the following format: **“{parameterOne} -> {parameterTwo} -> {parameterThree}”**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| let n = new Threeuple ('Hello World', [1], 312);  console.log(n.toString()); | Hello World -> 1 -> 312 |

## Car Dealership

Create two **generic interfaces:**

* **Dealership** that takes **one generic type parameter**: **dealershipName** and another property - **soldCars (number)**
* **Actions** that takes **one generic type parameter** and **defines** a **sellCar(dealerID , model)** method, where both parameters are of the generic type passed to the interface

Create a class **CarDealership** that implements both **Dealership** and **Actions** and has one property of its own:

* **Public** property **modelsSold** which is initially set to empty object

Note: The **dealershipName** is taken through the constructor and the **soldCars** is initially set to 0.

Make the following two methods:

* **sellCar(dealerID , model)** – add the dealerID as a key and the model as value in the modelsSold object and increase the soldCars by 1;
* **showDetails()** – return a message in the following format:

**“{Dealership}:**

**{dealerID} sold {model}**

**...”**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| let dealership = new CarDealer('SilverStar');  dealership.sellCar('BG01', 'C Class');  dealership.sellCar('BG02', 'S Class');  dealership.sellCar('BG03', 'ML Class');  dealership.sellCar('BG04', 'CLK Class');  console.log(dealership.showDetails()); | SilverStar:  BG01 sold C Class  BG02 sold S Class  BG03 sold ML Class  BG04 sold CLK Class |

## Bank Transactions

Create an **abstract generic class CreateAccount** that takes **two generic type parameters**: **bankName** and **bankID**.

Create a class **PersonalAccount** that extends **CreateAccount** and has three properties of its own:

* **Readonly** property **ownerName** (string)
* **Public** property **money** (number) which is initially set to 0
* **Public** property **recentTransactions** which is initially set to empty object

Only the **ownerName** is taken through the constructor.

Make the following three methods:

* **deposit(amount)** – simply add the **amount** to the **money** you currently have
* **expense(amount, expenseType)** – check if you have enough **money** (>= 0) to make the expense. If so put the **expenseType** as key and the amount as a value to the **recentTransactions** object. Furthermore, if have made the same expense more than once simply add the new amount to the existing one. However, if you don’t have enough money for the expense throw a new error: **“You cant make {expenseType} transaction”**
* **showDetails()** – Calculate the total amount of money spent on expenses and return a message in the following format:

**“Bank name: {bankName}**

**Bank ID: {bankID}**

**Owner name: {ownerName}**

**Money: {money}**

**Money spent: {totalMoneySpentOnExpenses}”**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| let account = new PersonalAccount('DSK', 101240, 'Ivan Ivanov');  account.deposit(1000);  account.deposit(1000);  account.expense(700, 'Buy new phone');  account.expense(400, 'Book a vacation');  account.expense(400, 'Book a vacation');  account.expense(100, 'Buy food');  console.log(account.showDetails()); | Bank name: DSK  Bank ID: 101240  Owner name: Ivan Ivanov  Money: 400  Money spent: 1600 |
| let account2 = new PersonalAccount('Fibank', 100000, 'Petar Petrol');  account2.deposit(10000);  account2.deposit(7000);  account2.expense(12000, 'Buy a new car');  account2.expense(200, 'Go to a fancy restaurant');  account2.expense(100, 'Go to a bar');  account2.expense(30, 'Go to the movies');  console.log(account2.showDetails()); | Bank name: Fibank  Bank ID: 100000  Owner name: Petar Petrol  Money: 15470  Money spent: 1530 |

### Note

You may try to make the problems from the previous exercise generic too.