TINA

Style Guide

Stylistic Guidelines for writing code in TypeScript for the Tina Framework.

This Style Guide is heavily derived from Google's Styleguide for JavaScript.

1 - Introduction

Next - Formatting

1 Introduction

1.1 File Structure

The file structure of all Tina projects can be found at Quick Start.

1 Introduction

1.2 Source Files

All TypeScript files should end with extension `.ts`, and should use a `PascalCase` naming scheme; any files that contain TSX code need to hold extension `.tsx`.

```
index.ts
Game.ts
/ui
   index.ts
   App.tsx
   PageRouter.tsx
```

1 Introduction

1.3 Source Folders

All folders with Source Code should be labeled in abbreviations or very short names, such as `/ui` for *User Interface*.

If a folder contains `.rbxm` or `.rbxmx` files, these should be explained with a \`meta.md\`.

```
index.ts
/lib
   /elements
      meta.md
   /ui
      index.ts
```

2 - Formatting

2 Formatting

2.1 Braces

Braces are **required** for all control structures (`if`, `else`, `for`, `do`, `while`).

Braces are always kept on the same line as the control structure initiated, there will be no C++*isms*. The only exception is a *quick-*`if`;

```
if (shortCondition()) return;
// or
if (shortCondition()) foo();
```

2 Formatting

2.2 Block Indentation

Blocks must always abide by the following core points:

- All indentation must be +4 **spaces**.
- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.

2 Formatting

2.3 Statements

Statements must always abide by the following core points:

One statement per line

Every statement must be followed by a line-break.

Semicolon after every statement

Every eol-statement must be directly terminated by a semicolon, missing semicolons will result in docked pay. This does not include control structures, class definitions, arrow functions, etc.

2 Formatting

2.4 Column Limit

All lines should end at 100 columns before being wrapped or scrapped.

Exceptions

- `import` statements that have less than 4 members being imported.
- Lines that need to be user-discovered, such as URLs or Commands.
- Mission-Crucial data such as encryption or API keys.

2 Formatting

2.5 Line-Wrapping

The prime directive of line-wrapping is: prefer to break at the highest syntactic level;

```
currentTotal = math.pow(currentTotal, (currentAggregator - currentTotal) * 1.2) / 2.0 * 3.8 - currentAggregator;
```

```
currentTotal =
    math.pow(
        currentTotal,
        (currentAggregator - currentTotal) * 1.2
    )
        / 2.0 * 3.8
        - currentAggregator
```

2 Formatting

2.6 Whitespace

2.6.1 Vertical Whitespace

A single blank line can — and *should*, appear when any of the following are true:

- Between unrelated method calls.
- Between unrelated method definitions in a class or object literal.
- Between unrelated property definitions in a class or object literal.

Do **NOT** include a vertical whitespace on creation of a block.

2.6.2 Horizontal Alignment

Do **NOT** try to align tokens above or below each other in any context.

Example:

```
{
    //    ------ added spaces
    small:       248_000
    veryDamnBig: 12
}
```

2.6.3 Horizontal Whitespace

General guidelines for placement of Horizontal Whitespaces;

- Do **NOT** include trailing whitespaces.
- Always place a whitespace after a reserved word (`if`, `switch`, ...); This rule is only violated by `function` and `super`
- Place a whitespace before any open curly brace.
- Place a whitespace before and after the members of a tuple definition. (e.g: `[ foo, bar ]`)
- Always place a whitespace after the beginning of a comment.
- Always place a whitespace after an inline multiline comment has ended. (e.g: `foo(/** @type {number} */ bar)`).

Generally, try not to glue stuff together.

2 Formatting

2.7 Grouping Parentheses

Grouping is generally recommended where possible and reasonable. The reader should not be expected to have the entire operator precedence table memorized.

That said, do **NOT** group around operators such as `delete`, `typeof`, `void`, `return`, `throw`, `case`, `in`, `of`, or `yield`.

2 Formatting

2.8 Comments

Comments are preferred in their multi-line form (`/* ... */`) whenever they take up a full line or more, and should primarily use a leading `//` when appending information to the end of a line.

`/** ... */` is **reserved** for TSDoc comments.

2 Formatting

2.9 TSDoc

TSDoc comments reserve the following comment syntax.

```
/**
 * description
 * @param {string} arg
 */
function foo(arg: string) {}
```

Do **NOT** leave a title on the leading line (`/** foo`), TSDoc comments do not need to be titled. TSDoc comments are optional.

3 - Language Features

3 Language Features

3.1 Variables

All variables where possible will use `const`, decreasing to `let` if needed. `var` is **banned**.

3 Language Features

3.2 Array Literals

Arrays will be defined using `[]` exclusively.

```
const values = [
    'foo',
    'bar',
]
```

3.2.1 Trailing Commas

Always use trailing commas when an array is on multiple lines. Single-line arrays are not *required* to have trailing commas.

3.2.2 Destructuring

Destructuring of arrays is permitted and syntax should be that of tuples, including when using spreead arguments;

```
const [ a, b, vec, ...items ] = calculate();
```

Unused elements should be omitted so as to not polute the environment,

```
let [ , b, , ...items ] = calculate();
```

3.2.3 Spread operator

Spread operators should be used without a space following the ellipsis (`...`). Naming of a spread argument should always be plural representing what it possesses.

3 Language Features

3.3 Object Literals

3.3.1 Trailing Commas

Always use trailing commas.

3.3.2 Key Flavors

Pick a single key flavor and stick with it, do not mix quoted/unquoted/symbollic keys. The following illustrates this;

```
const obj = {
    time: 42,
    'time_ms': 42_000,
    ["time_ns"]: 42_000_000
}
// Instead:
const obj = {
    time: 42,
    time_ms: 42_000,
    time_ns: 42_000_000
}
```

3.3.3 Method Shorthand

It is **forbidden** to use `methodName: () => foo` inside of an object literal, you **must** use `methodName() {}` shorthand instead.

```
const obj = {
    time: 42,
    foo(otherTime: number) {
        /* ... */
    },
}
```

3 Language Features

3.4 Enumerators

Enumerators (`enum`) should be used with static constants, all mission-critical enums should have all their mapped values stay static, so there will be no adding to the middle of an enum and suddenly breaking Production data. This also helps with readability.

```
enum ViewSetting {
    Panoramic = 0,
    Portrait = 1,
    Landscape = 2
}
```

3 Language Features

3.5 Classes

Classes should always be defined with the ES6 syntax `class Name {}`. Constructors are optional, and fields should either all be defined in the object body or in the constructor.

3.5.1 Field Composition

It is **NOT** allowed to mix Object Body/Constructor Fields if the constructor is collapsed or body-less.

```
class Example {
    public bar: number;
    constructor(public foo: string) {} // This is **NOT** okay.
}
// But,
class Example2 {
    public bar: number;
    constructor(
        public foo: string, // This is fine.
    ) { /* ... */ }
}
```

3.5.2 Class Field Stack

```
class E {
    private readonly thatOneBirthmarkOnMyBottom: Memory;
    private readonly loveForMyMother: number;

    private nonoSquare: Region;
    private deepestSecrets: Array<string>;

    protected readonly socialSecurity: number;
    protected bankPin: number;

    public readonly birthday: Date;
    public age: number;
    public friends: Array<number>;

    /* then static, then methods */
}
```

3 Language Features

3.6 Functions

3.6.1 Top-Level Functions

All Top-Level functions should be Arrow Functions (`() => {}`), usage of braces is as previously specified.

3.6.2 Trailing Commas in Parameters

Trailing Commas should always be used when parameters are expanded. Single-line definitions are allowed optional commas.

```
let bees = (
    str: string,
    n: number, // This comma is required.
) => {};
```

3.6.3 Return Types

Whenever a return type is unclear, it should be specified, this is left to author's discretion but will sometimes be remarked on more complex methods.

3 Language Features

3.7 String Literals

All string literals will use single quotes (`'`). If several lines are required, use template literals instead.

3.7.1 Template Literals

All template literals will use the backtick character (`` ` ``)

3.7.2 Line Continuations

Do **NOT** use `\` or terminate string, all line continuations are strictly banned and will sadly just have to be too long.

3 Language Features

3.8 Number Literals

Number literals may use `0x`, `0o`, and `0b` where possible as prefixes for hex, octal, and binary respectively. Never use any leading zeroes (except when used to illustrate data registers).

Feel free to use `200_000` syntax with underscores delimiting every 3rd power of 10, though please do not apply this to decimals.

3 Language Features

3.9 Control Structures

3.9.1 `for` loops

`for`-`of` loops are always preferred when their use is possible, `@rbxts/object-utils` provides `Object.keys()` to permit their continued efficient usage.

3.9.2 `try {} catch {}` statements

Exceptions should only be used when it is possible for the block to throw any errors, and shouldn't be used for fun. Very importantly **always** leave a comment exactly as follows whenever you leave an error unhandled.

```
try {} catch (e) { /* UNHANDLED ERROR */ }
```

If an error should not be handled, leave an appropriate comment stating that;

```
try {} catch (e) { /* Continue; we're okay if this didn't work */ }
```

3.9.3 `switch` statements

3.9.3.1 Fall-through in `switch`

Fall-through is a very useful feature but can often lead to confusion, leave comments whenever there is a fall-through as long as the current case has had any code;

Example:

```
switch (input) {
    case 1:
    case 2: // This fall through is okay and does not need to be commented.
    case 3:
        code();
        /* fall through */
        // This fall through needed to be commented as code has executed, it needs to be clear we're continuing downwards.
    case 4:
        break;
    default:
        handleNothing();
}
```

3.9.3.2 `default` block

The `default` block is **necessary** no matter what, even if it's not feasible for it to be reached, simply because code can change and evolve and we need to be prepared. It is not mandatory to use `break` to terminate it.

3 Language Features

3.10 `this`

The `this` keyword should only be used whenever inside of a class, so as to not confuse contexts.

3 Language Features

3.11 Operators

3.11.1 Equality Checks

Always use identity operators (`===` / `!==`).

3.11.2 Increment/Decrement Operators

Increment/Decrement *may* be used solely for integers, do not use `++` or `--` on floats or non-linear values. `+=` and `-=` are usually preferred.

3 Language Features

3.12 Ternary Statements

Ternary statements should be constructed with `<condition> ? <ifTrue> : <ifFalse>`, wrapping them should always have the `?` and `:` leading their own lines, at one tab deeper than the start line of the condition.

```
doFoo(
    longConditionalExpression.isValid()
        ? "valid"
        : "error"
);
```

3 Language Features

3.13 Namespaces

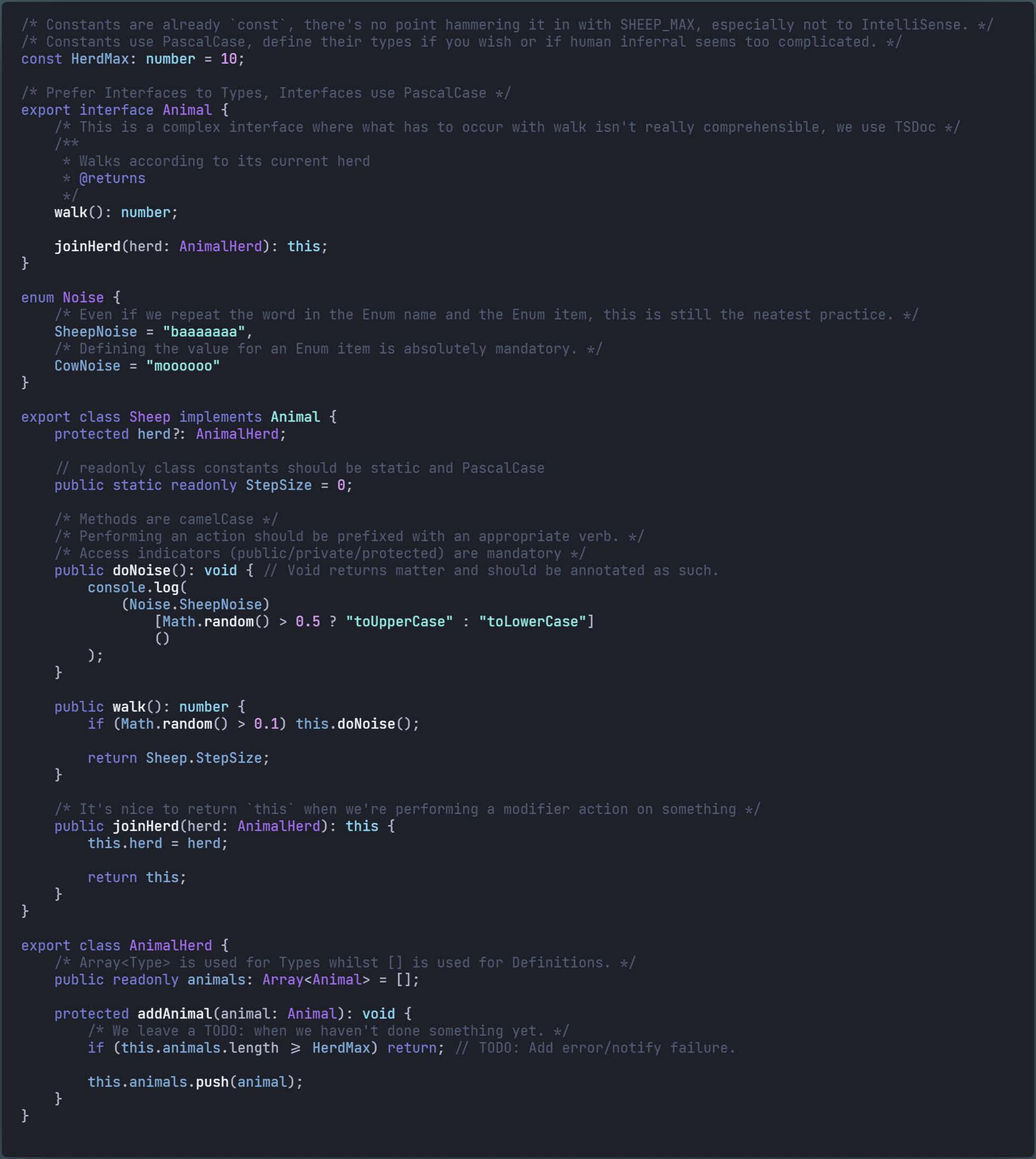
Namespaces should never contain nested classes inline, please include them through (ugly but better) syntax:

```
class FooCake {}

export namespace Foo {
    export const Cake = FooCake;
}
```

Any relevant Types should also be exported through Namespaces.

Please see Naming Scheme.



```typescript
/* Constants are already `const`, there's no point hammering it in with SHEEP_MAX, especially not to IntelliSense. */
/* Constants use PascalCase, define their types if you wish or if human inferral seems too complicated. */
const HerdMax: number = 10;

/* Prefer Interfaces to Types, Interfaces use PascalCase */
export interface Animal {
    /* This is a complex interface where what has to occur with walk isn't really comprehensible, we use TSDoc */
    /**
     * Walks according to its current herd
     * @returns
     */
    walk(): number;

    joinHerd(herd: AnimalHerd): this;
}

enum Noise {
    /* Even if we repeat the word in the Enum name and the Enum item, this is still the neatest practice. */
    SheepNoise = "baaaaaaa",
    /* Defining the value for an Enum item is absolutely mandatory. */
    CowNoise = "moooooo"
}

export class Sheep implements Animal {
    protected herd?: AnimalHerd;

    // readonly class constants should be static and PascalCase
    public static readonly StepSize = 0;

    /* Methods are camelCase */
    /* Performing an action should be prefixed with an appropriate verb. */
    /* Access indicators (public/private/protected) are mandatory */
    public doNoise(): void { // Void returns matter and should be annotated as such.
        console.log(
            (Noise.SheepNoise)
                [Math.random() > 0.5 ? "toUpperCase" : "toLowerCase"]
                ()
        );
    }

    public walk(): number {
        if (Math.random() > 0.1) this.doNoise();

        return Sheep.StepSize;
    }

    /* It's nice to return `this` when we're performing a modifier action on something */
    public joinHerd(herd: AnimalHerd): this {
        this.herd = herd;

        return this;
    }
}

export class AnimalHerd {
    /* Array<Type> is used for Types whilst [] is used for Definitions. */
    public readonly animals: Array<Animal> = [];

    protected addAnimal(animal: Animal): void {
        /* We leave a TODO: when we haven't done something yet. */
        if (this.animals.length >= HerdMax) return; // TODO: Add error/notify failure.

        this.animals.push(animal);
    }
}
```

6 - Abstract Patterns

6.1 - Structural Patterns

- 6.1.1 - Bridge
- 6.1.2 - Adapter
- 6.1.3 - Composite
- 6.1.4 - Decorator
- 6.1.5 - Facade
- 6.1.6 - Flyweight
- 6.1.7 - Proxy

6.2 - Creational Patterns

- 6.2.1 - Factory Method
- 6.2.2 - Abstract Factory
- 6.2.3 - Builder
- 6.2.4 - Prototype
- 6.2.5 - Singleton

6.1 Abstract Patterns - Structural Patterns

6.1.1 Bridge

The Bridge Pattern is a pattern that lets you split abstraction and implementation into two separate hierarchies, using it is authorized.

6.1.2 Adapter

Adapters are middle-men between data, using them is perfectly fine, though their use being required should be something looked into. All Adapters should explicitly be named "____Adapter"

6.1.3 Composite

Composites allow a branching tree of objects and data, this pattern should be avoided.

6.1.4 Decorator

Decorators are cryptic and side-effect-y, avoid them at all costs.

6.1.5 Facade

Facades present simpler and shorter APIs for complex sets of data or features, using them is fine. All Facades should explicitly be named "____API"

6.1.6 Flyweight

Flyweight is the sharing of global state or data, it is heavily encouraged and morphing it to Boids or other grouping techniques is great. Using Flyweight in ECS is complicated but can and should be accomplished.

6.1.7 Proxy

Proxies handle caches or access to complex and heavy APIs. Restrict Proxy usage to actually complex objects, name them discretely so as to not impersonate the eventual object.

6.2 Abstract Patterns - Creational Patterns

6.2.1 Factory Method

Factory Methods are fine but due to our hopeful lack of Singletons, usage of them should be limited.

6.2.2 Abstract Factory

Abstract Factories have quite ugly application and semblance, so let's keep it to data-instantiators for the ECS.

6.2.3 Builder

Builders are absolutely genuinely just banned, if you use a Builder in production code I will instantly ban you from existence. (I WILL KILL YOU)

(I'M NOT JOKING, DO NOT TEST ME)

6.2.4 Prototype

Prototypes are used for cloning objects without needing their class metadata, avoid their use.

6.2.5 Singleton

Using Singletons allows you to have a single overarching service, or some sort of public global access point to data, do not use Singletons.