Vue 3 Reactivity

In this course we will understand the new Vue 3 Reactivity system. Learning how this is built from the ground up will help you understand the design patterns used inside Vue, improve your Vue debugging skills, enable you to use the new Vue 3 modularized Reactivity library, and perhaps even contribute to the Vue 3 source code yourself.

In this lesson we will start building a simple reactivity system using the very same techniques you'll find in the Vue 3 source code.

Understanding Reactivity

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

And somehow Vue's Reactivity system 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's Reactivity system 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, programming usually doesn't work this way. For example, if I run this code:

```
let product = { price: 5, quantity: 2 }
let total = product.price * product.quantity  // 10 right?
product.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.

For the rest of this lesson and the next 2 after this one, we will be building a Reactivity System from scratch using the same methodology as Vue 3 (which is very different than Vue 2). We will then look into the Vue 3 source code to discover these patterns we wrote from scratch.

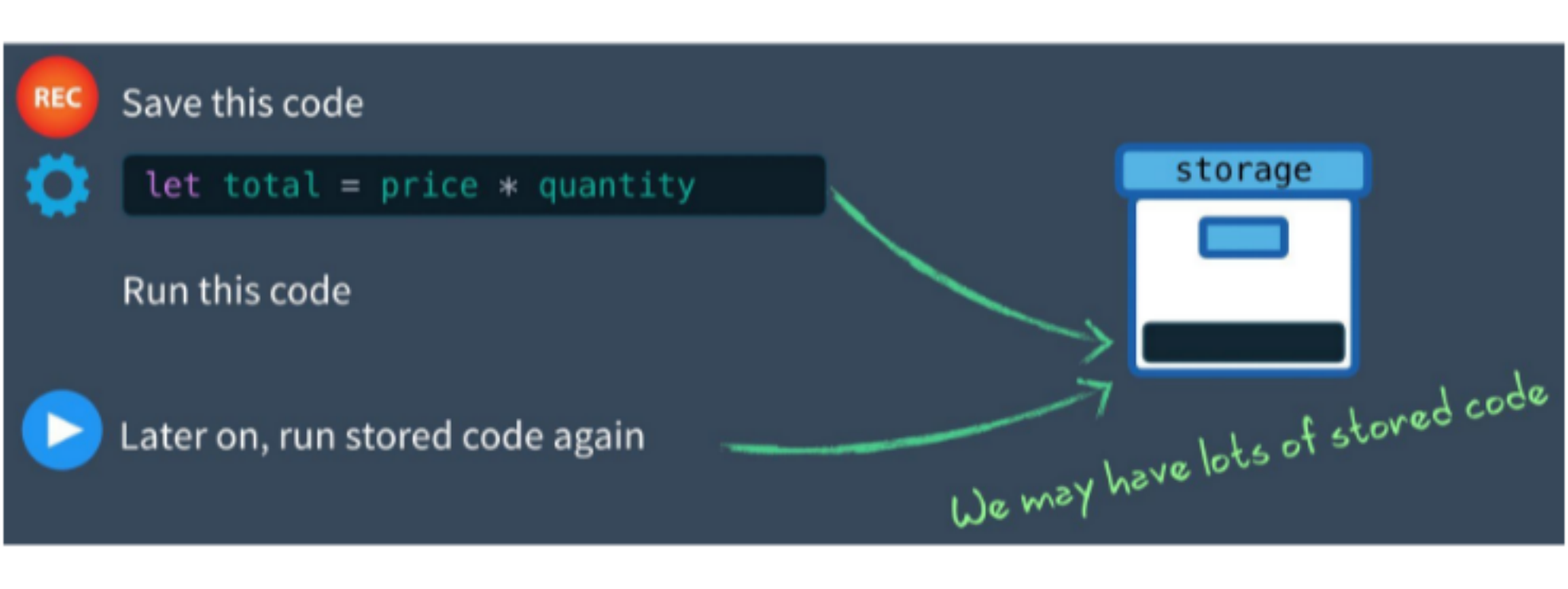
Saving Code to Run Later

Problem

As you saw with the code above, in order to start building reactivity 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, "Store the code (effect) I'm about to run, 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 (effect) so we can run it again.

```
let product = { price: 5, quantity: 2 }
let total = 0

let effect = function () {
  total = product.price * product.quantity
})

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

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

```
let effect = () => { total = product.price * product.quantity }
```

In order to define `track`, we need a place to store our effects, we may have many of them. We'll create a variable called `dep`, as in dependency. We call it dependency because typically with the Observer design pattern a dependency has subscribers (in our case effects) which will get notified when an object changes state. We might make dependency a class with an array of subscribers, like we did in the Vue 2 version of this tutorial. However, since all it needs to store is a set of effects, we can simply create a **Set**.

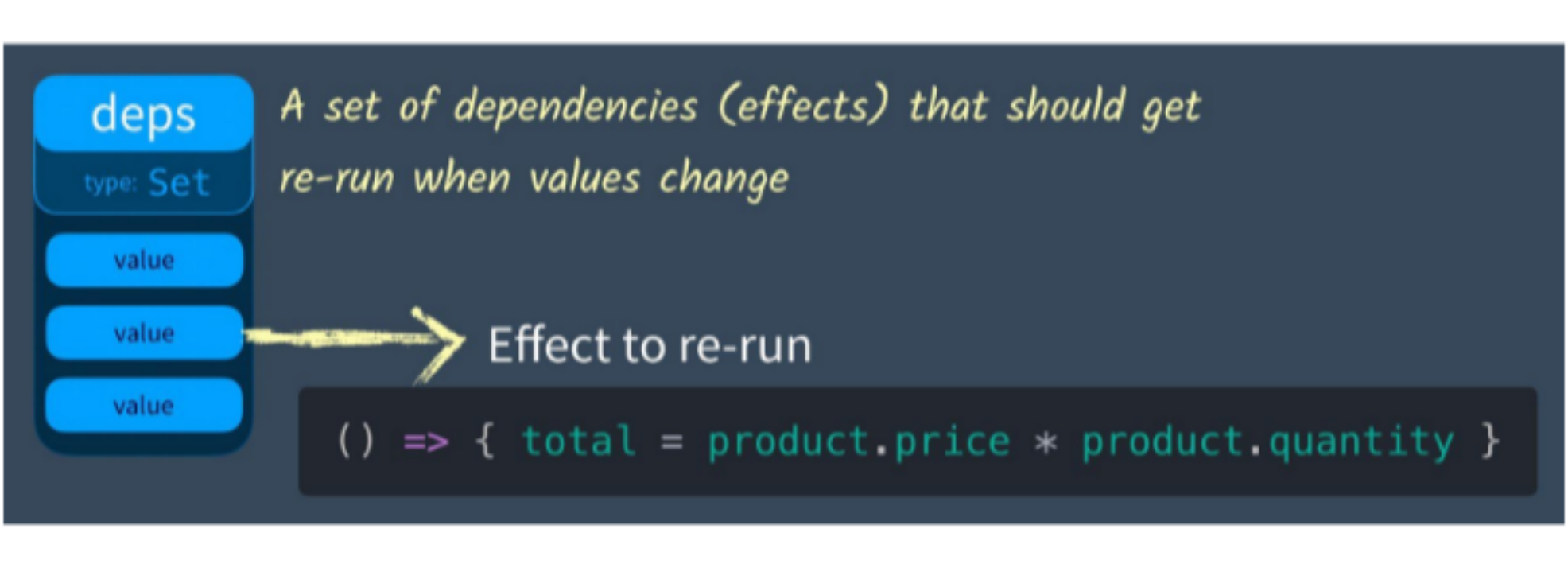
```
let dep = new Set() // Our object tracking a list of effects
```

Then our `track` function can simply add our effects to this collection:

```
function track () {
  dep.add(effect) // Store the current effect
}
```

In case you're not familiar, the difference between a JavaScript Array and Set, is that a Set cannot have duplicate values and it doesn't use an index like arrays. Learn more about Set's here if you're not familiar.

We're storing the `effect` (in our case the `{ total = price * quantity }`) so we can run it later. Here's a visualization this dep Set:



Let's write a trigger function that runs all the things we've recorded.

```
function trigger() {
  dep.forEach(effect => effect())
}
```

This goes through all the anonymous functions we have stored inside the `dep` Set and executes each of them. Then in our code, we can just:

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

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

```
let product = { price: 5, quantity: 2 }
let total = 0
let dep = new Set()

function track() {
  dep.add(effect)
}

function trigger() {
  dep.forEach(effect => effect())
}

let effect = () => {
  total = product.price * product.quantity
}

track()
effect()

product.price = 20
console.log(total) // => 10

trigger()
console.log(total) // => 40
```

```
10
40
```

Problem: Multiple Properties

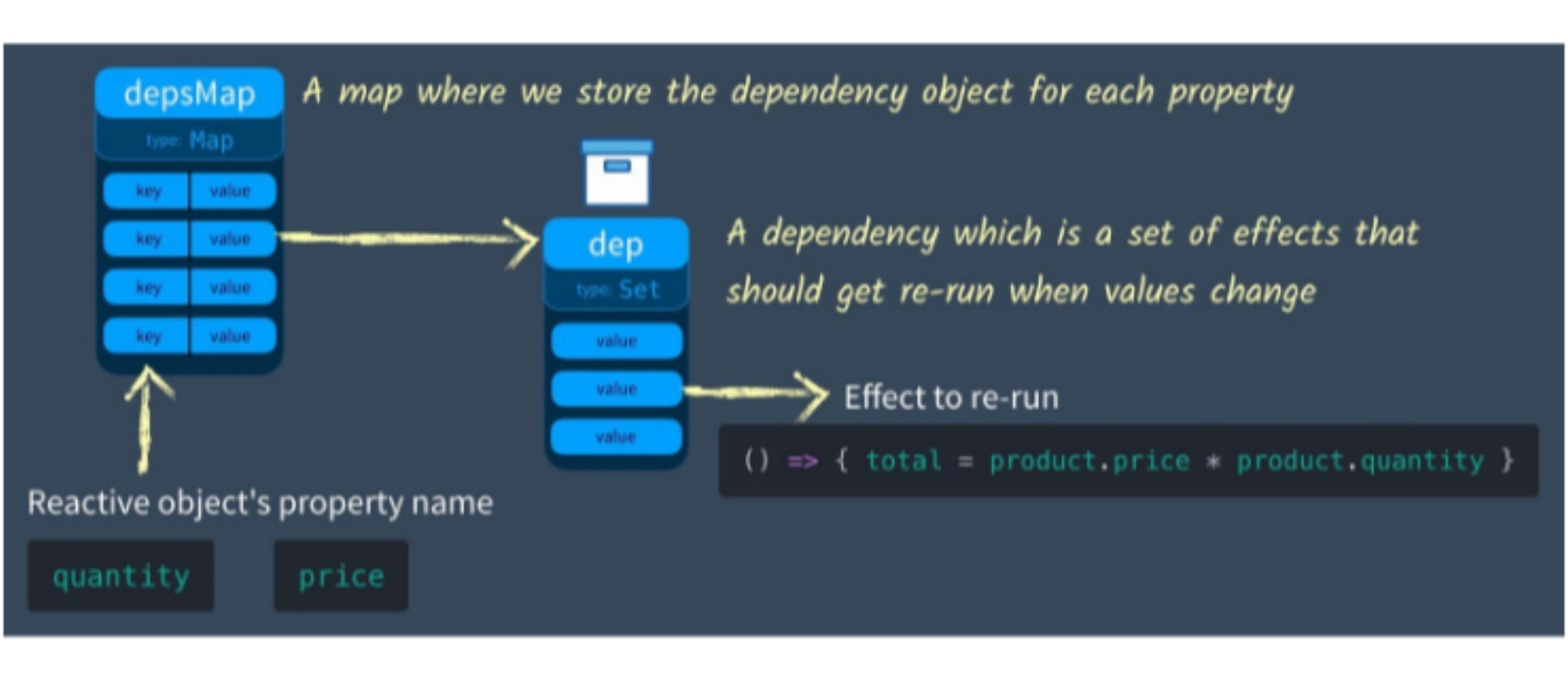
We could go on tracking effects as needed, but our reactive objects are going to have different properties, and those properties each need their own `dep` (which is a set of `effects`). Take a look at our object here:

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

Our `price` property needs it's own dep (set of `effects`) and our `quantity` needs it's own `dep` (set of `effects`). Let's build out our solution to properly record these.

Solution: depsMap

When we call track or trigger we now need to know which property in our object we're targeting (`price` or `quantity`). To do this we'll create a `depsMap`, which is of type **Map** (think keys and values). Here's how we might visualize it:



Notice how the `depsMap` has a key which will be the property name we want to add (or track) a new `effect` on. So we'll need to send in this key to the `track` function.

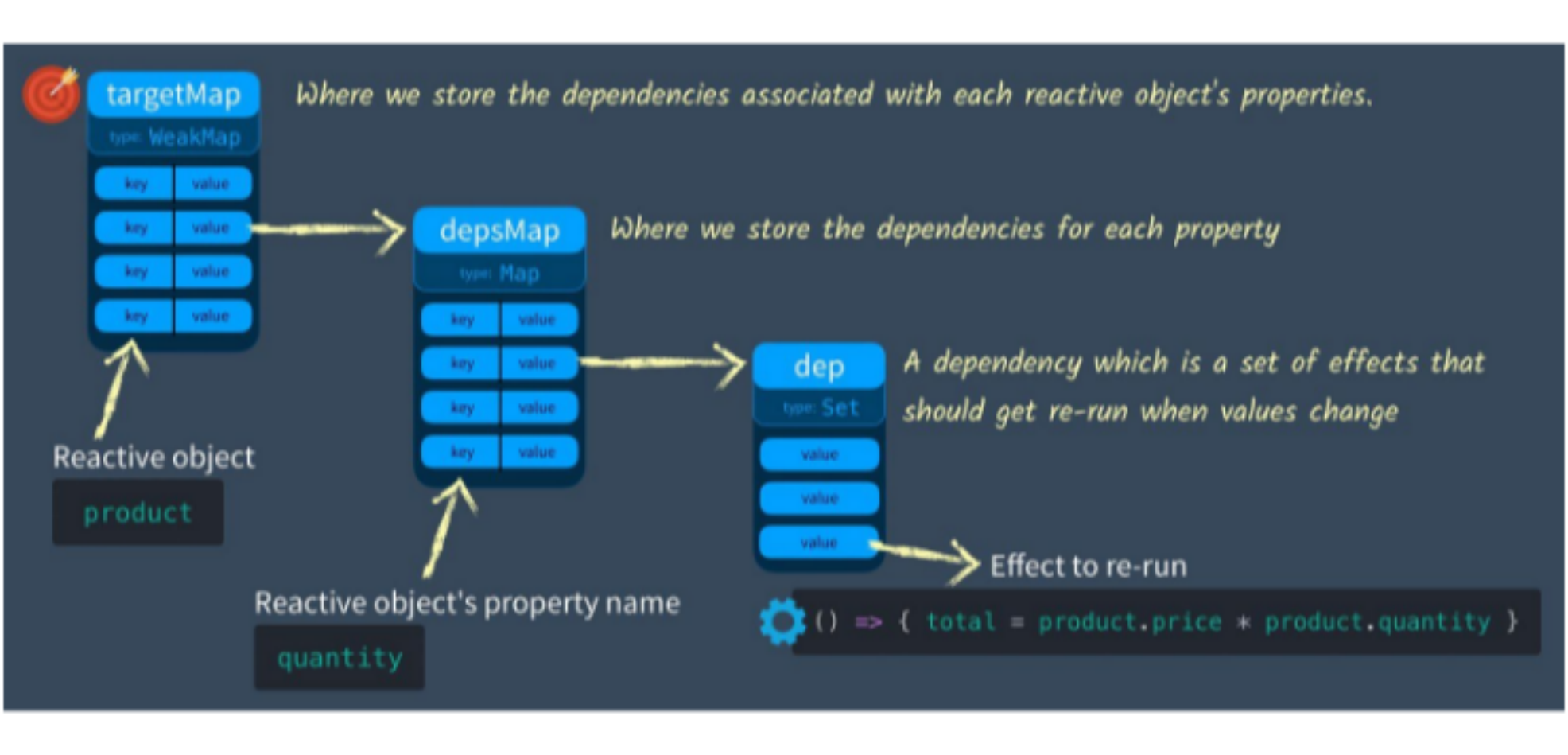
```
const depsMap = new Map()
function track(key) {
  // Make sure this effect is being tracked.
  let dep = depsMap.get(key) // Get the current dep (effects) that need to be run when this key (property) is set
  if (!dep) {
    // There is no dep (effects) on this key yet
    depsMap.set(key, (dep = new Set())) // Create a new Set
  }
  dep.add(effect) // Add effect to dep
}
  }
function trigger(key) {
  let dep = depsMap.get(key) // Get the dep (effects) associated with this key
  if (dep) { // If they exist
    dep.forEach(effect => {
      // run them all
      effect()
    })
  }
}

let product = { price: 5, quantity: 2 }
let total = 0

let effect = () => {
  total = product.price * product.quantity
}

track('quantity')
effect()
console.log(total) // --> 10

product.quantity = 3
trigger('quantity')
console.log(total) // --> 40
```

Problem: Multiple Reactive Objects

This works great, until we have multiple reactive objects (more than just product) which need to track effects. Now we need a way of storing a `depsMap` for each object (ex. product). We need another Map, one for each object, but what would be the key? If we use a **WeakMap** we can actually use the objects themselves as the key. **WeakMap** is a JavaScript Map that uses only objects as the key. For example:

```
let product = { price: 5, quantity: 2 }
const targetMap = new WeakMap()
targetMap.set(product, "example code to test")
console.log(targetMap.get(product)) // ---> "example code to test"
```

Obviously this isn't the code we're going to use, but I wanted to show you how our `targetMap` uses our product object as the key. We call our WeakMap `targetMap` because we'll consider target the object we're targeting. There's another reason it's called target which will become more obvious in the next lesson. Here is what we have visualized:



When we call `track` or `trigger` we now need to know which object we're targeting. So, we'll send in both the `target` and the key when we call it.

```
const targetMap = new WeakMap() // targetMap stores the effects that each object should re-run when it's updated

function track(target, key) {
  // We need to make sure this effect is being tracked.
  let depsMap = targetMap.get(target) // Get the current depsMap for this target

  if (!depsMap) {
    // There is no map.
    targetMap.set(target, (depsMap = new Map())) // Create one
  }

  let dep = depsMap.get(key) // Get the current dependencies (effects) that need to be run when this is set
  if (!dep) {
    // There is no dependencies (effects)
    depsMap.set(key, (dep = new Set())) // Create a new Set
  }

  dep.add(effect) // Add effect to dependency map
}

function trigger(target, key) {
  const depsMap = targetMap.get(target) // Does this object have any properties that have dependencies (effects)
  if (!depsMap) {
    return
  }

  let dep = depsMap.get(key) // If there are dependencies (effects) associated with this
  if (dep) {
    dep.forEach(effect => {
      // run them all
      effect()
    })
  }
}

let product = { price: 5, quantity: 2 }
let total = 0
let effect = () => {
  total = product.price * product.quantity
}

track(product, 'quantity')
effect()
console.log(total) // --> 10

product.quantity = 3
trigger(product, 'quantity')
console.log(total) // --> 15
```

So now we have a very effective way of tracking the dependencies on multiple objects, this is a big piece of the puzzle when building our reactivity system. Give yourself a pat on the back. The battle is half over. In the next lesson we will discover how to call `track` and `trigger` automatically using ES6 proxy.

Proxy and Reflect

In our last lesson we learned how Vue 3 keeps track of `effects` to re-run them when needed. However, we're still having to manually call `track` and `trigger`. In this lesson we'll learn how to use `Reflect` and `Proxy` to call them automatically.

Solution: Hooking onto Get and Set

We need a way to hook (or listen for) the get and set methods on our reactive objects.

GET property => We need to `track` the current `effect`

SET property => We need to `trigger` any tracked dependencies (effects) for this property

The first step to understanding how to do this, is to understand how in Vue 3 with ES6 `Reflect` and `Proxy` we can intercept GET and SET calls. Previously in Vue 2 we did this with ES5 `Object.defineProperty`.

Understanding ES6 Reflect

To print out an object property I can do this:

```
let product = { price: 5, quantity: 2 }
console.log('quantity is ' + product.quantity)
// or
console.log(`quantity is ` + product[`quantity`])
```

However, I can also GET values on an object by using `Reflect.get`. `Reflect` allows you to get a property on an object. It's just another way to do what I wrote above:

```
console.log('quantity is ' + Reflect.get(product, 'quantity'))
```

Why use `reflect`? Good question! Because it has a feature we'll need later, hold that thought.

Understanding ES6 Proxy

A `Proxy` is a placeholder for another object, which by default delegates to the object. So if I run the following code:

```
let product = { price: 5, quantity: 2 }
let proxiedProduct = new Proxy(product, {})
console.log(proxiedProduct.quantity)
```

The `proxiedProduct` delegates to the `product` which returns `2` as the quantity. Notice the second argument on `Proxy` with `{}`? This is called a `handler` and can be used to define custom behavior on the proxy object, like intercepting `get` and `set` calls. These interceptor methods are called `traps` and here's how we would set a `get` trap on our `handler`:

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

let proxiedProduct = new Proxy(product, {
  get() {
    console.log('Get was called')
    return 'Not the value'
  }
})

console.log(proxiedProduct.quantity)
```

In the console I'd see:

Get was called

Not the value

We've re-written what `get` returns when the property value is accessed. We should probably return the actual value, which we can do like:

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

let proxiedProduct = new Proxy(product, {
  get(target, key) {  // <--- The target (our object) and key (the property name)
    console.log('Get was called with key = ' + key)
    return target[key]
  }
})

console.log(proxiedProduct.quantity)
```

Notice that the `get` function has two parameters, both the `target` which is our object (`product`) and the `key` we are trying to get, which in this case is `quantity`. Now we see:

Get was called with key = quantity

2

This is also where we can use Reflect and add an additional argument to it.

```
let product = { price: 5, quantity: 2 }
let proxiedProduct = new Proxy(product, {
  get(target, key, receiver) {  // <--- notice the receiver
    console.log('Get was called with key = ' + key)
    return Reflect.get(target, key, receiver) // <----
  }
})
```

Notice our get has an additional parameter called `receiver` which we're sending as an argument into `Reflect.get`. This ensures that the proper value of `this` is used when our object has inherited values / functions from another object. This is why we always use `Reflect` inside of a `Proxy`, so we can keep the original behavior we are customizing.

Now let's add a setter method, there shouldn't be any big surprises here:

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

let proxiedProduct = new Proxy(product, {
  get(target, key, receiver) {
    console.log('Get was called with key = ' + key)
    return Reflect.get(target, key, receiver)
  },
  set(target, key, value, receiver) {
    console.log('Set was called with key = ' + key + ' and value = ' + value)
    return Reflect.set(target, key, value, receiver)
  }
})

proxiedProduct.quantity = 4
console.log(proxiedProduct.quantity)
```

Notice that `set` looks very similar to get except that it's using `Reflect.set` which receives the `value` to set the `target` (product). Our output as expected is:

Set was called with key = quantity and value = 4

Get was called with key = quantity

4

There's another way we can encapsulate this code, which is what you see in the Vue 3 source code. First, we'll wrap this proxying code in a `reactive` function which returns the proxy, which should look familiar if you've played with the Vue 3 Composition API. Then we'll declare our `handler` with it's `traps` separately and send them into our proxy.

```
function reactive(target) {
  const handler = {
    get(target, key, receiver) {
      console.log('Get was called with key = ' + key)
      return Reflect.get(target, key, receiver)
    },
    set(target, key, value, receiver) {
      console.log('Set was called with key = ' + key + ' and value = ' + value)
      return Reflect.set(target, key, value, receiver)
    }
  }
  return new Proxy(target, handler)
}

let product = reactive({ price: 5, quantity: 2 }) // <-- Returns a proxy object
product.quantity = 4
console.log(product.quantity)
```

This would return the same as above, but now we can easily create multiple reactive objects.

Combining Proxy + Effect Storage

If we take the code we have for creating reactive objects, and remember:

GET property => We need to `track` the current effect

SET property => We need to `trigger` any tracked dependencies (`effects`) for this property

We can start to imagine where we need to call `track` and `trigger` with the code above:

```
function reactive(target) {
  const handler = {
    get(target, key, receiver) {
      let result = Reflect.get(target, key, receiver)
      // track()
      return result
    },
    set(target, key, value, receiver) {
      let oldValue = target[key]
      let result = Reflect.set(target, key, value, receiver)
      if (result && oldValue != value) {  // Only if the value changes
        // trigger
      }
      return result
    }
  }
  return new Proxy(target, handler)
}
```

Now let's put the two pieces of code together:

```
const targetMap = new WeakMap() // targetMap stores the effects that each object should re-run when it's updated
function track(target, key) {
  // No need to make sure this effect is being tracked.
  let depsMap = targetMap.get(target) // Get the current depsMap for this target
  if (!depsMap) {
    // There is no map.
    targetMap.set(target, (depsMap = new Map())) // Create one
  }
  let dep = depsMap.get(key) // Get the current dependencies (effects) that need to be run when this is set
  if (!dep) {
    // There is no dependencies (effects)
    depsMap.set(key, (dep = new Set())) // Create a new Set
  }
  dep.add(effect) // Add effect to dependency map
}
function trigger(target, key) {
  const depsMap = targetMap.get(target) // Does this object have any properties that have dependencies (effects)
  if (!depsMap) {
    return
  }
  let dep = depsMap.get(key) // If there are dependencies (effects) associated with this
  if (dep) {
    dep.forEach(effect => {
      // run them all
      effect()
    })
  }
}

function reactive(target) {
  const handler = {
    get(target, key, receiver) {
      let result = Reflect.get(target, key, receiver)
      track(target, key) // If this reactive property (target) is GET inside then track the effect to rerun on SET
      return result
    },
    set(target, key, value, receiver) {
      let oldValue = target[key]
      let result = Reflect.set(target, key, value, receiver)
      if (result && oldValue != value) {
        trigger(target, key) // If this reactive property (target) has effects to rerun on SET, trigger them
      }
      return result
    }
  }
  return new Proxy(target, handler)
}

let product = reactive({ price: 5, quantity: 2 })
let total = 0

let effect = () => {
  total = product.price * product.quantity
}
effect()

console.log('before updated quantity total = ' + total)
product.quantity = 3
console.log('after updated quantity total = ' + total)
```

Notice how we no longer need to call `trigger` and `track` because these are getting properly called inside our `get` and `set` methods. Running this code gives us:

before updated quantity total = 10

after updated quantity total = 15

Wow, we've come a long way! There's only one bug to fix before this code is solid. Specifically, that we only want `track` to be called on a reactive object if it's inside an `effect`. Right now `track` will be called whenever a reactive object property is `get`. We'll polish this up in the next lesson.