# React Coding Standards

## Don’t use public accessor within classes

All members within the class are public by default (and always public in runtime, TS private/protected will "hide" particular class properties/methods only during compile time). Don't introduce extra churn to your codebase. Also using public accessor is not "valid/idiomatic javascript"

## Don’t use a private accessor within Component class

private accessor won't make your properties/methods on class private during runtime. It's just TypeScript "emulation during compile time". Don't get fooled and make things "private" by using well-known patterns like:

- name starting with underscore _someProp
- or if you really wanna make those properties private use Symbol for defining those. ( real runtime private properties are coming to ECMAscript )

## Don’t use protected accessor within Component class

Using protected is an immediate "RED ALERT" in terms of functional patterns leverage with React. There are more effective patterns like this for extending behavior of some components. You can use:

- just extract the logic to separate component and use it as seen above
- HoC (high order function) and functional composition.
- CaaF ( children as a function )

## Don’t use enum

- To use enum within TypeScript might be very tempting, especially if you're coming from languages like C# or Java. But there are better ways how to interpret both with well-known JS idiomatic patterns or as can be seen in "Better" example just by using compile-time type literals.
- Enums compiled output generates unnecessary boilerplate
- Non string Enums don’t narrow to proper number type literal, which can introduce unhandled bug within your app
- It’s not standard/idiomatic JavaScript (although enum is reserved word in ECMA standard)
- Cannot be used with babel for transpiling

## Don’t use the constructor for class Components

There is really no need to use a constructor within React Component. If you do so, you need to provide more code boilerplate and also need to call super with provided props ( if you forget to pass props to your super, your component will contain bugs as props will not be propagated correctly)

## Don’t use decorators for class Components

- You won’t be able to get the original/clean version of your class
- TypeScript uses an old version of the decorator proposal which isn’t gonna be implemented in ECMAscript standard
- It adds additional runtime code and processing time execution to your app
- What is most important though, in terms of type checking within JSX, is, that decorators don’t extend class type definition. That means (in our example), that our Container component, will have absolutely no type of information for consumer about added/removed props

## Use lookup types for accessing component State/Props types

- Exporting Props or State from your component implementation is making your API surface bigger.
- You should always ask a question, why consumers of your component should be able to import explicit State/Props type? If they really need that, they can always access it via type lookup functionality. So cleaner API but type information is still there for everyone.
- If you need to provide a complex Props type though, it should be extracted to models/types file exported as Public API.

## Always provide an explicit type for children Props

- children prop is annotated as optional within both Component and Functional Component in react.d.ts which just mirrors the implementation of how React handles children.
- While that's ok and everything, I prefer to be explicit with component API.
- if you plan to use children for content projection, make sure to explicitly annotate it with the type of your choosing and in the opposite, if your component doesn't use it, prevent its usage with never type.

## Use type inference for defining Component State or DefaultProps

- Less type boilerplate
- More readable code
- by adding read-only modifier and freezing the object, any mutation within your component will immediately end with a compile error, which will prevent any runtime error = happy consumers of your app!

## When using function factories instead of classes for models/entities, leverage declaration merging by exporting both type and implementation

- Less Boilerplate
- One token for both type and implementation / Smaller API
- Both type and implementation are in sync and most importantly, implementation is the source of truth

## Use default import to import React

- It’s confusing to import all contents from react library when you’re not using them.
- It’s more aligned to “idiomatic JS”
- You don’t need to import types defined on React namespace like you have to do with Flow as TS support declaration merging
- The “consider” example is even more explicit what is used within your module and may improve tree-shaking during compile time.

## Don’t use namespace

- namespace was kinda useful in pre ES2015 modules era. We don't need it anymore.
- Cannot be used with babel for transpiling

## Don’t use ES2015 module imports when importing types without any run-time code

- Your code is explicit for both humans and machines. If you don’t use any run-time code, annotate your code only via import('path')
- check this great post from David East to learn more

## Declare types before run-time implementation

- The first lines of the document clearly state what kind of types are used within the current module. Also, those types are compiled only code
- run-time and compile-time declarations are clearly separated
- in component user immediately knows what the component “API” looks like without scrolling

## Don’t use method declaration within interface/type alias

- --strictFunctionTypes enforces stronger type checks when comparing function types, but does not apply to methods. Check TS wiki to learn more
- explanation from TS issue
- consistency, all members of type/interface are defined via the same syntax

## Don’t use number for indexable type key

- In JavaScript object properties are always typeof string! don't create false type predicates within your apps!
- Annotating keys with number is OK for arrays (array definition from standard .d.ts lib), although in real life you should rarely come into a situation that you wanna define "custom" array implementation

## Don’t use JSX.Element to annotate function/component return type or children/props

- TypeScript supports locally scoped JSX to be able to support various JSX factory types and proper JSX type checking per factory. While current react types use still the global JSX
- namespace, it’s gonna change in the future.
- explicit types over-generalized ones

## Use type alias instead of interface for declaring Props/State

- consistency/clearness. Let’s say we use tip no.8 (defining state type from implementation). If you would like to use an interface with this pattern, you’re out of luck, as that’s not allowed within TypeScript
- interface cannot be extended by types created via union or intersection, so you would need to refactor your State/Props interface to type alias in that case.
- interfaces can be extended globally via declaration merging, if you wanna provide that kind of capabilities to your users you’re doing it wrong (exposing “private” API)

## Don’t use FunctionComponent<P>/FC<P> to define a function component

- consistency/simplicity (always prefer familiar vanilla JavaScript patterns without too much type noise/magic)
- FC defines optional children on props, which is not what your API may support as explained in tip no 8. API should be explicit!
- FC breaks defaultProps type resolution (introduced in TS 3.1) and unfortunately all other "static" props as well

Check here for the full specification.