Angular Framework Coding Standards

The coding practices in this doc apply only to development on Angular itself, not applications built *with* Angular. (Though you can follow them too if you really want).

Code style

The Google JavaScript Style Guide is the basis for Angular's coding style, with additional guidance here pertaining to TypeScript. The team uses `clang-format` to automatically format code; automatic formatting is enforced by CI.

Code practices

Write useful comments

Comments that explain what some block of code does are nice; they can tell you something in less time than it would take to follow through the code itself.

Comments that explain why some block of code exists at all, or does something the way it does, are *invaluable*. The "why" is difficult, or sometimes impossible, to track down without seeking out the original author. When collaborators are in the same room, this hurts productivity. When collaborators are in different timezones, this can be devastating to productivity.

For example, this is a not-very-useful comment:

```
// Set default tabindex.
if (!attributes['tabindex']) {
  element.setAttribute('tabindex', '-1');
}
```

While this is much more useful:

```
// Unless the user specifies otherwise, the calendar should not be a tab stop.
// This prevents ngAria from overzealously adding a tabindex to anything with an ng-model.
if (!attributes['tabindex']) {
  element.setAttribute('tabindex', '-1');
}
```

In TypeScript code, use JsDoc-style comments for descriptions (on classes, members, etc.) and use `//` style comments for everything else (explanations, background info, etc.).

API Design

Boolean arguments Generally avoid adding boolean arguments to a method in cases where that argument means "do something extra". In these cases, prefer breaking the behavior up into different functions.

```
// AVOID
function getTargetElement(createIfNotFound = false) {
  // ...
}

// PREFER
function getExistingTargetElement() {
  // ...
}

function createTargetElement() {
  // ...
}
```

You can ignore this guidance when necessary for performance reasons in framework code.

Optional arguments Use optional function arguments only when such an argument makes sense for an API or when required for performance. Don't use optional arguments merely for convenience in implementation.

TypeScript

Typing Avoid `any` where possible. If you find yourself using `any`, consider whether a generic or `unknown` may be appropriate in your case.

Getters and Setters Getters and setters introduce openings for side-effects, add more complexity for code readers, and generate additional code when targeting older browsers.

- Only use getters and setters for `@Input` properties or when otherwise required for API compatibility.
- Avoid long or complex getters and setters. If the logic of an accessor would take more than three lines, introduce a new method to contain the logic.
- A getter should immediately precede its corresponding setter.
- Decorators such as `@Input` should be applied to the getter and not the setter.
- Always use a `readonly` property instead of a getter (with no setter) when possible.

  ```
  /** YES */
  readonly active: boolean;

  /** NO */
  get active(): boolean {
    // Using a getter solely to make the property read-only.
  ```

```
    return this._active;
  }
```

Iteration Prefer `for` or `for of` to `Array.prototype.forEach`. The `forEach` API makes debugging harder and may increase overhead from unnecessary function invocations (though modern browsers do usually optimize this well).

JsDoc comments All public APIs must have user-facing comments. These are extracted for API documentation and shown in IDEs.

Private and internal APIs should have JsDoc when they are not obvious. Ultimately it is the purview of the code reviewer as to what is "obvious", but the rule of thumb is that *most* classes, properties, and methods should have a JsDoc description.

Properties should have a concise description of what the property means:

```
/** The label position relative to the checkbox. Defaults to 'after' */
@Input() labelPosition: 'before' | 'after' = 'after';
```

Methods blocks should describe what the function does and provide a description for each parameter and the return value:

```
/**
 * Opens a modal dialog containing the given component.
 * @param component Type of the component to load into the dialog.
 * @param config Dialog configuration options.
 * @returns Reference to the newly-opened dialog.
 */
open<T>(component: ComponentType<T>, config?: MatDialogConfig): MatDialogRef<T> { ... }
```

Boolean properties and return values should use "Whether…" as opposed to "True if…":

```
/** Whether the button is disabled. */
disabled: boolean = false;
```

Try-Catch Only use `try-catch` blocks when dealing with legitimately unexpected errors. Don't use `try` to avoid checking for expected error conditions such as null dereference or out-of-bound array access.

Each `try-catch` block **must** include a comment that explains the specific error being caught and why it cannot be prevented.

Variable declarations Prefer `const` wherever possible, only using `let` when a value must change. Avoid `var` unless absolutely necessary.

`readonly` Use `readonly` members wherever possible.

Naming

General

- Prefer writing out words instead of using abbreviations.
- Prefer *exact* names over short names (within reason). For example, `labelPosition` is better than `align` because the former much more exactly communicates what the property means.
- Except for `@Input()` properties, use `is` and `has` prefixes for boolean properties / methods.

Observables Don't suffix observables with `$`.

Classes Name classes based on their responsibility. Names should capture what the code *does*, not how it is used:

```
/** NO: */
class DefaultRouteReuseStrategy { }

/** YES: */
class NonStoringRouteReuseStrategy { }
```

Methods The name of a method should capture the action performed *by* that method rather than describing when the method will be called. For example:

```
/** AVOID: does not describe what the function does. */
handleClick() {
  // ...
}

/** PREFER: describes the action performed by the function. */
activateRipple() {
  // ...
}
```

Test classes and examples Give test classes and examples meaningful, descriptive names.

```
/** PREFER: describes the scenario under test. */
class FormGroupWithCheckboxAndRadios { /* ... */ }
class InputWithNgModel { /* ... */ }


/** AVOID: does not fully describe the scenario under test. */
class Comp { /* ... */ }
class InputComp { /* ... */ }
```

RxJS When importing the `of` function to create an `Observable` from a value, alias the imported function as `observableOf`.

```
import {of as observableOf} from 'rxjs';
```

Testing

Test names Use descriptive names for jasmine tests. Ideally, test names should read as a sentence, often of the form "it should...".

```
/** PREFER: describes the scenario under test. */
describe('Router', () => {
  describe('with the default route reuse strategy', () => {
    it('should not reuse routes upon location change', () => {
      // ...
    });
  })
});

/** AVOID: does not fully describe the scenario under test. */
describe('Router', () => {
  describe('default strategy', () => {
    it('should work', () => {
      // ...
    });
  })
});
```