Lab 1 - debug the casino-app

Prerequisites: latest version of vscode installed

1. Open vscode in an empty directory
2. Copy paste the "Casino app code" to a new file named "casino-app.js".
3. There might be some bugs in the code try to find them all WITHOUT running the code.
4. Now add this as a first line: `// @ts-check`
5. Can you now find more bugs?

Casino app code:

```
class Person {
  /**
   * @param {string} name
   */
  constructor(name) {
    this._name = name;
  }
}

class Player extends Person {
  /**
   *
   * @param {string} name
   * @param {number} chips
   */
  constructor(name, chips) {
    this.chips = chips;
  }

  toString() {
    return `${this.name} has ${chips} number of chips left`;
  }
}

var playerOne = new Player('Han', 46);
var playerTwo = new Player('Leia', 68);

var highestNumberOfChips = Math.max([playerOne.chips, playerTwo.chips]);
console.log(highestNumberOfChips + ' is the highest number of chips');

class RouletteBoard {
  constructor() {
    this.betRecords = [];
  }

  /**
   *
   * @param {Player} player
   * @param {number} bet
   */
  placeBet(player, bet) {
    var record = this.records.find((r) => r.player === player && r.bet === bet);
    if (!record) {
      record = { player: player, bet: bet, numberOfChips: 0 };
      this.betRecords.add(record);
    }
    record.numberOfChips++;
  }

  play() {
    var winner = Math.floor(Math.random() * 36);
    console.log('winning number: ' + winner);
    for (var record in this.betRecords) {
      if (this.betRecords[record].bet === winner) {
        var loot = this.betRecords[record].numberOfChips * 10;
        this.betRecords[record].player.chips += loot;
```

```
        console.log(
          this.betRecords[record].player.toString() + ' wins ' + loot
        );
      }
    }
    this.betRecords = [];
  }
}

var roulette = new RouletteBoard();
roulette.placeBet(playerOne, 20);
roulette.placeBet(playerOne, 20);
roulette.placeBet(playerTwo, 1);
roulette.placeBet(playerTwo, 2);
roulette.placeBet(playerTwo, 6);
roulette.placeBet(playerTwo, 31);
roulette.placeBet(playerTwo, 5);
roulette.placeBet(playerTwo, 4);

roulette.play();
```

Lab 2 - Getting started

Prerequisites: - NodeJS installed - NPM installed

1. Create an empty working directory. Open that directory in the command line of your choice.
2. Initialize a new TypeScript project.
   Create a `package.json`.
   ```
   npm init --yes
   ```
   Install TypeScript as a local dev dependency
   ```
   npm install --save-dev typescript
   ```
   Create a `tsconfig.json`
   ```
   npx tsc --init
   ```
   Open the `tsconfig.json` file and change it
   Compile your code to `es2022`
   Set `sourceMap` to `true`
   Set `outDir` to `dist`
3. Setup your favorite IDE.
   If using VSCode look in the slides on how to do this.
4. Create `src` directory, with a `main.ts`. This will be the home of our TypeScript application for now.
5. Make sure you are able to compile your code and debug the JavaScript output. Follow the instructions on the slides.
6. Compile the file in `--watch` mode.
7. Play around with a hello world type application. Make sure errors are displayed in your IDE.
8. Try out some constructs you know from JavaScript, like `functions`, `if-else`, `for`, `while`

Lab 3 - TypeQuiz

_Either do this lab in vscode, or use the kahoot quiz. For kahoot, go to https://create.kahoot.it/details/typescript-quiz-what-is-the-result/b37b6dc8-01e9-4f84-ae1d-16ec9d3c9a17 _

Preparations

Use the setup you made last lab.

Typequiz

For each of these expressions: try to guess what the outcome will be: `true`, `false`, compile error or something else. Next, write the expression in `main.ts` using a `console.log`. Run the file to see if you were correct. Keep your score so we can compare later :).

```
1. null === null;
2. true || false;
3. 2 === "2";
4. false === true;
5. null === undefined;
6. 2 + "2";
7. 2 * "2";
8. var a: string; console.log(typeof a);
9. var b: never; console.log(typeof b);
10. var c: any = 'test'; console.log(typeof c);
11. var d = true; console.log(d.charAt(1));
12. var e: any = true; console.log(e.charAt(1));
```

Lab 4 - Declare variables

Preparations

Create the setup you made the previous lab.

Case Explanation

Welcome to the greatest bank in the world. The first bank that decided to go all in! The future is bright indeed.

Our mission? TYPE SAFETY. Yes. All the way. We will program our entire code base 100% type safe. No customer will lose it's money because of a type error!

Our brilliant bank will be TypedBank™.

The success of this company depends solely on you. Good luck!

Exercise 1 - Declare constants

Create/empty the file `main.ts`.

Declare read-only variables:

1. A `string` called `DEFAULT_COUNTRY_CODE` with value `'NL'`
2. A `string` called `DEFAULT_BANK_CODE` with value `'TYPE'`
3. Try predict what the types of variables are. Are you correct? You can see the type by hovering over the variable name in your code editor.

»

Exercise 2 - Playing with variables - If time permits

1. Copy + paste the following boilerplate:

```
for (var i = 0; i < 2; i++) {
  let j = i;
  console.log(i, j);
}
```

2. Try to guess what the result is.
3. Execute the code and verify the result.
4. Now replace the `console.log(i, j)` statement with `setTimeout(function() { console.log(i, j); }, 0)`.
   Note: `setTimeout` will queue work to be done at a later time. In this case, the `console.log` will be executed *after* the for loop is executed.
5. Try to guess what the result might be.
6. Execute the code. Is it what you expected? Can you explain why?
7. Remove the for statement (keep only the result of exercise 1)

Lab 5 - Create bank helper functions

Preparations

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* folder.

Work in the `main.ts` file.

Exercise 1 - Format customer names

In our bank we will need to format customer names.

1. Create that function. The shape looks like this:

```
function formatName(
  firstName: string,
  lastName: string,
  insertion: string
): string {
  // TODO: Implement
}
```

2. Test that it works: `formatName('Pascalle', 'Vries', 'de')` should result in `'Pascalle de Vries'`.
3. Change `insertion` to make sure it is optional. `formatName('Foo', 'Bar')` should result in `'Foo Bar'`.

Exercise 2 - Generate IBAN account numbers

In the main.ts file, create a `generateIban` function. It should generate a new random IBAN account number. It does *not* have to be a valid IBAN, just one that looks like it.

1. Implement the following function:

```
function generateIban(bankCode: string, countryCode: string) {
    // TODO: implement
    return {
       countryCode: /* TODO */,
       bankCode: /* TODO */,
       accountNumber: /* TODO */,
       controlNumber: 0
    };
}
```

As the control number, you can use `'00'` for now.
Note: you can use `Math.floor(Math.random() * 10000000000).toString()` to generate random account number

2. Make the `bankCode` and `countryCode` optional, the default values are `DEFAULT_BANK_CODE` and `DEFAULT_COUNTRY_CODE` respectively.
3. Can you guess what the return type of this function is? Verify this by hovering over the function name in your code editor.

»

Exercise 3 - Format IBAN account numbers

1. Create and implement the `formatIban(iban: any)` function. It should format the IBAN in groups of 4 characters:

```
NL50 TYPE 3532 0409 92
NL97 INGB 9589 7465 22
DE09 DEUT 6102 3797 98
```

Verify that your method works using these examples:

```
const ibanTypedBank = generateIban();
const ibanIng = generateIban('INGB', 'NL');
const ibanDeutscheBank = generateIban('DEUT', 'DE');
console.log(formatIban(ibanTypedBank));
console.log(formatIban(ibanIng));
console.log(formatIban(ibanDeutscheBank));
```

2. Improve the function by using *Object matching* in the parameter names

»

Exercise 4 - Valid IBAN accounts - If time permits

Make sure that the `generateIban` function only generates *valid* IBAN account numbers by calculating the control number (https://nl.wikipedia.org/wiki/International_Bank_Account_Number).

Hint: you might need to use `BigInt` for this.

Lab 6 - Interfaces

Preparations

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* folder.

In this Lab you are going to create interfaces and improve the code base to make it more type safe.

Exercise 1: Create an `Iban` interface

1. Create an `Iban` interface. The interface should describe the shape returned by the `generateIban` function.
2. Use the interface as the return type of `generateIban`
3. Make sure that there are no compile errors.

Play around with the `Iban` interface. What happens if you add a property to it? What happens when you remove the explicit return type from the `Iban` function? Can you explain what's happening here?

Exercise 2: Create a `Customer` interface

1. Create an interface for a `Customer`. It should have a `firstName` (string), `lastName` (string) and an `insertion` (string). The `insertion` should be optional.
2. Make sure the `formatName` method accepts this new interface.
3. Test the code to see if it formats a name correctly.

Exercise 3: Create a `BankAccount` interface

1. Create an interface for a `BankAccount`. It should have a `customer` field (type `Customer`) and a `iban` field (type `Iban`)
2. Add a function `createBankAccount` with the following shape:

```
function createBankAccount(customer: Customer): BankAccount {
  // TODO
}
```

It should create a new `Iban` using `generateIban` and return a new `BankAccount`.

3. Create some bank accounts in an array:

```
const bankAccounts = [
  createBankAccount({
    firstName: 'Alfred',
    lastName: 'Kwak',
    insertion: 'Jodocus',
  }),
  createBankAccount({ firstName: 'Brad', lastName: 'Pitt' }),
  createBankAccount({ firstName: 'Jack', lastName: 'Sparrow' }),
];
```

4. Can you guess what the type of the `bankAccounts` variable is? Verify this by hovering over it in your code editor.

»

Exercise 4: If time permits

Add a method `toString` to the bank accounts created with the `createBankAccount` method. Use the `formatName` and `formatIban` functions to display them in the following order:

```
[NL15 TYPE 7608 1718] Alfred Jodocus Kwak
[NL65 TYPE 5016 0769] Brad Pitt
[NL76 TYPE 3727 8486] Jack Sparrow
```

Hint You might need to add the `toString` method to the `BankAccount` interface.

Lab 7 - Classes

Preparations

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* folder.

In this lab, we'll convert some interfaces to classes.

Exercise 1 - Converting interfaces to classes

1. Inside the main.ts file, change the `interface` keyword in front of `BankAccount` to `class`.
2. Change the `createBankAccount` function to be the `constructor` of `BankAccount`
3. Do the same for the `Customer` and `Iban` interfaces. Make sure that all attributes are assigned in their constructors. Be sure to let the `Iban` class be entirely `readonly`.
4. Convert `generateIban` to a static method in the `Iban` class. Feel free to rename it to `generate`.
5. Make sure that both `Customer` and `Iban` have a `format` method. Refactor `formatName` and `formatIban` for this purpose.

Exercise 2 - Creating a Bank class.

We want to add a `Bank` class. We expect it to have a lot of configuration, so we decide to create a interface called `BankConfig`.

1. Inside the main.ts file, create an `interface` called `BankConfig`.
2. Add the following fields (type: `string`)
   `name`, `countryCode` and `bankCode`.
3. Create a class called `Bank`, with:
   A `public` field `config` of type `BankConfig`
   A `private` field `accounts` of type `BankAccount[]`
   A `constructor` which accepts a `BankConfig` and assigns it to its own `config`.
4. Create a (public) method called `createAccount` which creates a bank account for a given customer and adds it to the private `accounts` array. It should also print `'[${bankName}] welcomes ${account}'` to the console.
5. Make sure the `bankCode` from the `config` attribute is used to instantiate the `Iban` instances inside the `createAccount` method.

Feel free to remove the old `DEFAULT_BANK_CODE` and `DEFAULT_COUNTRY_CODE` fields.

6. Both fields `config` and `accounts` should never be reassigned. Make sure this is the case.
7. Test it out! Remove the old `bankAccounts` array. Instead, create a new bank and create some accounts. Make sure everything works and you don't have compile errors.

```
const bank = new Bank({
  bankCode: 'TYPE',
  countryCode: 'NL',
  name: 'Typed bank',
});
bank.createAccount(new Customer('Alfred', 'Kwak', 'Jodocus'));
bank.createAccount(new Customer('Brad', 'Pitt'));
bank.createAccount(new Customer('Jack', 'Sparrow'));
```

Lab 8 - Generics

Preparations

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* folder.

In this lab, we'll implement an audit log function.

Exercise 1 - Create an `auditLog` function

1. Make sure both `Iban` and `Customer` have a `format` method. It should be without parameters and return a string. Copy it from the lab-solutions if it is missing.
2. Inside the main.ts file, add an `auditLog` function. It should take an argument called "subject" of type `Iban` and an argument called "action" of type string.
3. Inside the `auditLog` function, log the message in this form "[subject]: action". Log it to console for now (we'll have to implement an actual audit log later on). Make sure you call the `format` method on `subject` when you're logging it.
4. Call the `auditLog` function from inside the `BankAccount` constructor (just after you've created the `Iban`).

   ```
   auditLog(this.iban, 'created');
   ```

5. Now also use the `auditLog` function to log when a customer is assigned to a bank account. Add `auditLog(customer, 'assigned');` to the `createAccount` method of the `Bank` class. To make it work, make the `auditLog` function generic. Change the type of `subject` to be of generic type `T`. **Hint:** You might need a type constraint to make this work.

Exercise 2 - My very own `call` - if time permits

This is a fun exercise. Try to make your own `call` function. It takes a function as its first argument and a list of parameters as its following arguments. It should execute the function for you. This is what it looks like without generic type arguments:

```
function call(fn: any, ...args: any[]): any {
  return fn(...args);
}
```

Now introduce generic type arguments to make the function type-safe.

When you're done, you should be able to use it like this:

```
function increment(n: number) {
  return ++n;
}

console.log(call(increment, 41));
call(console.log, 'test');
```

However, this should result in compile errors:

```
call(increment, '42'); // => ERROR!
const str: string = call(increment, 42); // => ERROR!
```

Lab 9 - Modules

Preparations

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* folder.

In this lab, we will split our code into multiple files.

Exercise 1 - Split your code into multiple files.

Split all classes into their own files. Use `modules` with `import` and `export` to make sure that everything works.

The goal is to have a folder structure like this:

```
my-app/
├── node_modules/
├── package.json
├── src
│   ├── server
│   │   ├── bank.ts
│   │   ├── audit-log.ts
│   │   └── main.ts
│   └── shared
│       ├── bank-account.ts
│       ├── bank-config.ts
│       ├── customer.ts
│       └── iban.ts
└── tsconfig.json
```

Note: it is common to use lower-kebab-case for file names in JavaScript.

You can follow these steps to get there, but you can also try it yourself without following these steps. It's your choice.

1. Open your `tsconfig.json` and update your configuration:
   Set `module` and `moduleResolution` to `Node16`
   Enable `verbatimModuleSyntax`.
   Enable `isolatedModules`.
2. Open your `package.json` file and set `"type": "module"`.
3. Now move the code you intend to reuse across the client and server in the shared folder
   Create a directory `src/shared`.

Add `Customer` class to `src/shared/customer.ts`

Add `Iban` class to `src/shared/iban.ts`

Add `BankAccount` class to `src/shared/bank-account.ts`

Remove the call to the `auditLog` for now.

Add `BankConfig` interface to `src/shared/bank-config.ts`.

4. Move server-specific code to a src/server directory

Create a directory `src/server`

Add the `Bank` class to `src/server/bank.ts`.

Add the `auditLog` function to `src/server/audit-log.ts`.

Add the remaining code from the `main.ts` file to `src/server/main.ts`

5. Make sure your code compiles without errors.

Run the `src/server/main.js` file to make sure it still works as expected.

If you're using a `launch.json` file, make sure it runs your new `dist/src/server/main.js` file:

```
// .vscode/launch.json
{
   "version": "0.2.0",
   "configurations": [
      {
         "type": "node",
         "request": "launch",
         "name": "Launch BankServer",
-         "program": "${workspaceFolder}/dist/main.js",
+         "program": "${workspaceFolder}/dist/server/main.js",
         "outFiles": [
            "${workspaceFolder}/dist/**/*.js"
         ]
      }
   ]
},
```

»

Exercise 2 - Create a shared index.ts - If time permits

Create an `index.ts` file inside of the `src/shared` directory. Make sure to re-export all dependencies from the shared module. Use that index file to import from.

This way you can import multiple classes directly from the shared module, without knowing exactly where the file lives.

For example:

```
import { BankAccount, BankConfig, Customer } from '../shared/index.js';
```

The `index.ts` file can be seen as the "public API", while other files from `shared` can be viewed as the internal API (like C#'s `internal` or Java's modules).

Make sure your code compiles without errors. Run the `src/server/main.js` file to make sure it still works as expected.

Lab 10 - Create the bank server

Preparations

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* folder.

In this lab, we will create the bank web server.

Exercise 1 - Create a bank web server

For this exercise, we'll use the express web server.

1. Install express using `npm i express`
2. Install the *typings* for express `npm i -D @types/express`
3. Add a property `port` to the `BankConfig` interface. Be sure to update the instance of the BankConfig in your code (follow the compile error). Choose port number `8080` for the `'Typed bank'`.
4. Create a new class `BankServer`. The constructor should take a `Bank`. You can use this boilerplate:

```
import express from 'express';
export class BankServer {
  private app;

  constructor(private bank: Bank) {
    this.app = express();
  }
}
```

5. Add a `listen` method to the `BankServer`. It should create a webserver that listens to the port configured in the `config` of that `Bank`. Log a line to the console in the `listen` method to indicate that you are running the webserver.

```
this.app.listen(this.bank.config.port);
console.log(
  `Bank ${this.bank.config.name} listening on port
${this.bank.config.port}`
);
```

6. Use the official express documentation to implement the following methods:

HTTP GET on `/api/bank` should provide the config of the bank as a JSON object

HTTP GET on `/api/accounts` should provide the bank accounts belonging to the bank as a JSON array.

Hint: you can use `resp.json(myObject)` to respond with a JSON body and a `Content-Type: application/json` header.

7. In main.ts, create a new `BankServer` and pass the `Bank` instance to it. Then call the `listen` method on the `BankServer`.
8. Test it out! Start the webserver and navigate to `/api/bank` and `/api/accounts` in your browser. You should see the JSON response.

Note: Since the run command now opens a webserver, it keeps running until you manually stop it. You can stop it by pressing `Ctrl+C` in the terminal or by pressing the stop button in vscode.

You can also make `node` automatically reload using the new `--watch` feature. You can use it like this:

```
$ node --watch dist/src/server/main.js
```

Or when using vscode, you can change the `launch.json`:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceRoot}/dist/server/main.js",
      "cwd": "${workspaceRoot}",
      "sourceMaps": true,
+     "runtimeArgs": ["--watch"],
      "outFiles": ["${workspaceRoot}/**/*.js"]
    }
  ]
}
```

»

Exercise 2 (if time permits)

1. Implement the HTTP POST on `/api/customers`. It should add the provided JSON customer to the current bank and return a `204 - No Content`. Tip: you will need to use `app.use(express.json())` to enable express to parse JSON.
2. Try it out using the following cURL command. You can also use a tool like Postman or an HttpRequester browser plugin.

```
$ curl -H 'Content-Type: application/json' -d '{ "firstName": "James",
"lastName": "Bond" }' http://localhost:8080/api/customers
```

3. Feel free to implement some validation. These customers are invalid:

```
$ curl -H 'Content-Type: application/json' -d '{ "firstName": "Only first
name" }' http://localhost:8080/api/customers
```

```
$ curl -H 'Content-Type: application/json' -d '{ "lastName": "Only last
name" }' http://localhost:8080/api/customers
```

```
$ curl -H 'Content-Type: application/json' -d '{"firstName":
"John","lastName": "Doe","insertion": 42}'
http://localhost:8080/api/customers
```

Lab 11 - Add client-side code

Preparations

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* folder.

In this lab, we'll first set up the code with project references and ES module syntax to allow for sharing code between client and server.

Next, we'll create a banking website.

Exercise 1 - Setup project references

Set up your project to use project references. Your project structure should end up like this:

```
src
├── client
│   └── tsconfig.json
├── shared
│   └── tsconfig.json
└── server
    └── tsconfig.json
tsconfig.settings.json
tsconfig.json
```

In the `tsconfig.base.json` you can put all the shared compiler options. It should at least have these settings:

```
// tsconfig.base.json
{
  "compilerOptions": {
    "target": "es2022",
    "module": "Node16",
    "esModuleInterop": true,
    "verbatimModuleSyntax": true,
    "isolatedModules": true,
    "moduleResolution": "Node16",
    "strict": true,
    "sourceMap": true,
    "composite": true,
    "declarationMap": true,
    "declaration": true,
    "rootDir": "./src".
    "outDir": "./dist"
  }
}
```

The specific `tsconfig.json` files should extend the `tsconfig.base.json` file. They should also declare the `outDir` property to make sure code is outputted in the `dist` directory. Finally, both "client" and "server" should reference the "shared" project. This is an example of the server/tsconfig.json file. You can base the "client" and "shared" tsconfig files on it.

```
// src/server/tsconfig.json
{
  "extends": "../../tsconfig.settings.json",
  "references": [
    {
      "path": "../shared"
    }
  ]
}
```

The root tsconfig.json file should not refer to any ".ts" files directly. Instead, it only needs to reference all projects. The file looks like this:

```
// tsconfig.json
{
  "files": [],
  "references": [
    {
      "path": "src/client",
    },
    {
      "path": "src/shared",
    },
    {
      "path": "src/server",
    },
  ]
}
```

Create a dummy `client.ts` in the "client" project. It should contain the following code:

```
import { BankAccount } from '../shared/bank-account.js';
console.log('Hello ', BankAccount.name);
```

Make sure the output after `npx tsc -b` looks like this:

```
dist
├── client
│   ├── *.js.map
│   ├── *.d.ts
│   ├── *.d.ts.map
│   ├── *.js
│   └── tsconfig.tsbuildinfo
├── shared
│   ├── *.js.map
│   ├── *.d.ts
│   ├── *.d.ts.map
│   ├── *.js
│   └── tsconfig.tsbuildinfo
└── server
    ├── *.js.map
    ├── *.d.ts
    ├── *.d.ts.map
    ├── *.js
    └── tsconfig.tsbuildinfo
```

»

Exercise 2 - restrict types

Make sure you cannot use `document` (a browser global variable) in your backend code, or `Buffer` (a node global variable) in your frontend code! Hint: use the `lib` and `types` compiler options. In shared, you are not allowed to use either of them.

Exercise 3 - Create and serve the client app

In this exercise, we'll let our webserver also serve static files. We will load an html page in a browser which will function as our banking application.

Since all projects are emitting ES module code, we can easily share code between both NodeJS and the browser. Note that, in a real-life scenario, you would probably be using a bundler like webpack, rollup or parcel to package your client code.

1. Copy the `static` directory (`lab11/static`) right **next to** your `src` directory.
2. Alter your `src/server/bank-server.ts` file. Make sure it can serve client files and TS sources can be served. You can do this by registering these handlers in your express app. First, create a `resolve` function that can resolve relative file paths to absolute. You can put this in the `src/server/bank-server.ts` file.

```
import { fileURLToPath } from 'url';

const resolve = (relativePath: string) =>
  fileURLToPath(new URL(`../../${relativePath}`, import.meta.url));
```

3. With this new `resolve` function, we can now register the routes.

```
this.app.use(express.static(resolve('static')));
this.app.use(express.static(resolve('dist')));
this.app.use('/src', express.static(resolve('src')));
```

4. Now run your webserver (`dist/server/main.js`). Open a browser and go to http://localhost:8080. The index.html file should be visible in your browser.
5. Open the console in your browser (`F12` tools). The log message "Hello BankAccount" should be visible.
6. 🥧 congrats! You've got things up and running.

Notice that you can debug your *TypeScript* source code in the browser (with the `F12` tools). This is the magic of using source maps. *Source maps are useful for testing, but you would turn this off for a production application.*

Lab 12 - Advanced types

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* directory.

Note: Make sure that typescript compiler is running in watch mode (`npx tsc -b -w`).

Exercise 1 - Support for languages

1. Add a property called `language` in `BankConfig`. Make sure the only correct values are `'nl'`, `'en'` and `'fr'`.
2. In the `createAccount` method of `Bank`, change the `console.log` statement. Based on the `language` setting, it should either say
   '*[bank]* verwelkomt *[customer]*' (for `'nl'`)
   '*[bank]* welcomes *[customer]*' (for `'en'`)
   '*[bank]* accueille *[customer]* (for `'fr'`)

Make sure your code compiles and runs.

Exercise 2 - Type guard for new customers

In the `BankServer` class. Review the `HTTP POST` express `POST` handler for `/api/customers`. If you didn't get the chance to finish it, copy that part from the lab-solutions of lab `11`.

Make sure the `request.body` is an actual *valid* customer. Implement the validation method called `isValid` using a type guard. Be sure to type the HTTP POST payload as `unknown`, in order to see that the type guard works. You can use this code snippet to call the `isValid` method:

```
const maybeCustomer: unknown = request.body;
if (this.isValid(maybeCustomer)) {
  this.bank.createAccount(
    new Customer(
      maybeCustomer.firstName,
      maybeCustomer.lastName,
      maybeCustomer.insertion
    )
  );
  response.status(204);
  response.end();
} else {
  response.status(422);
  response.end('Customer entity invalid');
}
```

Make sure your code compiles and runs. Try it out using the following cURL command. You can also use a tool like Postman or an HttpRequester browser plugin.

```
$ curl -H 'Content-Type: application/json' -d '{ "firstName": "James",
"lastName": "Bond" }' http://localhost:8080/api/customers
```

Lab 13 - Advanced use cases

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* directory.

Note: Make sure that typescript compiler is running in watch mode (`npx tsc -b -w`).

Exercise 1 - Exhaustiveness checking for languages

In the `Bank` class, we've implemented a welcome message based on the language.

Let's now use exhaustiveness checking to make sure that we have a welcome message in all languages.

Verify that your exhaustiveness check works. Add a language to the `Language` union type, for example, `'de'`. It should result in a compile error.

Lab 14 - Await async calls

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* directory.

In this lab, we'll change the client to retrieve information from the server and show it on the screen. You can reuse the classes from the 'src/shared' folder so you don't have to recreate the `BankConfig` or `BankAccount` classes.

Note: Make sure that typescript compiler is running in watch mode (`npx tsc -b -w`).

Exercise 1 - Show name of the `BankConfig`

1. In the "client" directory, create a new file "bank-service.ts" to hold a new `bankService` object, responsible for http calls to the server.
2. In this object, create a method `retrieveBank` that will retrieve the `BankConfig` from the server (HTTP GET `/api/bank`). You can use the HTML5's `fetch` API. It returns a `Promise` for all calls:

   ```
   fetch('/api/bank').then(
     (response) => response.json() as Promise<BankConfig>
   );
   ```

3. In your 'client.ts' file, use the newly created function to retrieve the name of the current bank and to display it on the screen. You can select the title on the screen with: `document.querySelector('h1')`.
4. Test it out and see that you can actually see the name of the bank.
   If it doesn't work, be sure to check both consoles for error messages.
5. Now try to replace `promise.then()` calls to `await promise` calls in your 'client.ts' and 'backend-service.ts' files.

Exercise 2 - Show `BankAccounts`

1. Also create a method for retrieving `BankAccounts` from the server in your `Backend` class. Give it the name `retrieveBankAccounts`.
2. We'll now show our bank accounts inside an HTML table. There is already a *custom element* prepared for this. Create `src/client/bank-accounts-table-component.ts` and add copy paste this content:

```
import { BankAccount } from '../shared/bank-account.js';
export class BankAccountsTableComponent extends HTMLElement {
  private _accounts: BankAccount[] = [];

  public get accounts(): BankAccount[] {
    return this._accounts;
  }

  public set accounts(value: BankAccount[]) {
    this._accounts = value;
    this.updateTable();
  }

  public updateTable() {
    this.innerHTML = `<table class="table">
        <thead>
            <tr>
                <th>Account</th>
                <th>Name</th>
            </tr>
        </thead>
        <tbody>
        ${this.accounts
          .map(
            (account) =>
              `<tr><td>${account.iban.format()}</td><td>
${account.customer.format()}</td></tr>`
          )
          .join('')}
        </tbody>
        </table>`;
  }
}
customElements.define('bank-accounts-table', BankAccountsTableComponent);
declare global {
  interface HTMLElementTagNameMap {
    'bank-accounts-table': BankAccountsTableComponent;
  }
}
```

3. Now add this component to the page. Open "client.ts" and import it:

```
import './bank-accounts-table-component.js';
```

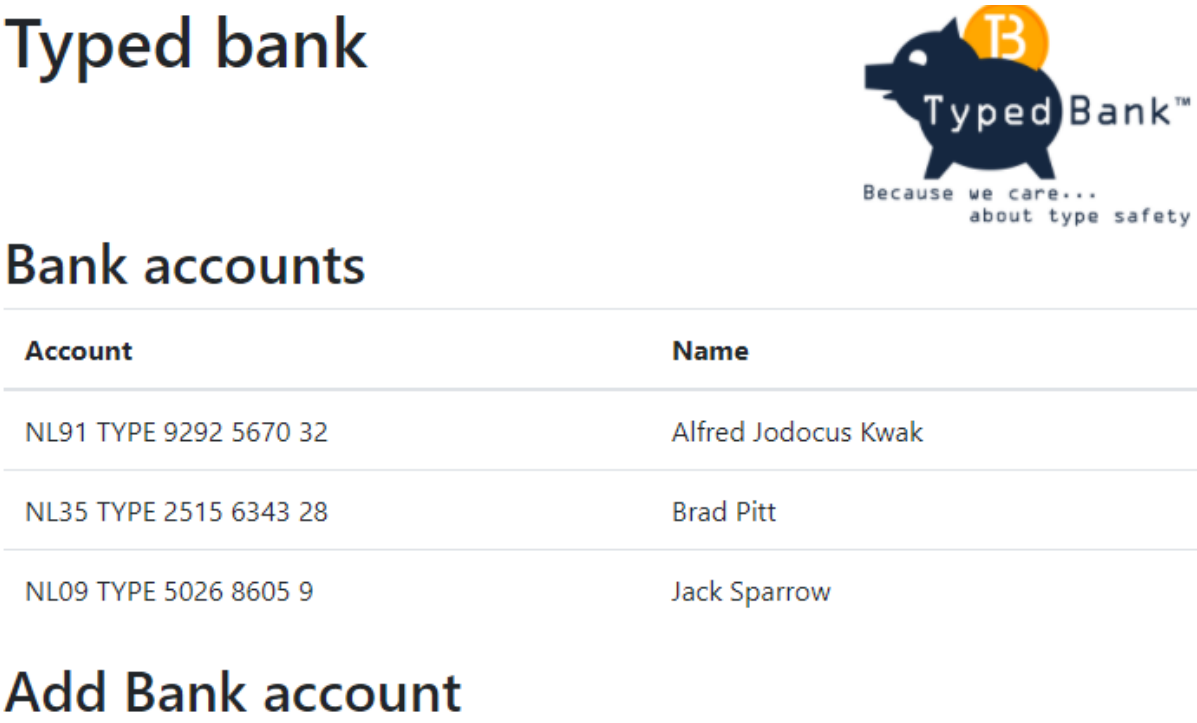
4. Now bind the bank accounts to the correct element using this code snippet:

```
const accounts = await backend.retrieveBankAccounts();
const bankAccountsTable = document.querySelector('bank-accounts-table')!;
bankAccountsTable.accounts = accounts;
```

5. Run the code and make sure it works. Be sure to check the console for errors if you cannot see the bank accounts on screen. You're free to change the code in order to

show the accounts.

If everything went well, it should look like this:



»

Exercise 3 (if time permits) - Add new `BankAccounts`

1. Add a new `addCustomer` method to your `Backend` class. It should perform a HTTP POST call to create a new customer on the server. You can use this code snippet.

```
await fetch('api/customers', {
  method: 'POST',
  body: JSON.stringify(customer),
  headers: { 'Content-Type': 'application/json' },
});
```

2. We'll create a new "Add customer" form to the page. Again, here is the custom element prepared for the form. Create `src/client/add-customer-component.ts` and copy-paste this content:

```
import { Customer } from '../shared/index.js';

export class AddCustomerComponent extends HTMLElement {
  connectedCallback() {
    this.render();
  }

  private render() {
    this.innerHTML = `<form class="form" name="customer">
         <div class="form-group">
             <label for="firstNameInput">First name</label>
             <input id="firstNameInput" name="firstName" type="text"
class="form-control">
         </div>
         <div class="form-group">
             <label for="lastNameInput">Last name</label>
             <input id="lastNameInput" name="lastName" type="text"
class="form-control">
         </div>
         <div class="form-group">
             <label for="insertionInput">Insertion</label>
             <input id="insertionInput" name="insertion" type="text"
class="form-control">
         </div>
         <button type="submit" class="btn btn-primary">Add</button>
     </form>`;
    this.form.addEventListener('submit', (event) => this.submit(event));
  }

  private submit(event: Event) {
    event.preventDefault();
    event.stopPropagation();

    const addCustomerEvent = new CustomEvent('customer-added', {
      detail: new Customer(
        this.form.firstName.value,
        this.form.lastName.value,
        this.form.insertion.value
      ),
    });
    this.dispatchEvent(addCustomerEvent);
  }

  private get form(): HTMLFormElement {
    return this.querySelector('form')!;
  }
}
customElements.define('add-customer', AddCustomerComponent);

declare global {
  interface HTMLElementTagNameMap {
    'add-customer': AddCustomerComponent;
  }
}
```

3. Now add this component to the page. Open "client.ts" and import it:

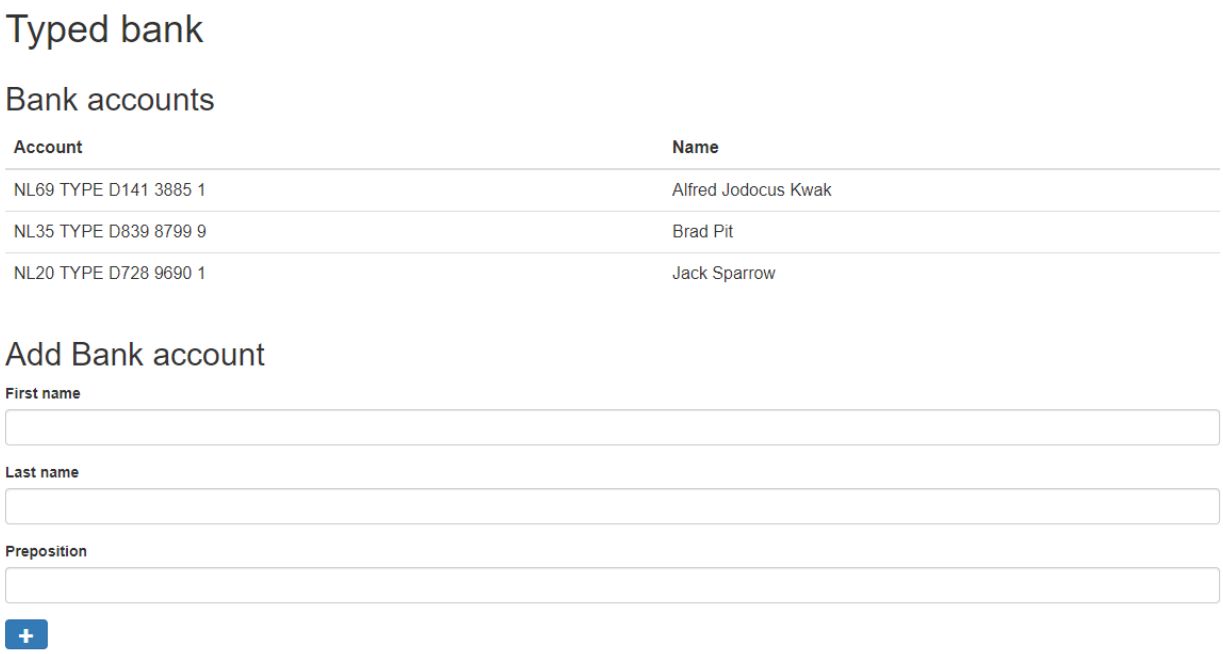
```
import './add-customer-component.js';
```

4. Now add a event listener to the `add-customer` element that is responsible for handling the `customer-added` event. You can use this starter code snippet (in "client.ts"):

```
document.querySelector('add-customer')?.addEventListener('customer-added',
((
  event: CustomEvent<Customer>
) => {
  // TODO, add customer in "event.detail"
}) as EventListener);
```

   Be sure to update the account table after the customer is successfully added.

If everything went well, it should look like this:



Lab 15 - Mapped & Conditional types

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* directory.

Note: Make sure that typescript compiler is running in watch mode (`npx tsc -b -w`).

Exercise 1 - Optional BankConfig

Right now, we are required to provide all bank config options when we create a new bank, even if we can think of sane defaults for the values. Let's change that using a mapped type.

1. Add an object containing the sane defaults to the `bank.ts` file. Create a constant variable called `DEFAULT_BANK_CONFIG`. Add these values as defaults:

```
{
    port: 8080,
    bankCode: 'TYPE',
    language: 'nl' as const,
    name: 'unknown bank',
    countryCode: 'NL'
}
```

2. Change the constructor of `Bank` so it now accepts a partial implementation of the `BankConfig` interface.
3. Inside the constructor, copy over the values to use. Use the defaults, but override with the provided values. **Hint**: If you use an ES2015 construct here it can be a one-liner.
4. Test your setup by only providing the bank name to the constructor in the `main.ts` file.
5. Bonus: make sure the default object is entirely read-only. Compile time as well as runtime.

Exercise 2 - Improve JSON types (if time permits)

In the `bank-service.ts` file (in the client project) we have a method for retrieving `BankAccount`s. However, the bank accounts we receive from the server are plain JS objects and don't have the `format` method. This is because they are serialized with `JSON.stringify` and deserialized with `JSON.parse`.

Create a mapped type using conditional types to improve the typing. The new type is based on the `BankAccount`, but with the methods removed. It looks like this (pseudo code):

```
// The goal, using pseudo code
type Jsonified<BankAccount> = {
  customer: {
    firstName: string;
    lastName: string;
    insertion?: string;
  };
  iban: {
    countryCode: string;
    bankCode: string;
    accountNumber: string;
    controlNumber: number;
  };
};
```

Start with this code, and store it in "src/shared/jsonified.ts":

```
export type Jsonified<T> = {
  [K in keyof T]: T[K];
};
```

Tip: You should end up with a recursive type.

Make sure this type is used when appropriate.

Lab 16 - A type-safe `fetch`

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* directory.

Note: Make sure that typescript compiler is running in watch mode (`npx tsc -b -w`).

Until now, we've been using the browser's `fetch` API as a means to call the backend. This is unfortunately not type-safe. In this lab, we'll try to improve this a bit.

Exercise 1 - Create routes

Create a new file called "routes.ts" in our "shared" project. This file will hold the routes within our application. Start with this code:

```
import type { BankAccount } from './bank-account.js';
import type { BankConfig } from './bank-config.js';
import type { Customer } from './customer.js';

export const contextRoot = 'api';

export interface Entities {
  accounts: BankAccount[];
  bank: BankConfig;
  customers: Customer[];
}

export type BankRoute = string; // TODO! Template literal type here
```

Alter the `BankRoute` type so it can only contain a valid bank route. For example:

```
const route: BankRoute = '/api/accounts'; // VALID
const route2: BankRoute = '/app/accounts'; // INVALID
const route3: BankRoute = '/api/acc0unts'; // INVALID
```

Be sure to use the `Entities` interface in your *template literal type*.

Don't forget to export `BankRoute`, `BankRoute` and `contextRoot` from the shared "index.ts" file.

Exercise 2 - Create a type-safe `get`

Create a new file "http.ts" inside the "client" project. Start with this code:

```
import { BankRoute } from '../shared/index.js';

export async function get(route: BankRoute): Promise<unknown> {
  const response = await fetch(route);
  if (!response.ok) {
    throw new Error(`Response was not OK: ${response.status}`);
  }
  return response.json() as Promise<unknown>;
}

export async function post(route: BankRoute, body: unknown): Promise<void> {
  const response = await fetch(route, {
    method: 'POST',
    body: JSON.stringify(body),
    headers: { 'Content-Type': 'application/json' },
  });
  if (!response.ok) {
    throw new Error(`Response was not OK: ${response.status}`);
  }
}
```

Start using this code from the `bankService` object:

```
// [...]
import * as http from './http.js';

export class Backend {
  public async retrieveBank(): Promise<BankConfig> {
    return http.get('/api/bank');
  }

  public async retrieveBankAccounts(): Promise<BankAccount[]> {
    const bankAccounts = await http.get('/api/accounts');
    return bankAccounts.map((bankAccount) => BankAccount.fromJson(bankAccount));
  }

  public async addCustomer(customer: Customer): Promise<void> {
    await http.post('/api/customers', customer);
  }
}
```

As you can see, this will result in multiple compile errors. Try to alter the `get` function in "http.ts", to make the function type-safe using template literal types and the `Entities` mapped type.

Hint: you might have to use the `infer` keyword.

»

Exercise 3 - Create a type-safe `post` - if time permits

Next, do the same for `post`. I.e. this should result in a compile error:

```
// [...]
export class Backend {
  // [...]

  public async addCustomer(customer: Customer): Promise<void> {
    await http.post('/api/accounts', customer);
    //                               ^^^^^^^^
    // Argument of type 'Customer' is not assignable to parameter of type
'BankAccount'.
  }
}
```

Lab 17 - Implement `@customElement`

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* directory.

Note: Make sure that typescript compiler is running in watch mode (`npx tsc -b -w`).

Exercise 1 - Create `@customElement`

Create a `@customElement()` decorator that can be used to create a custom element. It should be used like this:

```
@customElement('bank-accounts-table')
export class BankAccountsTableComponent extends HTMLElement {
  // [...]
}
```

The decorator should use `customElements.define` to register the custom element class. Use it from "bank-accounts-table-component.ts" (and "add-customer-component.ts" if it exists)

Exercise 2 - Create `@auditLog` (if time permits)

Alter the `auditLog` function to now be a decorator. You should be able to use it in the bank like this:

```
@auditLog
public createAccount(customer: Customer) {
  // [...]
}
```

It should still log the same. For example: `[Alfred Jodocus Kwak]: createAccount`

Lab 18 - Using typed-html

If you couldn't finish the previous exercise, you can copy and paste the previous solution from the *lab-solutions* directory.

Note: Make sure that typescript compiler is running in watch mode (`npx tsc -b -w`).

Exercise 1 - setup TypedHtml

1. Install typed-html: `npm i typed-html`
2. Configure your client tsconfig.json for JSX support. Add the following configuration:

```
{
  "jsx": "react",
  "jsxFactory": "typedHtml.createElement"
}
```

3. Update your file extension of `bank-accounts-table-component.ts` to `.tsx`. In that file, import `typedHtml` using:

```
import * as typedHtml from '../../node_modules/typed-
html/dist/esm/src/elements.js';
```

4. Make sure your `BankServer` also serves files from the `node_modules` directory. For example add this line:

```
this.app.use('/node_modules', express.static(resolve('node_modules')));
```

5. Try it out. You should be able to use HTML tags from inside the `bank-accounts-table-component.tsx` file.

Exercise 2 - Replace the string template in the BankAccountsTableComponent

Change the implementation of the `updateTable()` method. It should now use the XML-like syntax instead of a string to define the template.

Exercise 3 - Replace string template in the AddCustomerComponent - if time permits

Now also change the implementation of the `render()` method in "add-customer-component.ts".