thinkdev #6

Bindings again

Variables

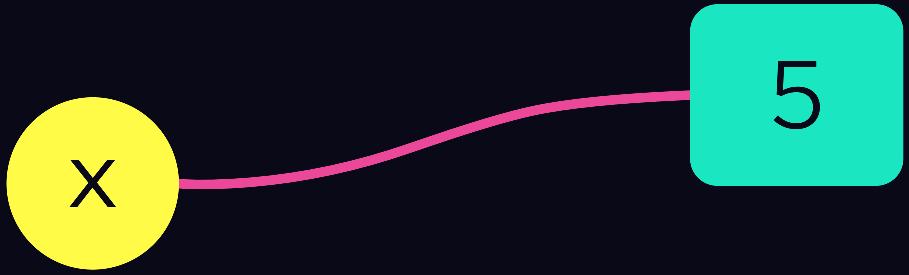
```
let x = 5
```

What does this do again?

```
let x = 5
```

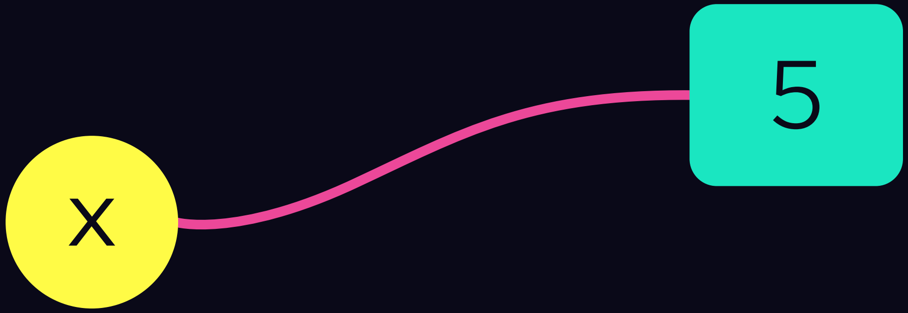
X

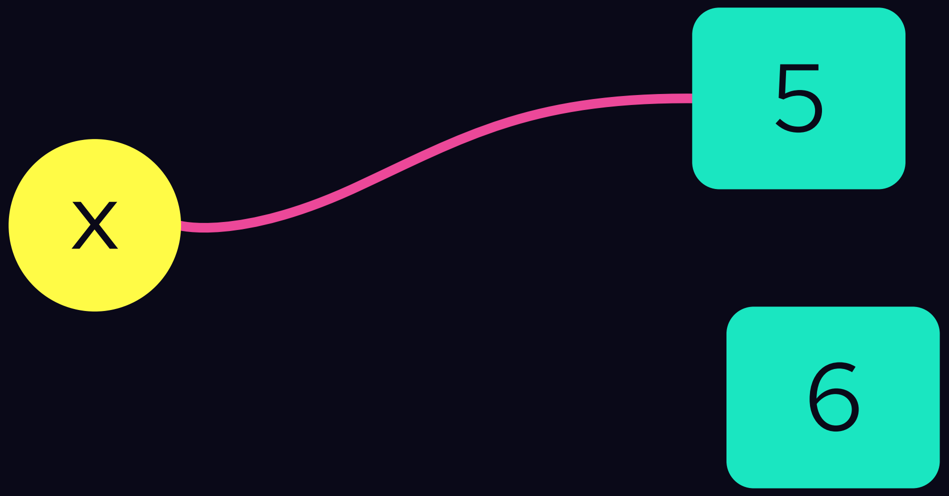
x
5

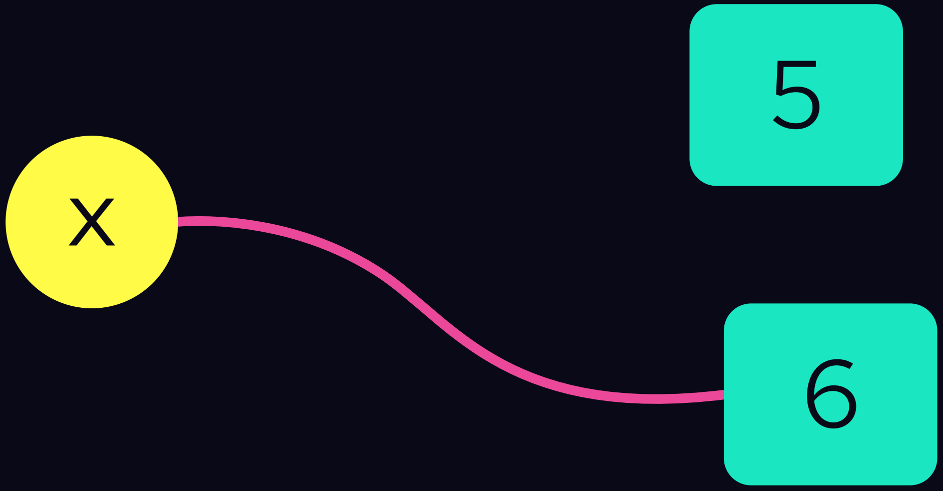

x
5

Because we’re using `let`, we can always bind the name again to another value later:

```
let x = 5
x = 6
```


x
5


x
5
6


x
5
6

We can't do this with `const` because it creates a *constant* binding that cannot be changed:

```
const x = 5
x = 6 // TypeError: Assignment to constant variable
```

After declaring, we can use the name to refer to its bound value:

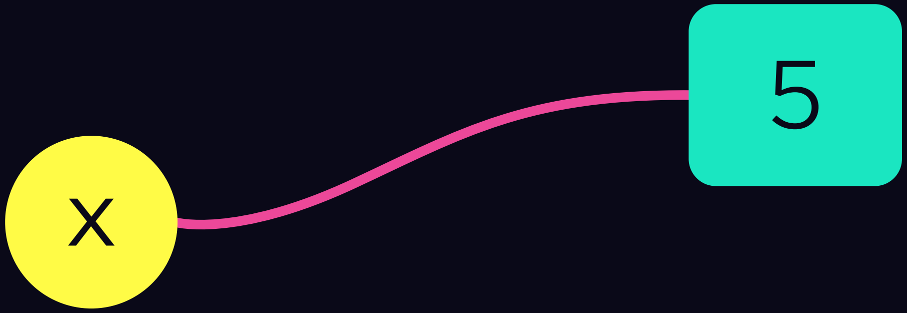
```
let x = 5
console.log(x) // 5
```

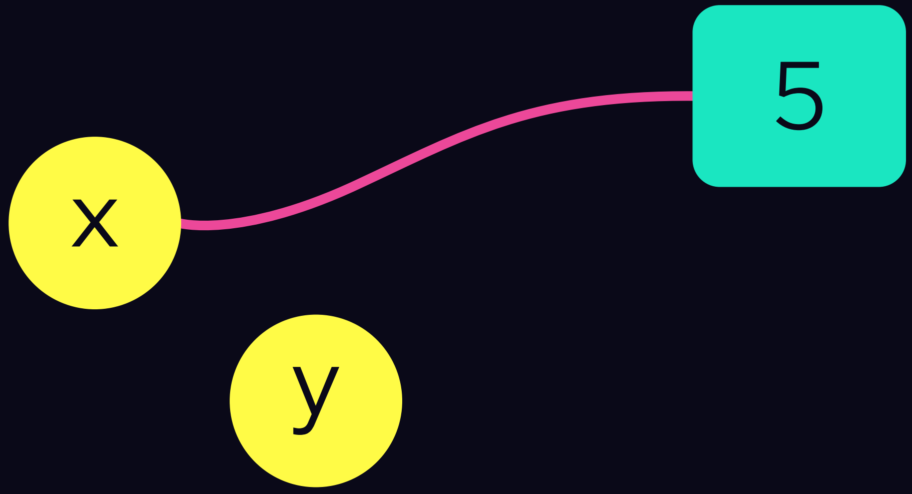
We can operate on the name as we would on the value:

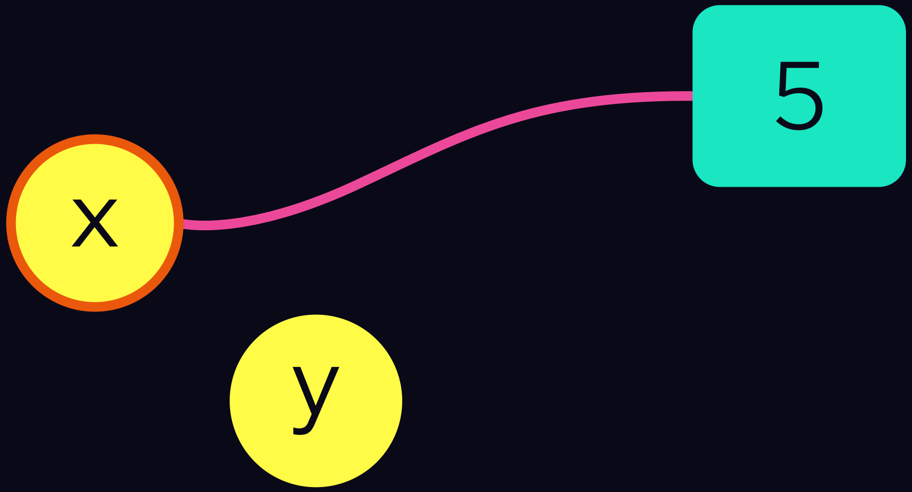
```
let x = 5
console.log(x + 2) // 7
```

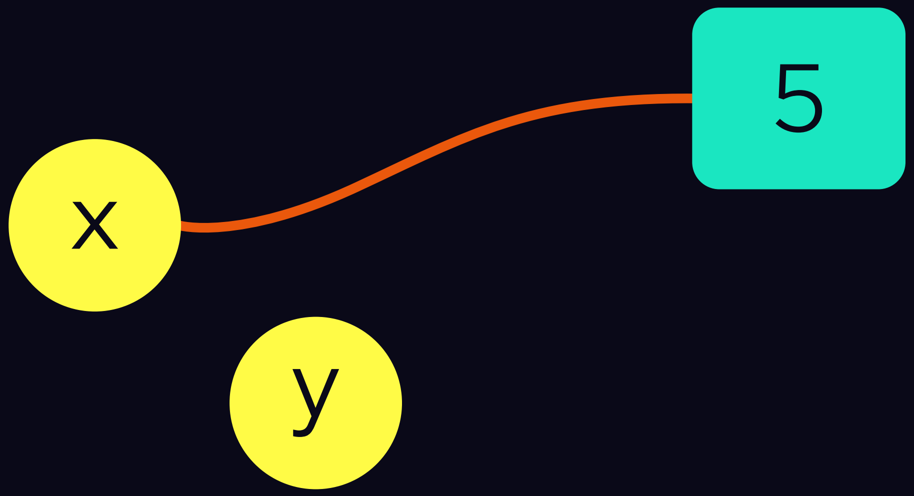
We can also bind a name to the value of another name:

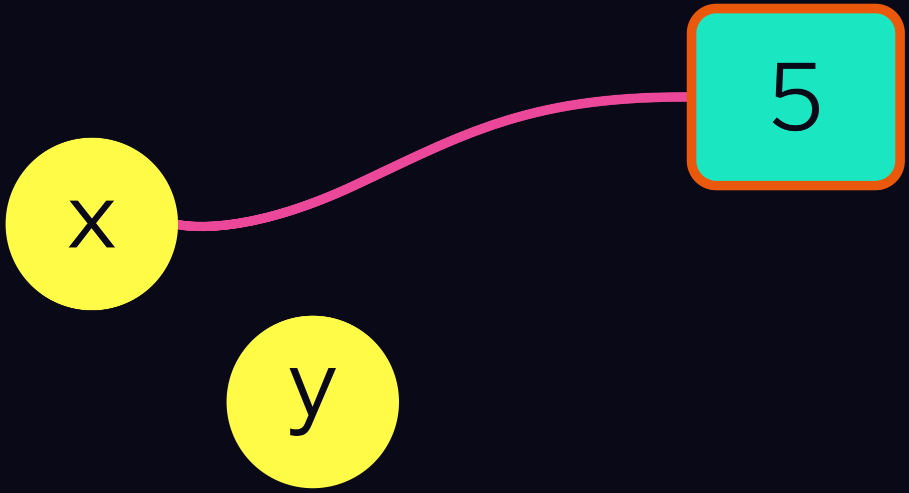
```
let x = 5
let y = x
```

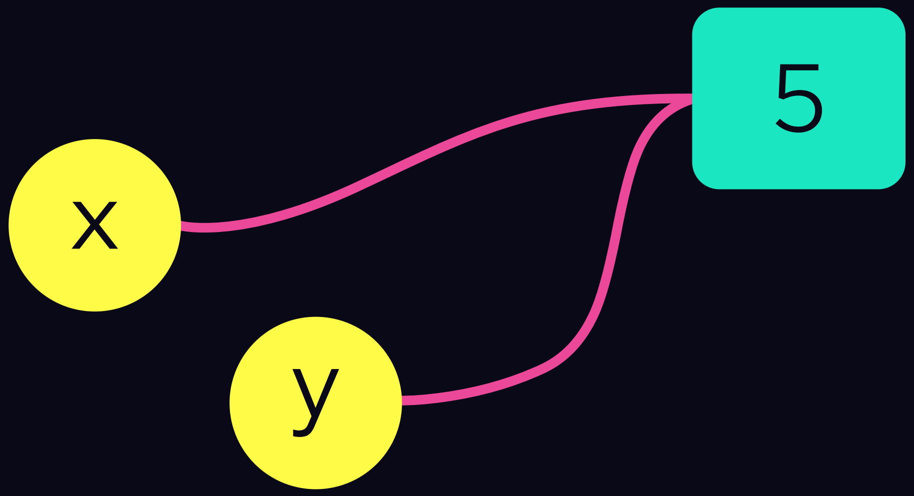

x
5


5
x
y


5
x
y

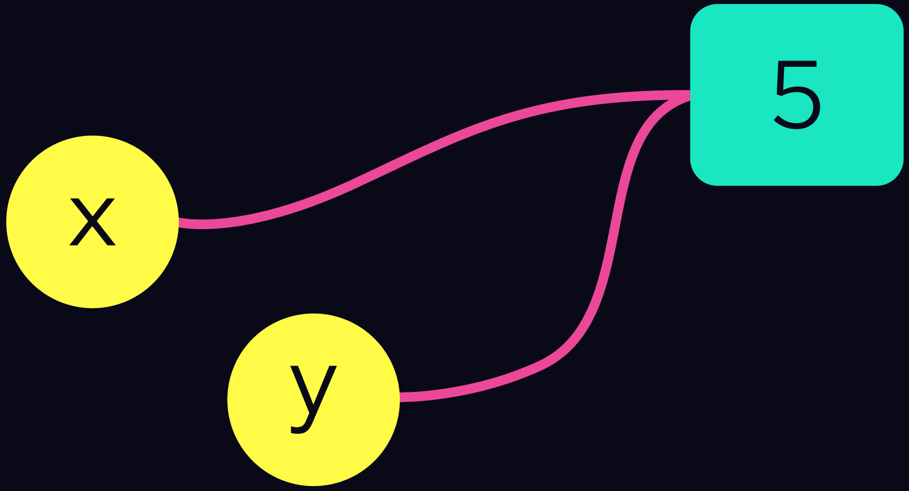

5
x
y

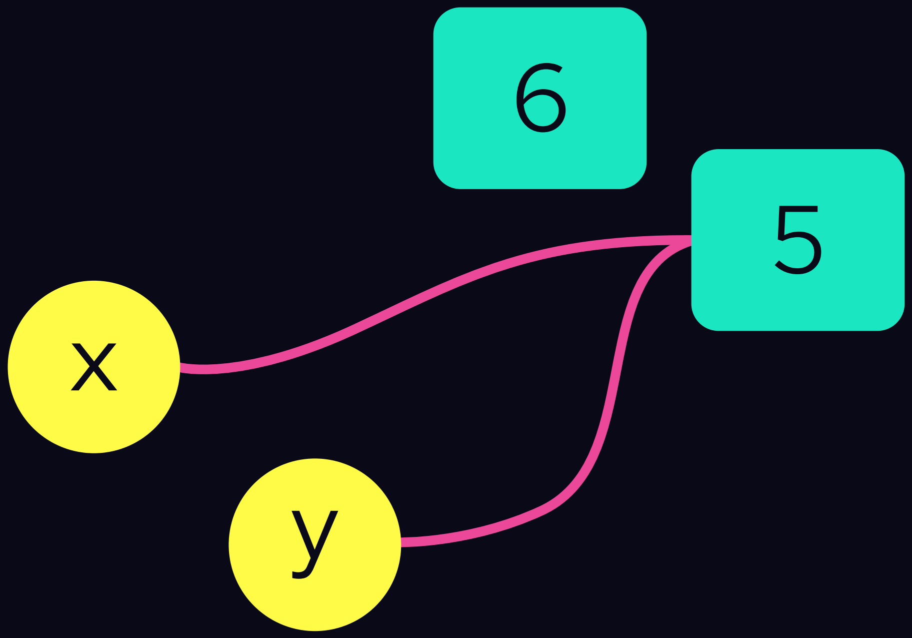

5
x
y

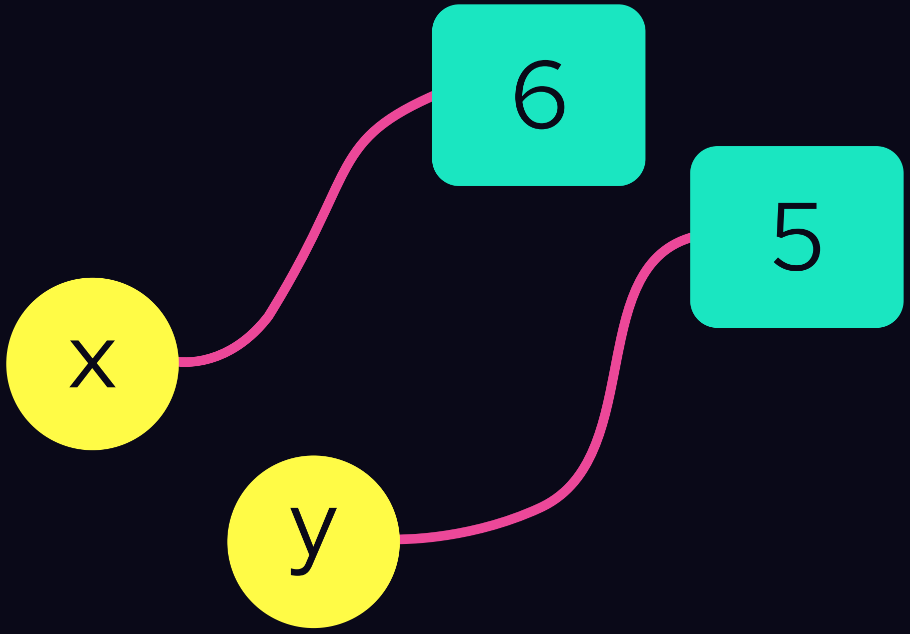

x
y
5

What if we reassign one of them?

```
let x = 5
let y = x
x = 6
```


x
y
5


6
5
x
y


6
5
x
y

Reserved words

You can’t use some words as names because they are reserved for special use in JavaScript.

await	enum	instanceof	this
break	export	let	throw
case	extends	new	true
catch	false	null	try
class	finally	package	typeof
const	for	private	var
continue	function	protected	void
debugger	if	public	while
default	implements	return	with
delete	import	static	yield
do	interface	super	
else	in	switch	

You don’t have to memorise them;
JavaScript will complain if you use any of them as a name:

```
let if = 5 // SyntaxError: Unexpected token 'if'
```

Now, let's try something different

```
let x = 5
```

What does this do?

```
let x = 5
let x = 6
```

You can't redeclare an existing variable *in the same scope*.

```
let x = 5
let x = 6 // SyntaxError
```

Scope

```
let x = 5
if (true) {
  let x = 6
}
```

This is allowed because the braces create a *block* and the block in turn creates a new *scope* for its variables.

```
let x = 5
if (true) {
  let x = 6
}
```

In that scope, the new x *shadows* the old one:

```
let x = 5
if (true) {
  let x = 6
  console.log(x) // 6
}
```

But only in that scope:

```
let x = 5
if (true) {
  let x = 6
  console.log(x) // 6
}
console.log(x) // 5
```

We say that variables declared in a block are *local* to the block:

```
let x = 5
if (true) {
  let x = 6 // x is local to the block
  console.log(x)
}
```

JavaScript also has a *global scope* containing several built-in bindings. Some global bindings we’ve already used are `console`, `Number`, `String`, and `Boolean`.

Variables are *visible* in their scope and in inner scopes:

```
let x = 5
if (true) {
  console.log(x) // 5
}
```

You can’t access a variable where it’s not visible:

```
if (true) {
  let x = 6
}
console.log(x) // ReferenceError
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x)
    console.log(y)
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x)
    console.log(y)
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x)
    console.log(y)
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x)
    console.log(y)
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x) // 6
    console.log(y)
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x) // 6
    console.log(y)
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x) // 6
    console.log(y)
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x) // 6
    console.log(y)
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x) // 6
    console.log(y)
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x) // 6
    console.log(y) // 7
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x) // 6
    console.log(y) // 7
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x) // 6
    console.log(y) // 7
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x) // 6
    console.log(y) // 7
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x) // 6
    console.log(y) // 7
    let z = 8
  }
  console.log(z)
}
```

The same rules apply to deeper scopes:

```
let x = 5
let y = 7
if (true) {
  let x = 6
  if (true) {
    console.log(x) // 6
    console.log(y) // 7
    let z = 8
  }
  console.log(z) // ReferenceError
}
```

A final note is that you can’t use a variable before it is declared, even in the same scope:

```
console.log(x) // ReferenceError
let x = 5
```