PLAAY Coding Standard

TypeScript

Source file basics

Program structure

The module dependency must be acyclic. If module A depends, directly or indirectly, on module B, then module B must not depend, whether director or indirectly, on module A.

Module dependency should be documented in file "dependence.gv".

File names

Source file names should be the same as the module in the file.

One file per module. One module per file.

Source File Structure

Each source file consists of

- References for external libraries
- References for other modules in alphabetical order
- Imports for external libraries
- Imports for other modules in alphabetical order
- One module declaration
- The export assignment

Example

```
/// <reference path="jquery.d.ts" />
/// <reference path="jqueryui.d.ts" />

/// <reference path="assert.ts" />
/// <reference path="collections.ts" />
/// <reference path="pnode.ts" />
/// <reference path="pnodeEdits.ts" />
/// <reference path="sharedMkHtml.ts" />
/// <reference path="treeManager.ts" />


import assert = require('./assert') ;
import sharedMkHtml = require('./sharedMkHtml');
import collections = require( './collections' );
import pnode = require( './pnode');
import pnodeEdits = require( './pnodeEdits');
import treeManager = require( './treeManager');

/** The editor module ... */
module editor {
    ....
}

export = editor;
```

Use of reference directives and imports.

Although the references directives and imports seem redundant, as I understand it, we need them both. So when a new import is added (or removed) the corresponding reference directive should be added (or removed). Also the dependence should be documented in "dependence.gv" and possibly "dependence-js.gv" as mentioned above.

To give simple names for entities within imported modules, use import declarations within the module declaration like this.

```
....

/** The editor module ... */
module editor {
    import list = collections.list;
    import Option = collections.Option ;
    import some = collections.some ;
    import none = collections.none ;
    import snoc = collections.snoc;
    import Selection = pnodeEdits.Selection;
    ....
}

export = editor;
```

Formatting

Semicolons

Semicolons should be used even when they are optional.

Braces

Braces should be used even when they are optional. An exception is for short lambda expressions like `(x : int) => x+1`.

Where braces are placed is up to you. My preferred brace style is this

```
function redo() : void {
    if (redostack.length != 0) {
        undostack.push(currentSelection);
        currentSelection = redostack.pop();
        generateHTMLSoon();
    } else {
        beep() ;
    }
}
```

or even this

```
function redo() : void {
    if (redostack.length != 0) {
        undostack.push(currentSelection);
        currentSelection = redostack.pop();
        generateHTMLSoon(); }
    else {
          beep() ; } }
```

Line breaking

Lines shouldn't be longer than about 80 characters.

When breaking a statement over multiple lines, be careful of Typescript's rules for inserting semicolons.

Usually the best solution to long lines is to rewrite the code.

Usually the best way to break a long statement over multiple lines is to stack the arguments to a function. In this case it's all or nothing. Be sure to align the arguments. E.g.,

```
                // BAD Line is too long
                return opt.map( newNode =>
                    new Selection( newNode, cons( kTrg, newTrgSeln.path())
, newTrgSeln.anchor(), newTrgSeln.focus() ) ) ;

                // BAD. Mix of horizontal and vertical layout for args of
same call.
                return opt.map( newNode =>
                    new Selection( newNode, cons( kTrg, newTrgSeln.path())
,
                                   newTrgSeln.anchor(), newTrgSeln.focus()
 ) ) ;

                // GOOD
                return opt.map( newNode =>
                    new Selection( newNode,
                                   cons( kTrg, newTrgSeln.path()),
                                   newTrgSeln.anchor(),
                                   newTrgSeln.focus() ) ) ;
```

An exception is when the parameters are in logical groups.

```
    return doubleReplaceOnePathEmpty( node,
                                      srcStart, srcEnd, newNodes4Src,
                                      trgPath, trgStart, trgEnd, newNodes4
Trg,
                                      allowTrgAncestorOverwrite, true ) ;
```

Indentation

4 spaces

Variable declarations

One variable per declaration.

Use `const` or `let` for local variables. Never use `var`.

Prefer `const` to `let`.

Use `const` or `let` for module level variables. (I think `var` is the same as `let` at the modules level, but use `let` anyway.)

Naming

Use Camel Case

Use camel case. Avoid underscores, except as below.

An exception is enum members, which are all upper case with words separated by underscores.

Case of first letter

- modules (namespaces): lower case.
- classes, enums, type names: upper case
- type variable: upper case, usually one letter
- variables: lower case
- fields: either lower case or underscore
- functions: lower case
- methods: lower case

Method and Function names are usually verbs or verb phrases. Exceptions can be made for accessors and pure functions. For example `squareRoot` is better than `computeSquareRoot`.

Class declarations

The order of declarations is usually

- Fields
- Constructor
- Methods

Methods should either be accessors or mutators but not both.

Fields should usually be `private`. Where possible, fields should be declared `readonly`. (`readonly` indicates that the field will not change after construction.)

Methods should be `private` unless there is a reason for them to not be `private`.

Whether `private`, `protected`, or `public`, the accessibility level should be explicitly declared.

Object invariants should be carefully documented.

Consider checking object invariants after construction and after each mutator.

```
    /** A Selection indicates ...
    *
    * Invariant:
    *
    * * The path must identify a node under the root.
    *       I.e. the path can be empty or its first item must be
    *       the index of a child of the root and the rest of the path must
    *       identify a node under that child.
    * * The focus and anchor must both be integers greater or equal to 0 a
nd
    *     less or equal to the number of children of the node identified b
y the path.
    */
    export class Selection {

        private readonly _root : PNode ;
        private readonly _path : List<number> ;
        private readonly _anchor : number ;
        private readonly _focus : number ;

        constructor( root : PNode, path : List<number>,
                    anchor : number, focus : number ) {
            assert.checkPrecondition( checkSelection( root, path, anchor,
focus ),
                                      "Attempt to make a bad selection" )
;
            this._root = root;
            this._path = path;
            this._anchor = anchor ;
            this._focus = focus ; }

        public root() : PNode { return this._root ; }

        public path() : List<number> { return this._path ; }

        .
        .
        .
    }
```

Initialization

Local variables should almost always be initialized. Delay declaring a variable until it is ready to be initialized.

Types

Strict null and undefined checking

We compile with the `strictNullChecks` option. This means that `null` and `undefined` are not considered to be members of most types.

If a variable might hold `null`, this needs to be explicitly declared. For example

```
    let pendingAction : number|null = null ;
```

This sometimes means that a cast must be used where it wouldn't otherwise, in order that the compiler know that the variable is not `null`. For example `window.clearTimeout` takes a `number` and so we write.

```
    if( pendingAction !== null ) {
        window.clearTimeout( pendingAction as number ) ; }
```

Declare return types

All functions and methods should have a declared return type. This includes functions with a return type of void. For
example

```
    function redo() : void { ... } // GOOD

    function redo() { ... } // BAD
```

Typescript will infer the return type based on the return statements. So for very short functions, where the type is obvious it is ok to omit the result type. The following is acceptable

```
    applyEdit( a : A ) : Option<A> {
        let result : Option<A> = this._first.applyEdit( a ) ;
        return result.choose(
                    (a0 : A) => result,
                    () => this._second.applyEdit( a ) ) ;
    }
```

For the first lambda expression, there is no need to declare the type of the result since it is clearly the same as the type of the variable `result` and its type can be seen 2 lines earlier. For the second lambda expression, the reader may have to think a little more, but if they know about `choose`, they know that both argument should have the same result type.

Declare parameter types

Parameters should always have declared types.

Declare module-level variable types

Declare field types in classes

Optionally declare local variable types

The compiler can infer the types of local variables if they are initialized. For example in

```
function foo( j : number ) : number {
    let i = j ; // GOOD
    i = "hello" ;  // Error reported.
    return i ;
}
```

The compiler will infer the type of `i` from the type of `j` and so the compiler will detect the error in the assignment.

However for readability it is often a good idea to declare the type of the variable anyway.

```
function foo( j : number ) : number {
    let i : number = j ; // GOOD
    i = "hello" ;  // Error reported.
    return i ;
}
```

For uninitialized variables, the type should *always* be given. For example

```
function foo( j : number ) : number {
    let i ;  // BAD
    i = 1 ;
    i = "hello" ; // No error reported.
    return i ;  // No error reported.
}
```

In the code above, no error is reported at compile time. The variable `i` is inferred to have type `any`.

Lambda expressions

JavaScript and Typescript have two kinds of lambda expressions. A `function`-keyword-style lambda expression looks like this

```
function( param : ParamType ) : resultType { code }
```

A fat-arrow lambda expression looks like this

```
( param : ParamType ) : resultType => { code }
```

As an abbreviation, when the code part is of the form `return exp ;` we can write

```
( param : ParamType ) : resultType => exp
```

Use fat-arrow lambda expressions mostly

Prefer the fat-arrow form of lambda expressions to the `function`-keyword-style of lambda expressions.

The expressions `(x : int) : int => x+1` and `function( x : int) : int {return x+1}` have the same meaning. However the first is shorter.

In JavaScript (and thus also Typescript), whether to use the fat arrow or function keyword is often not a matter of style, but of correctness.

The treatment of the keyword `this` differs between the two syntaxes. Usually the way that the fat arrow does it is what you want. For example consider:

```
// GOOD
let x : boolean[] = this._children.map(
    (a:PNode) : boolean =>
        a.label() === this._label ) ;
```

Here the use of the keyword `this` within the lambda expression makes it necessary to use the fat arrow style of lambda expression.

Using the `function` keyword is incorrect here

```
// VERY VERY BAD
let x : boolean[] = this._children.map(
    function(a:PNode) : boolean {
        return a.label() === this._label ; } ) ;
```

When the code is executed, the two uses of "this" are not bound to the same object.

Since we compile with the "noImplicitThis" option turned on, this sort of mistake should almost always be caught by the compiler.

Use `function keyword`-style for JQuery and Mocha

For HTML or jQuery event handlers, the "this" object will be the HTML element. So here it is more appropriate to use the `function` keyword style.

When the code of the lambda expression mentions `this`, you should declare the type of `this` for the duration of the function, as illustrated here:

```
    const keyUpHandlerForInputs
        = function(this : HTMLElement, e : JQueryKeyEventObject ) : void {

            if (e.keyCode === 13) {
                console.log( ">>keyup handler") ;
                updateLabelHandler.call( this, e ) ;
                console.log( "<< keyup handler") ;
            } } ;
```

(Note: be careful what type you declare `this` to have. The typescript compiler will typically not check whether you've declared the appropriate type for this. For example in the above code, you can change `HTMLElement` to `number` and the compiler will still compile it. And it will also then compile `inputs.keyup(keyUpHandlerForInputs);` where `input` is a `JQuery` object. Even though, this is complete hogwash. So be careful.)

A similar case is in mocha tests. In Mocha, `this` is bound to the test context and so the `function`-keyword style is preferred.

If `this` is used within functions passed to Mocha, its type should be declared. For example:

```
describe( 'pnodeEdits.CopyEdit',
    function( this : Mocha.IContextDefinition) : void {
         this.timeout( 500 ) ;

        it( 'should copy a single node',
            function(this: Mocha.ITestDefinition) : void {
                this.timeOut(1000) ;
                if( thisTestNotApplicableForSomeReason ) {
                    this.skip() ; }
                else {
                    .... }
        } ) ;
} ) ;
```

Assertion checking

The check functions from the `assert.ts` module should be used as much as reasonable.

In particular, preconditions to functions should be checked using `assert.checkPrecondition( condition, message)`. If the precondition is known to have failed use `assert.failedPrecondition( message )`.

Class invariants should be checked at the end of each constructor and mutator using `assert.checkInvariant( condition, message)`.

Unreachable code should be marked by `assert.unreachable( message)`.

Places where code remains to be written can be marked by `assert.todo( message )`.

Other checks should use either `assert.check( condition, message)` or `assert.checkEqual( expected, found, message)`.

Use of `never`

The return type of `assert.unreachable` and `assert.failedPrecondition` is `never`, which indicates that they do not return.

Consider a function

```
function  squareRoot( n : number ) : number {
    if( n < 0 ) {
        assert.failedPrecondition( "negative argument to squareRoot" ) ;
    } else {
        ...
        return result ; }
}
```

Even though the return type of `assert.failedPrecondition` is `never`, the compiler seems to consider that this code is equivalent to

```
function  squareRoot( n : number ) : number {
    if( n < 0 ) {
        assert.failedPrecondition( "negative argument to squareRoot" ) ;
        return undefined ;
    } else {
        ...
        return result ; }
}
```

And since we use strict null checking, this is a type error, since `undefined` is not considered a `number` under strict null checking.

The solution is to write the function like this

```
function  squareRoot( n : number ) : number {
    if( n < 0 ) {
        return assert.failedPrecondition( "negative argument to squareRoot
" ) ;
```

```
    } else {
        ...
        return result ; }
}
```

The issue could have been avoided more elegantly like this

```
function  squareRoot( n : number ) : number {
    assert.checkPrecondition( n >= 0, "negative argument to squareRoot" )
;
    ...
    return result ; }
}
```

Comments

Comments should be used to document modules, classes, and nonprivate methods. We use `typedoc` to turn documentation into html, so comments should use the `typedoc` format, which is similar to `javadoc`. See http://typedoc.org/guides/doccomments/

Miscellaneous

Use `===` rather than `==` and `!==` rather than `!=`.

Use `as` for casting. I.e. `f as number` rather than `<number> f`.

Remember that casting is not checked at runtime. Thus it is important to be sure that any necessary checks are explicitly coded. For example

Consider this method

```
        public getPendingNode() : PNode {
            assert.checkPrecondition( !this.isDone() ) ;
            const p = this.pending as List<number> ;
            return this.root.get( p ) ;
        }
```

Here the type of `this.pending` is `null | List<number>`. However, if `!this.isDone()` is true, then `this.pending` can not be null. Therefore the case expression is safe.

The `any` type should be avoided.

Assignments and other side effects, should never occur with the guards of `if`, `while`, `for` commands. For example

```
    // BAD
    if( a = b ) ...

    // GOOD
    a = b ;
    if( a ) ...
```

Likewise avoid assignments and other side effects in arguments.

Avoid `throw` and `catch`.

Avoid use of `null` and `undefined` as much as possible.