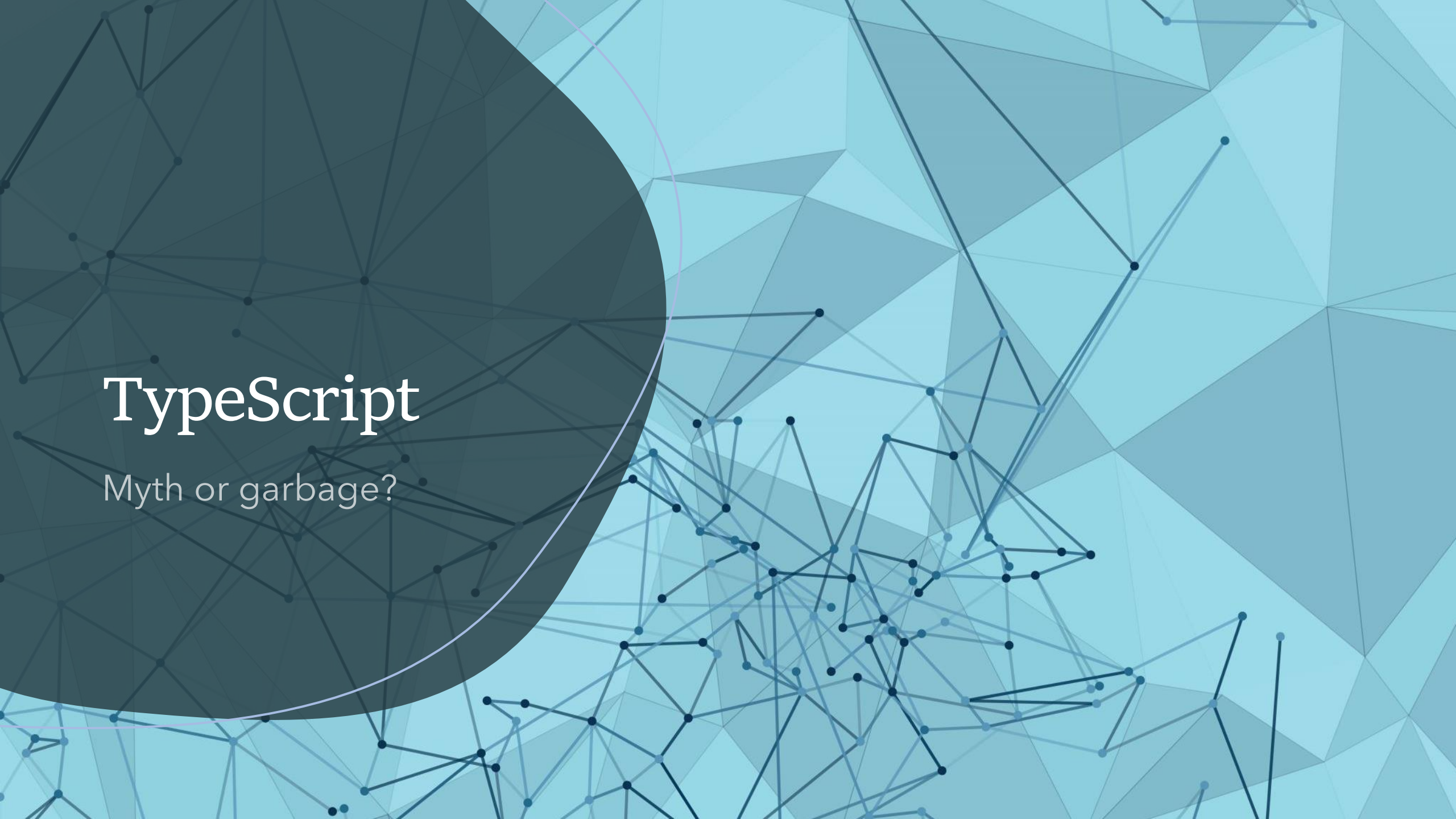
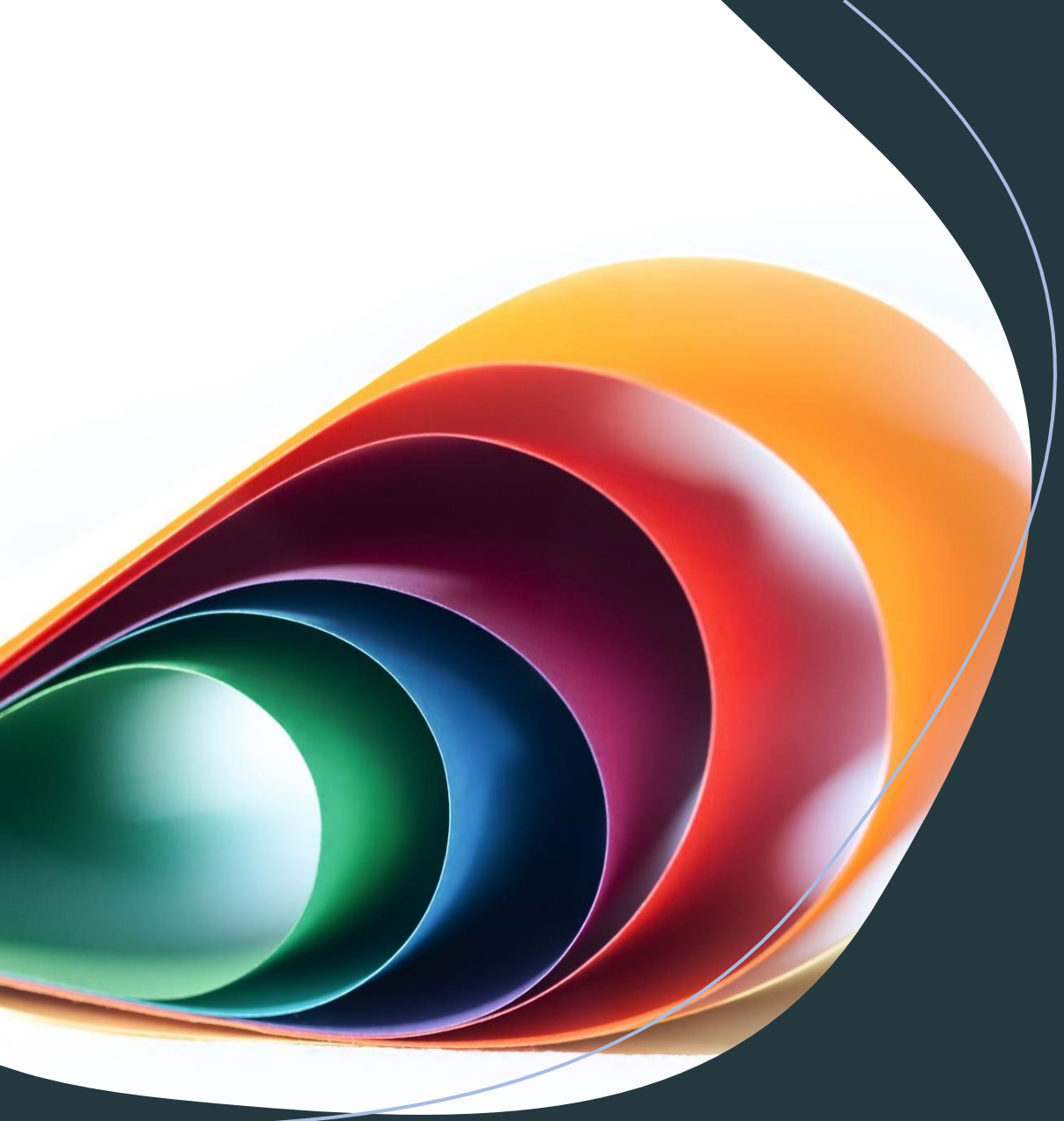

TypeScript
Myth or garbage?



JavaScript

- Interpreted
- Multi-paradigm
  - Event-driven
  - Functional
  - Imperative
- Dynamically typed
- Object oriented, based on prototypes
- Meant to be simple and fast

TypeScript

- Strongly typed
  - Static code analysis
  - Better tooling
  - Type safety
- Compiled
  - Transpiles to JavaScript
  - Provides compile-time errors
- Superset of JavaScript
  - Any JavaScript program is valid TypeScript
  - Not the other way around

Why choose TypeScript over JavaScript

- Large projects involving many teams and developers
- Complex data modeling
- Third party library usage
- Personal taste

Variables declaration

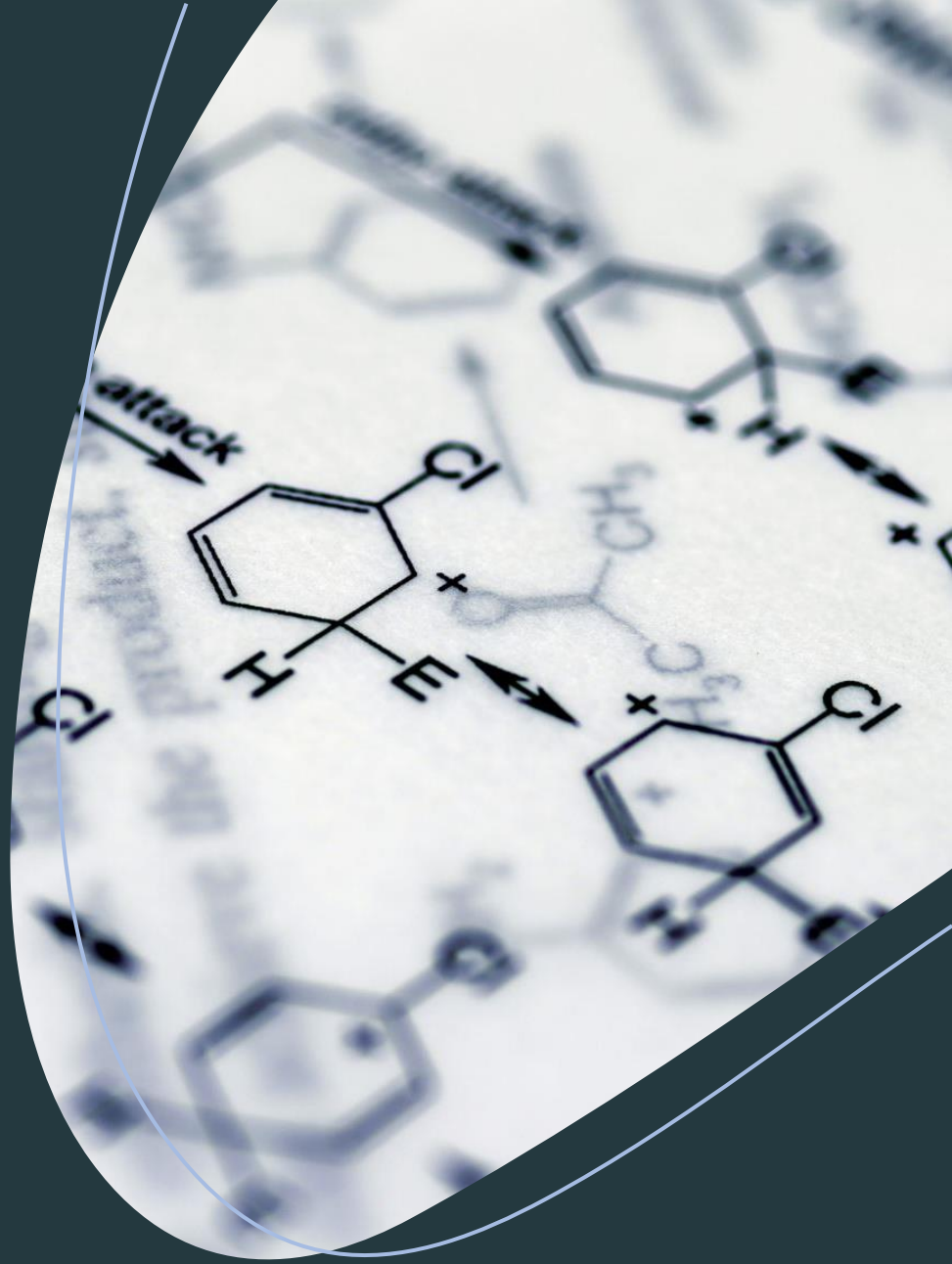
- Variables not meant to be reassigned
  - const
- Variables that will be reassigned
  - let
- Unscoped variables (that can be reassigned)
  - var
  - Don’t use this, as it’s meant for backwards compatibility

Some primitive types

Number	String	Boolean
Object	Array	Null and Undefined

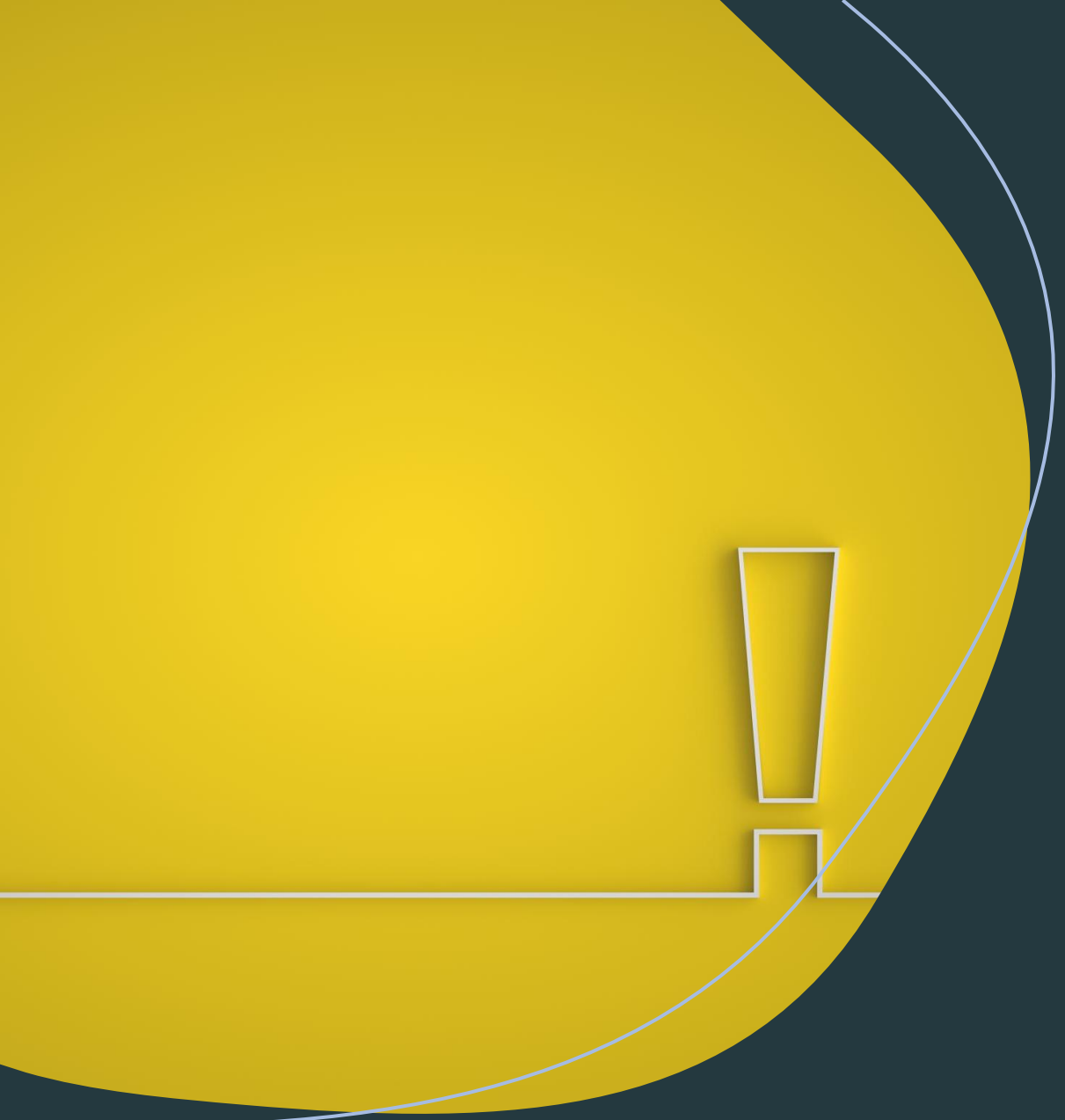
Type inference

- Variable types are inferred from the context
- The compiler always tries its best to infer variable types
- Type annotations should be omitted whenever possible
  - Example: variable initialization
  - Counter example: function arguments



Arrays and type inference

- Array item type is inferred by the compiler
- TypeScript supports mixed type arrays
  - Might result in union type inference (more on this later)
- Autocompletion is provided for commonly used JavaScript array methods (e.g. map, reduce, etc.)



Null and undefined

- Both are «falsy»
- Undefined
  - No value has yet been assigned to the variable
  - It’s usually unintentional
- Null
  - The variable has no value
  - Has to be set intentionally

Handling null-related errors

- Optional chaining
  - Does not replace proper error handling
- Null-coalescing operator
  - A simple way to provide a default value when assignment would give null
- Non null assertion
  - Use carefully, as it's one of those operators that «break» type inference
- Fun with flags: --strictNullChecks

Union types

- A variable that can be a type OR another is best described with a union type
  - string | number
- Nullable variables are inherently union types between the main type and null (or undefined)
  - number | null | undefined
- Mixed type arrays are inferred as union type arrays
  - (number | string)[]

Intersection types

- They are the opposite of union types, as they are supposed to contain all properties from source types

Tuples

- Can be thought as a record in a database, i.e. a group of values in the right order
- They use an array-like syntax and actually compile to plain JavaScript arrays
  - It’s just syntax sugar, but also provides type safety

The «any» type

- Can be thought of as the «subtype» of all types
- It can be assigned to anything
- No, really, don’t use it
- Fun with flags: --noImplicitAny

The «unknown» type

- «any» can be assigned to anything
- «unknown» type can be only assigned to
  - «unknown»
  - «any»
- «unknown» becomes assignable if «typeof» checks are performed in advance

Implementing type guards

- If a value is somehow inferred as unknown, type guards provide a type safe approach to use it as their intended type
- Type guards are simply functions accepting unknown arguments with a special return type annotation
  - function(value: unknown): value is IntendedType {…}

The «never» type

- Nothing can be assigned to «never»
- This special type is used by the type inference system to describe situations that can never happen
  - Example: a string union type and an if/else chain covering all cases. The final else can never happen.

Literal types

Type inference always narrows down as much as possible

«Hello world» is a string, but a string is not «Hello world»

In this example, «Hello world» is the literal type

const assignment usually infers literal types

let assignment cannot infer literal types, as the content of the variable can change: narrowing is not possible

Enum types

- Types used to provide mnemonic (and readable) representations of values
  - PlaybackStatus.Playing
- They compile to plain JavaScript objects

Type aliases

- Developers can assign custom names to types: this is called type aliasing
- Aliasing works on any type definition (union types, object structure, etc.)
- Type aliases are meant to be reused

Interfaces

- Best suited to define custom object types
- Support property modifiers
  - Nullable
  - Readonly
- Open for extension

Type assertion

- When the compiler is not able to narrow down enough, developers can do it explicitly
- This should be done carefully, as it could cause runtime errors
- Two possible syntax sugars
  - «as» keyword
  - <Angular brackets> syntax, previously known as casting
- Type assertion can only be used to narrow down to a more specific type
  - OK: HTMLElement as HTMLInputElement
  - Not OK: number as string (although this can be overcome by first casting to any)

TypeScript functions

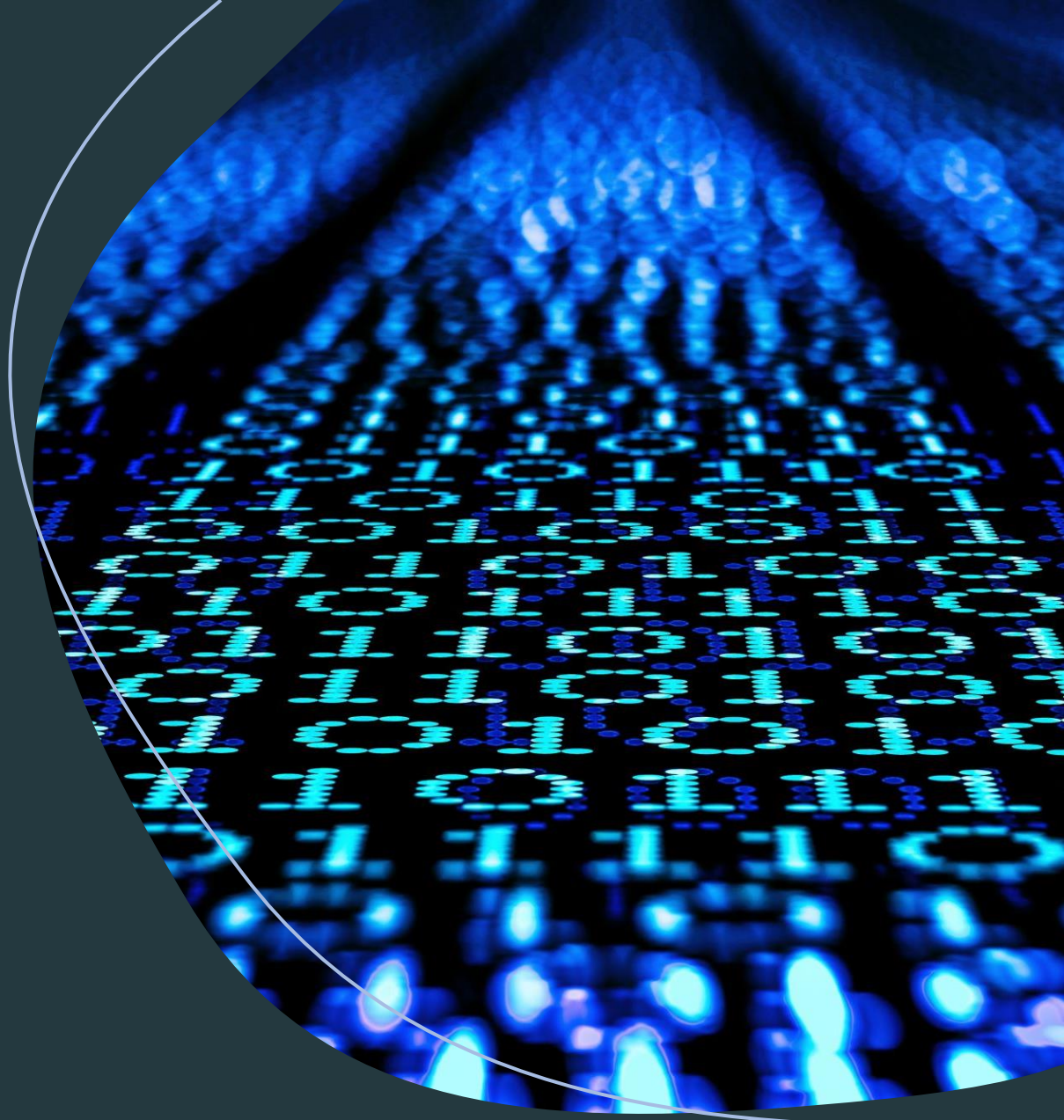
- Normal functions vs arrow functions
  - Arrow functions preserve the «this» context
  - Example: saveCourse function
- Functions support default argument values
  - Arguments with default values can be in any position

Functions in depth

- Function return types are inferred from the implementation
  - Type annotation can help building consistent return values
- As in JavaScript, functions are first-class citizens
  - Functions can be treated and passed as any other value type
- Function types can have type aliases
  - Not all arguments need to be passed when specifying a function type

TypeScript generics

- Generics help writing reusable and maintainable code by
  - Decluttering and deduplicating code that only depends on types
  - Helping the type inference system while writing clean code
- Example: Array<T>, Promise<T>

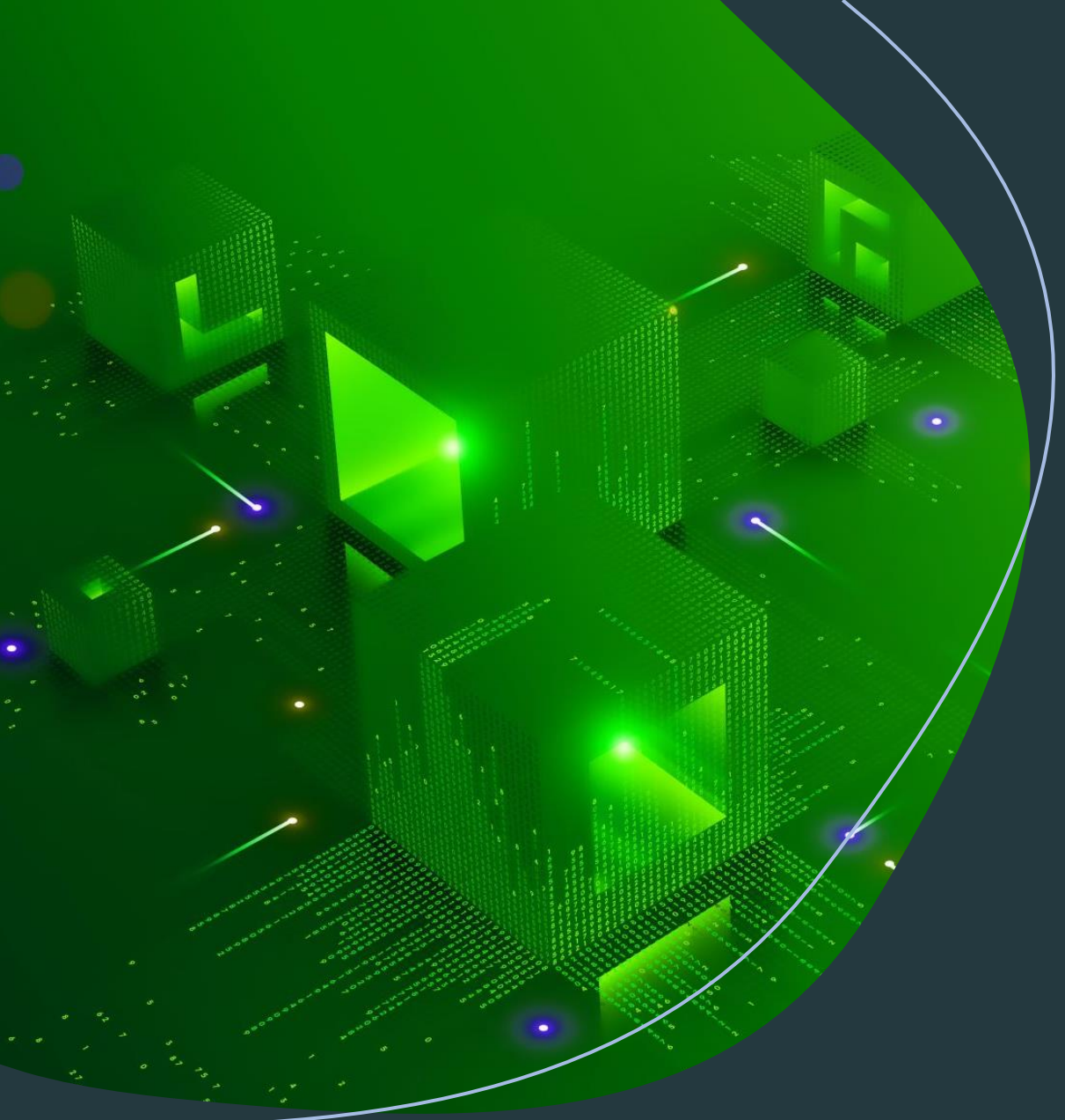


TypeScript generics

- Utility types
  - Partial<T> builds a type with the properties of T set to optional
  - Readonly<T> builds a type with the properties of T set to readonly
  - More in the TypeScript handbook
- Generic functions
- Generic classes
- Generics leveraging the «keyof» operator
- And all the utility you can think of

TypeScript generics

- Type constraints can be stated when declaring symbols with generic type arguments
  - E.g.: function someFun<T extends object>()
- A powerful feature for stating that a generic type implements an interface



TypeScript modules

- Each file is considerer an isolated module
- Nothing is exposed to the outside world which is not exported
- Modules talk to each other by importing symbols
- Anything can be exported (constants, types, etc.)
- Commonly used pattern: barrel import

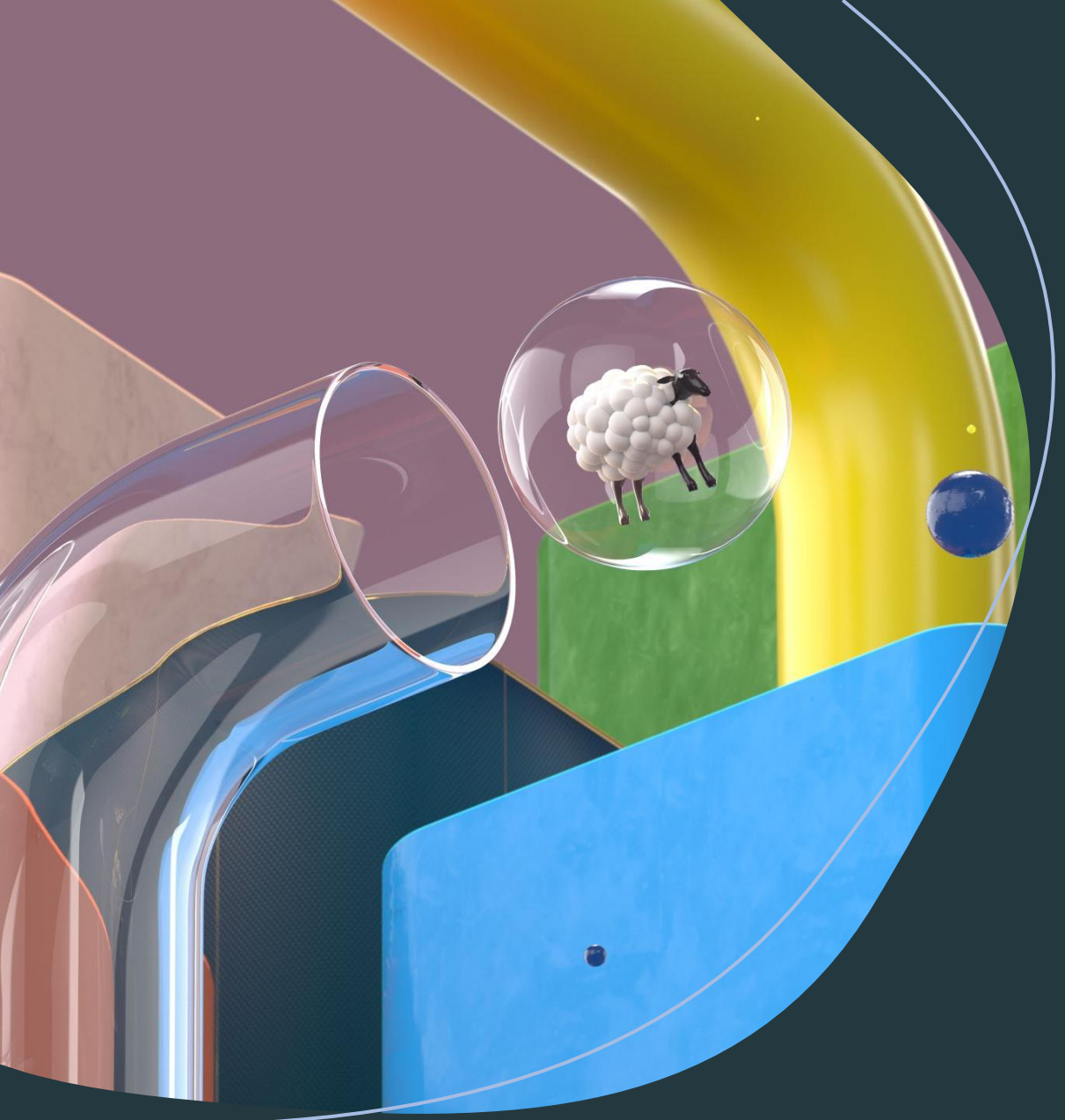
TypeScript modules – patterns

Barrel import/export

- A convenient way to segregate application features
- The most used pattern to make feature-specific functionalities available to other application modules

Default export

- Rarely used
- Allows to declare a default export symbol for a file

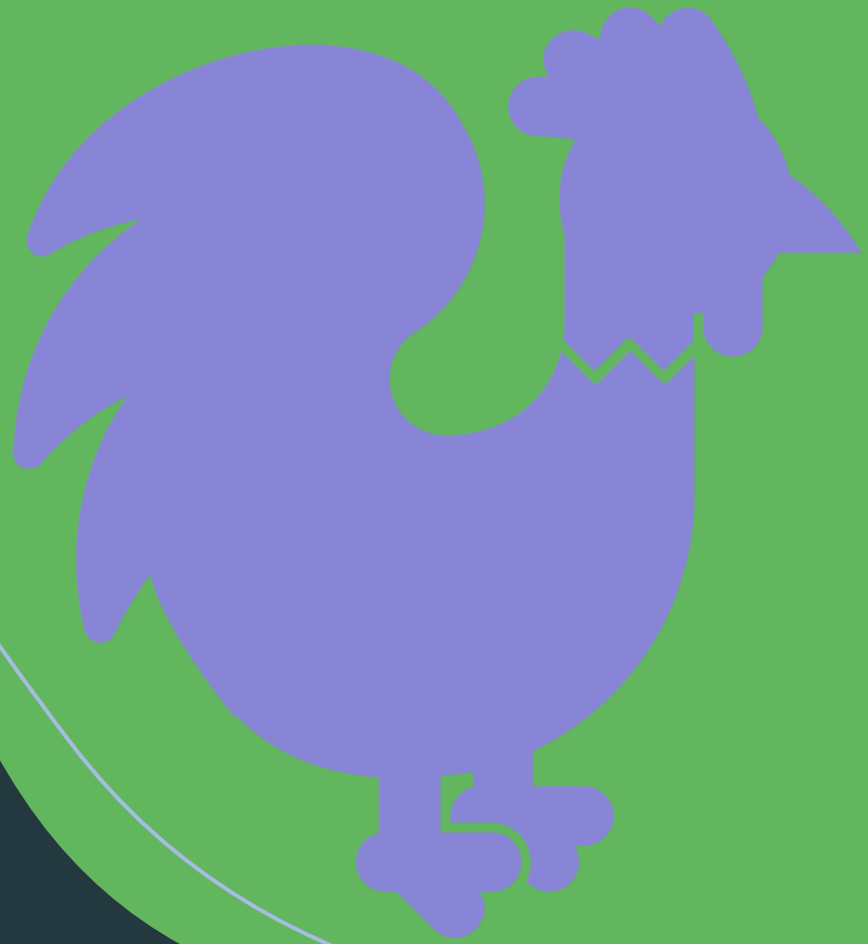


Objects/arrays in depth

- Spread operator
  - Useful to create shallow copies of objects and arrays
  - Enables rest arguments in functions
- Destructuring assignment
  - Makes code that tediously extracts object properties more readable
  - Works similarly for array, but with indices instead of properties
- Shorthand notation (objects only)

Object-oriented programming – classes

- TypeScript (and JavaScript) classes are just syntax sugar for function contexts: JavaScript does not have proper classes
  - The «new» keyword just returns the context of the called function
- Class programming (properties , inheritance, etc.) itself is just syntax sugar built around the manipulation of prototypes
- Classes are treated as types for the TypeScript type system

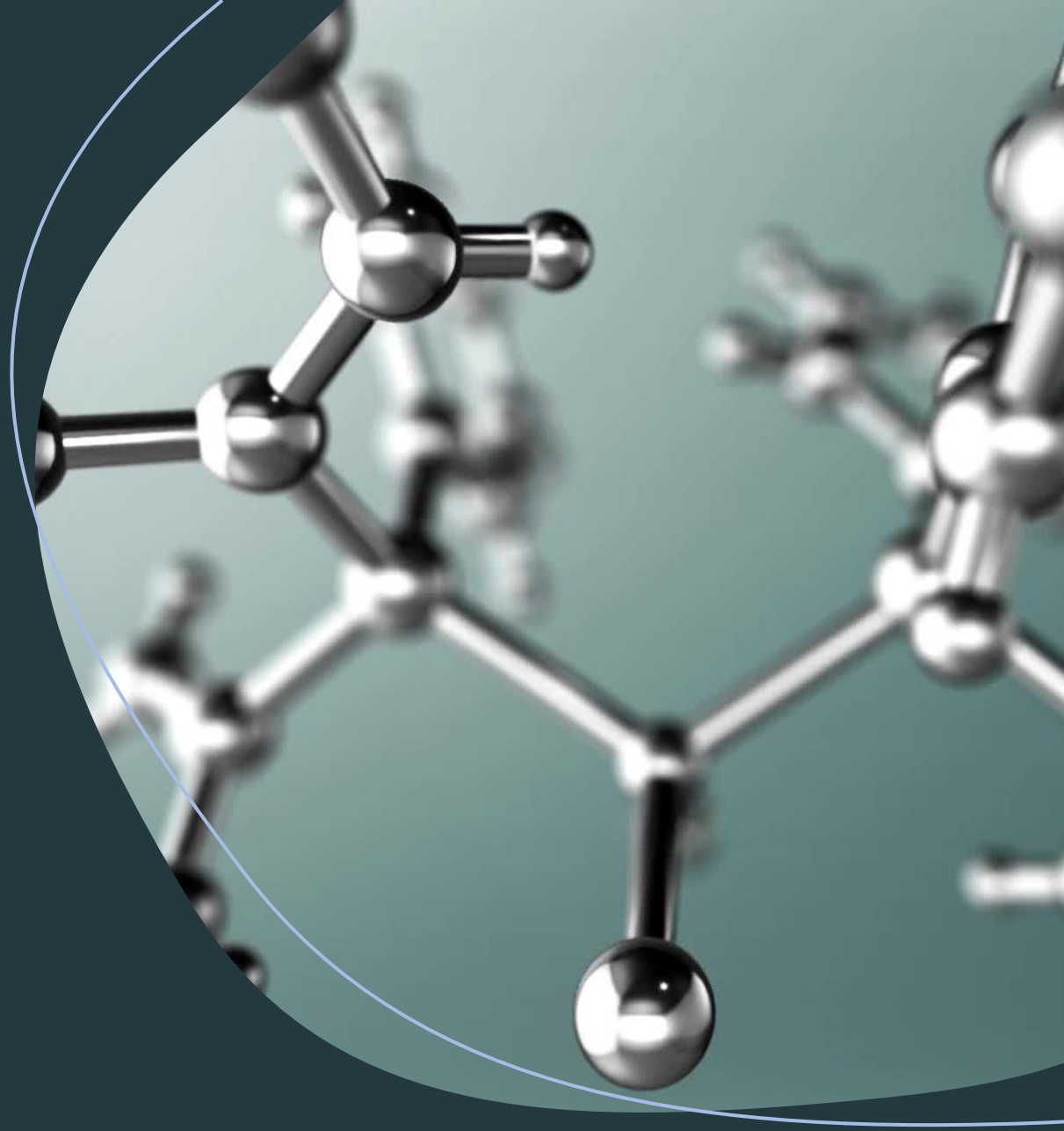


Classes in depth (basics)

- Member variables are mutable and publicly visible by default
  - So beware: encapsulation must be explicitly enforced
- Get/Set properties are supported
  - And it is generally a good practice to use them
- Constructor overloads are not supported
  - But default parameters definitely are
- Access modifiers can be applied to constructor parameters
  - If so, such parameters are also assumed to be member variables
- Static properties and methods are supported
  - Use them to share data among all instances or to provide common logic

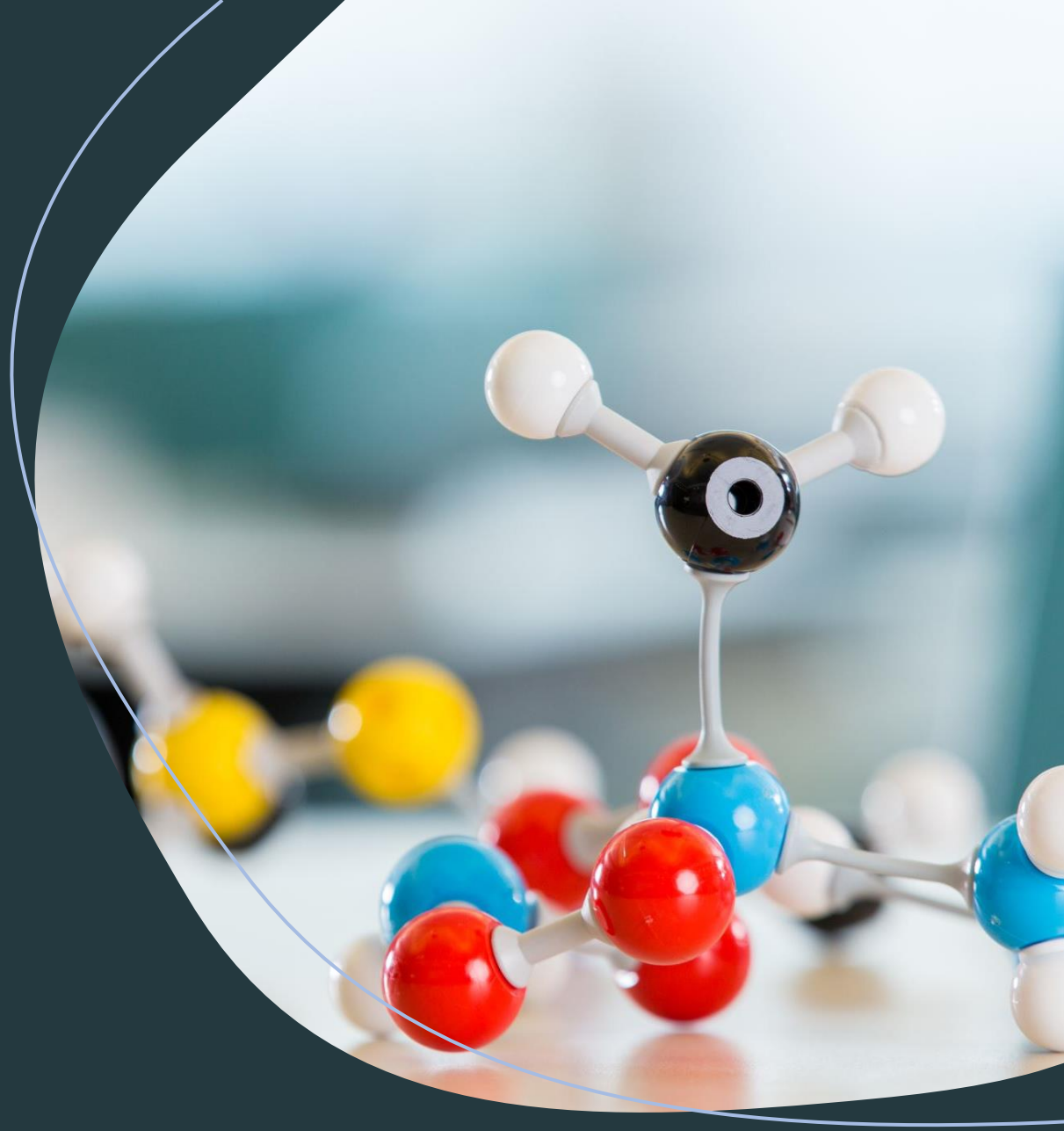
Classes in depth (inheritance)

- Class inheritance is allowed as in most languages implementing object-oriented programming and supports
  - Overrides
  - Superclass reference
  - Protected members



Classes in depth (abstract classes and interfaces)

- Abstract classes
  - Are actual classes (can contain logic, declare members, etc.)
  - Cannot be instantiated (must be inherited)
- Interfaces
  - Are not classes, but can be used to ensure a class implements them
  - Classes can implement multiple interfaces



Aspect-oriented programming – decorators

- Add functionality to symbols without leveraging inheritance
  - Class decorators
  - Method decorators
  - Property decorators

Method decorators

- Are simply functions that wrap other functions
- Can leverage advanced meta-programming features that build on native JavaScript prototypes
- Can execute logic before and after calling the original function, and might even not call it at all
  - Example: the Log decorator
- Are composable
  - Decorators encapsulate each other, starting from the innermost (the one declared just before the actual method)
  - Example: the Perf decorator
- Can be also defined at class level
  - Remember: classes do not exist, they're just syntax sugar for functions!

Property decorators

- A special decorator that transforms a simple object property into a getter/setter pair where logic can be executed
  - Leverages the Object.defineProperty API to define the decorator logic
  - Example: the DatabaseId decorator

Bonus stage: debugging TypeScript programs