**Vue 3**

## **Declaring Reactive State**

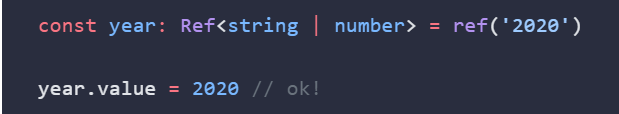
### ref()

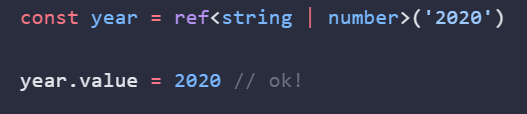
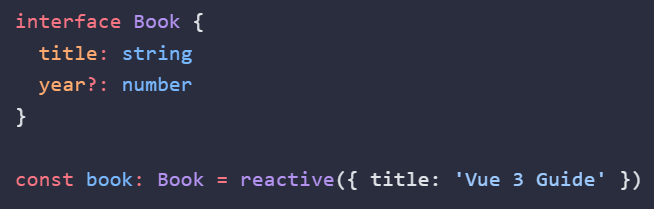
Refs can hold any value type, including deeply nested objects, arrays, or JavaScript built-in data structures like Map. A ref will make its value deeply reactive. This means you can expect changes to be detected even when you mutate nested objects or arrays: It is also possible to opt-out of deep reactivity with [shallow refs](https://vuejs.org/api/reactivity-advanced.html#shallowref). For shallow refs, only .value access is tracked for reactivity. Shallow refs can be used for optimizing performance by avoiding the observation cost of large objects, or in cases where the inner state is managed by an external library.

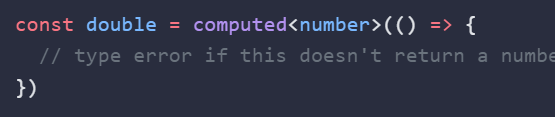
### reactive()

Reactive objects are JavaScript Proxies and behave just like normal objects. The difference is that Vue is able to intercept the access and mutation of all properties of a reactive object for reactivity tracking and triggering. reactive() converts the object deeply: nested objects are also wrapped with reactive() when accessed. It is also called by ref() internally when the ref value is an object. Similar to shallow refs, there is also the shallowReactive() API for opting-out of deep reactivity. It is important to note that the returned value from reactive() is a [Proxy](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy) of the original object, which is not equal to the original object:

### Code Snippets for reference



Typing computed()  


## Computed

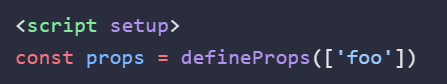
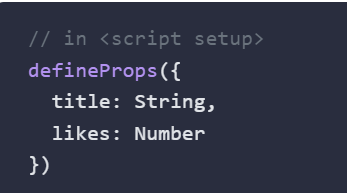
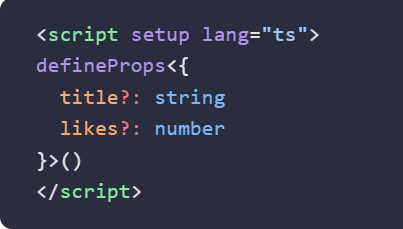
In Vue 3, the **computed** property is an essential feature used to create derived state, which is reactive data that depends on other reactive data. Computed properties are particularly useful because they are cached based on their dependencies and only re-evaluate when one of these dependencies changes. This makes them highly efficient for scenarios where you need to perform costly computations or transform data in a reactive way.

Let's consider a practical example where you fetch a list of users from an API and you want to display them sorted by their last name. You also want to compute the total number of active users. Here, using computed properties can efficiently handle the dynamic sorting and counting based on the fetched data.

A screen shot of a computer program

Description automatically generated

## Props

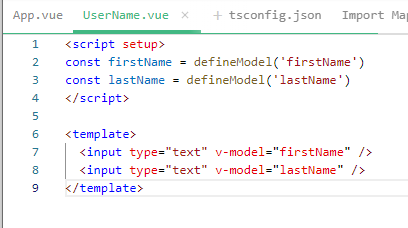
  
  
  
  
Typescript version  
  




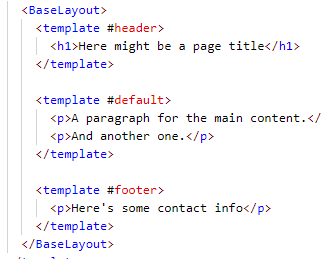
## Events

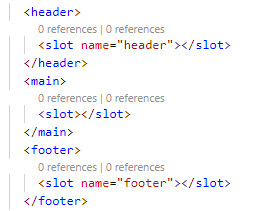


## Model

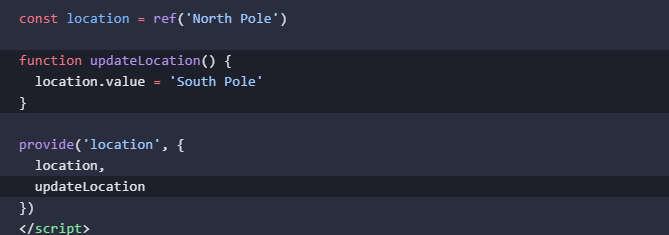
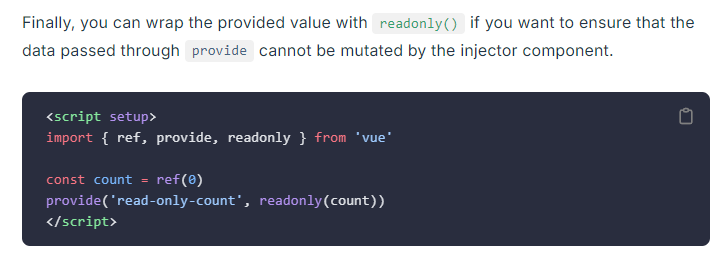
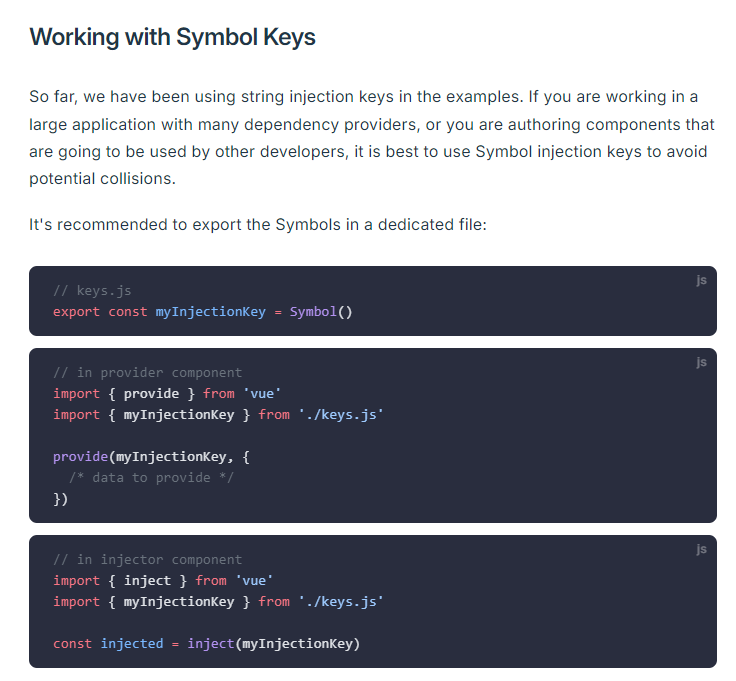
## Slots







## Provide Inject

## Async components



## Composable


# Typescript

## Class Basic

class Person {

  protected name: string

  protected age: number

  constructor(*name*: string, *age*: number) {

    this.name = *name*

    this.age = *age*

  }

  public greet(): void {

    console.log(`Hello, my name is ${this.name} from Person class.`)

  }

}

Derived Class: Employee

class Employee extends Person {

  private jobTitle: string

  private salary: number

  constructor(*name*: string, *age*: number, *jobTitle*: string, *salary*: number) {

*super*(*name*, *age*)

    this.jobTitle = *jobTitle*

    this.salary = *salary*

  }

  public greet(): void {

    console.log(`Hello, my name is ${this.name} from Employee class. I work as a ${this.jobTitle}.`)

  }

  public displayJob(): void {

    console.log(`My job title is ${this.jobTitle} and my salary is ${this.salary}`)

  }

  public promote(*newJobTitle*: string, *newSalary*: number): void {

    this.jobTitle = *newJobTitle*

    this.salary = *newSalary*

    console.log(`I have been promoted to ${this.jobTitle} with a salary of ${this.salary}`)

  }

}

Further Derived Class: Manager

class Manager extends Employee {

  private teamSize: number

  constructor(*name*: string, *age*: number, *jobTitle*: string, *salary*: number, *teamSize*: number) {

*super*(*name*, *age*, *jobTitle*, *salary*)

    this.teamSize = *teamSize*

  }

  public manageTeam(): void {

    console.log(`I manage a team of ${this.teamSize} members.`)

  }

  public displayDetails(): void {

    this.greet() *// This will call the greet method in Employee*

  }

}

const manager = new Manager('Alice', 40, 'Engineering Manager', 120000, 10)

manager.greet() *// Output: Hello, my name is Alice from Employee class. I work as a Engineering Manager.*

manager.displayDetails() *// Output: Hello, my name is Alice from Employee class. I work as a Engineering Manager.*

# SASS

## **Variables**

A screenshot of a computer code

Description automatically generated

## **Nesting**

A screenshot of a computer code

Description automatically generated

## Modules

**A screenshot of a computer program

Description automatically generated**

## **Mixins**

## Extend/Inheritance



## Operators

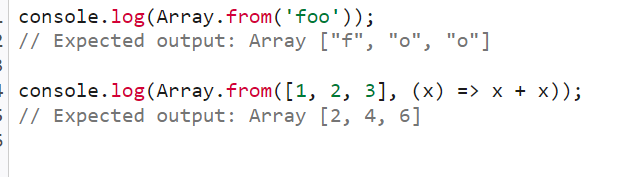
A screenshot of a computer code

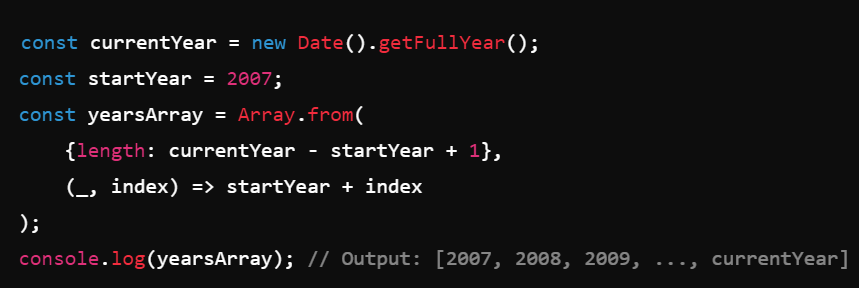
Description automatically generated

# ES6

## Array

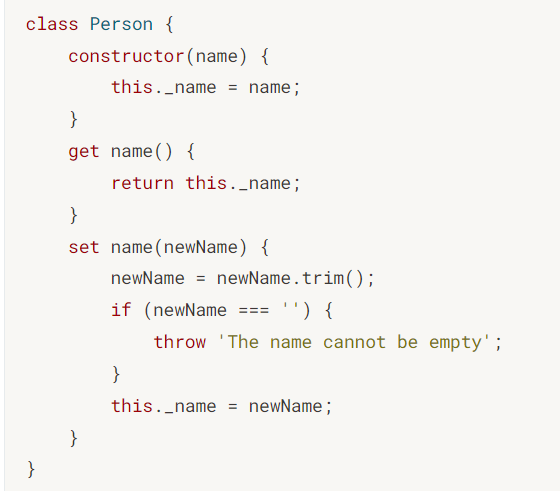
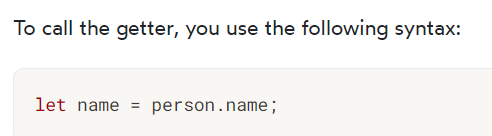
### **Array.from()**

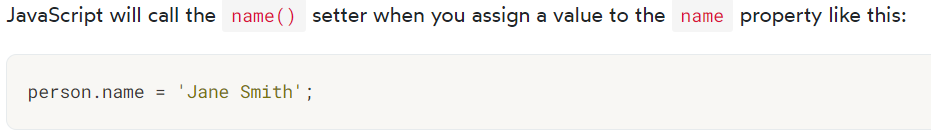






## **Getters and Setters**

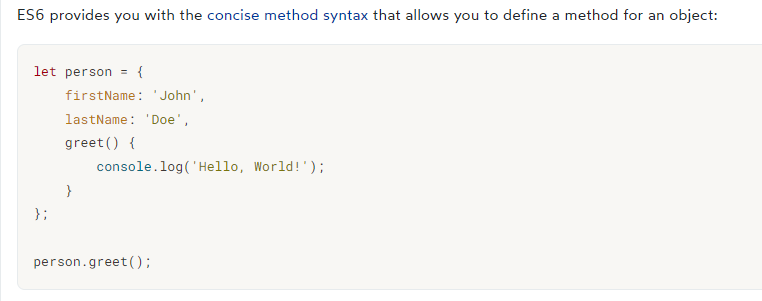
  




## **JavaScript object destructuring assignment**



## **Object method shorthand**



## JavaScript Constructor Function

## JavaScript Prototype