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.

3. Objects

Exercise 3.1

Define an object literal type `AuthUser` with three properties:

- `id` of type `number`
- `name` of type `string`
- `surname` of type `string`

Write a function which sorts an array of `AuthUser` instances in place by either `id`, `name`, or `surname`:

```
sortUsers(users: AuthUser[], sortBy: "id"|"name"|"surname")
```

To sort an array `a: T[]` in place you can use the method `a.sort(compareFn)`, where the comparison function `compareFn(a: T, b: T): number` should return:

- `+1` if `a` is greater than `b` according to the chosen sorting criterion
- `-1` if `a` is lesser than `b` according to the chosen sorting criterion
- `0` if `a` is equal to `b` according to the chosen sorting criterion

Here is an example:

```
type Pair = [number, number];
const compareFn = (a: Pair, b: Pair): number => {
  // Compares pairs according to first entry.
  if (a[0] < b[0]) { return -1; }
  if (a[0] > b[0]) { return +1; }
  return 0;
}
let arr: Pair[] = [[2, 1], [0, 2], [1, 3]];
arr.sort(compareFn); // in place
console.log(arr); // Output: [[0, 2], [1, 3], [2, 1]]
```

Hint. Remember that properties can be accessed both with dot notation and with dictionary notation. This can be used to significantly simplify your code.

Exercise 3.2

Write a class `Frac` for fractions:

- a property `num` (numerator) of type `bigint`
- getter and setter for a property `den` (denominator) of type `bigint`
- a method `add(other: Frac): Frac` that performs addition of fractions:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

- a method `toString(): string` which returns the fraction as a string, e.g. $\frac{-3}{2}$ is returned as `-3/2` and $\frac{2}{1}$ is returned as `2`;
- a method `valueOf(): number` which returns the floating point number corresponding to the fraction

The value for `den` should be held internally a property `_den` (this should be a `private` property, if we've already gone over those in the lecture).
The setter `den(x: bigint)` should perform some validation:

- it should throw an error if `x == 0n`
- it should ensure that the denominator is always set to a positive integer, by negating both `x` and the numerator when `x < 0n`

The constructor should take a value for the numerator, and an optional value for the denominator (default is `1n`).

Test your implementation by creating a few fractions and adding them, comparing them, printing them to console:

```
const frac1 = new Frac(3n,2n);
const frac2 = new Frac(2n);
console.log(`${frac1} + ${frac2} = ${frac1.add(frac2)}`);
console.log(`${frac1} <= ${frac2} ? ${frac1 <= frac2}`);
```

Note that fractions can be compared directly (`valueOf` is called under the hood). Also, using fractions into template strings—or manually converting them to strings as `String(frac)`—automatically calls `toString()`.

Exercise 3.3

A *bag* is an unordered collection with multiplicity: it keeps track of which elements are in it and how many copies it has for each element, but has no information about the order of elements.

Implement a class for a Bag of strings, following the skeleton below (if we've not yet gone over private properties in class, ignore the `private` modifier for `_elements`):

```

/**
 * Class for a bag of strings.
 */
class Bag {
    /**
     * Dictionary (type with index signature)
     * mapping each unique string in the bag to
     * the number of copies that appear (a bigint).
```

```
    */
    private _strings: {
      [str: string]: bigint
    } = {};

    constructor(...strings: string[]) {
        /* Add all strings to this bag.*/
        ...
    }

    /**
     * The total number of strings in this bag,
     * each counted with its multiplicity.
     */
    get size(): bigint { ... }

    /**
     * Adds a new string to the bag.
     */
    public add(str: string): void { ... }

    /**
     * deletes a string from the bag.
     */
    public delete(str: string): void { ... }

    /**
     * Counts how many copies of a given string
     * exist in the bag.
     *
     * Hint: use the nullish coalescing operator ??
     */
    public count(str: string): bigint { ... }

    /**
     * Returns a new array containing all unique
     * strings in this bag (each appearing once).
     */
    public values(): string[] {...}

    /**
     * Returns a new array containing all strings
     * in this bag, counted with their multiplicity.
     */
    public toArray(): string[] {...}

    /**
     * Compares this bag with another bag for equality
     * (same strings, same multiplicities).
     *
     * Hint: use 'values' and 'count' for both bags,
     * but don't rely on the order of values.
     */
    public equals(other: Bag): boolean {...}
}
```

For `size`, `toArray` and `equals`, recall that you can iterate they keys of a dictionary by using a `for..in` loop. For `equals`, know that you can access `other._strings` within the code of `Bag` methods (even though it is private). Test your implementation:

```
const bag1 = new Bag("a", "c", "a", "b", "a", "b");
const bag2 = new Bag("c", "a", "b", "b");
console.log(bag1.size); // 6n
console.log(bag2.count("b")); // 2n
console.log(bag1.toArray()) // [ 'a', 'a', 'a', 'c', 'b', 'b' ]
console.log(bag1.values()) // ['a', 'c', 'b']
console.log(bag2.values()) // ['c', 'a', 'b']
console.log(bag2.toArray()) // [ 'c', 'a', 'b', 'b' ]
console.log(bag1.equals(bag2)); // false
bag1.delete("a");
bag2.add("a");
console.log(bag1.equals(bag2)); // true
```

4. SOLID Principles

Exercise 4.1

Refactor the `Bag` from into three classes:

```
abstract class AbstractBag {
    get size(): bigint { ... }
    public has(str: string): boolean { ... }
    public equals(other: AbstractBag): boolean
    public toArray(): string[]
    public abstract count(str: string): bigint
    public abstract values(): string[]
}
class ReadonlyBag extends AbstractBag{
    private _strings: { [str: string]: bigint } = {};
    constructor(...strings: string[]) { ... }
    public count(str: string): bigint { ... }
    public values(): string[] { ... }
    // + one protected method of your design
}
class MutableBag extends ReadonlyBag{
    constructor(...strings: string[]) { ... }
    public add(str: string) { ... }
    public delete(str: string) { ... }
}
```

The `count` and `values` methods are enough to implement all other methods of `AbstractBag`, without the need to access `_strings` directly. You might have to slightly alter some of your previous code (depending on your implementation choices).

The `_strings` property is `private` in `ReadonlyBag` and cannot be accessible directly by `MutableBag`: this will cause you some issues with the implementation of `add` and `delete`. You should solve this issue by defining a suitable protected method in `ReadonlyBag`, allowing string counts to be modified safely (e.g. disallow setting negative counts).

Exercise 4.2

Following the Interface segregation principle—and inspired by the example in the slides—create small interfaces that define small blocks of functionality implemented by the bag classes from the previous exercise. Focus on features they have in common with the builtin `Set<string>` class, and define interfaces which work for both bags and `Set<string>` (by slightly widening the types, if necessary).

Make the bag classes implement the relevant interfaces and check that you get no error. Make an interface `ISet` by extending all interfaces that you think apply to `Set<string>`: if you're right, the following code will compile without error.

```
const testISet: (x: Set<string>) => ISet = x => x;
```

Hint. Using IntelliSense to inspect the types of methods for `Set<string>`, you might encounter the type `IterableIterator<string>`. When writing your interfaces, it might help you to know that both this type and `string[]` are sub-types of `Iterable<string>`.

Exercise 4.3

The following code presents a basic implementation for the moving logic of some chess pieces. Refactor it to use Composition over Inheritance: the `Piece` class should no longer be abstract and individual piece types should not be implemented by subclasses (a possible solution is to implement them by evocatively named functions instead).

```
type Colour = 'Black' | 'White';
type Rank = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8;
type File = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H';
type Pos = readonly [File, Rank];

function fileToNumber(file: File): number{
    return 1+['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'].indexOf(file);
}
function file(pos: Pos): File { return pos[0]; }
function rank(pos: Pos): Rank { return pos[1]; }
function fileDifference(from: Pos, to: Pos): number {
    return Math.abs(fileToNumber(file(to))-fileToNumber(file(from)));
}
function rankDifference(from: Pos, to: Pos): number {
    return Math.abs(rank(to)-rank(from));
}

abstract class Piece {
    constructor(public readonly colour: Colour, private _pos: Pos){}
    get position(): Pos{
        return this._pos;
    }
    abstract canMoveTo(position: Pos): boolean;
}

class Queen extends Piece {
    canMoveTo(position: Pos): boolean {
        const fileDiff = fileDifference(this.position, position);
        const rankDiff = rankDifference(this.position, position);
        return fileDiff == rankDiff || fileDiff == 0 || rankDiff == 0;
```

```
    }
}

class Pawn extends Piece {
    canMoveTo(position: Pos): boolean{
        const [currFile, currRank] = this.position;
        const [nextFile, nextRank] = position;
        return currFile == nextFile && nextRank == currRank+1;
    }
}

class Bishop extends Piece {
    canMoveTo(position: Pos): boolean {
        const fileDiff = fileDifference(this.position, position);
        const rankDiff = rankDifference(this.position, position);
        return fileDiff == rankDiff;
    }
}
```

Exercise 4.4

Refactor the following interface into a few smaller interfaces, according to the Single responsibility principle.

Note. This is a rather open-ended question: don't overthink it!

```
type MessageID = bigint;
type EmailAddress = string;
interface EmailClient {
    createMessage(to: readonly EmailAddress[], subject: string): MessageID
    getMessageBody(id: MessageID): string
    setMessageBody(id: MessageID, newContents: string): void
    isMessageSent(id: MessageID): boolean // checks in sent folder
    sendMessage(id: MessageID): void // additionally stores in sent folder
    getEmailAddress(name: string): EmailAddress|undefined
    setEmailAddress(name: string, EmailAddress: EmailAddress): void
    get getContactNames(): readonly string[]
    filterInbox(subjectContains?: string, from?: EmailAddress): readonly
MessageID[]
    filterArchive(subjectContains?: string, from?: EmailAddress): readonly
MessageID[]
    filterSent(subjectContains?: string): readonly MessageID[]
    storeInInbox(message: MessageID): void
    storeInArchive(message: MessageID): void
}
```