OOP Exercises

(Credit for exercises 1.5, 2.6 and 2.7 goes to Mike Spivey, Gavin Lowe and Joe Pitt-Francis.)

0. Introduction

Exercise 0.1

Follow the instructions on slides 6, 7 and 8 of `0-how-to-ts` and make sure that everything works as expected.

Exercise 0.2

Create another file called `math.ts`. Write in `main.ts` and `math.ts` the code from slide 20 of `0-how-to-ts`. Compile, run and make sure that the number `4` is printed to console.

1. Basic Types

Exercise 1.1

Write a function which returns the largest perfect square which is less than or equal to `x`:

```
function largestSquare(x: number): number
```

Handle invalid inputs by throwing `RangeError`. You must use a for loop: you cannot use any of the `Math` functions.

Exercise 1.2

Recall that Fibonacci numbers are defined inductively by:

- $F_0 = 0$ and $F_1 = 1$
- $F_{n+2} = F_{n+1} + F_n$

Write a function which returns the `n`-th Fibonacci number F_n, using recursion (i.e. `fibRec` is allowed to call itself in its own code):

```
function fibRec(n: bigint): bigint
```

The `bigint` type ensures that `n` is an integer, but not necessarily that `n` is non-negative. Handle invalid inputs by throwing `RangeError`.

Exercise 1.3

Write a function which returns the `n`-th Fibonacci number F_n, without using recursion (`fib` is not allowed to call itself, instead you should use a `for` loop):

```
function fib(n: bigint): bigint
```

The `bigint` type ensures that `n` is an integer, but not necessarily that `n` is non-negative. Handle invalid inputs by throwing `RangeError`.

Exercise 1.4

Write a function which determines whether a string is a palindrome:

```
function isPalindrome(str: string): boolean
```

Exercise 1.5*

Write function which prints (using template strings) the tree of recursive calls to `fibRec`, e.g.

```
printFib(3n)
/* Output:
fib(3)
| fib(2)
| | fib(1)
| | = 1
| | fib(0)
| | = 0
| = 1
| fib(1)
| = 1
= 2
*/
```

Use an auxiliary function to keep track of the recursion depth, starting at 0, as well as the actual Fibonacci numbers.

Exercise 1.6

The following are all basic types, write down which ones:

1. `number | never`
2. `number | unknown`
3. `string & never`
4. `string & unknown`
5. `number & bigint`
6. `number & boolean`
7. `number & (number | undefined)`
8. `string | (boolean & bigint)`
9. `number | (number & number)`

Is `number|undefined` a basic type? Is `number&undefined`?

2. Arrays and Functions

Exercise 2.1

Write a function which computes and returns the sum of the elements in the given array:

```
function sum(a: number[]): number
```

Exercise 2.2

Write a function `map` which returns a new array containing `f(x)` for all elements `x` of `a`, in the original order:

```
function map(a: number[], f: (x: number) => number): number[]
```

Exercise 2.3

Write a function `filter` which returns a new array of elements `x` of `a` for which `f(x) == true`, in the original order:

```
function filter(a: number[], f: (x: number) => boolean): number[]
```

Exercise 2.4

Use ordinary and optional parameters to write a function `range` which takes one or two arguments of type `number` and returns a `number[]` defined as:

- `range(n)` returns the integers `i` such that `0<=i<n`, in increasing order;
- `range(s, n)` returns the integers `i` such that `s<=i<n`, in increasing order.

Exercise 2.5

Use ordinary and rest parameters to write a function `max` which takes one or more arguments of type `number` and returns their maximum. Calling `max()` with no arguments should result in compile-time error:

```
max(1, 5, 3); // OK: 5
max(2); // OK: 2
max();
/* Error (2555):
Expected at least 1 arguments, but got 0.
*/
```

Exercise 2.6

Given an array of numbers `a`, we say that a *hit* occurs at index `j` if all element at the left of position `j` are less than the element `a[j]` (i.e. if `a[i]<a[j]` for all `0<=i<j`). Write a function which uses a loop to compute and return the number of hits in the given array `a`:

```
function hits(a: number[]): bigint
```

You code should run in time proportional to the length of the array (i.e. linear time, not quadratic time).

Exercise 2.7 (harder)

Alice is thinking of a positive integer $x \geq 1$ unknown to Bob. She provides a function `tooBig(y: bigint): boolean` that returns `true` if `x < y` and `false` if `x >= y`. Write a function that Bob can use to determine the number `x` by calling the function `tooBig` logarithmically many times:

```
function findX(tooBig: (y: bigint) => boolean): bigint
```

The function works as follows:

1. Set $a_1 = 1$, so that $a_1 \leq x$. Find the smallest $k_1 \geq 0$ such that $x < a_1 + 2^{k_1}$. If $k_1 = 0$, then $x = a_1$ and we're done.
2. Set $a_2 = a_1 + 2^{k_1 - 1}$, so that $a_2 \leq x$. Find the smallest $k_2 \geq 0$ such that $x < a_2 + 2^{k_2}$. If $k_2 = 0$, then $x = a_2$ and we're done.
3. Set $a_3 = a_2 + 2^{k_2 - 1}$, so that $a_2 \leq x$. Find the smallest $k_3 \geq 0$ such that $x < a_3 + 2^{k_3}$. If $k_3 = 0$, then $x = a_3$ and we're done.
4. (...hopefully you get the gist...)

Remember that you don't have direct access to x: to test whether $a_i + 2^{k_i} \leq x$, you have to query the `tooBig` function.