AOT metadata errors

The following are metadata errors you may encounter, with explanations and suggested corrections.

Expression form not supported Reference to a local (non-exported) symbol Only initialized variables and constants Reference to a non-exported class Reference to a non-exported function Function calls are not supported Destructured variable or constant not supported Could not resolve type Name expected Unsupported enum member name Tagged template expressions are not supported Symbol reference expected

{@a expression-form-not-supported} ## Expression form not supported

The compiler encountered an expression it didn't understand while evaluating Angular metadata.

Language features outside of the compiler's restricted expression syntax can produce this error, as seen in the following example:

```
// ERROR
export class Fooish { ... }
...
const prop = typeof Fooish; // typeof is not valid in metadata
  ...
  // bracket notation is not valid in metadata
  { provide: 'token', useValue: { [prop]: 'value' } };
  ...
```

You can use `typeof` and bracket notation in normal application code. You just can't use those features within expressions that define Angular metadata.

Avoid this error by sticking to the compiler's restricted expression syntax when writing Angular metadata and be wary of new or unusual TypeScript features.

{@a reference-to-a-local-symbol} ## Reference to a local (non-exported) symbol

Reference to a local (non-exported) symbol 'symbol name'. Consider exporting the symbol.

The compiler encountered a referenced to a locally defined symbol that either wasn't exported or wasn't initialized.

Here's a `provider` example of the problem.

```
// ERROR
let foo: number; // neither exported nor initialized

@Component({
  selector: 'my-component',
  template: ... ,
  providers: [
```

```
    { provide: Foo, useValue: foo }
  ]
})
export class MyComponent {}
```

The compiler generates the component factory, which includes the `useValue` provider code, in a separate module. *That* factory module can't reach back to *this* source module to access the local (non-exported) `foo` variable.

You could fix the problem by initializing `foo`.

```
let foo = 42; // initialized
```

The compiler will fold the expression into the provider as if you had written this.

```
  providers: [
    { provide: Foo, useValue: 42 }
  ]
```

Alternatively, you can fix it by exporting `foo` with the expectation that `foo` will be assigned at runtime when you actually know its value.

```
// CORRECTED
export let foo: number; // exported

@Component({
  selector: 'my-component',
  template: ... ,
  providers: [
    { provide: Foo, useValue: foo }
  ]
})
export class MyComponent {}
```

Adding `export` often works for variables referenced in metadata such as `providers` and `animations` because the compiler can generate *references* to the exported variables in these expressions. It doesn't need the *values* of those variables.

Adding `export` doesn't work when the compiler needs the *actual value* in order to generate code. For example, it doesn't work for the `template` property.

```
// ERROR
export let someTemplate: string; // exported but not initialized

@Component({
  selector: 'my-component',
  template: someTemplate
})
export class MyComponent {}
```

The compiler needs the value of the `template` property *right now* to generate the component factory. The variable reference alone is insufficient. Prefixing the declaration with `export` merely produces a new error, "`Only initialized variables and constants can be referenced`".

{@a only-initialized-variables} ## Only initialized variables and constants

Only initialized variables and constants can be referenced because the value of this variable is needed by the template compiler.

The compiler found a reference to an exported variable or static field that wasn't initialized. It needs the value of that variable to generate code.

The following example tries to set the component's `template` property to the value of the exported `someTemplate` variable which is declared but *unassigned*.

```
// ERROR
export let someTemplate: string;

@Component({
  selector: 'my-component',
  template: someTemplate
})
export class MyComponent {}
```

You'd also get this error if you imported `someTemplate` from some other module and neglected to initialize it there.

```
// ERROR - not initialized there either
import { someTemplate } from './config';

@Component({
  selector: 'my-component',
  template: someTemplate
})
export class MyComponent {}
```

The compiler cannot wait until runtime to get the template information. It must statically derive the value of the `someTemplate` variable from the source code so that it can generate the component factory, which includes instructions for building the element based on the template.

To correct this error, provide the initial value of the variable in an initializer clause *on the same line*.

```
// CORRECTED
export let someTemplate = '<h1>Greetings from Angular</h1>';

@Component({
  selector: 'my-component',
  template: someTemplate
```

```
})
export class MyComponent {}
```

{@a reference-to-a-non-exported-class} ## Reference to a non-exported class

Reference to a non-exported class . Consider exporting the class.

Metadata referenced a class that wasn't exported.

For example, you may have defined a class and used it as an injection token in a providers array but neglected to export that class.

```
// ERROR
abstract class MyStrategy { }

  ...
  providers: [
    { provide: MyStrategy, useValue: ... }
  ]
  ...
```

Angular generates a class factory in a separate module and that factory can only access exported classes. To correct this error, export the referenced class.

```
// CORRECTED
export abstract class MyStrategy { }

  ...
  providers: [
    { provide: MyStrategy, useValue: ... }
  ]
  ...
```

{@a reference-to-a-non-exported-function} ## Reference to a non-exported function

Metadata referenced a function that wasn't exported.

For example, you may have set a providers `useFactory` property to a locally defined function that you neglected to export.

```
// ERROR
function myStrategy() { ... }

  ...
  providers: [
    { provide: MyStrategy, useFactory: myStrategy }
  ]
  ...
```

Angular generates a class factory in a separate module and that factory can only access exported functions. To correct this error, export the function.

```
// CORRECTED
export function myStrategy() { ... }

  ...
  providers: [
    { provide: MyStrategy, useFactory: myStrategy }
  ]
  ...
```

{@a function-calls-not-supported} ## Function calls are not supported

Function calls are not supported. Consider replacing the function or lambda with a reference to an exported function.

The compiler does not currently support function expressions or lambda functions. For example, you cannot set a provider's `useFactory` to an anonymous function or arrow function like this.

```
// ERROR
  ...
  providers: [
    { provide: MyStrategy, useFactory: function() { ... } },
    { provide: OtherStrategy, useFactory: () => { ... } }
  ]
  ...
```

You also get this error if you call a function or method in a provider's `useValue`.

```
// ERROR
import { calculateValue } from './utilities';

  ...
  providers: [
    { provide: SomeValue, useValue: calculateValue() }
  ]
  ...
```

To correct this error, export a function from the module and refer to the function in a `useFactory` provider instead.

```
// CORRECTED
import { calculateValue } from './utilities';

export function myStrategy() { ... }
export function otherStrategy() { ... }
export function someValueFactory() {
  return calculateValue();
}
  ...
  providers: [
```

```
    { provide: MyStrategy, useFactory: myStrategy },
    { provide: OtherStrategy, useFactory: otherStrategy },
    { provide: SomeValue, useFactory: someValueFactory }
  ]
  ...
```

{@a destructured-variable-not-supported} ## Destructured variable or constant not supported

Referencing an exported destructured variable or constant is not supported by the template compiler. Consider simplifying this to avoid destructuring.

The compiler does not support references to variables assigned by destructuring.

For example, you cannot write something like this:

```
// ERROR
import { configuration } from './configuration';

// destructured assignment to foo and bar
const {foo, bar} = configuration;
  ...
  providers: [
    {provide: Foo, useValue: foo},
    {provide: Bar, useValue: bar},
  ]
  ...
```

To correct this error, refer to non-destructured values.

```
// CORRECTED
import { configuration } from './configuration';
  ...
  providers: [
    {provide: Foo, useValue: configuration.foo},
    {provide: Bar, useValue: configuration.bar},
  ]
  ...
```

{@a could-not-resolve-type} ## Could not resolve type

The compiler encountered a type and can't determine which module exports that type.

This can happen if you refer to an ambient type. For example, the `Window` type is an ambient type declared in the global `.d.ts` file.

You'll get an error if you reference it in the component constructor, which the compiler must statically analyze.

```
// ERROR
@Component({ })
```

```
export class MyComponent {
  constructor (private win: Window) { ... }
}
```

TypeScript understands ambient types so you don't import them. The Angular compiler does not understand a type that you neglect to export or import.

In this case, the compiler doesn't understand how to inject something with the `Window` token.

Do not refer to ambient types in metadata expressions.

If you must inject an instance of an ambient type, you can finesse the problem in four steps:

1. Create an injection token for an instance of the ambient type.
2. Create a factory function that returns that instance.
3. Add a `useFactory` provider with that factory function.
4. Use `@Inject` to inject the instance.

Here's an illustrative example.

```
// CORRECTED
import { Inject } from '@angular/core';

export const WINDOW = new InjectionToken('Window');
export function _window() { return window; }

@Component({
  ...
  providers: [
    { provide: WINDOW, useFactory: _window }
  ]
})
export class MyComponent {
  constructor (@Inject(WINDOW) private win: Window) { ... }
}
```

The `Window` type in the constructor is no longer a problem for the compiler because it uses the `@Inject(WINDOW)` to generate the injection code.

Angular does something similar with the `DOCUMENT` token so you can inject the browser's `document` object (or an abstraction of it, depending upon the platform in which the application runs).

```
import { Inject }   from '@angular/core';
import { DOCUMENT } from '@angular/common';

@Component({ ... })
export class MyComponent {
```

```
  constructor (@Inject(DOCUMENT) private doc: Document) { ... }
}
```

{@a name-expected} ## Name expected

The compiler expected a name in an expression it was evaluating.

This can happen if you use a number as a property name as in the following example.

```
// ERROR
provider: [{ provide: Foo, useValue: { 0: 'test' } }]
```

Change the name of the property to something non-numeric.

```
// CORRECTED
provider: [{ provide: Foo, useValue: { '0': 'test' } }]
```

{@a unsupported-enum-member-name} ## Unsupported enum member name

Angular couldn't determine the value of the enum member that you referenced in metadata.

The compiler can understand simple enum values but not complex values such as those derived from computed properties.

```
// ERROR
enum Colors {
  Red = 1,
  White,
  Blue = "Blue".length // computed
}


  ...
  providers: [
    { provide: BaseColor,   useValue: Colors.White } // ok
    { provide: DangerColor, useValue: Colors.Red }   // ok
    { provide: StrongColor, useValue: Colors.Blue }  // bad
  ]
  ...
```

Avoid referring to enums with complicated initializers or computed properties.

{@a tagged-template-expressions-not-supported} ## Tagged template expressions are not supported

Tagged template expressions are not supported in metadata.

The compiler encountered a JavaScript ES2015 tagged template expression such as the following.

```
// ERROR
const expression = 'funky';
```

```
const raw = String.raw`A tagged template ${expression} string`;
 ...
 template: '<div>' + raw + '</div>'
 ...
```

`String.raw()` is a *tag function* native to JavaScript ES2015.

The AOT compiler does not support tagged template expressions; avoid them in metadata expressions.

{@a symbol-reference-expected} ## Symbol reference expected

The compiler expected a reference to a symbol at the location specified in the error message.

This error can occur if you use an expression in the `extends` clause of a class.