

# Вычисляемые свойства и наблюдатели

- Введение
- Вычисляемые свойства
- Вычисляемое свойство против метода
- Геттеры и сеттеры вычисляемых свойств
- Методы - наблюдатели
- Отслеживание изменений в массивах

## Введение

Встраиваемые в шаблоны выражения удобны, но могут предназначаться только для **простых операций**. Добавление слишком большого количества логики в ваши шаблоны может сделать их раздутыми и трудными в обслуживании. При усложнении логики вычислений такие шаблоны трудно поддерживать.

### example\_1. Постановка задачи

```
<div id="app">

  <h2>{{ surname + ' ' + name + ' ' + patronymic }} </h2>

  <h3>Персональные данные</h3>

  <div>

    Логин: Нет данных<br>

    E-mail: Нет данных<br>

    Возраст: {{ new Date().getFullYear() - new
    Date(dateOfBirth).getFullYear() }}

  </div>

</div>

<script>
```

```
// создаем экземпляр приложения

const app = Vue.createApp({

  data() {

    // набор данных приложения

    return {

      surname: "Иванов",

      name: "Петр",

      patronymic: "Сергеевич",

      dateOfBirth: "12.03.1975"

    }

  }

}).mount('#app');

</script>
```

Представленный пример шаблона уже не выглядит простым и декларативным.

```
13
14 <div id="app">
15   <h2>{{ surname + ' ' + name + ' ' + patronymic }} </h2>
16   <h3>Персональные данные</h3>
17   <div>
18     Логин: Нет данных<br>
19     E-mail: Нет данных<br>
20     Возраст: {{ new Date().getFullYear() - new Date(dateOfBirth).getFullYear() }}
21   </div>
22 </div>
23
```

Ситуация станет ещё хуже, если эту логику потребуется использовать в нескольких местах шаблона. И совсем недопустимо использование встраиваемых выражений, если в настройке data вы используете данные сложной структуры.

От встраиваемых выражений легко уйти, используя **методы** в выражении шаблона.

#### example\_2. Методы шаблона

```
<div id="app">
```

```
<h2>{{ getFullName() }} </h2>

<h3>Персональные данные:</h3>

<div>

  Логин: Нет данных<br>

  E-mail: Нет данных<br>

  Возраст: {{ getAge() }}

</div>

</div>

<script>

  // создаем экземпляр приложения

  const app = Vue.createApp({

    data() {

      // набор данных приложения

      return {

        // ...

      }

    },

    methods: {

      getFullName() {

        let fullName = this.surname + ' ' + this.name +
          ' ' + this.patronymic;

        return fullName;

      },

      getAge() {

        let age = new Date().getFullYear() - new
          Date(this.dateOfBirth).getFullYear();

        return age;

      }

    }

  });
```

```
}  
  
}).mount('#app');  
  
</script>
```

Вот теперь шаблон приложения выглядит гораздо лучше.

```
13  
14 <div id="app">  
15   <h2>{{ getFullName() }} </h2>  
16   <h3>Персональные данные:</h3>  
17   <div>  
18     Логин: Нет данных<br>  
19     E-mail: Нет данных<br>  
20     Возраст: {{ getAge() }}  
21   </div>  
22 </div>  
23
```

Но Vue предоставляет еще одну возможность, избежать лишних операций можно с помощью так называемых **вычисляемых свойств**. Эти свойства работают также, как и обычные, но отличаются тем, что они не присутствуют в **data** изначально, а вычисляются в процессе работы скрипта.

## Вычисляемые свойства

Для создания вычисляемых свойств предназначена настройка **computed**.

**Важно.** Обычные свойства хранятся в настройке **data** и представляют собой некоторые **данные** (числа, строки, массивы и т.п.), **вычисляемые свойства** хранятся в настройке **computed** и представляют собой **функции**.

Посмотрим практику применения вычисляемых свойств на конкретном примере.

**example\_3.** Вычисляемые свойства

```
<div id="app">  
  
  <h2>{{ cmpFullName }} </h2>  
  
  <h3>Персональные данные:</h3>
```

```
<div>

  Логин: Нет данных<br>

  Е-mail: Нет данных<br>

  Возраст: {{ cmpAge }}

</div>

</div>

<script>

  // создаем экземпляр приложения

  const app = Vue.createApp({

    data() {

      // набор данных приложения

      return {

        surname: "Иванов",

        name: "Петр",

        patronymic: "Сергеевич",

        dateOfBirth: "12.03.1975"

      }

    },

    computed: {

      cmpFullName() {

        let fullName = this.surname + ' ' + this.name +

          ' ' + this.patronymic;

        return fullName;

      },

      cmpAge() {

        let age = new Date().getFullYear() - new

          Date(this.dateOfBirth).getFullYear();

        return age;

      }

    }

  });

  app.mount('#app')
```

```
    }  
  
    }  
  
  }).mount('#app');  
  
</script>
```

Как видно из примера, **в шаблоне нет разницы** между обычными свойствами и вычисляемыми. В этом особое удобство использования вычисляемых свойств.

Следующий пример демонстрирует возможности вычисляемых свойств.

Попробуйте изменить значения свойств настройки **data** и вычисляемые свойства соответственно **обновятся**.

#### example\_4. Обновление вычисляемых свойств

```
<div id="app">  
  
  <h2>{{ cmpFullName }} </h2>  
  
  <h3>Персональные данные:</h3>  
  
  <div>  
  
    Логин: Нет данных<br>  
  
    E-mail: Нет данных<br>  
  
    Возраст: {{ cmpAge }}  
  
  </div>  
  
</div>  
  
<script>  
  
  // создаем экземпляр приложения  
  
  const app = Vue.createApp({  
  
    data() {  
  
      // набор данных приложения  
  
      return {  
  
        // ...  
  
      }  
  
    }  
  
  })
```

```
    },  
  
    computed: {  
  
      cmpFullName () {  
  
        // ...  
  
      },  
  
      cmpAge () {  
  
        // ...  
  
      }  
  
    }  
  
  }) .mount ('#app');  
  
  // изменим фамилию  
  app.surname = "Сумароков-Эльстон";  
  
  // изменим дату рождения  
  app.dateOfBirth = "10.05.1985";  
  
  // от изменяемых данных зависят вычисляемые свойства  
  
  // -> значит и вычисляемые свойства обновятся  
  
</script>
```

Вы можете привязывать данные к вычисляемым свойствам в шаблонах точно так же, как к обычному свойству. Vue знает, что вычисляемые свойства зависят от данных, поэтому он обновит любые привязки в вычисляемых свойствах, **как только обновятся данные**.

## Вычисляемое свойство против метода

Таким образом вместо метода мы можем использовать вычисляемое свойство. Но можно ли считать эти два способа равнозначными?

Для конечного результата два подхода действительно **абсолютно одинаковы**.

**Однако** разница в том, что вычисляемые свойства кэшируются на основе их

реактивных зависимостей. Вычисляемое свойство будет повторно оцениваться только тогда, когда некоторые из его реактивных зависимостей изменились.

Это означает, что до тех пор, пока в свойстве `app.dateOfBirth` не произошло изменений, обращение к вычисляемому свойству `cmpAge` немедленно вернет ранее вычисленный результат без необходимости повторного запуска функции получения.

**Важно.** Вычисляемые свойства пересчитываются только тогда, когда изменились их реактивные зависимости.

Для сравнения, **вызов метода всегда будет запускать функцию, когда происходит повторный рендеринг шаблона.**

Плюсы кэширования не совсем очевидны на простых примерах, но представьте, что у нас есть некоторое дорогостоящее вычисляемое свойство `cmpList`, которое требует перебора огромного массива и выполнения большого количества вычислений. В приложении могут быть другие вычисляемые свойства, которые, в свою очередь, зависят от `cmpList`. Без кэширования мы бы выполняли функцию получения данных намного больше раз, чем необходимо!

В случаях, когда вы не хотите кэширования, используйте вместо этого вызов метода. При этом не забывайте, использование метода может быть не очень оптимальным с точки зрения производительности.

Рассмотрим использование реактивных зависимостей на еще одном демонстрационном примере. На этот раз рассчитаем корни квадратного уравнения.

#### example\_5. Реактивные зависимости в вычисляемых свойствах

```
<div id="app">
  <p>Дискриминант: {{ cmpDiscr }} </p>
  <p>Корни: {{ cmpResp }} </p>
</div>
```



```
<script>
```

```
const app = Vue.createApp({  
  data() {  
    return {  
      a : 1,  
      b : 6,  
      c : 4,  
      d : undefined  
    }  
  },  
  computed: {  
    cmpDiscr: function() {  
      // вычисляем дискриминант  
      let d = this.b**2 - 4 * this.a * this.c;  
      this.d = d;  
      // возвращаем  
      return d;  
    },  
    cmpResp: function() {  
      // вычисляем корни  
      if (this.d < 0) {  
        return "Нет корней";  
      }  
      if (this.d == 0) {  
        x = (-this.b - Math.sqrt(this.d) ) / 2 *  
          this.a;  
        return `x = ${x.toFixed(2)}`;  
      }  
    }  
  }  
})
```

```
};

    if (this.d > 0) {

        x1 = (-this.b - Math.sqrt(this.d) ) / 2 *
        this.a;

        x2 = (-this.b + Math.sqrt(this.d) ) / 2 *
        this.a;

        return `x1 = ${x1.toFixed(2)}, x2 =
        ${x2.toFixed(2)}`;

    };

}

}

}).mount('#app');

// меняем значение коэффициента - a
// порождаем цепочку связанных событий
// -- пересчитываем Дискриминант - d
// -- пересчитываем Корни - x1, x2
// app.a = 4;
// меняем напрямую значение дискриминанта
// -- пересчитываем Корни - x1, x2
// app.d = 2;

</script>
```

Усложним логику приложения, подключим **внешний файл** с данными объекта заказа.

**example\_6.** Вычисляемые свойства для внешних данных

```
<div id="app">

    <h2>{{ cmpFullName }} </h2>

    <h3>Персональные данные</h3>
```

```
<div>

  Логин: {{ data.vendor.login }} <br>

  Е-mail: {{ data.vendor.email }} <br>

  Действующая скидка: {{ data.vendor.discount }}% <br>

</div>

<h3>Данные заказа</h3>

<div>

  Дата заказа: {{ data.vendor.date }} <br>

  Счет: {{ data.vendor.bill }} <br>

  Количество заказов: {{ cmpAmount }}

</div>

<h3>Позиции заказа</h3>

<ul>

  <li v-for="n in 3"> {{ n }} ... </li>

</ul>

</div>

<script>

  // создаем экземпляр приложения

  let app = Vue.createApp({

    data() {

      // набор данных приложения

      return { data : orders }

    },

    computed: {

      cmpFullName() {

        return this.data.vendor.surname + " " +

          this.data.vendor.name + " " +

          this.data.vendor.patronymic;
```

```
    },  
  
    cmpAmount () {  
  
        return this.data.product.length;  
  
    }  
  
}  
  
)).mount("#app");  
  
</script>
```

Важно помнить, что вычисляемые функции получения данных должны выполнять только чистые вычисления и быть свободны от побочных эффектов. Например, не выполняйте асинхронные запросы или **не изменяйте DOM внутри вычисляемого свойства**.

Думайте о вычисляемом свойстве как о **декларативном описании того, как получить значение на основе других значений** - его единственной обязанностью должно быть вычисление и возврат этого значения.

С вычисляемыми свойствами разобрались, идем дальше.

## Геттеры и сеттеры вычисляемых свойств

Вычисляемые свойства по умолчанию доступны только для метода **получения** (геттер - get). Если вы попытаетесь присвоить вычисляемому свойству новое значение, вы получите предупреждение во время выполнения.

В редких случаях, когда вам требуется вычисляемое свойство, доступное для записи, вы можете создать его, предоставив как **метод получения** (get), так и **метод установки** (сеттер - set).

## Избегайте изменения вычисляемого значения

Возвращаемое значение из вычисляемого свойства является производным состоянием. Думайте об этом как о временном снимке - каждый раз, когда изменяется исходное состояние, создается новый снимок.

Изменять снимок не имеет смысла, поэтому вычисленное возвращаемое значение следует рассматривать как доступное только для чтения и никогда не изменять - вместо этого обновите исходное состояние, от которого оно зависит, чтобы запустить новые вычисления.

example\_7. Геттер и сеттер вычисляемого свойства

```
<div id="app">

  <h2>{{ surname }} {{ name }} {{ patronymic }}</h2>

  <h3>Персональные данные:</h3>

  <div>

    Login: {{ login }} <br>

    E-mail: {{ cmpEmail }}<br>

    Возраст: {{ new Date().getFullYear() - new
    Date(dateOfBirth).getFullYear() }}

  </div>

</div>

<script>

  // создаем экземпляр приложения

  const app = Vue.createApp({

    data() {

      // набор данных приложения

      return {

        surname: "Иванов",

        name: "Петр",

        patronymic: "Сергеевич",

        dateOfBirth: "12.03.1975",

        login: "ivan-off"

      }

    },

  },
```

```
    computed: {  
      cmpEmail: {  
        // геттер свойства cmpEmail  
        get () {  
          return this.login + "@ppet-group.ru";  
        },  
        // сеттер свойства cmpEmail  
        set(str) {  
          let arr = str.split("@");  
          this.login = arr[0];  
        }  
      }  
    }  
  }).mount('#app');  
  // app.cmpEmail = "master@home.ru";  
  // console.log(app.cmpEmail);  
</script>
```

**Важно.** Вместо изменения вычисляемого свойства, измените его реактивную зависимость.

В следующем примере вычисляемый геттер свойства **cmpFullName** получает исходные данные объекта, сеттер свойства **обновляет** реактивные зависимости.

**example\_8.** Применение геттеров и сеттеров вычисляемых свойств

```
<div id="app">  
  <h2>{{ cmpFullName }} </h2>  
  <h3>Персональные данные</h3>
```

```
<div>

  Логин: {{ data.vendor.login }} <br>

  Е-mail: {{ data.vendor.email }} <br>

  Действующая скидка: {{ data.vendor.discount }}% <br>

</div>

<h3>Данные заказа</h3>

<div>

  Дата заказа: {{ data.vendor.date }} <br>

  Счет: {{ data.vendor.bill }} <br>

  Количество заказов: --

</div>

<h3>Позиции заказа</h3>

<ul>

  <li v-for="n in 3"> {{ n }} ... </li>

</ul>

</div>

<script>

  // создаем экземпляр приложения

  let app = Vue.createApp({

    data() {

      // набор данных приложения

      return { data : orders }

    },

    computed: {

      cmpFullName: {

        // геттер свойства cmpFullName

        get () {
```

```
        return this.data.vendor.surname + " " +
        this.data.vendor.name + " " +
        this.data.vendor.patronymic;

    },

    // сеттер свойства cmpFullName

    set (str) {

        let arr = str.split(" ");

        this.data.vendor.surname = arr[0];

        this.data.vendor.name = arr[1];

        this.data.vendor.patronymic = arr[2];

    }

}

}).mount("#app");

// обновим свойство cmpFullName

// сеттер свойства обновит свойства объекта: name, surname,
patronymic

// app.cmpFullName = "Иванов Иван Иванович";

</script>
```

Хорошего кода много не бывает.

Обратите внимание, в следующем примере сеттером вычисляемого свойства я меняю не само свойство, а элементы его зависимостей (скидку на каждый товар заказа).

#### example\_9. Применение сеттеров в вычислениях свойств

```
<div id="app">

  <h2>{{ cmpfullName }} </h2>

  <h3>Персональные данные</h3>
```



```
<div>

  <!-- -->

</div>

<h3>Данные заказа</h3>

<div>

  <!-- -->

  Сумма заказа: <b>{{ cmpTotalPrice }}</b> <br>

</div>

<h3>Позиции заказа</h3>

  <!-- -->

</div>

<script>

  // создаем экземпляр приложения

  let app = Vue.createApp({

    data() {

      // набор данных приложения

      return { data : orders }

    },

    computed: {

      cmpfullName: {

        // ...

      },

      cmpTotalPrice: {

        // геттер вычисляемого свойства

        get() {

          console.log("Сработал геттер вычисляемого свойства");

        }

      }

    }

  })

  app.mount('#app')
```

```
    let totalPrice = 0;

    with (this.data) {

        for ( item of product){

            totalPrice += item.price;

        }

    }

    return totalPrice;

},

// сеттер вычисляемого свойства

set(val) {

    console.log("Сработал сеттер вычисляемого свойства");

    with (this.data) {

        for ( item of product){

            item.price = item.price - (item.price
            * val);

        }

    }

}

}) .mount("#app");

// установим скидку в размере 20% на каждый товар

// app.cmpTotalPrice = 0.02;

</script>
```



Хотя в большинстве случаев лучше использовать вычисляемые свойства, иногда необходимы пользовательские **методы-наблюдатели**. Поэтому Vue предоставляет более общий способ реагирования на изменения в данных через настройку **watch**.

Эта возможность полезна для *дорогих* или асинхронных операций, выполняемых в ответ на изменение данных.

Рассмотрим пример.

#### example\_10. Методы-наблюдатели

```
<div id="app">
  S = {{ square }} <br>
</div>
<script>
  const app = Vue.createApp({
    data() {
      return {
        r : undefined,
        square : undefined
      }
    },
    watch: {
      r() {
        this.square = 3.14 * this.r ** 2;
      }
    }
  }).mount('#app');
```

```
// назначаем свойство  
  
app.r = 7;  
  
</script>
```

В демонстрационном примере **example\_11** эта же задача решена с помощью вычисляемого свойства.

В следующем примере *понаблюдаем* за свойством **dateOfBirth**.

#### example\_12. Применение наблюдателей

```
<div id="app">  
  
  <!-- -->  
  
  Возраст: {{ age }} <br>  
  
</div>  
</div>  
<script>  
  
  const app = Vue.createApp({  
  
    data() {  
  
      return {  
  
        dateOfBirth : "12.03.1975",  
  
        age : "Нет данных"  
  
      }  
  
    },  
  
    watch: {  
  
      dateOfBirth() {  
  
        let today = new Date();  
  
        let yearCurrent = today.getFullYear();  
  
        let dateOfBirth = new Date(this.dateOfBirth);  
  
        let yearOfBirth = dateOfBirth.getFullYear();
```

```
        this.age = yearCurrent - yearOfBirth;

    }

}

}).mount('#app');

// обновляем свойство

app.dateOfBirth = "12.03.1971";

</script>
```

В случае желания наблюдать за вложенными свойствами, путь к свойству необходимо **обернуть в кавычки**.

#### example\_13. Наблюдатели для вложенных свойств

```
<div id="app">

  <h2> {{ fullName }} </h2>

</div>

<script>

  const app = Vue.createApp({

    data() {

      return {

        data : orders,

        fullName : "Нет данных"

      }

    },

    watch: {

      'data.vendor.surname' (val) {

        this.fullName = val + ' ' + this.data.vendor.name

      },

      'data.vendor.name' (val) {

        this.fullName = this.data.vendor.surname + ' ' +
```

```
        val
      }
    }

  }).mount('#app');

  // обновим вложенное свойство name,
  // сформируем свойство fullName

  app.data.vendor.name = "Венедикт";

</script>
```

## Отслеживание изменений в массивах

---

### Методы, изменяющие массив

Во Vue для каждого массива определен ряд методов, с помощью которых можно управлять элементами массива:

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

**Эти методы являются обертками над одноименными стандартными методами JavaScript для управления массивами и работают точно также.**

Единственное отличие их от стандартных методов состоит в том, что эти методы-обертки информируют систему Vue о том, что с массивом были произведены некоторые действия, и соответственно для этого массива может быть произведен **повторный рендеринг на веб-странице.**

В следующем примере добавление нового элемента в массив немедленно отобразится в представлении.

#### example\_14. Методы, изменяющие массив

```
<div id="app">

  <h2>Выбор языка программирования:</h2>

  <select>

    <option v-for="elem in langs">{{ elem }}</option>

  </select>

</div>

<script>

  let app = Vue.createApp({
    data() {
      return {
        // одномерный массив данных
        langs: ["PHP", "JavaScript", "Python", "Perl",
          "Java", "C++"]
      }
    }
  }).mount("#app");

  // добавим в массив элементы
  app.langs.push("ActionScript");
  app.langs.unshift("Visual Basic");

  // вывод массива в консоль
  console.log(app.langs);

</script>
```

**Важно.** Методы, изменяющие массив, как следует из названия, будут изменять

исходный массив, на котором они вызваны.

## Методы, создающие новый массив

Но существуют и другие методы, например:

- `filter()`
- `concat()`
- `slice()`,

которые не изменяют исходный массив, а **всегда возвращают новый массив**.

При их использовании можно просто заменять старый массив на новый.

**example\_15.** Методы создающие массив

```
<div id="app">

  <h2>Курсы по программированию</h2>

  <select>

    <option v-for="elem in course">{{ elem }}</option>

  </select>

  <h2>Курсы по программированию на Java и JavaScript</h2>

  <select>

    <option v-for="elem in cmpCourse">{{ elem }}</option>

  </select>

</div>

<h1>Учись у лучших ! ;)</h1>

<script>

  let app = Vue.createApp({

    data() {

      return {

        // одномерный массив данных
```



```
        course: [  
            "JavaScript разработчик с нуля",  
            "Python-разработчик",  
            "Data Scientist",  
            "Фронтенд-разработчик на JavaScript",  
            "Тестирование веб-приложений на Python",  
            "Разработчик C++",  
            "Android-разработчик",  
            "HTML, CSS, JavaScript для чайников",  
            "Программирование на Java"  
        ],  
        },  
        computed : {  
            cmpCourse() {  
                // создадим новый массив  
  
                return this.course.filter(val =>  
                    val.includes("Java"))  
            }  
        }  
    }) .mount("#app");  
  
    // добавим новый элемент в массив  
    app.course.push ("Асинхронный JavaScript");  
  
    // обратите внимание,  
  
    // новый элемент отобразится сразу в двух списках  
  
</script>
```

**Примечание.** Преобразование и вывод массивов отработаем на следующих уроках курса.

## Задача 1

Напишите **метод-наблюдатель** за свойствами **a** и **h**, вычисляющий **площадь треугольника** ( $S = (a * h) / 2$ ) при назначении свойствам начальных значений.

## Задача 2

Напишите **вычисляемое свойство** создающее **новый** массив из исходного. Пусть в новый массив попадут только те авторы, для которых родным языком является - **русский**.

## Задача 3

Напишите **геттер вычисляемого свойства**, в котором от полного имени будет выводиться сокращенный формат записи (например - Баранов Е.И.).

## Задача 4

Напишите приложение, запрашивающее у пользователя дату рождения в формате ДД.ММ.ГГГГ. Используя возможности условной отрисовки блоков

шаблона выведете в браузер картинку согласно **вычисленного возраста**:

---

**P.S.**

Для отработки и закрепления учебного курса **донам** группы предоставляется следующий раздаточный материал.

К каждому уроку курса:

- Файлы **демонстрационного кода** (example);
- **Задачи** с решениями в контексте рассматриваемых вопросов урока (task).

К каждой теме курса:

- **Практические** работы.