Vue Mastery

Advanced Components

18:11

Lessons

1. Introduction
1:43

2. Build a Reactivity System
16:50

3. Evan You on Proxies
9:48

Build a Reactivity System

In this lesson we will build a simple reactivity system using the very same techniques you'll find in the Vue source code. This will give you a better understanding of Vue.js and it's design patterns, as well as get you familiar with watchers and the Dep class.

The Reactivity System

Vue's reactivity system can look like magic when you see it working for the first time.

Take this simple app:

```
<div id="app">
  <div>Price: ${{ price }}</div>
  <div>Total: ${{ price * quantity }}</div>
  <div>Taxes: ${{ totalPriceWithTax }}</div>
</div>



<script src="https://cdn.jsdelivr.net/npm/vue"></script>
<script>
  var vm = new Vue({
    el: '#app',
    data: {
      price: 5.00,
      quantity: 2
    },
    computed: {
      totalPriceWithTax() {
        return this.price * this.quantity * 1.03
      }
    }
  })
</script>
```

And somehow Vue just knows that if `price` changes, it should do three things:

- Update the `price` value on our webpage.
- Recalculate the expression that multiplies `price` `*` `quantity`, and update the page.
- Call the `totalPriceWithTax` function again and update the page.

But wait, I hear you wonder, how does Vue know what to update when the `price` changes, and how does it keep track of everything?

This is not how JavaScript programming usually works

If it's not obvious to you, the big problem we have to address is that programming usually doesn't work this way. For example, if I run this code:

```
let price = 5
let quantity = 2
let total = price * quantity  // 10 right?
price = 20
console.log(`total is ${total}`)
```

What do you think it's going to print? Since we're not using Vue, it's going to print `10`.

```
>> total is 10
```

In Vue we want `total` to get updated whenever `price` or `quantity` get updated. We want:

```
>> total is 40
```

Unfortunately, JavaScript is procedural, not reactive, so this doesn't work in real life. In order to make `total` reactive, we have to use JavaScript to make things behave differently.

Problem

We need to save how we're calculating the `total`, so we can re-run it when `price` or `quantity` changes.

Solution

First off, we need some way to tell our application, "The code I'm about to run, **store this**, I may need you to run it at another time." Then we'll want to run the code, and if `price` or `quantity` variables get updated, run the stored code again.

We might do this by recording the function so we can run it again.

```
let price = 5
let quantity = 2
let total = 0
let target = null
```

```
target = function () {
  total = price * quantity
})


record() // Remember this in case we want to run it later
target() // Also go ahead and run it
```

Notice that we store an anonymous function inside the `target` variable, and then call a `record` function. Using the ES6 arrow syntax I could also write this as:

```
target = () => { total = price * quantity }
```

The definition of the `record` is simply:

```
let storage = [] // We'll store our target functions in here

function record () { // target = () => { total = price * quantity }
  storage.push(target)
}
```

We're storing the `target` (in our case the `{ total = price * quantity }`) so we can run it later, perhaps with a `replay` function that runs all the things we've recorded.

```
function replay (){
  storage.forEach(run => run())
}
```

This goes through all the anonymous functions we have stored inside the storage array and executes each of them.

Then in our code, we can just:

```
price = 20
console.log(total) // => 10
replay()
console.log(total) // => 40
```

Simple enough, right? Here's the code in it's entirety if you need to read through and try to grasp it one

more time. FYI, I am coding this in a particular way, in case you're wondering why.

```
let price = 5
let quantity = 2
let total = 0
let target = null
let storage = []

function record () {
  storage.push(target)
}

function replay () {
  storage.forEach(run => run())
}

target = () => { total = price * quantity }

record()
target()

price = 20
console.log(total) // => 10
replay()
console.log(total) // => 40
```

```
10
40
```

Problem

We could go on recording targets as needed, but it'd be nice to have a more robust solution that will scale with our app. Perhaps a class that takes care of maintaining a list of targets that get notified when we need them to get re-run.

Solution: A Dependency Class

One way we can begin to solve this problem is by encapsulating this behavior into its own class, a **Dependency Class** which implements the standard programming observer pattern.

So, if we create a JavaScript class to manage our dependencies (which is closer to how Vue handles things), it might look like this:

```
    class Dep { // Stands for dependency
      constructor () {
        this.subscribers = [] // The targets that are dependent, and
should be
                              // run when notify() is called.
      }
      depend() {  // This replaces our record function
        if (target && !this.subscribers.includes(target)) {
          // Only if there is a target & it's not already subscribed
          this.subscribers.push(target)
        }
      }
      notify() {  // Replaces our replay function
        this.subscribers.forEach(sub => sub()) // Run our targets, or
observers.
      }
    }
```

Notice instead of `storage` we're now storing our anonymous functions in `subscribers`. ****Instead of our `record` function we now call `depend` ****and we now use `notify` instead of `replay`. To get this running:

```
    const dep = new Dep()

    let price = 5
    let quantity = 2
    let total = 0
    let target = () => { total = price * quantity }
    dep.depend() // Add this target to our subscribers
    target()  // Run it to get the total

    console.log(total) // => 10 .. The right number
    price = 20
    console.log(total) // => 10 .. No longer the right number
    dep.notify()       // Run the subscribers
    console.log(total) // => 40  .. Now the right number
```

It still works, and now our code feels more reusable. Only thing that still feels a little weird is the setting and running of the `target`.

Problem

In the future we're going to have a Dep class for each variable, and it'll be nice to encapsulate the

In the future we're going to have a Dep class for each variable, and it'd be nice to encapsulate the behavior of creating anonymous functions that need to be watched for updates. Perhaps a `watcher` function might be in order to take care of this behavior.

So instead of calling:

```
target = () => { total = price * quantity }
dep.depend()
target()
```

(this is just the code from above)

We can instead just call:

```
watcher(() => {
  total = price * quantity
})
```

Solution: A Watcher Function

Inside our Watcher fucntion we can do a few simple things:

```
function watcher(myFunc) {
  target = myFunc // Set as the active target
  dep.depend()       // Add the active target as a dependency
  target()           // Call the target
  target = null      // Reset the target
}
```

As you can see, the `watcher` function takes a `myFunc` argument, sets that as a our global `target` property, calls `dep.depend()` to add our target as a subscriber, calls the `target` function, and resets the `target`.

Now when we run the following:

```
price = 20
console.log(total)
dep.notify()
console.log(total)
```

```
10
40
```

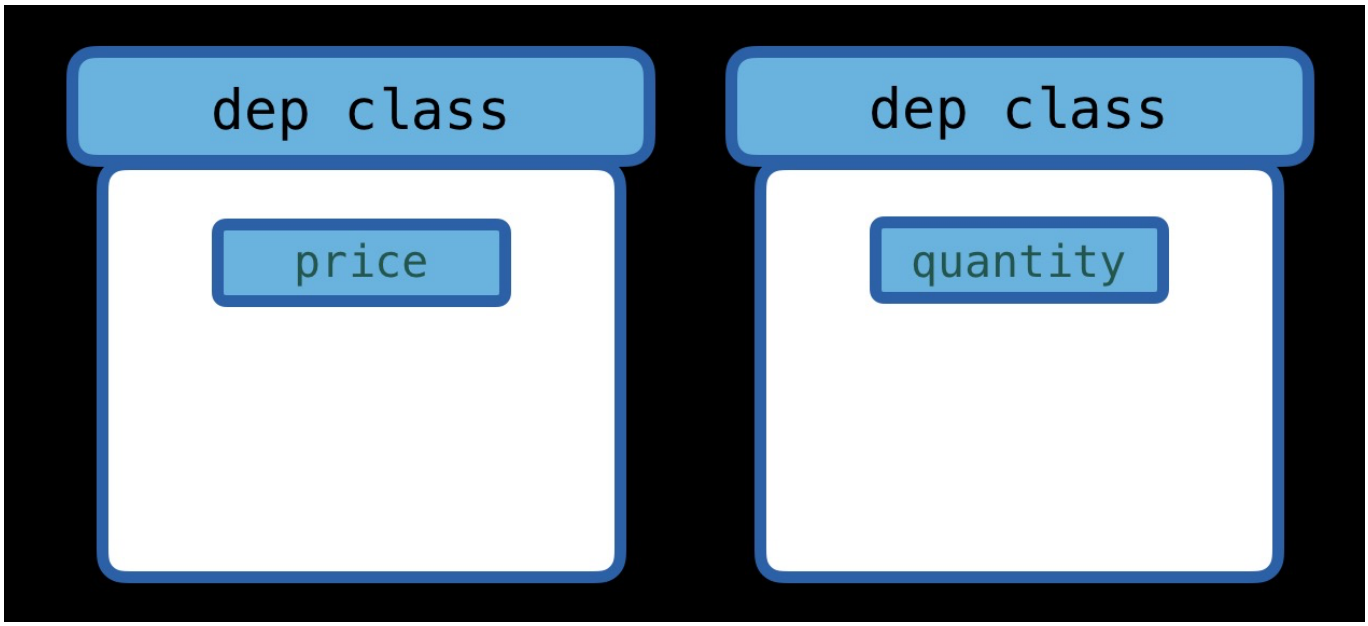
You might be wondering why we implemented `target` as a global variable, rather than passing it into our functions where needed. There is a good reason for this, which will become obvious by the end of our article.

Problem

We have a single `Dep class`, but what we really want is each of our variables to have its own Dep. Let me move things into properties before we go any further.

```
let data = { price: 5, quantity: 2 }
```

Let's assume for a minute that each of our properties (`price` and `quantity`) have their own internal Dep class.

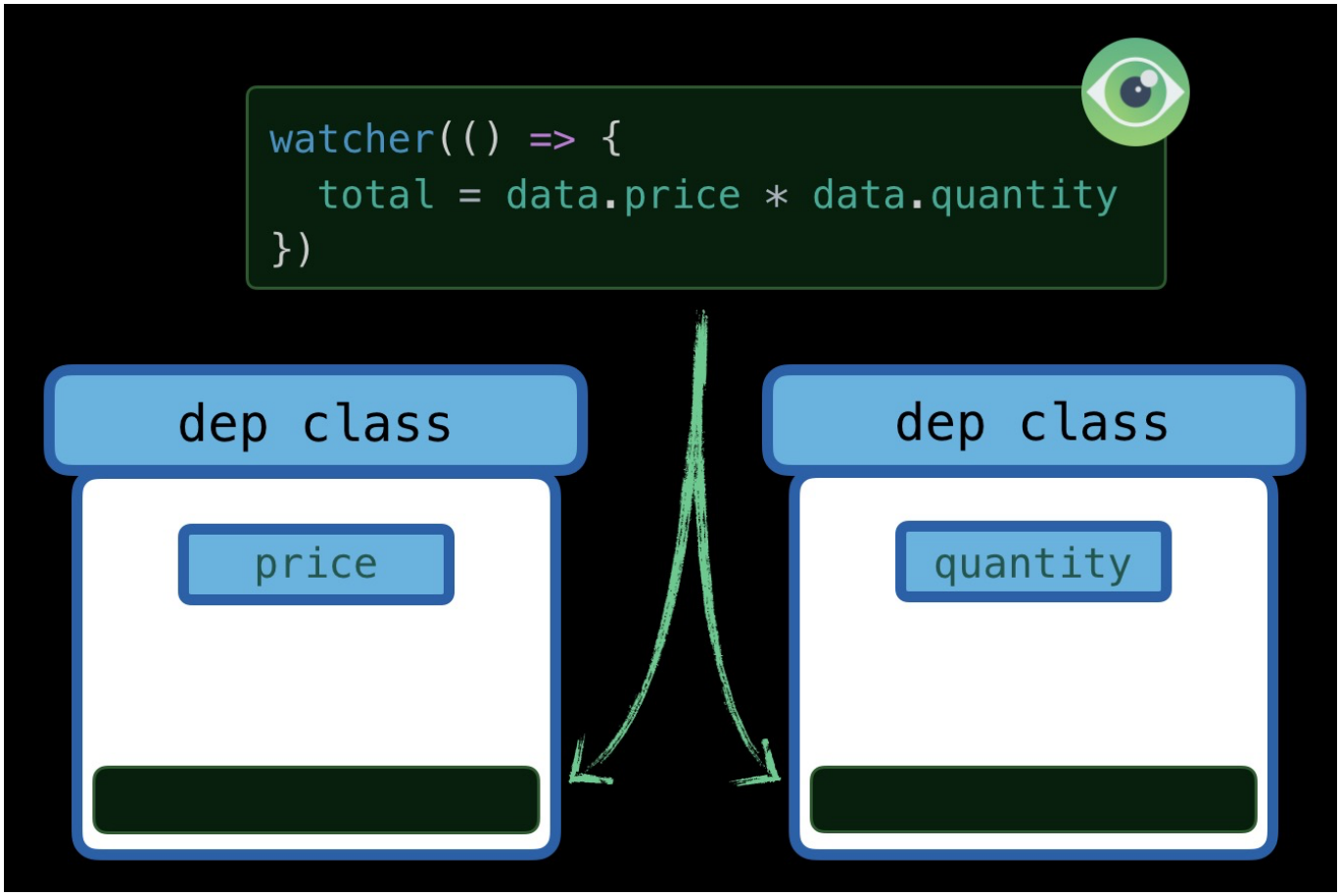


Now when we run:

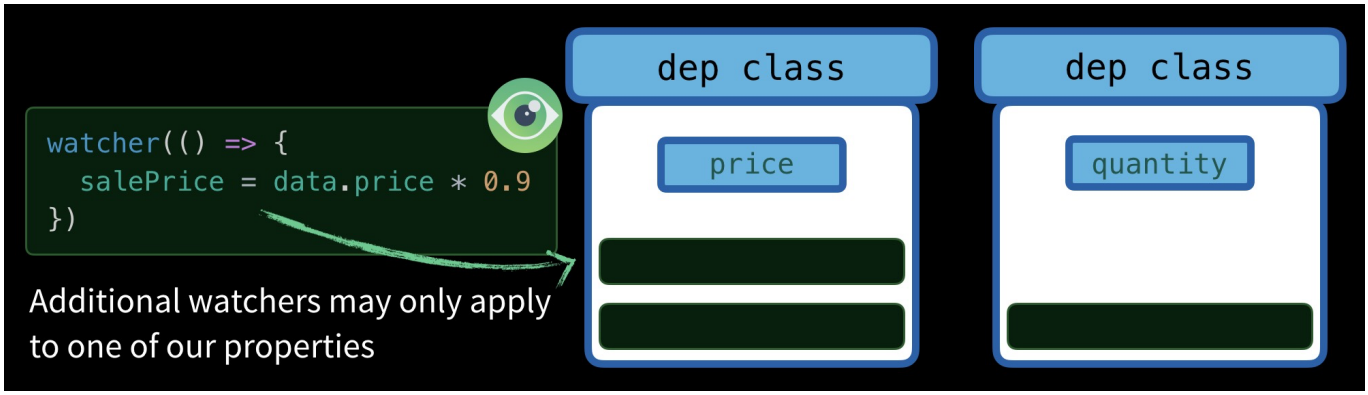
```
watcher(() => {
  total = data.price * data.quantity
})
```

Since the `data.price` value is accessed (which it is), I want the `price` property's Dep class to

push our anonymous function (stored in `target`) onto its subscriber array (by calling `dep.depend()`). Since `data.quantity` is accessed I also want the `quantity` property Dep class to push this anonymous function (stored in `target`) into its subscriber array.



If I have another anonymous function where just `data.price` is accessed, I want that pushed just to the `price` property Dep class.



When do I want `dep.notify()` to be called on `price`'s subscribers? I want them to be called when `price` is set. By the end of the article I want to be able to go into the console and do:

```
>> total
```

```
10
>> price = 20  // When this gets run it will need to call notify() on
the price
>> total
40
```

We need some way to hook into a data property (like `price` or `quantity`) so when it's accessed we can save the `target` into our subscriber array, and when it's changed run the functions stored our subscriber array.

Solution: Object.defineProperty()

We need to learn about the Object.defineProperty() function which is plain ES5 JavaScript. It allows us to define getter and setter functions for a property. Lemme show you the very basic usage, before I show you how we're going to use it with our Dep class.

```
let data = { price: 5, quantity: 2 }

Object.defineProperty(data, 'price', {  // For just the price
property

    get() {  // Create a get method
      console.log(`I was accessed`)
    },

    set(newVal) {  // Create a set method
      console.log(`I was changed`)
    }
})
data.price // This calls get()
data.price = 20  // This calls set()
```

```
I was accessed
I was changed
```

As you can see, it just logs two lines. However, it doesn't actually `get` or `set` any values, since we over-rode the functionality. Let's add it back now. `get()` expects to return a value, and `set()` still needs to update a value, so let's add an `internalValue` variable to store our current `price` value.

```
let data = { price: 5, quantity: 2 }
```

```
let data = { price: 5, quantity: 2 }

let internalValue = data.price // Our initial value.

Object.defineProperty(data, 'price', {  // For just the price property

  get() {  // Create a get method
    console.log(`Getting price: ${internalValue}`)
    return internalValue
  },

  set(newVal) {  // Create a set method
    console.log(`Setting price to: ${newVal}` )
    internalValue = newVal
  }
})
total = data.price * data.quantity  // This calls get()
data.price = 20  // This calls set()
```

Now that our get and set are working properly, what do you think will print to the console?

```
Getting price: 5
Setting price to: 20
```

So we have a way to get notified when we get and set values. And with some recursion we can run this for all items in our data array, right?

FYI, `Object.keys(data)` returns an array of the keys of the object.

```
let data = { price: 5, quantity: 2 }

Object.keys(data).forEach(key => { // We're running this for each item in data now
  let internalValue = data[key]
  Object.defineProperty(data, key, {
    get() {
      console.log(`Getting ${key}: ${internalValue}`)
      return internalValue
    },
    set(newVal) {
      console.log(`Setting ${key} to: ${newVal}` )
      internalValue = newVal
    }
  })
})
```

```
})
total = data.price * data.quantity
data.price = 20
```

Now everything has getters and setters, and we see this on the console.

```
Getting price: 5
Getting quantity: 2
Setting price to: 20
```

Putting both ideas together

```
total = data.price * data.quantity
```

When a piece of code like this gets run and **gets** the value of `price`, we want `price` to remember this anonymous function (`target`). That way if `price` gets changed, or is **set** to a new value, it'll trigger this function to get rerun, since it knows this line is dependent upon it. So you can think of it like this.

Get => Remember this anonymous function, we'll run it again when our value changes.

Set => Run the saved anonymous function, our value just changed.

Or in the case of our Dep Class

Price accessed (get) => call `dep.depend()` to save the current `target`

Price set => call `dep.notify()` on price, re-running all the `targets`

Let's combine these two ideas, and walk through our final code.

```
let data = { price: 5, quantity: 2 }
let target = null

// This is exactly the same Dep class
class Dep {
  constructor () {
    this.subscribers = []
  }
  depend() {
    if (target && !this.subscribers.includes(target)) {
      // Only if there is a target & it's not already subscribed
      this.subscribers.push(target)
    }
  }
  notify() {
    this.subscribers.forEach(sub => sub())
  }
}

// Same observe function
function observe(obj) {
  Object.keys(obj).forEach(key => {
    let internalValue = obj[key]
    let dep = new Dep()
    Object.defineProperty(obj, key, {
      get() {
        dep.depend()
        return internalValue
      },
      set(newVal) {
        internalValue = newVal
        dep.notify()
      }
    })
  })
}

observe(data)

function watcher(myFunc) {
  target = myFunc
  target()
  target = null
}

watcher(() => {
  data.total = data.price * data.quantity
})
```

```
      this.subscribers.push(target)
    }
  }
  notify() {
    this.subscribers.forEach(sub => sub())
  }
}

// Go through each of our data properties
Object.keys(data).forEach(key => {
  let internalValue = data[key]

  // Each property gets a dependency instance
  const dep = new Dep()

  Object.defineProperty(data, key, {
    get() {
      dep.depend() // <-- Remember the target we're running
      return internalValue
    },
    set(newVal) {
      internalValue = newVal
      dep.notify() // <-- Re-run stored functions
    }
  })
})

// My watcher no longer calls dep.depend,
// since that gets called from inside our get method.
function watcher(myFunc) {
  target = myFunc
  target()
  target = null
}

watcher(() => {
  data.total = data.price * data.quantity
})
```

And now look at what happens in our console when we play around.

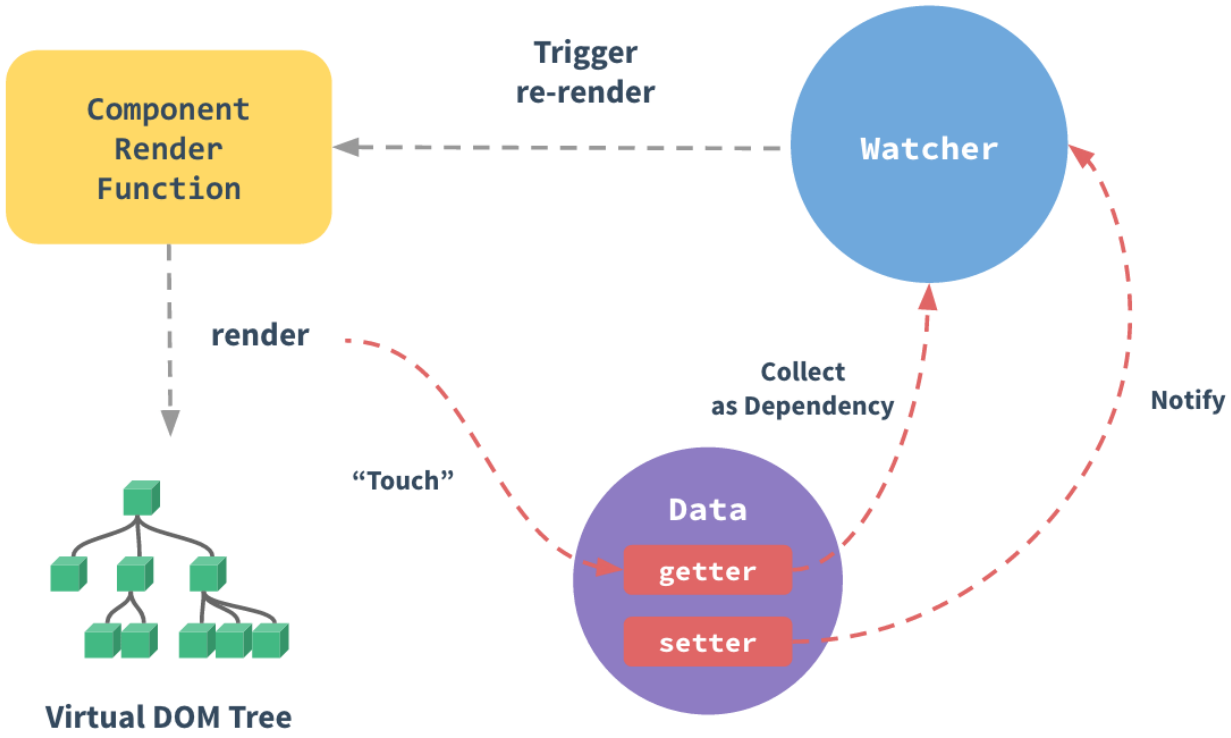
```
> data.total
<- 10
> data.price = 20
<- 20
> data.total
<- 40
> data.quantity = 3
```

```
> data.quantity = 3
< 3
> data.total
< 60
```

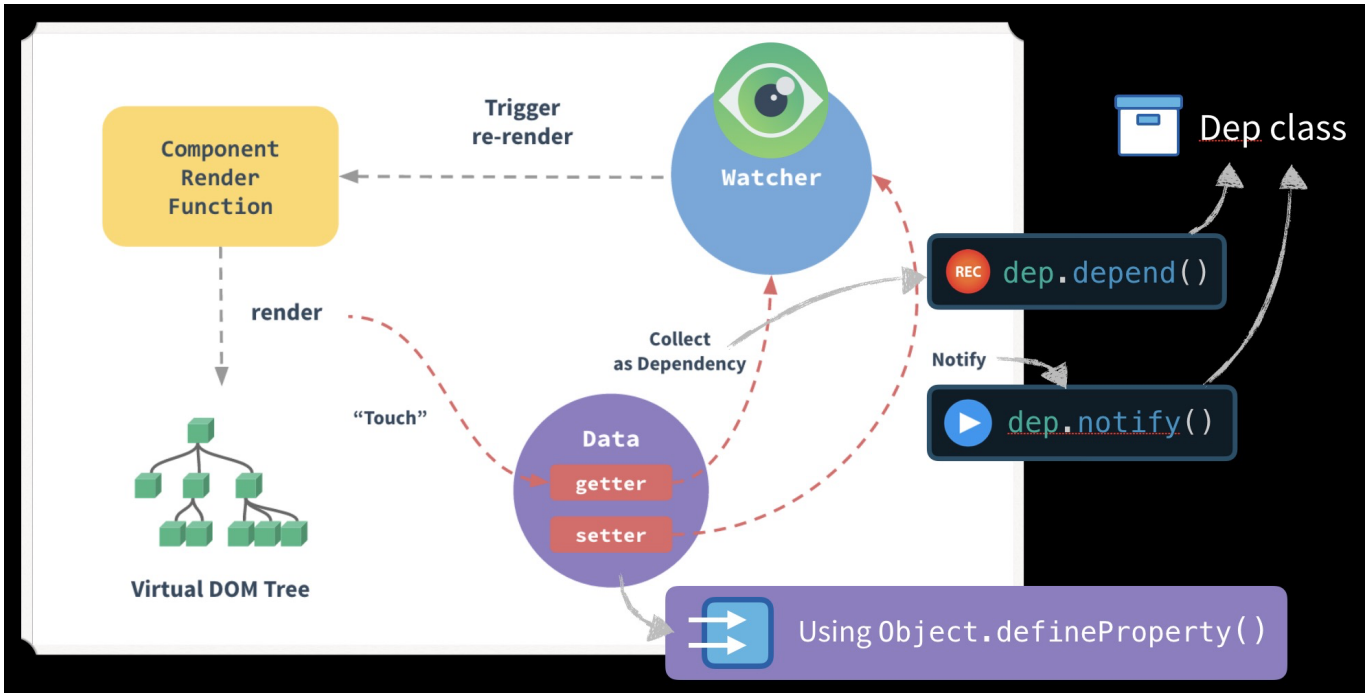
Exactly what we were hoping for! Both `price` and `quantity` are indeed reactive! Our total code gets re-run whenever the value of `price` or `quantity` gets updated.

Jumping to Vue

This illustration from the Vue docs should start to make sense now.



Do you see that beautiful purple Data circle with the getters and setters? It should look familiar! Every component instance has a `watcher` instance (in blue) which collects dependencies from the getters (red line). When a setter is called later, it **notifies** the watcher which causes the component to re-render. Here's the image again with some of my own annotations.



Yeah, doesn't this make a whole lot more sense now?

Obviously how Vue does this under the covers is more complex, but you now know the basics. In the next lesson we'll dive under the hood with Vue, and see if we can find this pattern inside the source code.

So what have we learned?

- How to create a **Dep class** which collects a dependencies (depend) and re-runs all dependencies (notify).
- How to create a **watcher** to manage the code we're running, that may need to be added (target) as a dependency.
- How to use **Object.defineProperty()** to create getters and setters.

Share Lesson

Discuss in our Facebook Group

Send us Feedback

Previous lesson Next lesson

The Official Vue.js News

We also help produce the Official Vue.js News. It's a free community resource where we curate the most impactful Vue.js news and tutorials. Offered up in both text and audio versions. Consider subscribing today.

Read Official Vue.js News

Listen to the Podcast

Nuxt Cheat Sheet

As the ultimate resource for Vue.js developers, Vue Mastery produces weekly lessons so you can learn what you need to succeed as a Vue.js Developer.