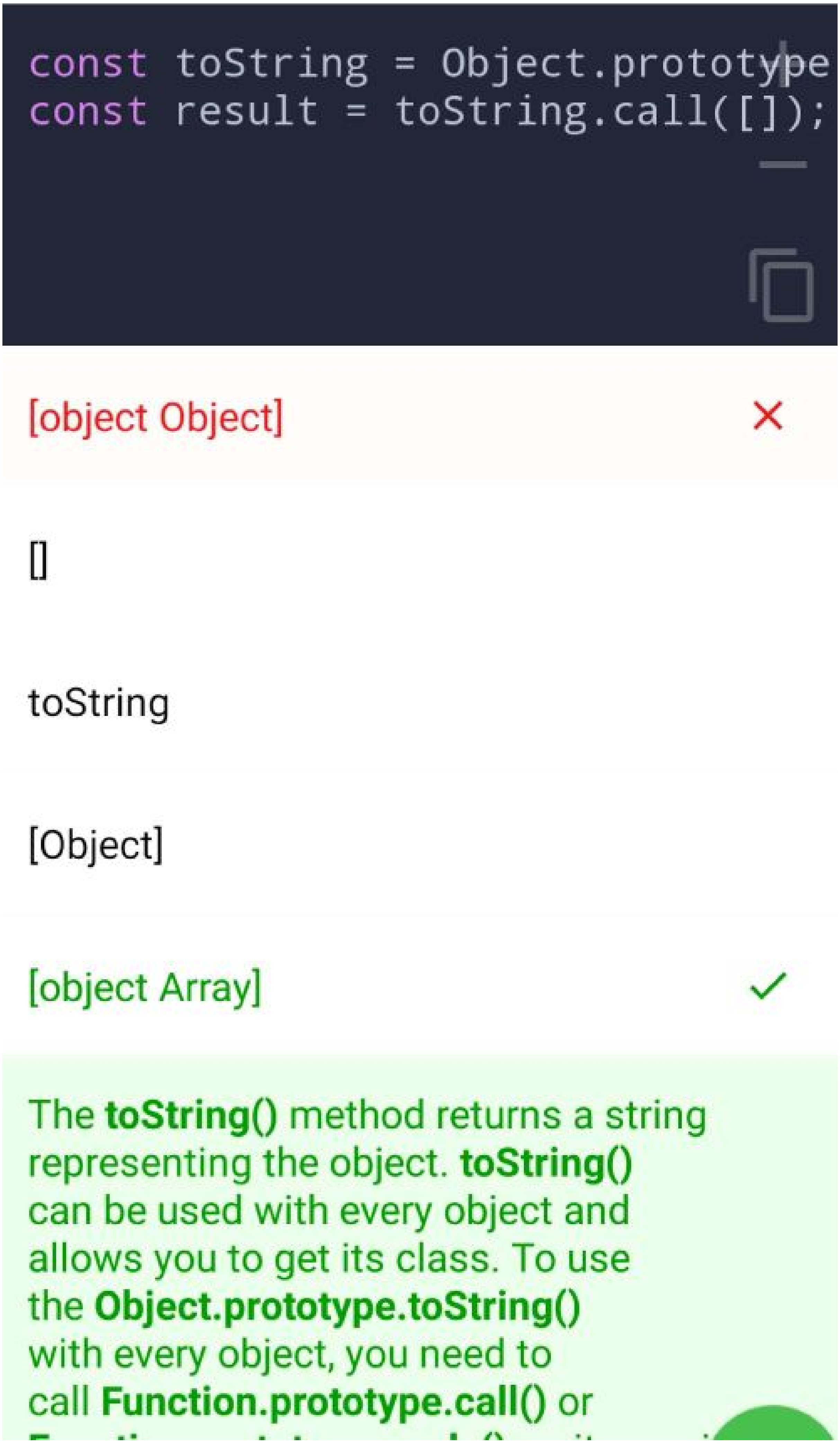
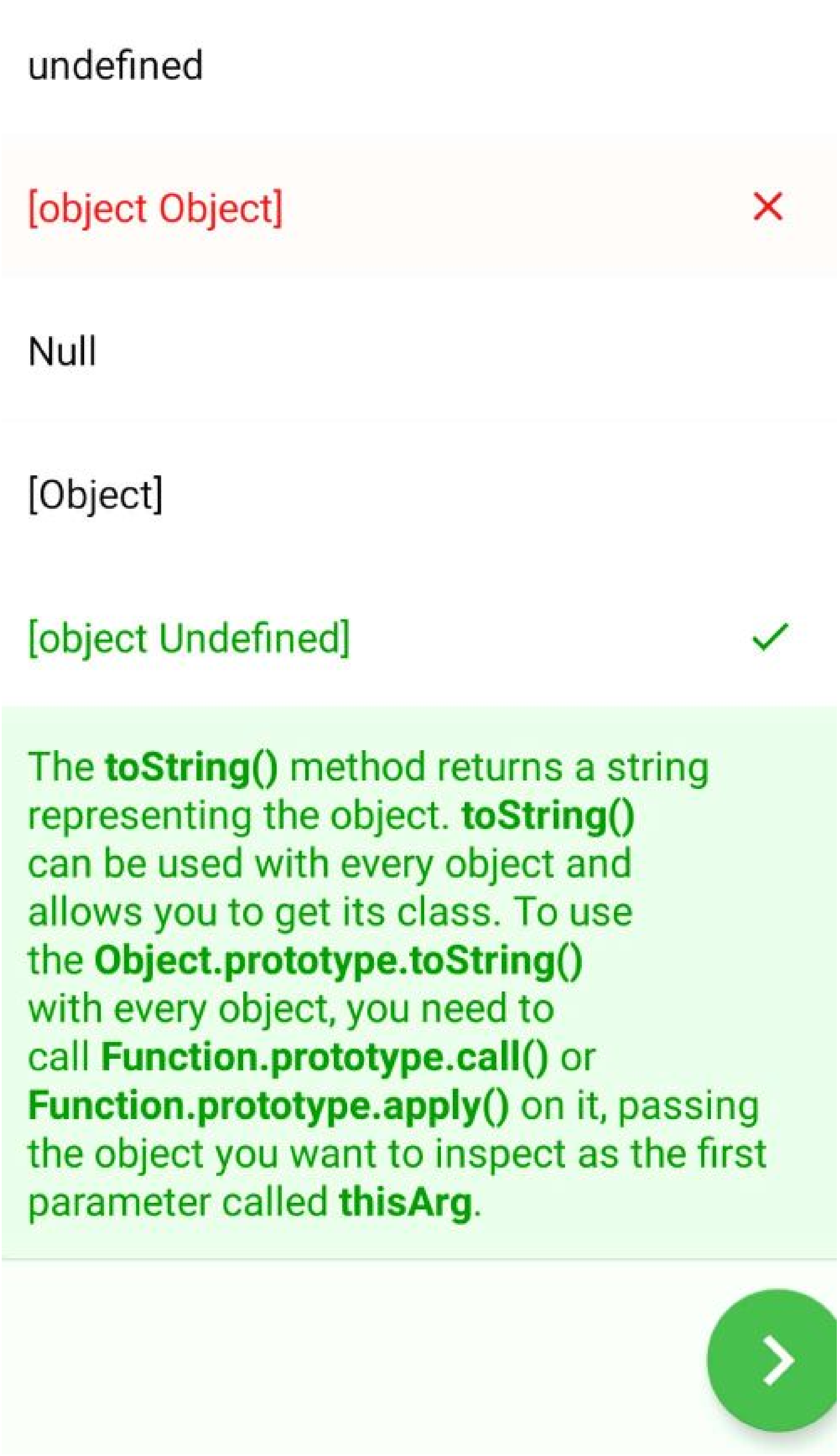


```
const toString = Object.prototype
const result = toString.call([]);
```

- [object Object] ✗
- []
- toString
- [Object]
- [object Array] ✓

The **toString()** method returns a string representing the object. **toString()** can be used with every object and allows you to get its class. To use the **Object.prototype.toString()** with every object, you need to call **Function.prototype.call()** or

Scanned by CamScanner



undefined

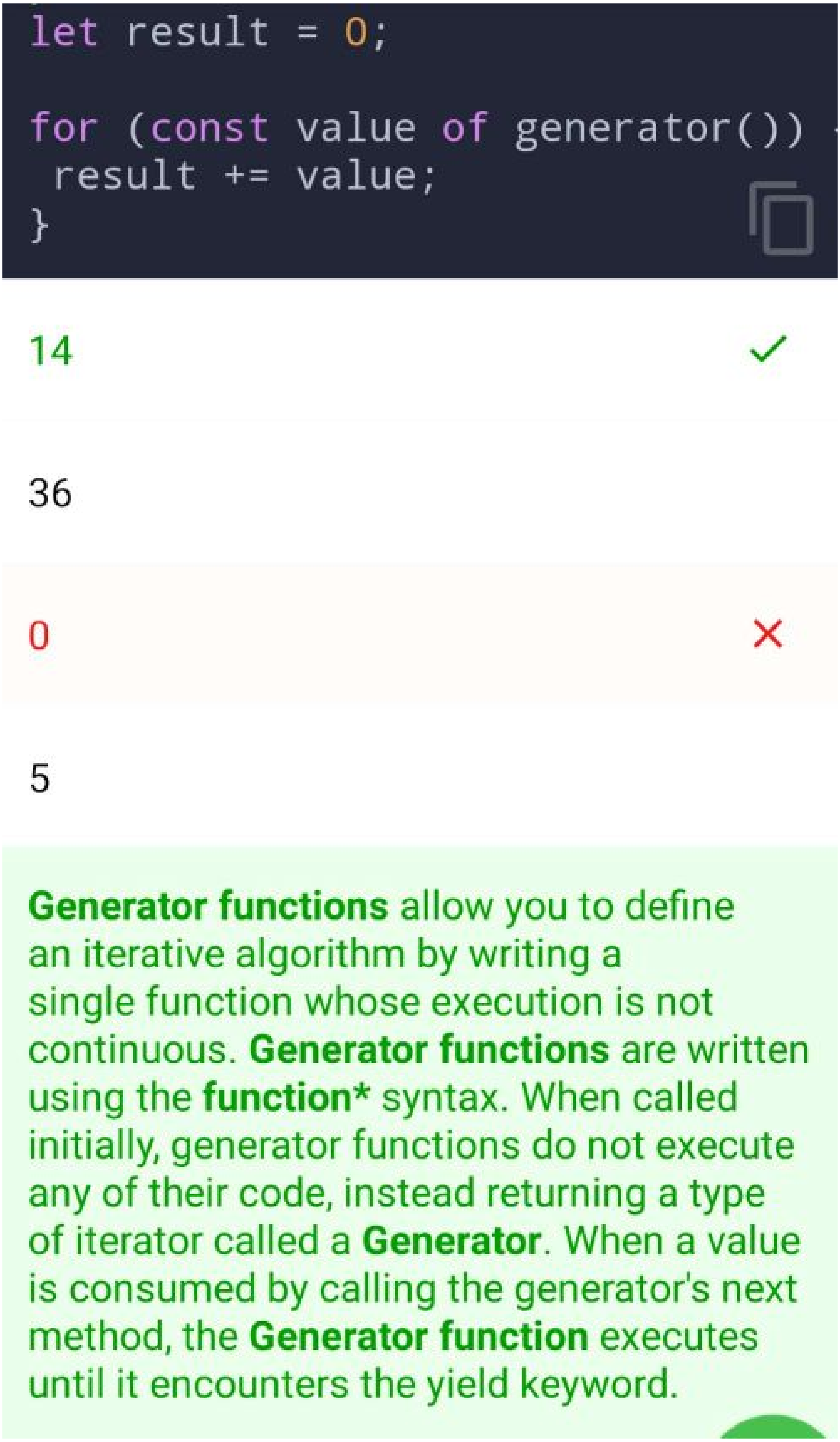
[object Object] ✗

Null

[Object]

[object Undefined] ✓

The **toString()** method returns a string representing the object. **toString()** can be used with every object and allows you to get its class. To use the **Object.prototype.toString()** with every object, you need to call **Function.prototype.call()** or **Function.prototype.apply()** on it, passing the object you want to inspect as the first parameter called **thisArg**.



```
let result = 0;

for (const value of generator())
  result += value;
}
```

14 ✓

36

0 ✗

5

Generator functions allow you to define an iterative algorithm by writing a single function whose execution is not continuous. **Generator functions** are written using the **function*** syntax. When called initially, generator functions do not execute any of their code, instead returning a type of iterator called a **Generator**. When a value is consumed by calling the generator's next method, the **Generator function** executes until it encounters the yield keyword.

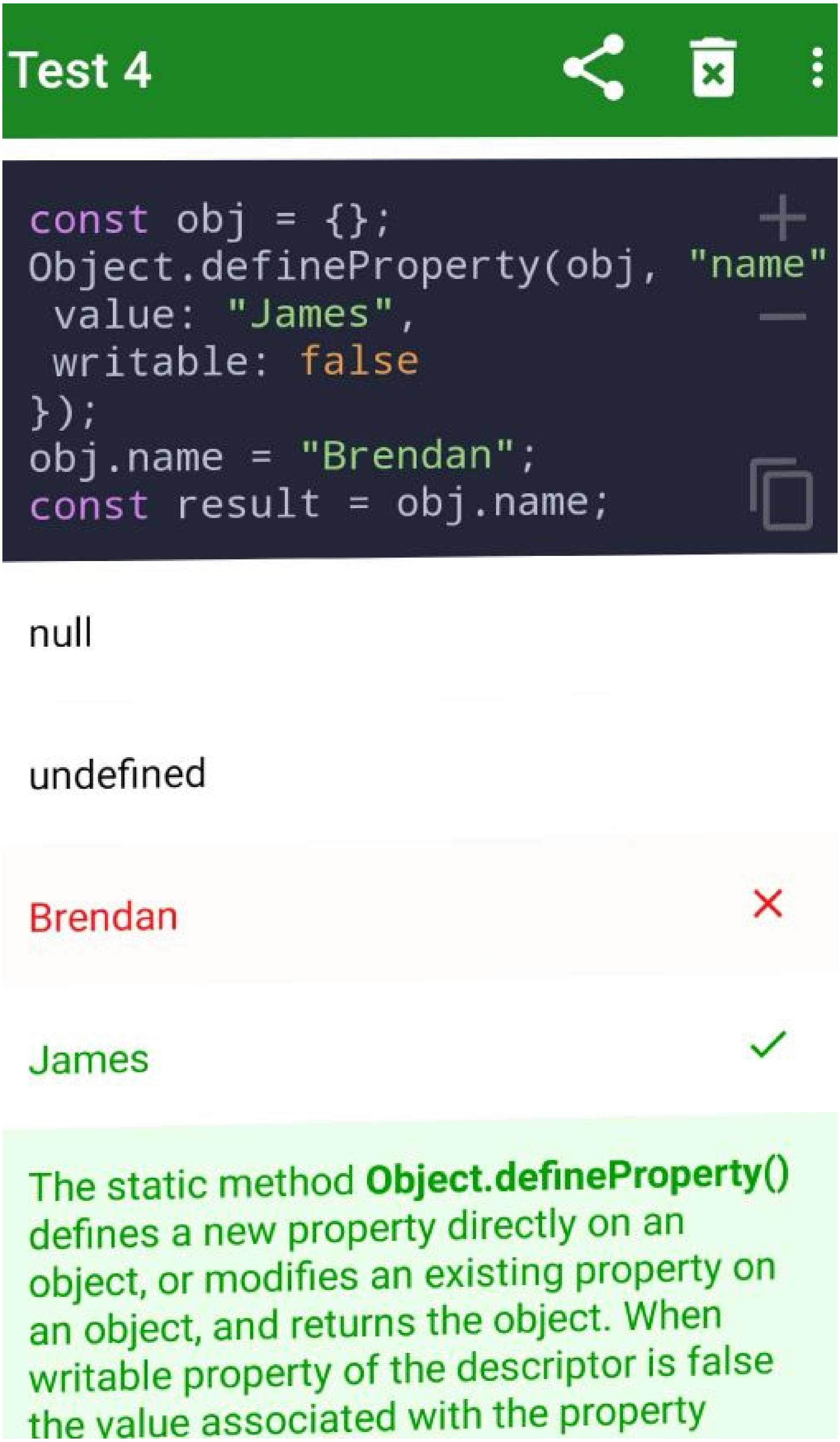
000123 ✗

123123 ✓

456456

123000

The **copyWithin()** method takes up to three arguments target, **start** and end. The **copyWithin()** method shallow copies part of an array to another location in the same array and returns it, without modifying its size. **target** is zero based index at which to copy the sequence to. **start** is zero based index at which to **start** copying elements from. If **start** is omitted, copyWithin will copy from the **start** (defaults to 0). **end** is zero based index at which to **end** copying elements from. **copyWithin()** copies up to but not including end. If **end** is omitted, copyWithin will copy until the **end** (default to **array.length**).



Test 4

```
const obj = {};
Object.defineProperty(obj, "name"
 value: "James",
 writable: false
});
obj.name = "Brendan";
const result = obj.name;
```

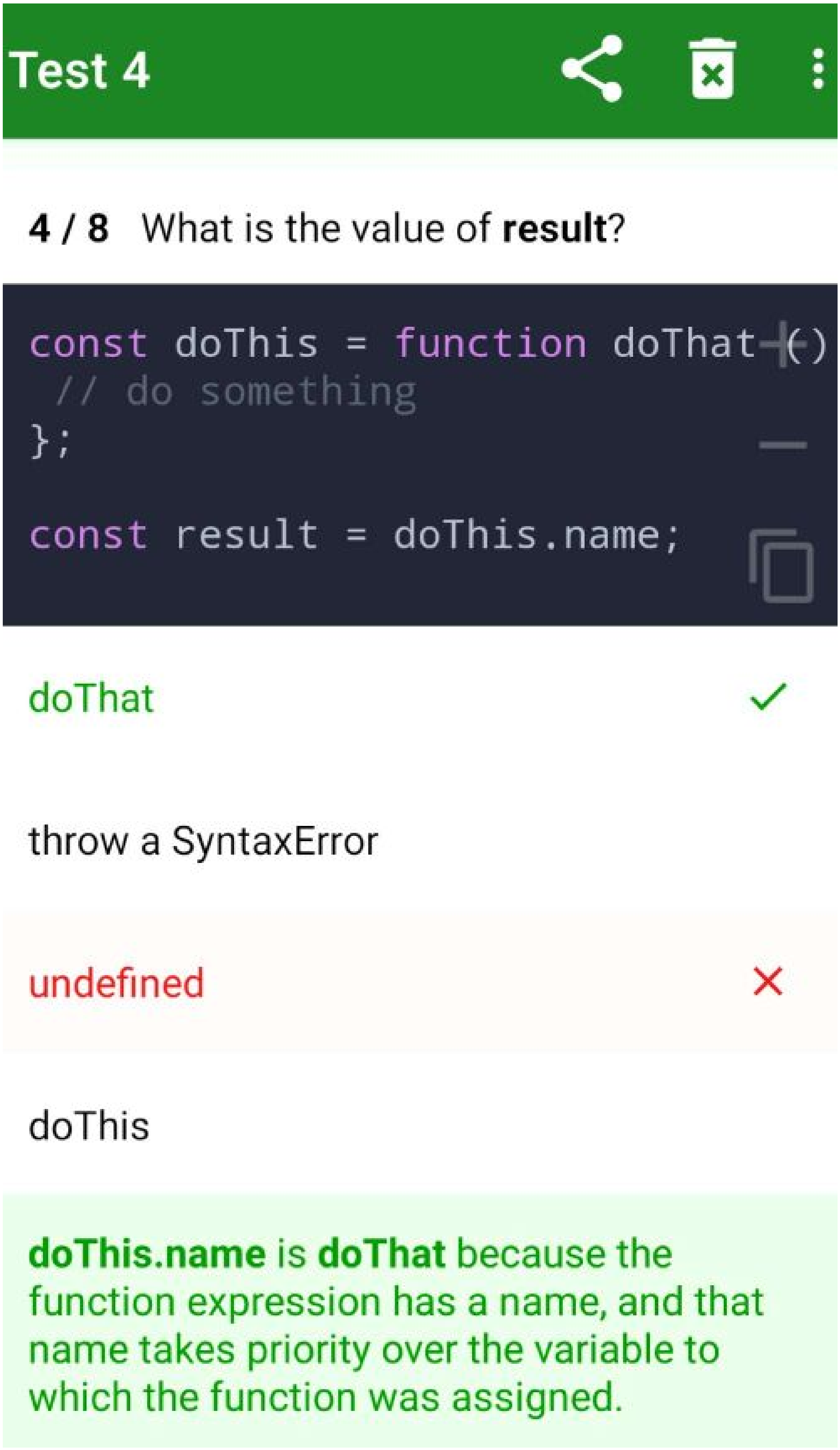
null

undefined

Brendan ✗

James ✓

The static method **Object.defineProperty()** defines a new property directly on an object, or modifies an existing property on an object, and returns the object. When writable property of the descriptor is false the value associated with the property



Test 4

4 / 8 What is the value of **result**?

```
const doThis = function doThat()
  // do something
};

const result = doThis.name;
```

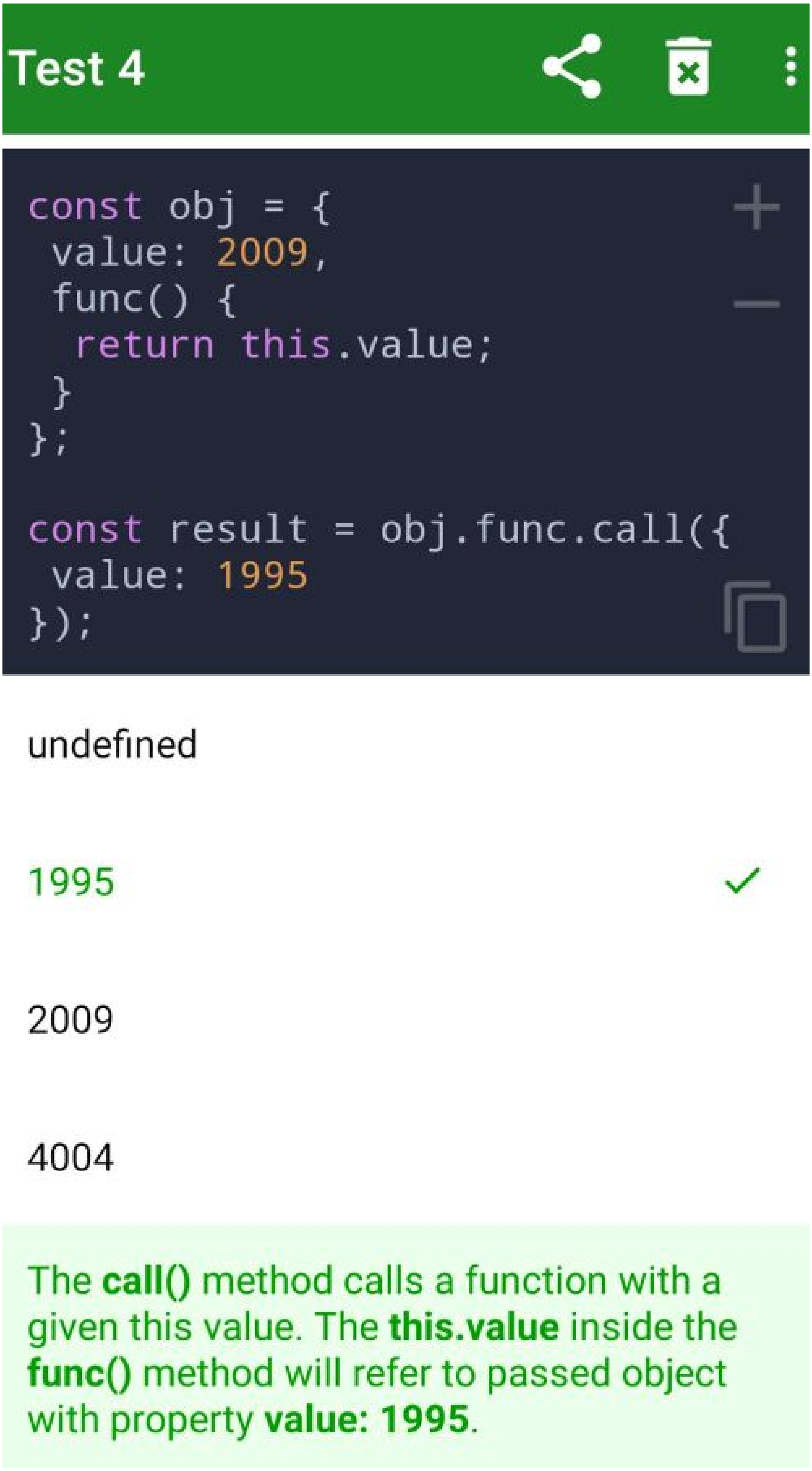
doThat ✓

throw a SyntaxError

undefined ✗

doThis

doThis.name is **doThat** because the function expression has a name, and that name takes priority over the variable to which the function was assigned.



Test 4

```
const obj = {
 value: 2009,
 func() {
  return this.value;
 }
};

const result = obj.func.call({
 value: 1995
});
```

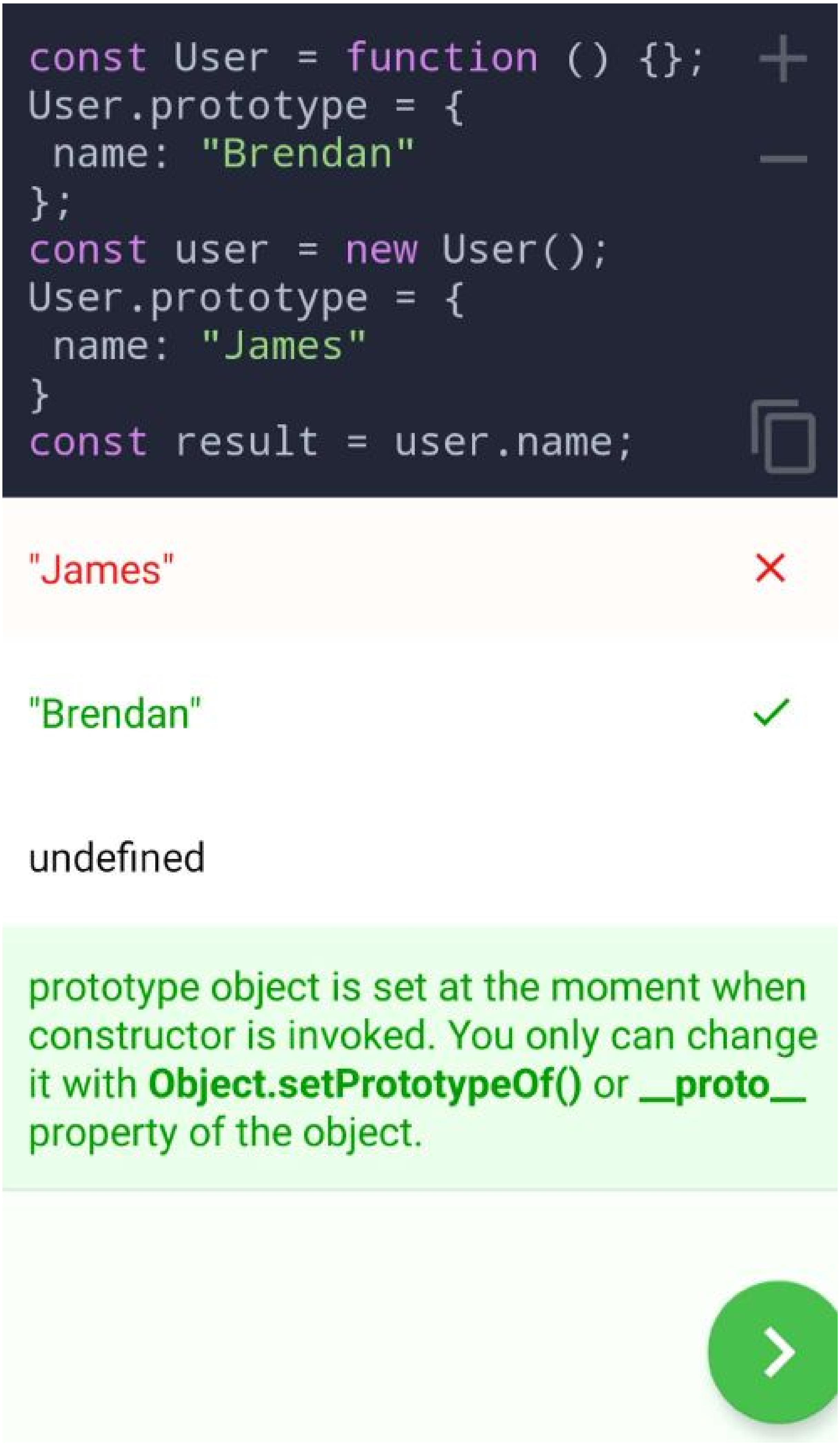
undefined

1995 ✓

2009

4004

The **call()** method calls a function with a given this value. The **this.value** inside the **func()** method will refer to passed object with property **value: 1995**.



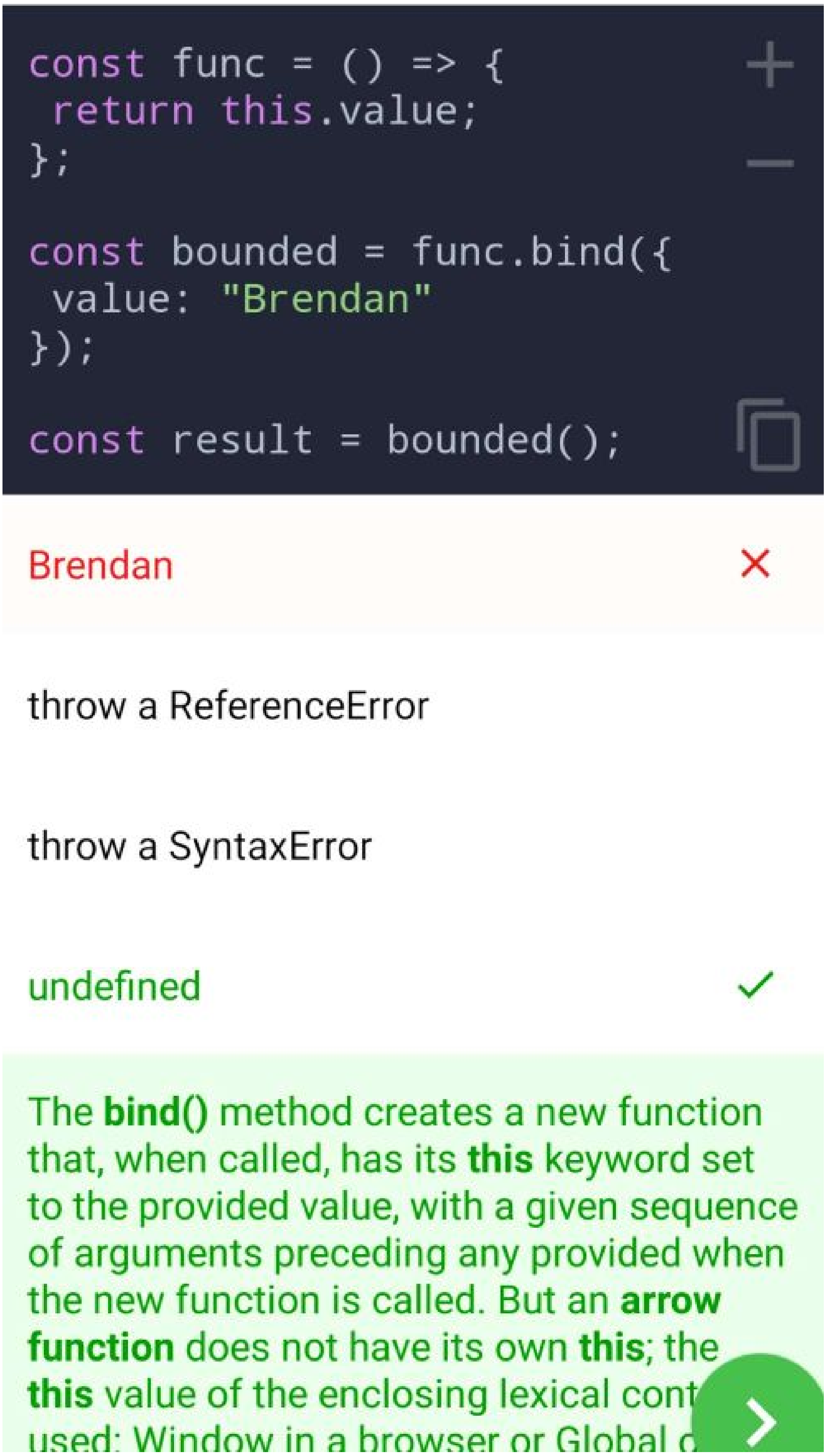
```
const User = function () {};
User.prototype = {
 name: "Brendan"
};
const user = new User();
User.prototype = {
 name: "James"
}
const result = user.name;
```

"James" ✗

"Brendan" ✓

undefined

prototype object is set at the moment when constructor is invoked. You only can change it with **Object.setPrototypeOf()** or **__proto__** property of the object.



```
const func = () => {
 return this.value;
};

const bounded = func.bind({
 value: "Brendan"
});

const result = bounded();
```

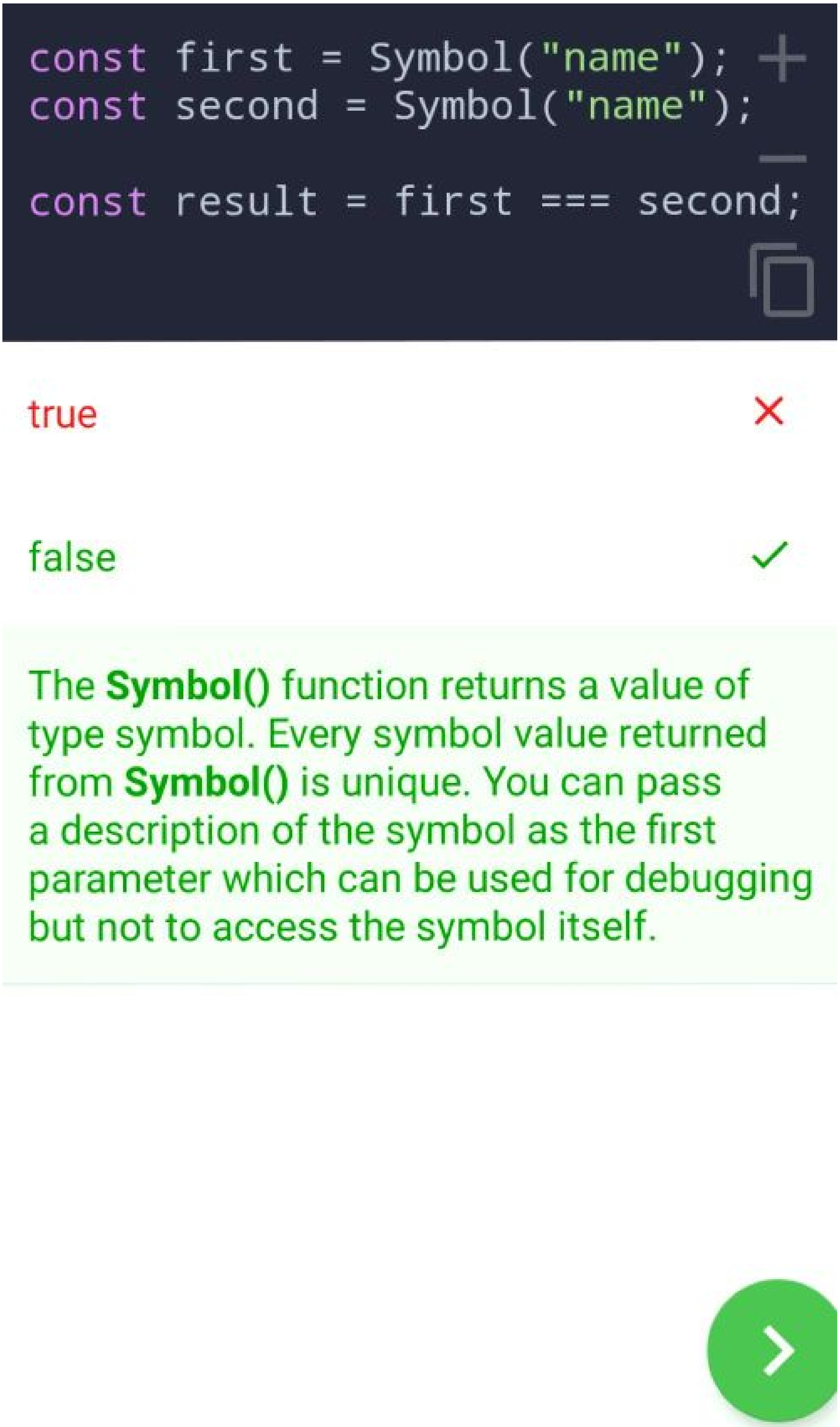
Brendan ✗

throw a ReferenceError

throw a SyntaxError

undefined ✓

The **bind()** method creates a new function that, when called, has its **this** keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called. But an **arrow function** does not have its own **this**; the **this** value of the enclosing lexical cont
used: Window in a browser or Global c

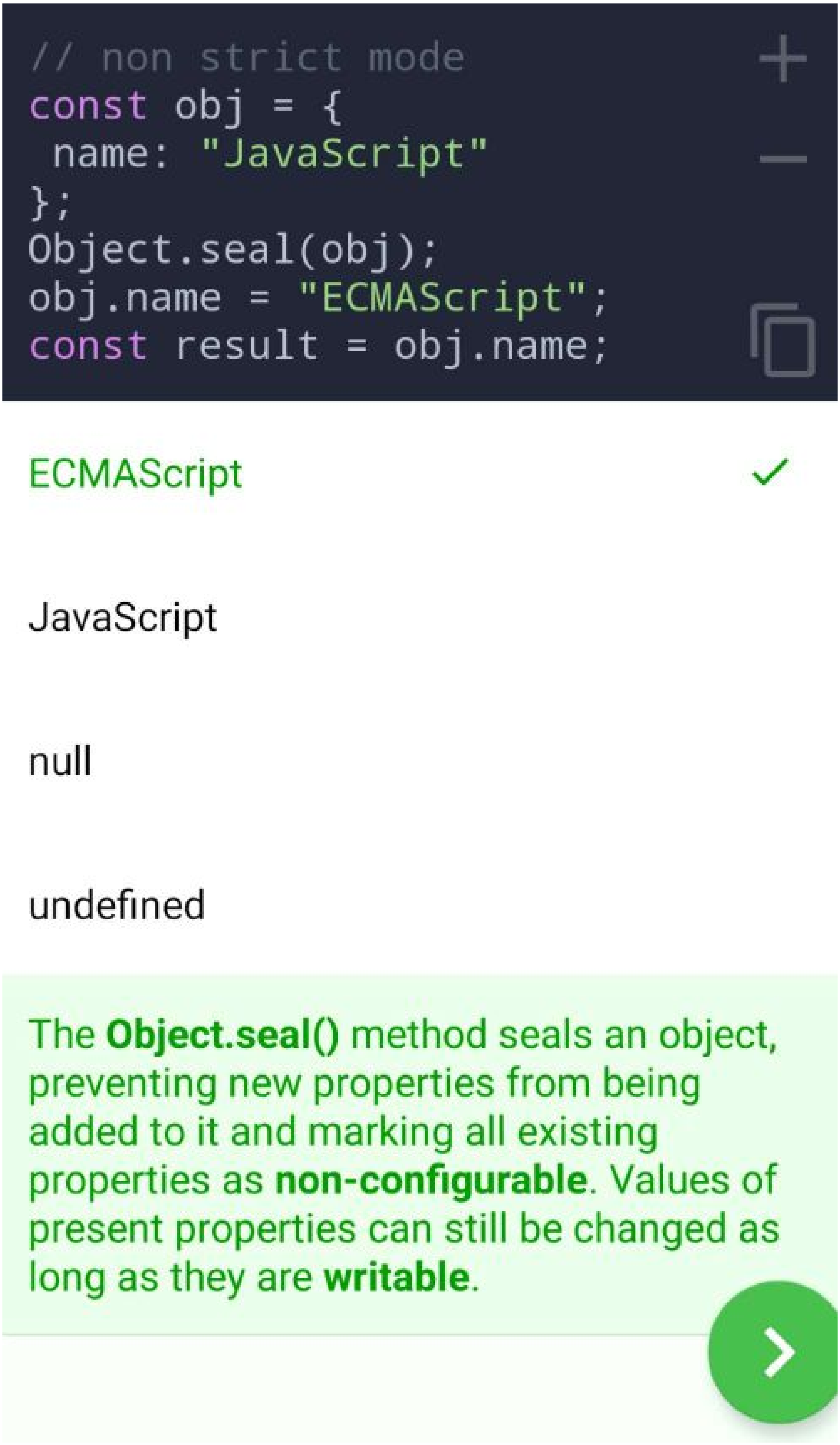


```
const first = Symbol("name");
const second = Symbol("name");

const result = first === second;
```

true ✗

false ✓

The **Symbol()** function returns a value of type symbol. Every symbol value returned from **Symbol()** is unique. You can pass a description of the symbol as the first parameter which can be used for debugging but not to access the symbol itself.



```
// non strict mode
const obj = {
 name: "JavaScript"
};
Object.seal(obj);
obj.name = "ECMAScript";
const result = obj.name;
```

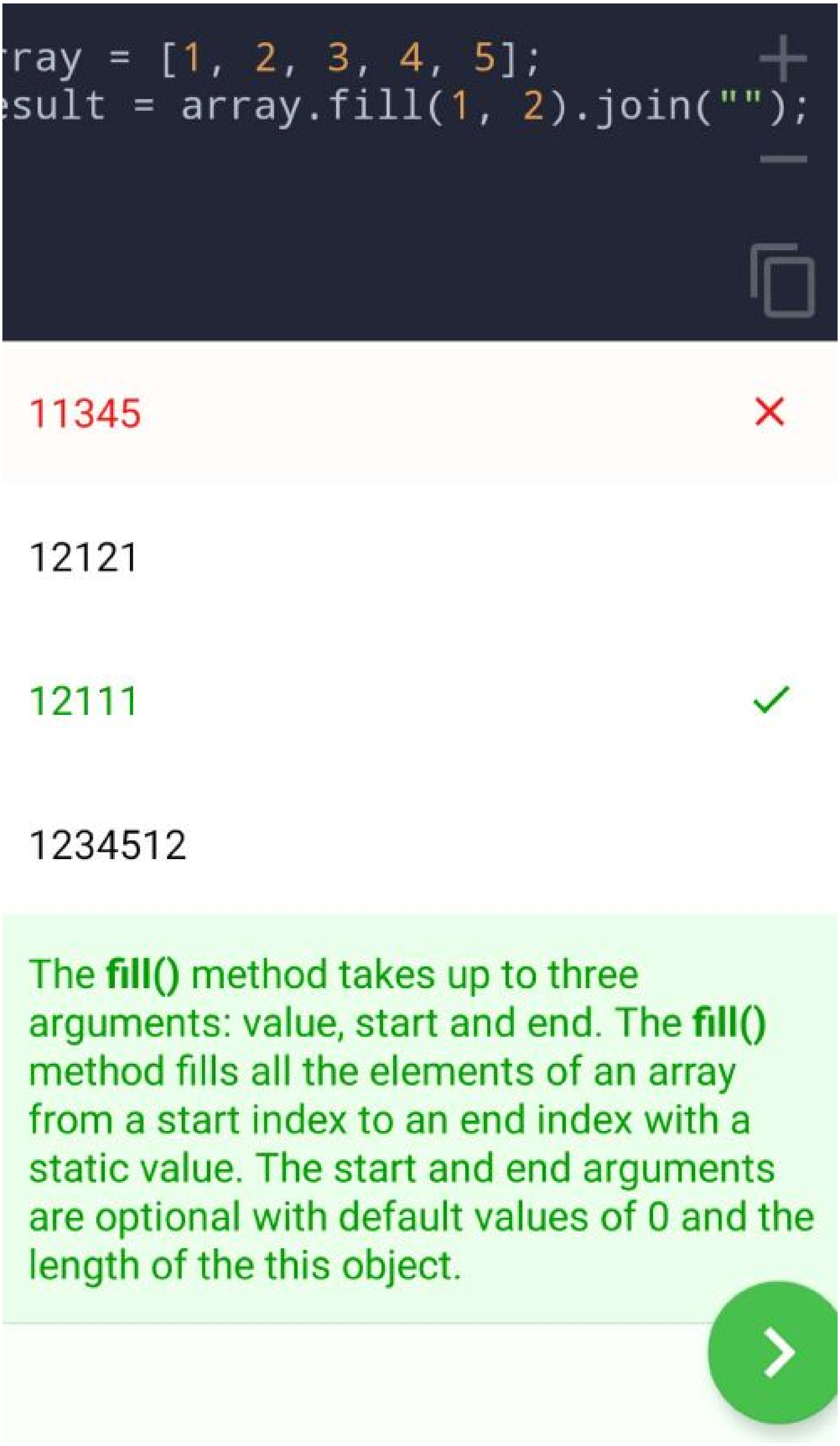
[✓ ECMAScript]

JavaScript

null

undefined

The **Object.seal()** method seals an object, preventing new properties from being added to it and marking all existing properties as **non-configurable**. Values of present properties can still be changed as long as they are **writable**.



```
ray = [1, 2, 3, 4, 5];
esult = array.fill(1, 2).join("");
```

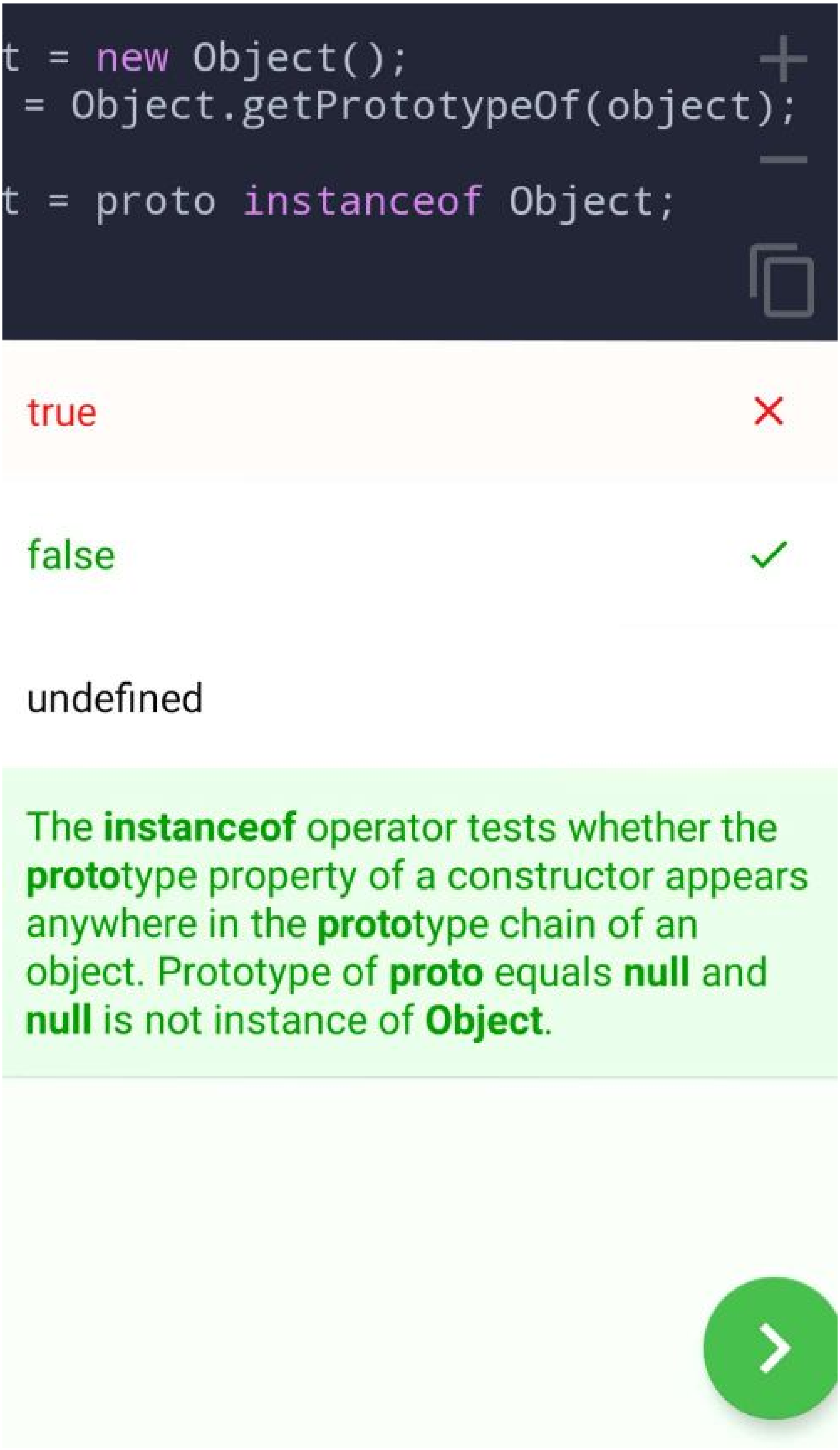
11345 ✗

12121

12111 ✓

1234512

The **fill()** method takes up to three arguments: value, start and end. The **fill()** method fills all the elements of an array from a start index to an end index with a static value. The start and end arguments are optional with default values of 0 and the length of the this object.



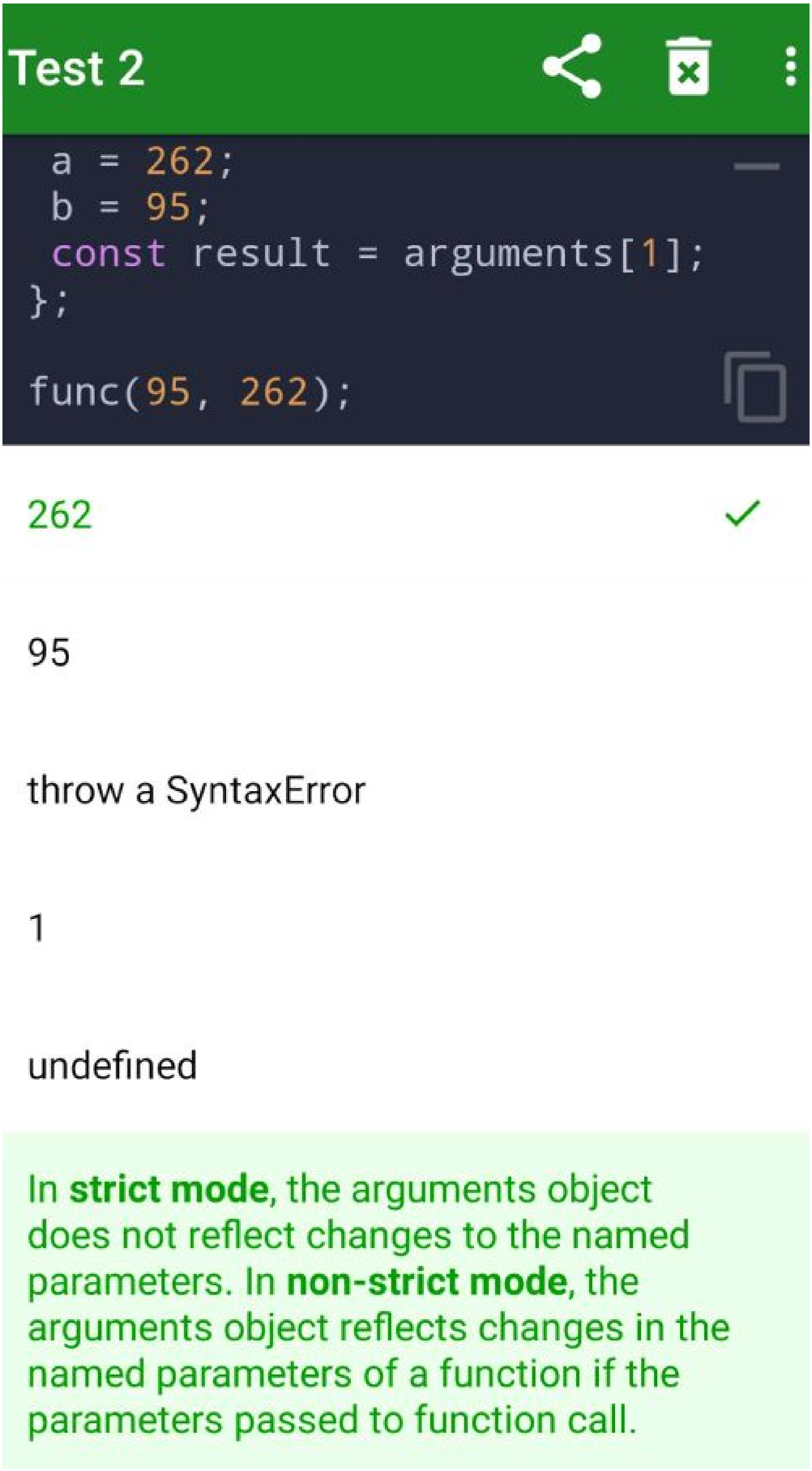
```
t = new Object();
 = Object.getPrototypeOf(object);

t = proto instanceof Object;
```

true ✗

false ✓

undefined

The **instanceof** operator tests whether the **proto**type property of a constructor appears anywhere in the **proto**type chain of an object. Prototype of **proto** equals **null** and **null** is not instance of **Object**.



Test 2

```
 a = 262;
 b = 95;
 const result = arguments[1];
};

func(95, 262);
```

[✓ 262]

95

throw a SyntaxError

1

undefined

In **strict mode**, the arguments object does not reflect changes to the named parameters. In **non-strict mode**, the arguments object reflects changes in the named parameters of a function if the parameters passed to function call.

```
const params = [1, 2, 3, 4, 5];
const result = Math.max.apply(10,
```

6

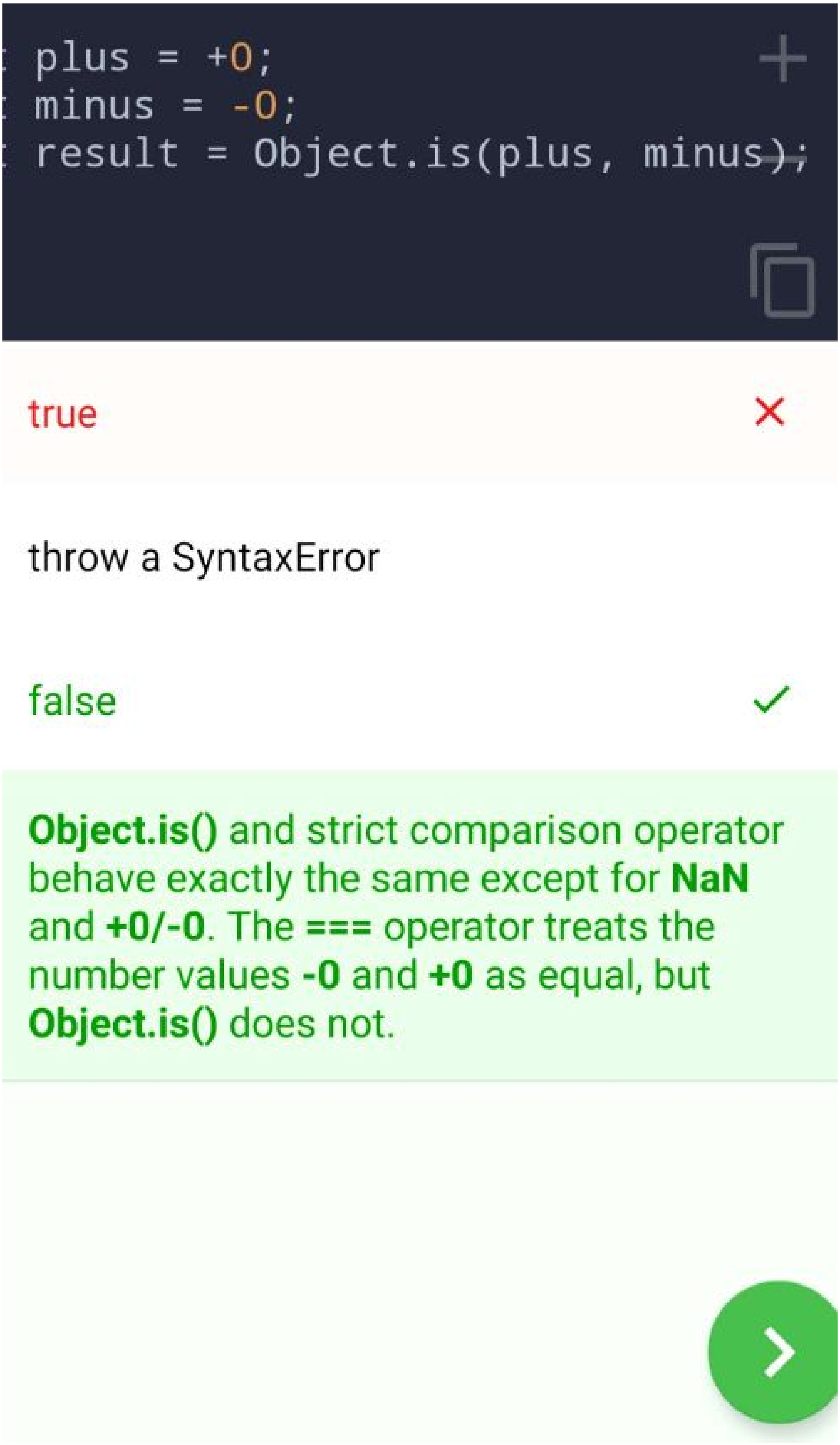
throw a SyntaxError

10 ✗

undefined

5 ✓

The **Math.max()** function returns the largest of zero or more numbers. The **apply()** method calls a function with a given **this** value, and arguments provided as an array. In the case **10** is ignored by the **Math.max()**.



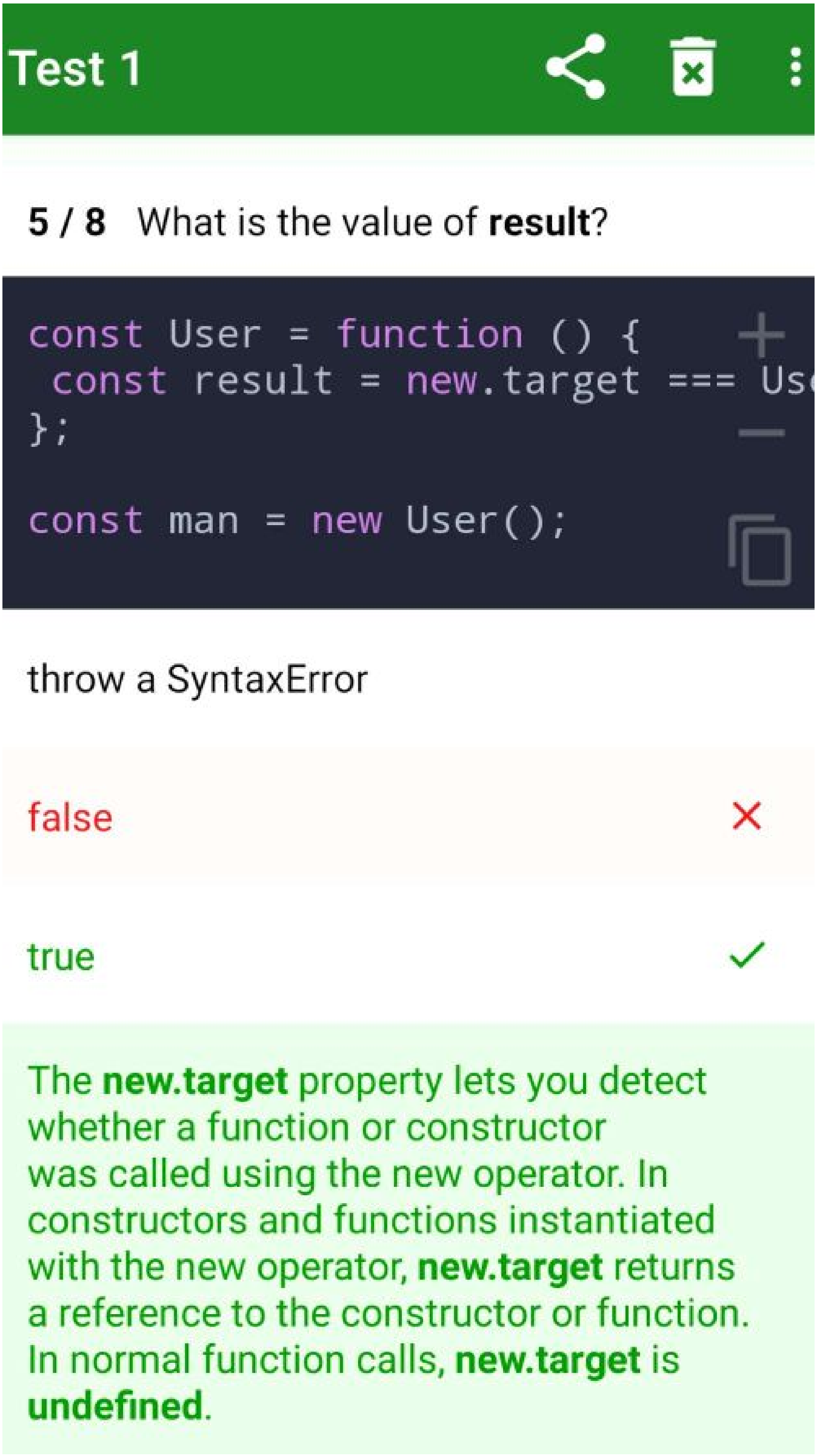
```
plus = +0;
minus = -0;
result = Object.is(plus, minus);
```

true ✗

throw a SyntaxError

false ✓

Object.is() and strict comparison operator behave exactly the same except for **NaN** and **+0/-0**. The **===** operator treats the number values **-0** and **+0** as equal, but **Object.is()** does not.


Test 1

5 / 8 What is the value of **result**?

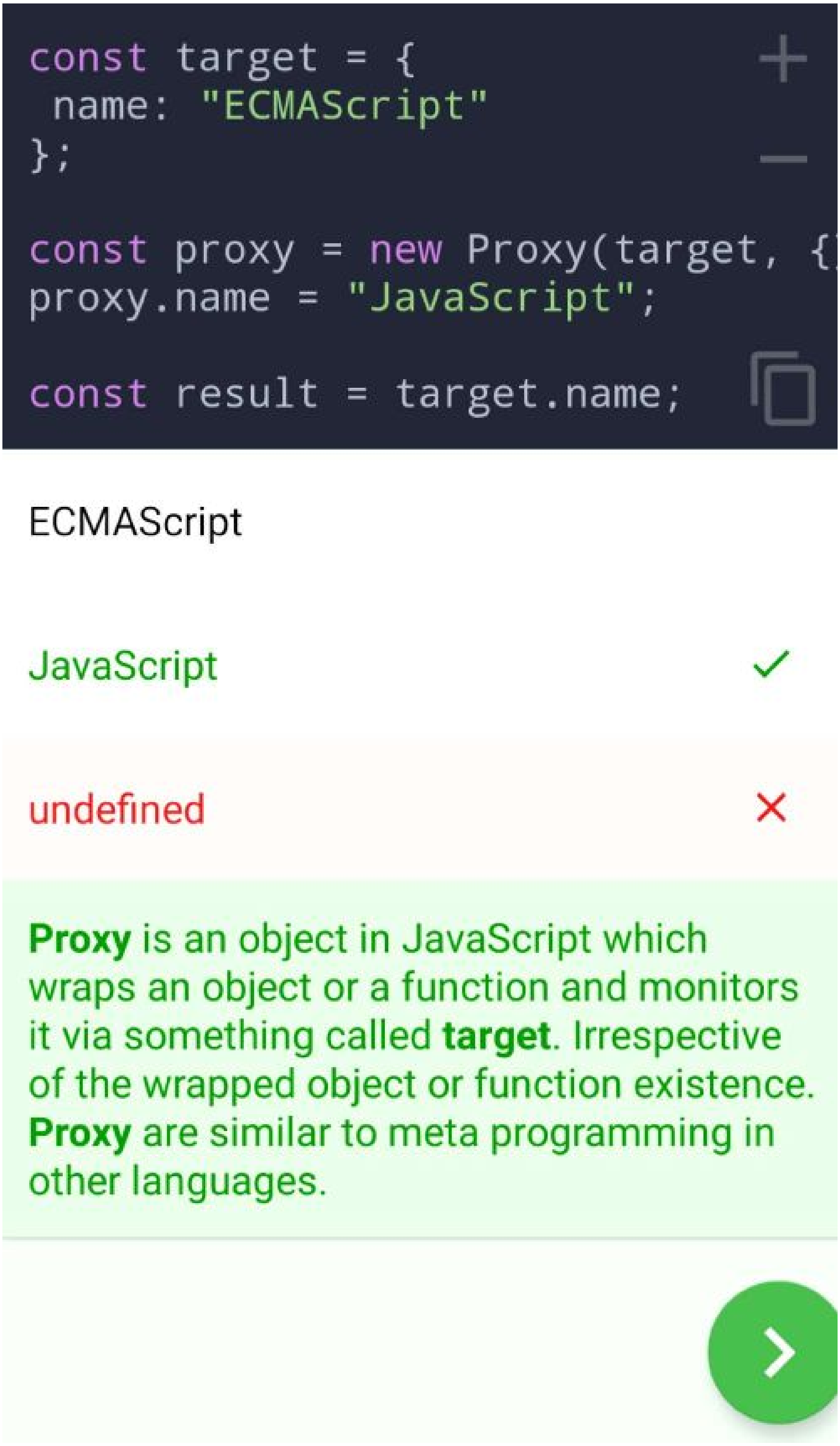
```
const User = function () {
 const result = new.target === Us
};

const man = new User();
```

throw a SyntaxError

false ✗

true ✓

The **new.target** property lets you detect whether a function or constructor was called using the new operator. In constructors and functions instantiated with the new operator, **new.target** returns a reference to the constructor or function. In normal function calls, **new.target** is **undefined**.



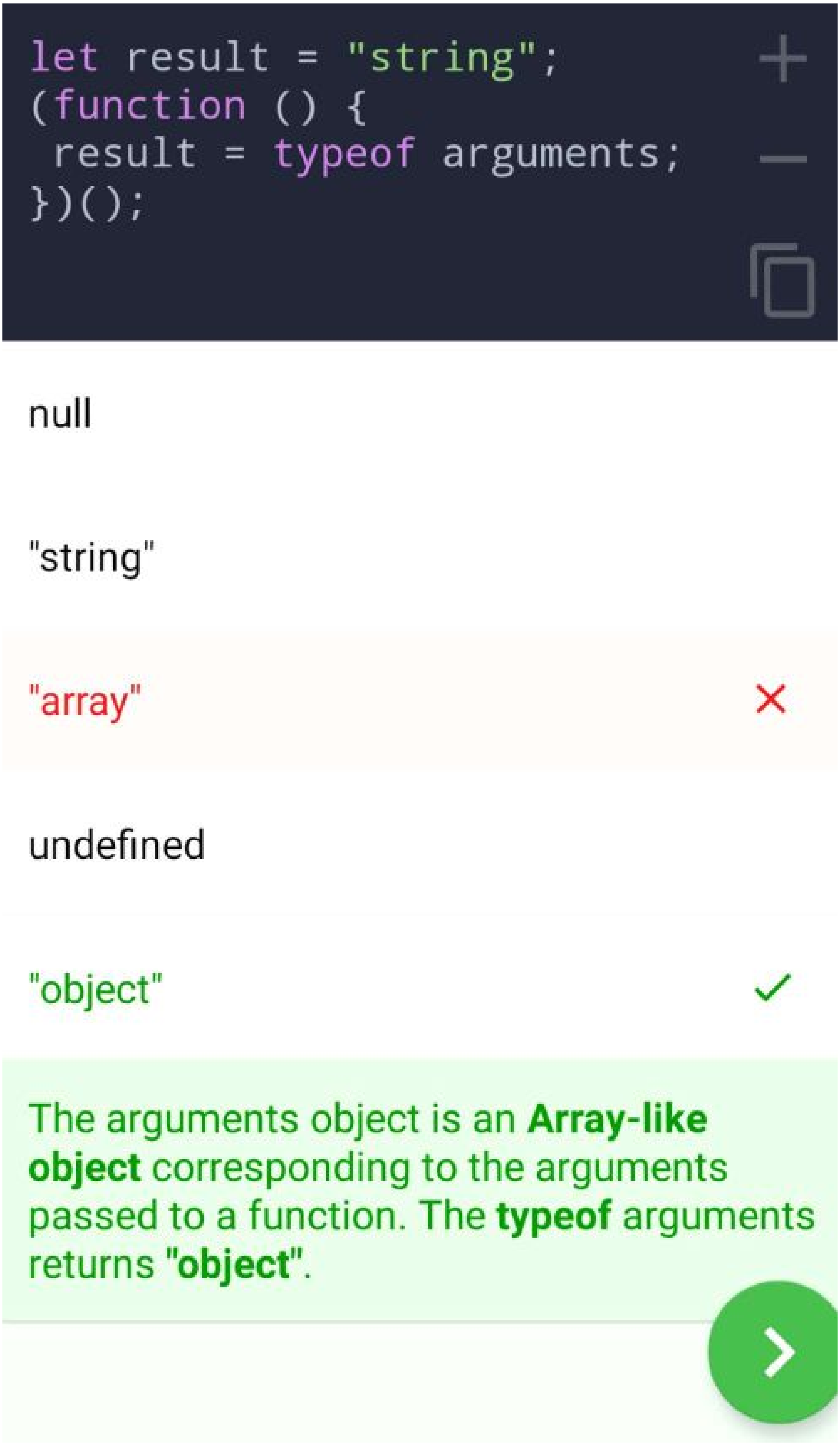
```
const target = {
 name: "ECMAScript"
};

const proxy = new Proxy(target, {
proxy.name = "JavaScript";

const result = target.name;
```

ECMAScript

JavaScript ✓

undefined ✗

Proxy is an object in JavaScript which wraps an object or a function and monitors it via something called **target**. Irrespective of the wrapped object or function existence. **Proxy** are similar to meta programming in other languages.



```
let result = "string";
(function () {
 result = typeof arguments;
})();
```

null

"string"

"array" ✗

undefined

"object" ✓

The arguments object is an **Array-like object** corresponding to the arguments passed to a function. The **typeof** arguments returns **"object"**.

```
const func = function () {
 return arguments.join("");
}
const result = func(2009, 262);
```

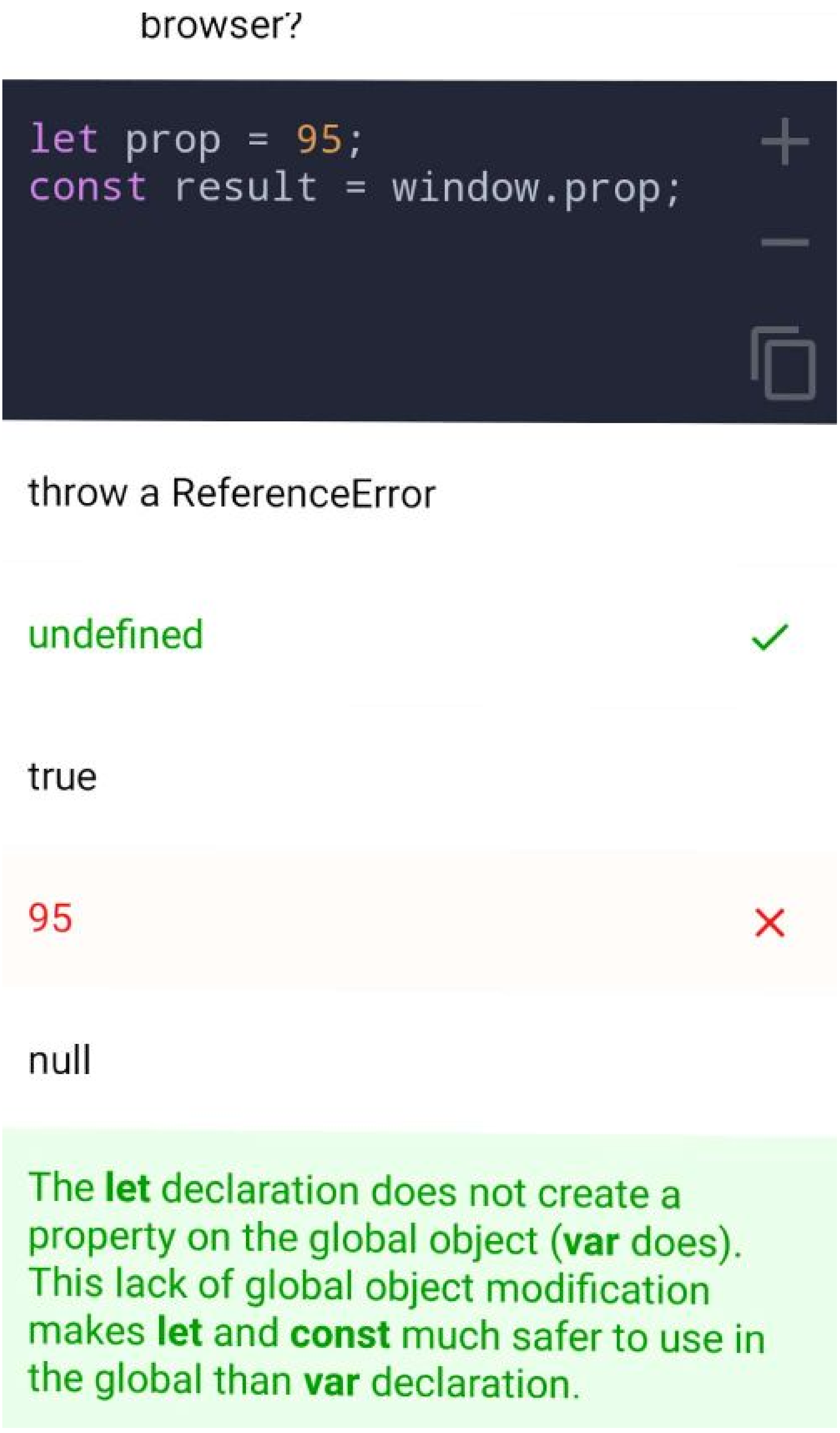
2271

throw a TypeError ✓

NaN

2009262 ✗

The arguments object is an **Array-like object** corresponding to the arguments passed to a function. The arguments object is not an **Array**. It is similar to an **Array**, but does not have any **Array** properties except **length**. It does not have the **join()** method.



browser?

```
let prop = 95;
const result = window.prop;
```

throw a ReferenceError

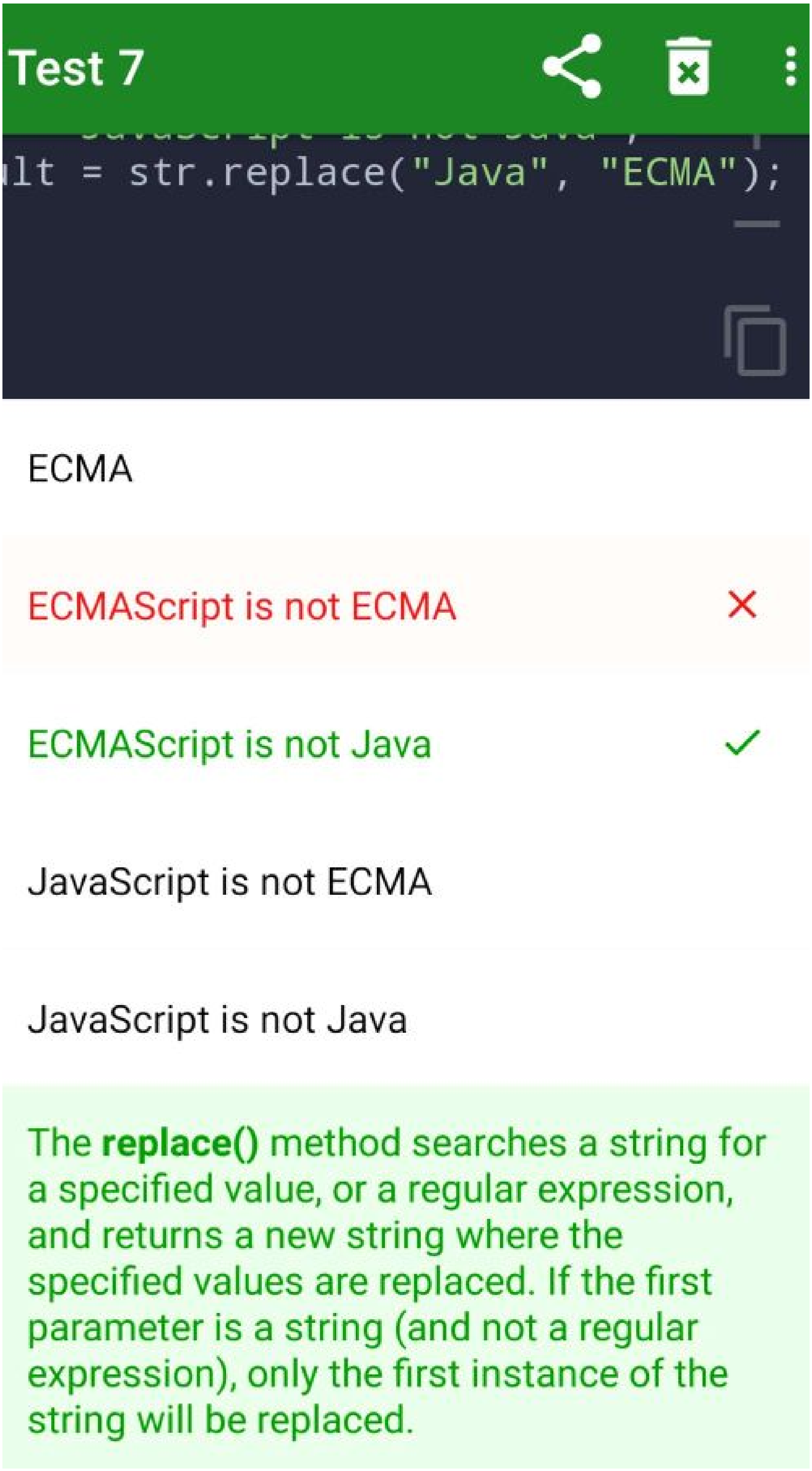
undefined ✓

true

95 ✗

null

The **let** declaration does not create a property on the global object (**var** does). This lack of global object modification makes **let** and **const** much safer to use in the global than **var** declaration.



Test 7

```
lt = str.replace("Java", "ECMA");
```

ECMA

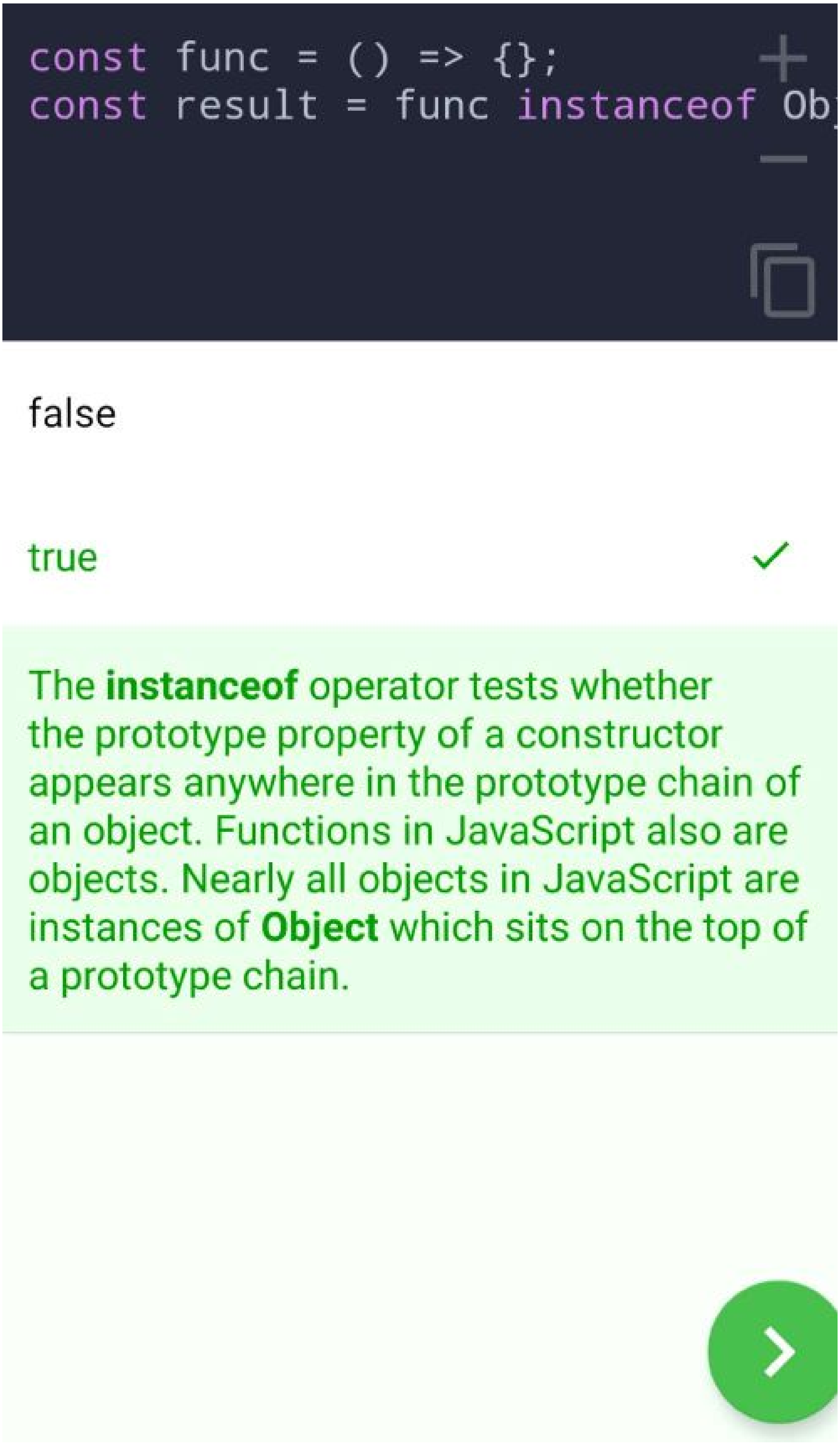
ECMAScript is not ECMA ✗

ECMAScript is not Java ✓

JavaScript is not ECMA

JavaScript is not Java

The **replace()** method searches a string for a specified value, or a regular expression, and returns a new string where the specified values are replaced. If the first parameter is a string (and not a regular expression), only the first instance of the string will be replaced.

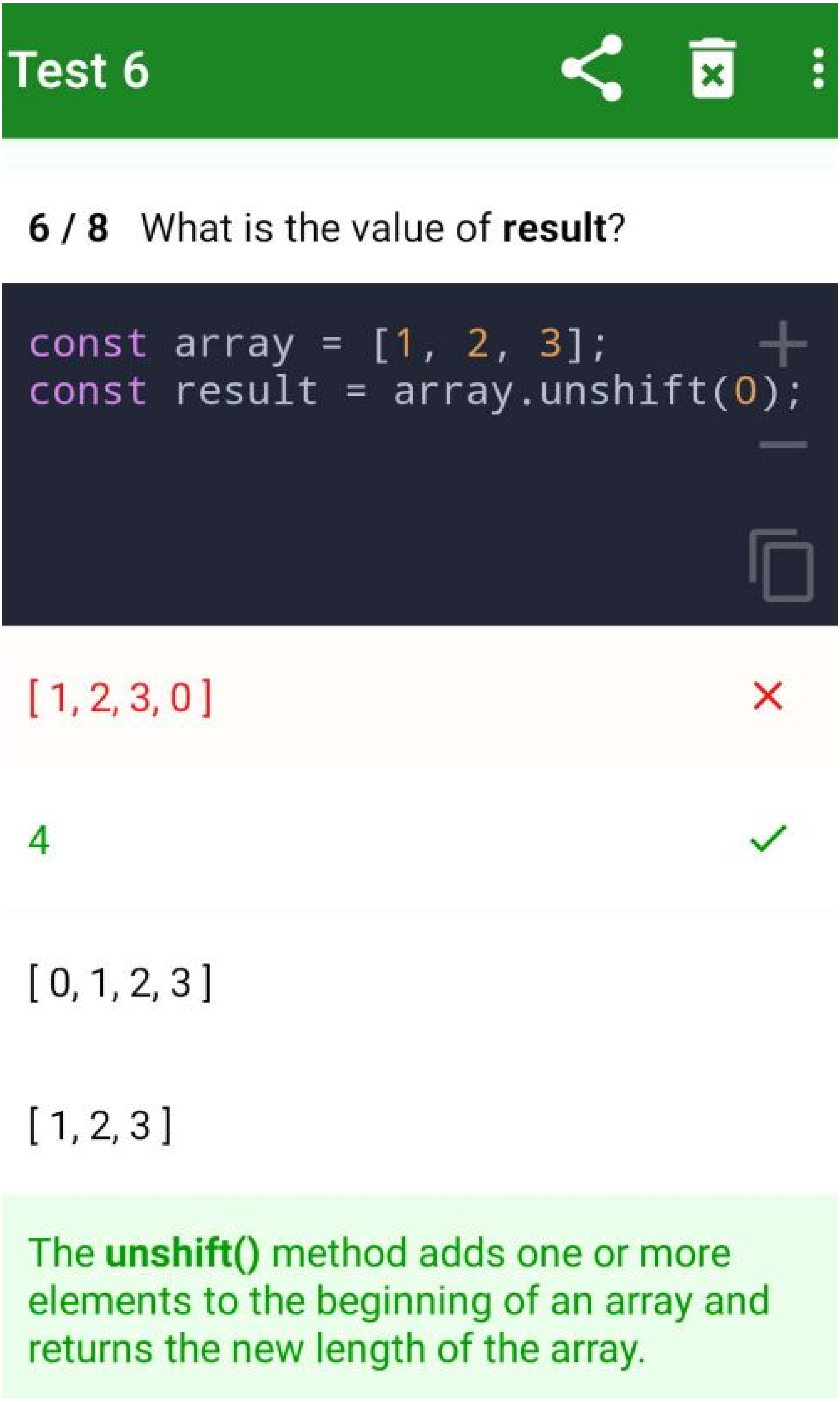


```
const func = () => {};
const result = func instanceof Ob
```

false

[✓ true]

The **instanceof** operator tests whether the prototype property of a constructor appears anywhere in the prototype chain of an object. Functions in JavaScript also are objects. Nearly all objects in JavaScript are instances of **Object** which sits on the top of a prototype chain.



6 / 8 What is the value of **result**?

```
const array = [1, 2, 3];
const result = array.unshift(0);
```

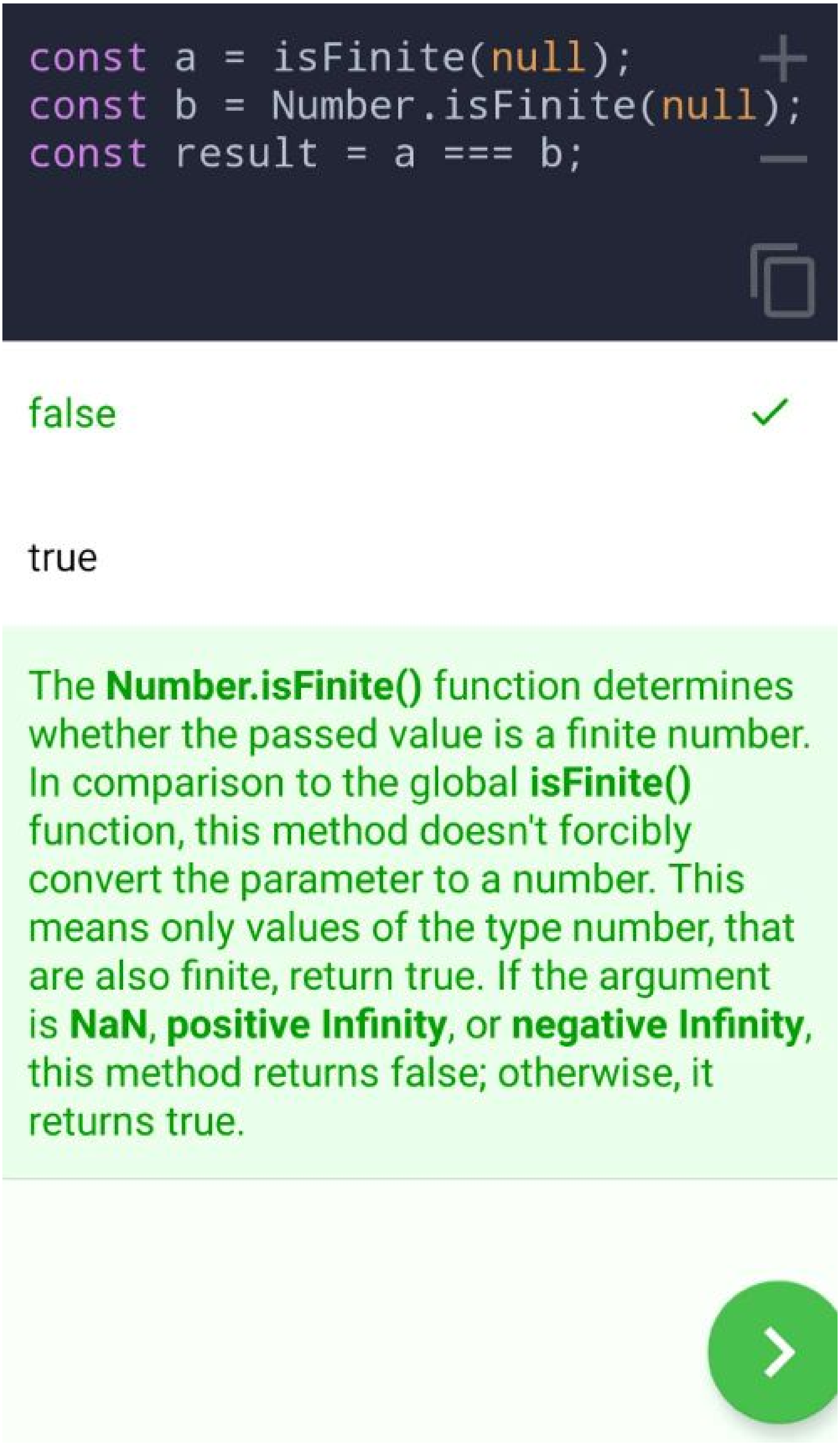
[1, 2, 3, 0] ✗

4 ✓

[0, 1, 2, 3]

[1, 2, 3]

The **unshift()** method adds one or more elements to the beginning of an array and returns the new length of the array.

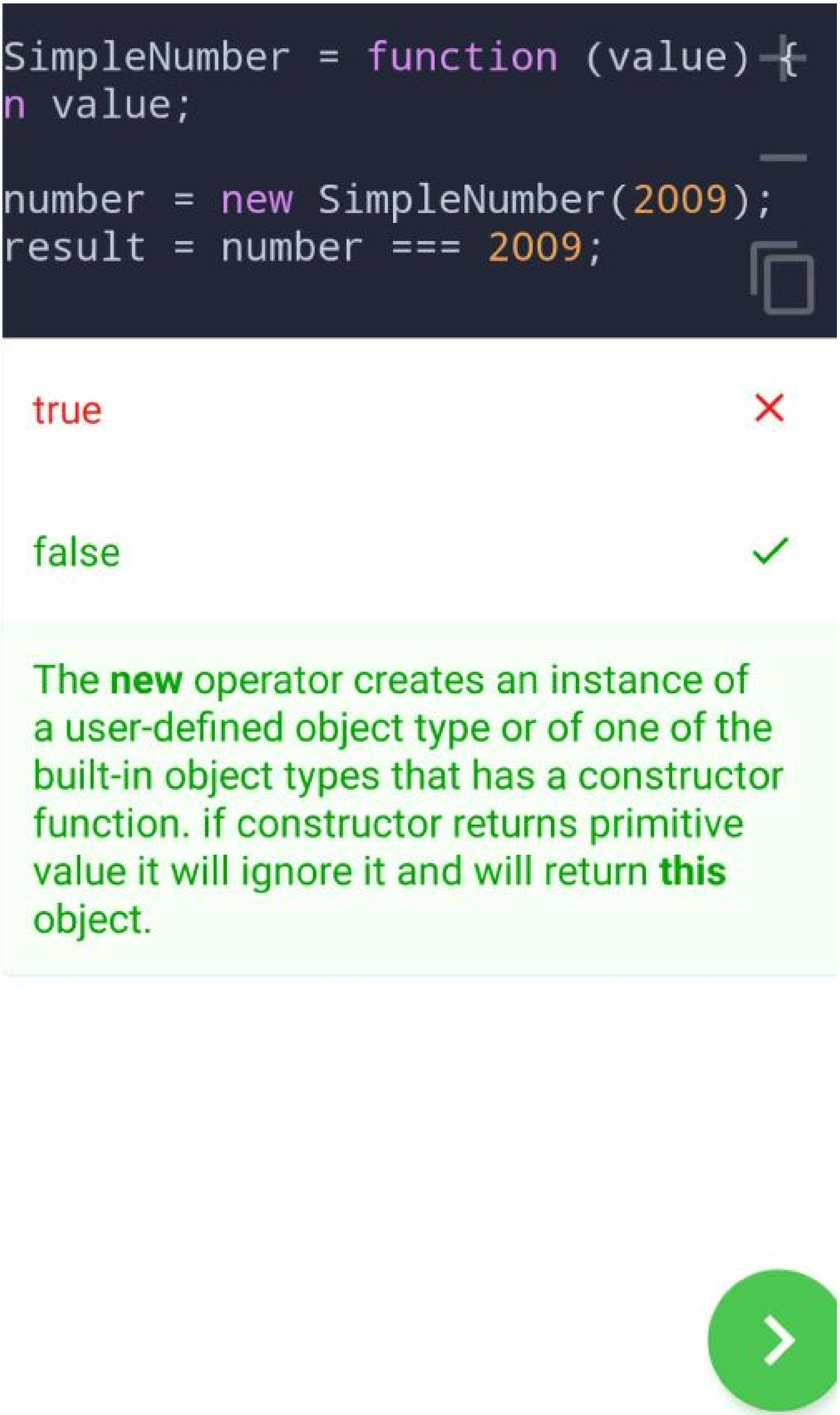


```
const a = isFinite(null);
const b = Number.isFinite(null);
const result = a === b;
```

[✓ false]

true

The **Number.isFinite()** function determines whether the passed value is a finite number. In comparison to the global **isFinite()** function, this method doesn't forcibly convert the parameter to a number. This means only values of the type number, that are also finite, return true. If the argument is **NaN**, **positive Infinity**, or **negative Infinity**, this method returns false; otherwise, it returns true.

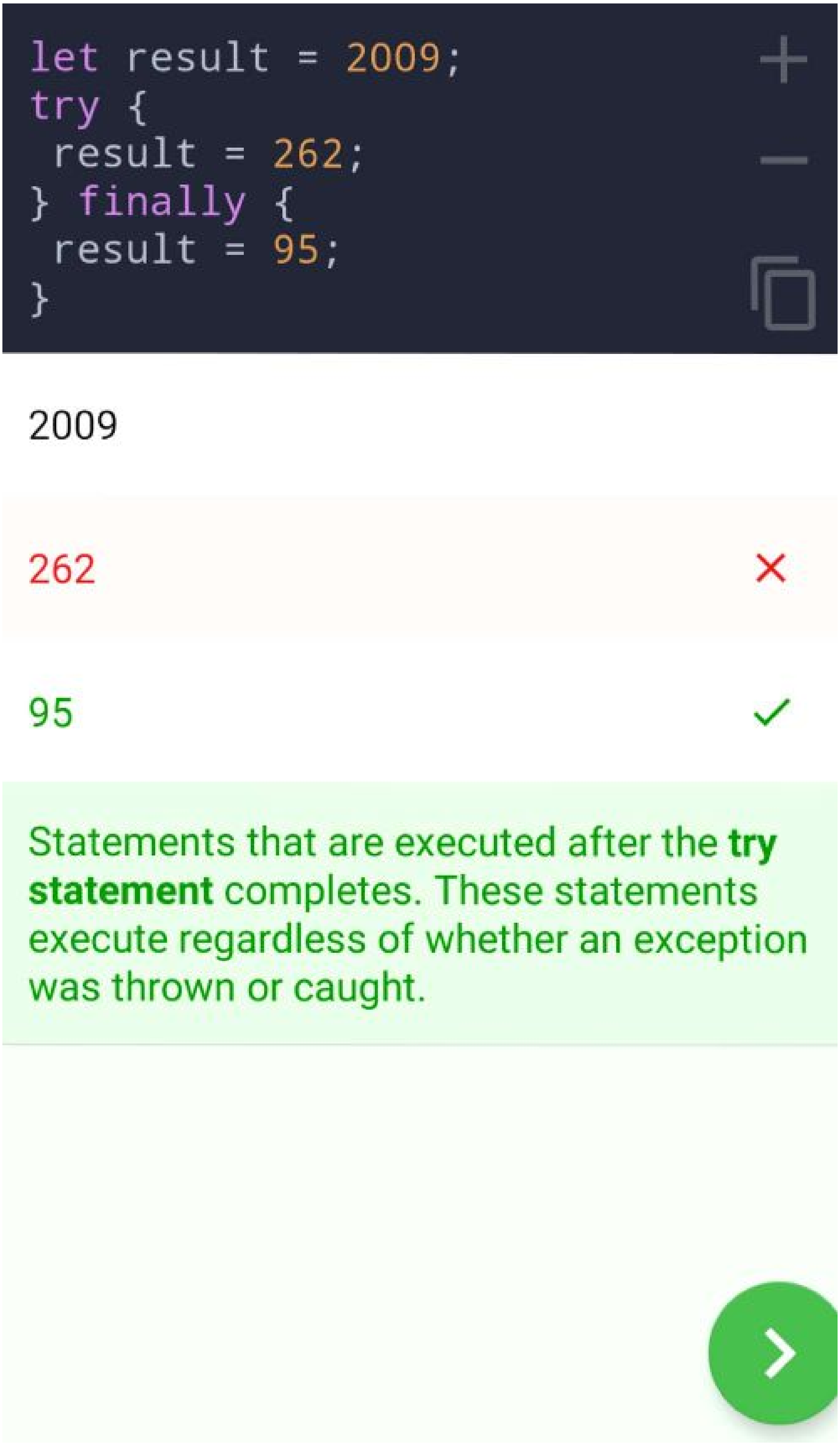


```
SimpleNumber = function (value) {
n value;

number = new SimpleNumber(2009);
result = number === 2009;
```

true ✗

false ✓

The **new** operator creates an instance of a user-defined object type or of one of the built-in object types that has a constructor function. if constructor returns primitive value it will ignore it and will return **this** object.



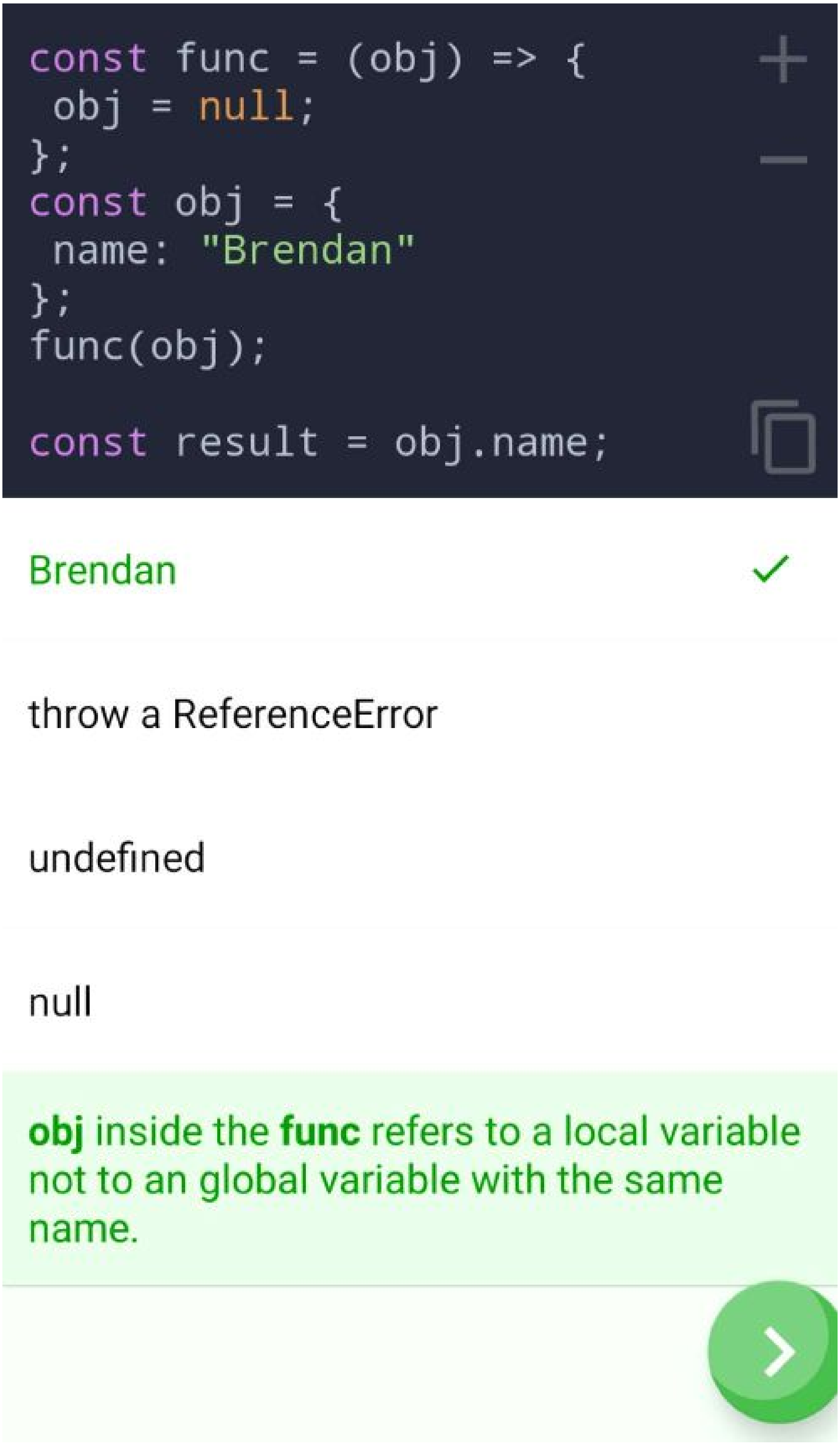
```
let result = 2009;
try {
 result = 262;
} finally {
 result = 95;
}
```

2009

262 ✗

95 ✓

Statements that are executed after the **try statement** completes. These statements execute regardless of whether an exception was thrown or caught.



```
const func = (obj) => {
 obj = null;
};
const obj = {
 name: "Brendan"
};
func(obj);

const result = obj.name;
```

Brendan ✓

throw a ReferenceError

undefined

null

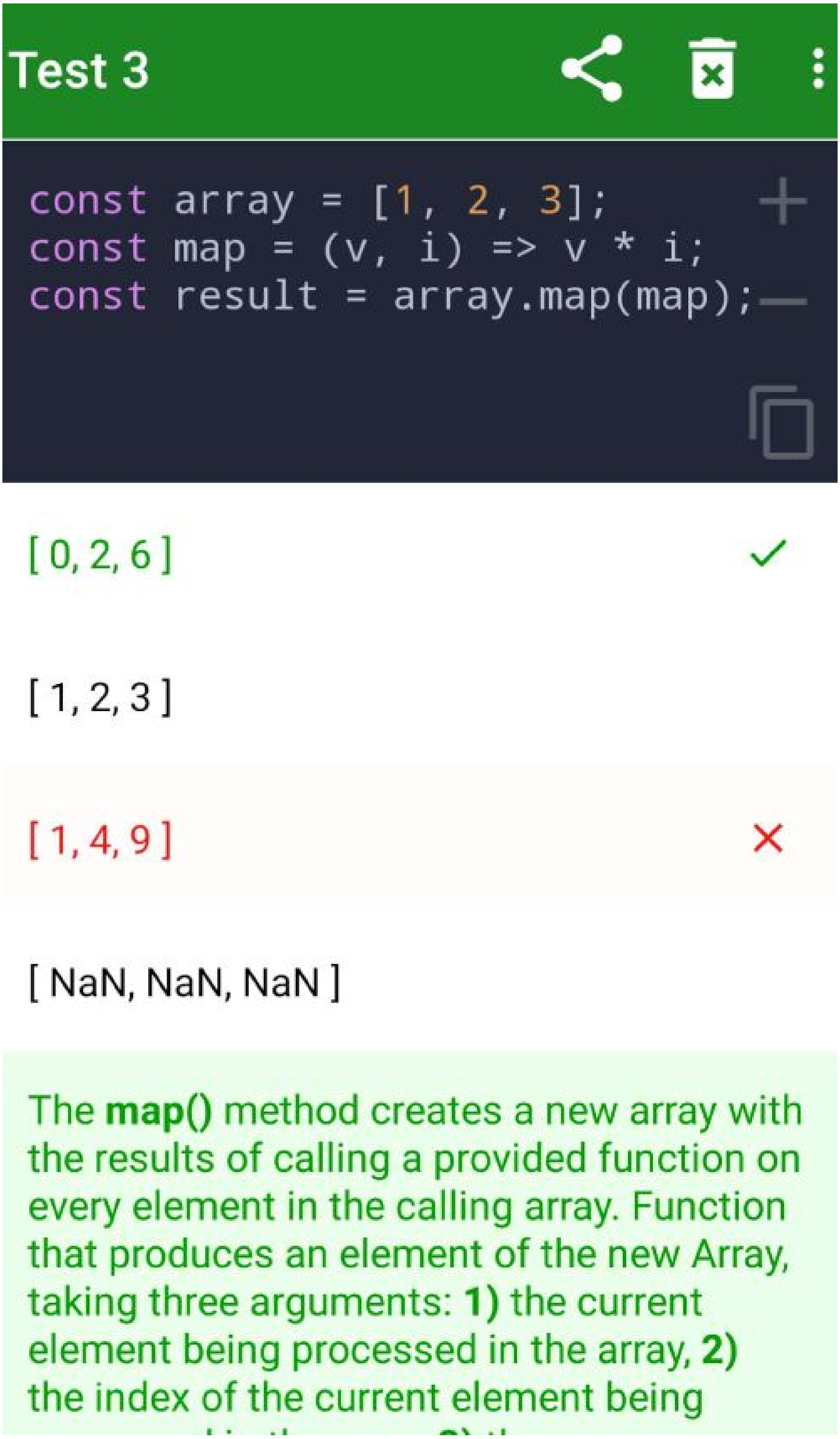
obj inside the **func** refers to a local variable not to an global variable with the same name.

```
const value = "2009";
const result = isFinite(value);
```

false ✗

true ✓

The global **isFinite()** function determines whether the passed value is a finite number. If needed, the parameter is first converted to a number. If the argument is **NaN**, positive infinity, or negative infinity, this method returns **false**; otherwise, it returns **true**.



Test 3

```
const array = [1, 2, 3];
const map = (v, i) => v * i;
const result = array.map(map);
```

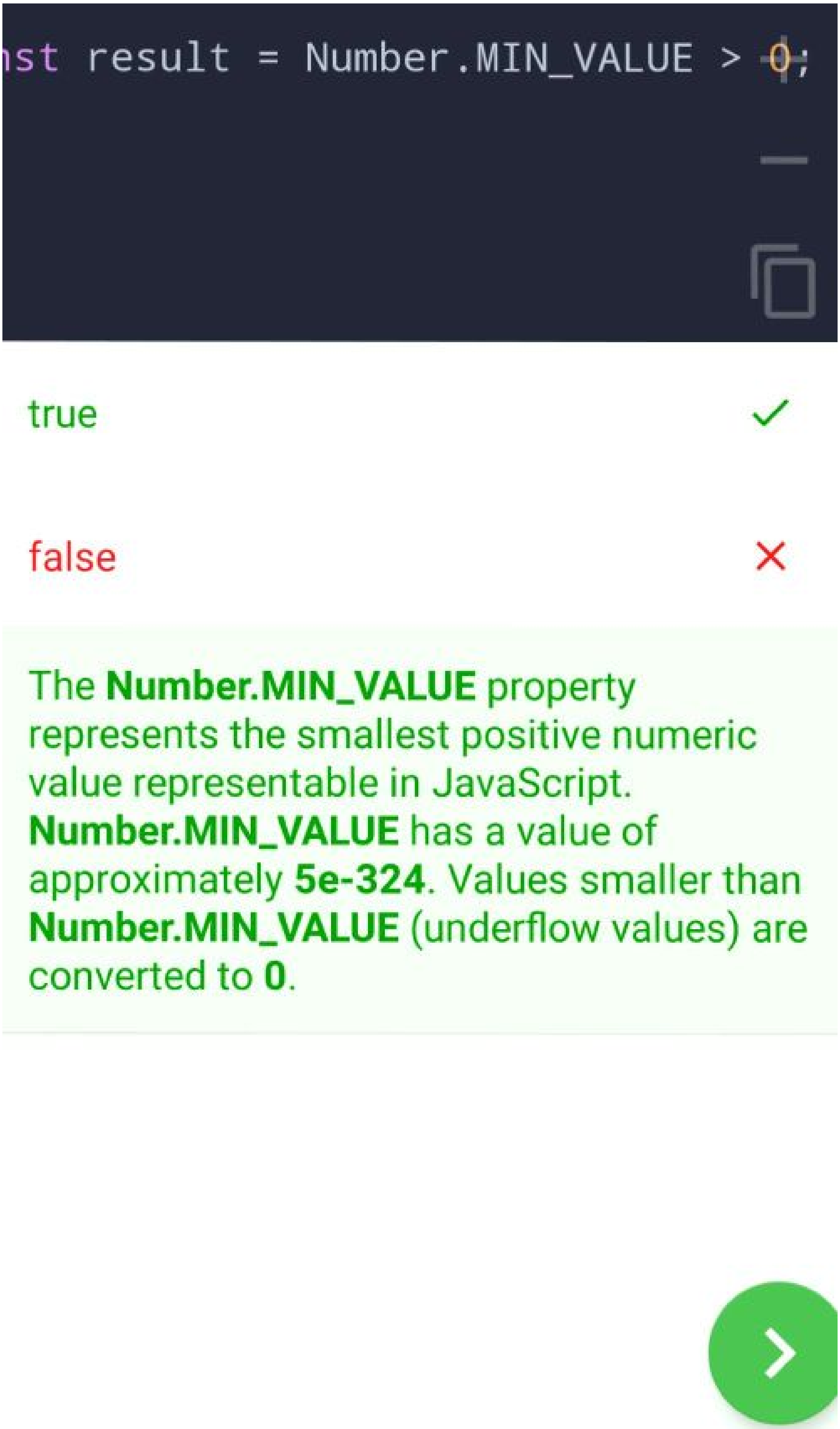
[0, 2, 6] ✓

[1, 2, 3]

[1, 4, 9] ✗

[NaN, NaN, NaN]

The **map()** method creates a new array with the results of calling a provided function on every element in the calling array. Function that produces an element of the new Array, taking three arguments: **1)** the current element being processed in the array, **2)** the index of the current element being

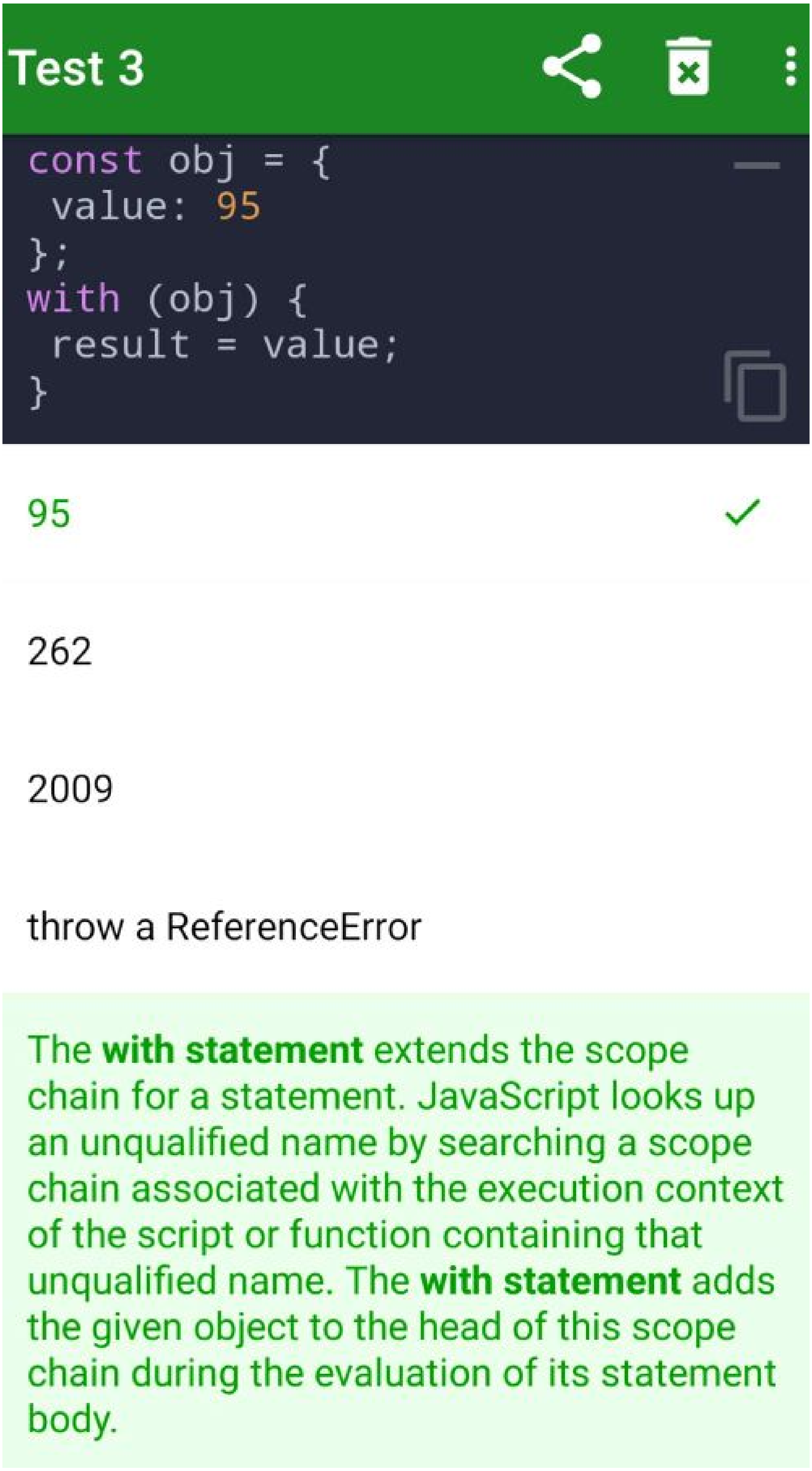


```
nst result = Number.MIN_VALUE > 0;
```

true ✓

false ✗

The **Number.MIN_VALUE** property represents the smallest positive numeric value representable in JavaScript. **Number.MIN_VALUE** has a value of approximately **5e-324**. Values smaller than **Number.MIN_VALUE** (underflow values) are converted to **0**.



Test 3

```
const obj = {
 value: 95
};
with (obj) {
 result = value;
}
```

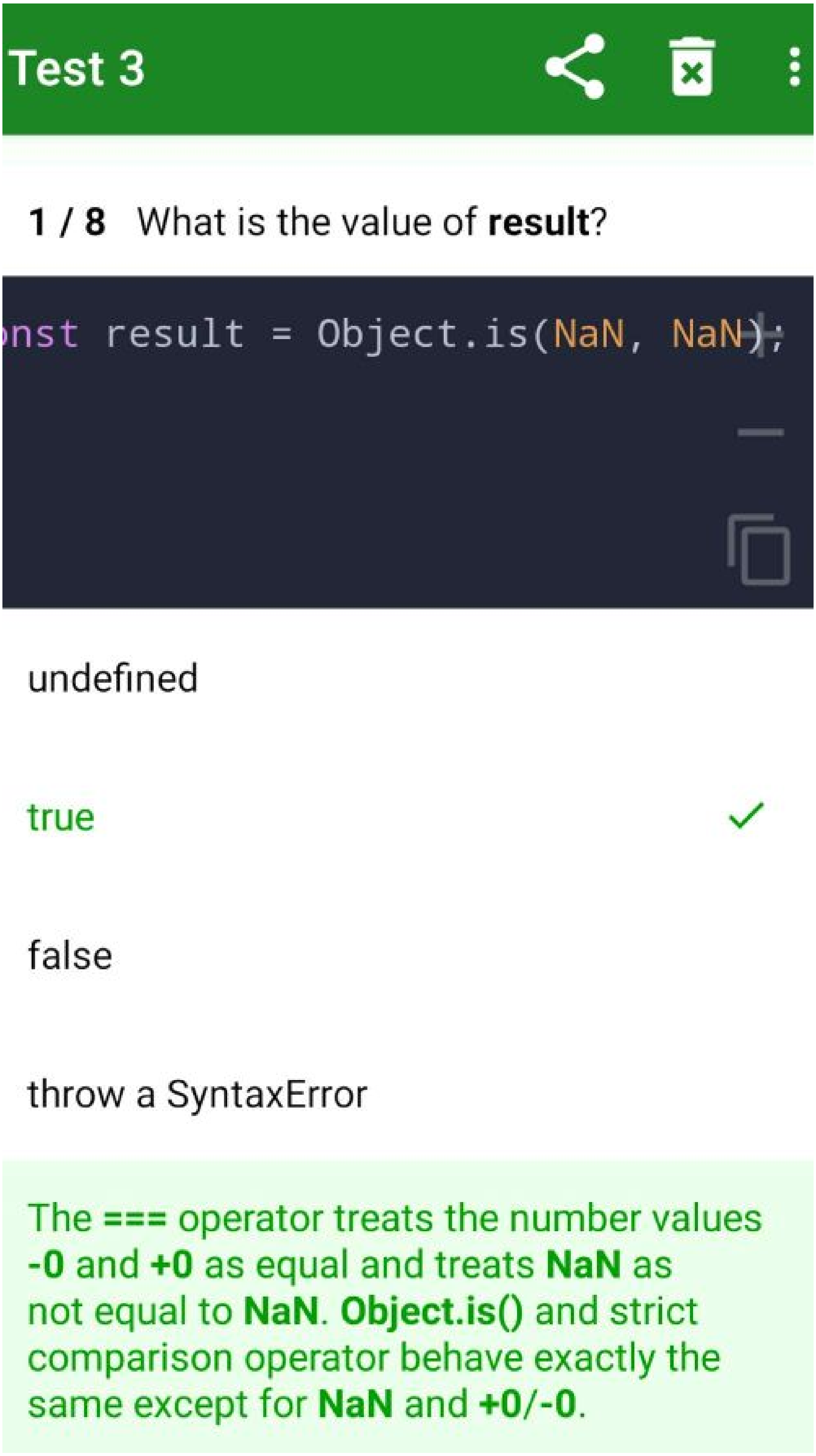
[✓ 95]

262

2009

throw a ReferenceError

The **with statement** extends the scope chain for a statement. JavaScript looks up an unqualified name by searching a scope chain associated with the execution context of the script or function containing that unqualified name. The **with statement** adds the given object to the head of this scope chain during the evaluation of its statement body.



Test 3

1 / 8 What is the value of **result**?

```
nst result = Object.is(NaN, NaN);
```

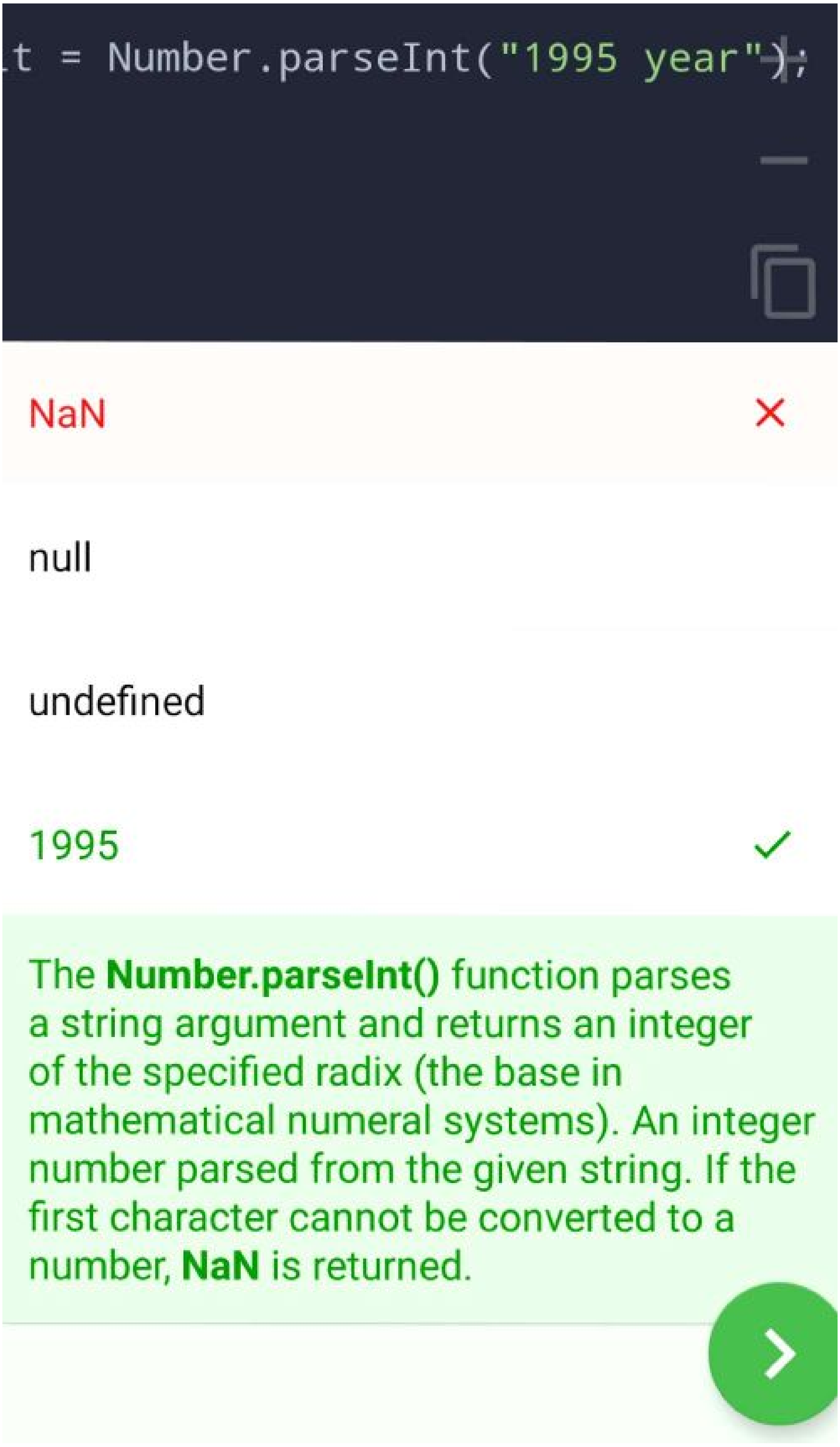
undefined

[✓ true]

false

throw a SyntaxError

The **===** operator treats the number values **-0** and **+0** as equal and treats **NaN** as not equal to **NaN**. **Object.is()** and strict comparison operator behave exactly the same except for **NaN** and **+0/-0**.



```
t = Number.parseInt("1995 year");
```

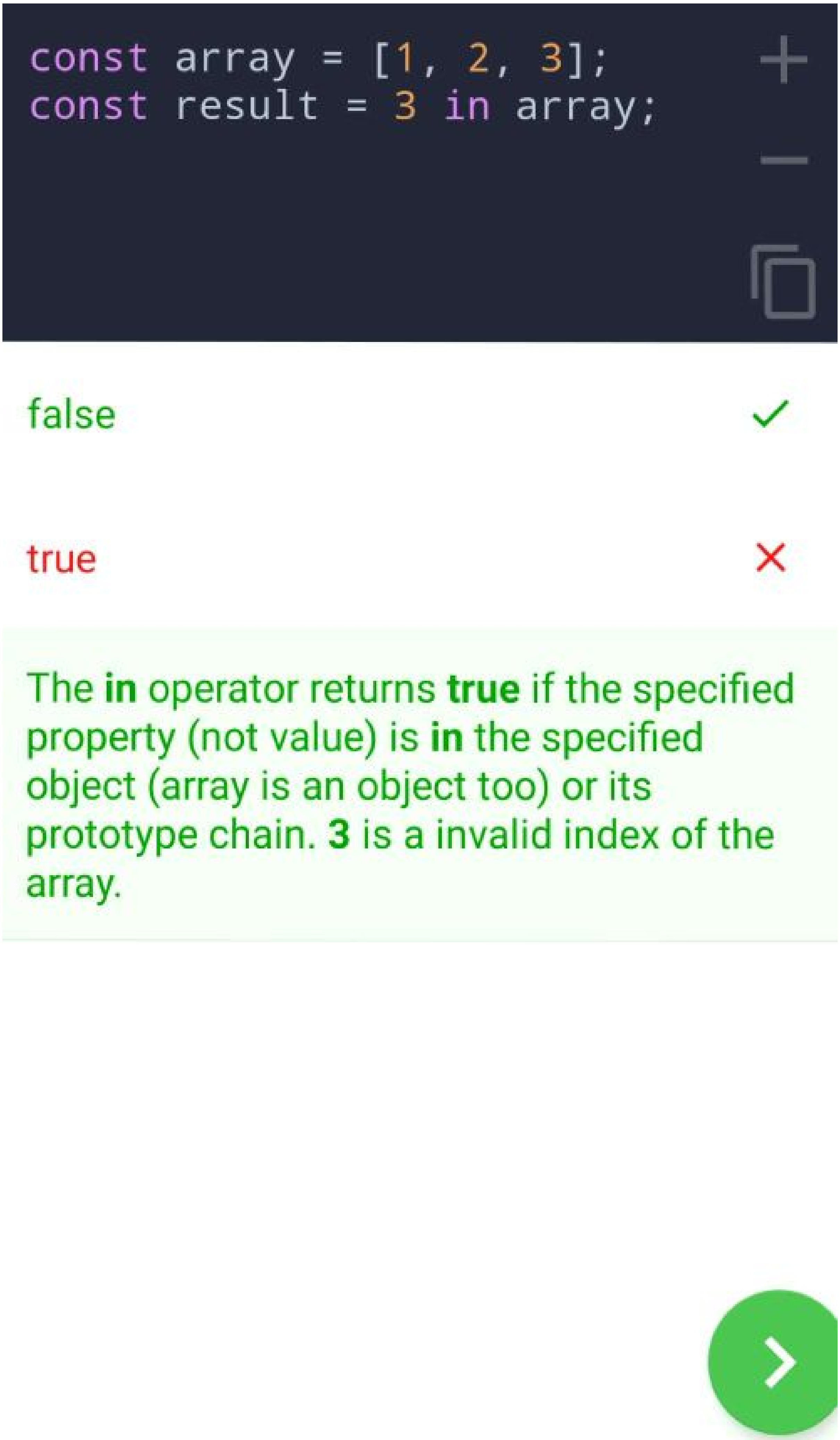
NaN ✗

null

undefined

1995 ✓

The **Number.parseInt()** function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems). An integer number parsed from the given string. If the first character cannot be converted to a number, **NaN** is returned.

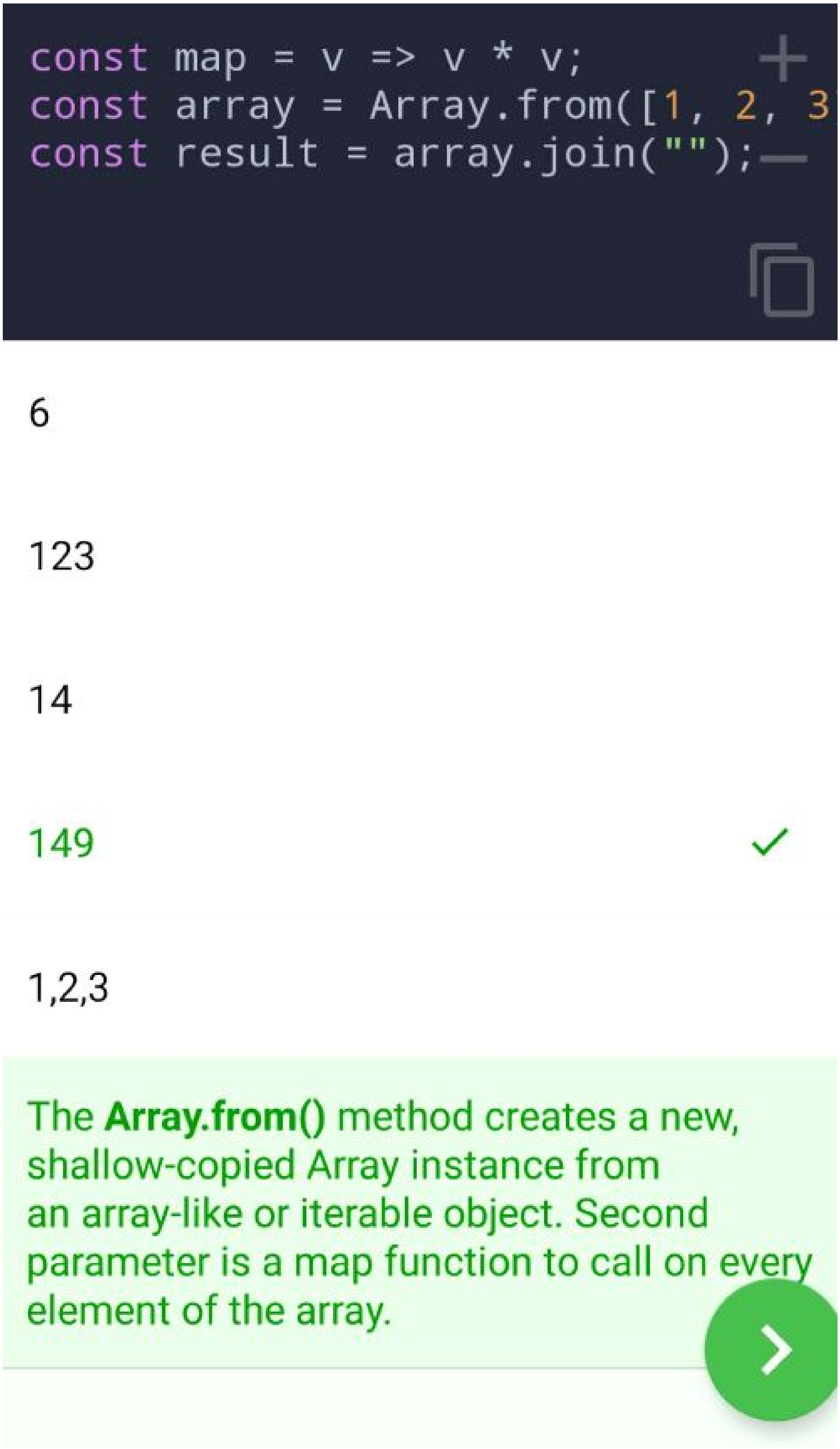


```
const array = [1, 2, 3];
const result = 3 in array;
```

false ✓

true ✗

The **in** operator returns **true** if the specified property (not value) is **in** the specified object (array is an object too) or its prototype chain. **3** is a invalid index of the array.



```
const map = v => v * v;
const array = Array.from([1, 2, 3
const result = array.join("");
```

6

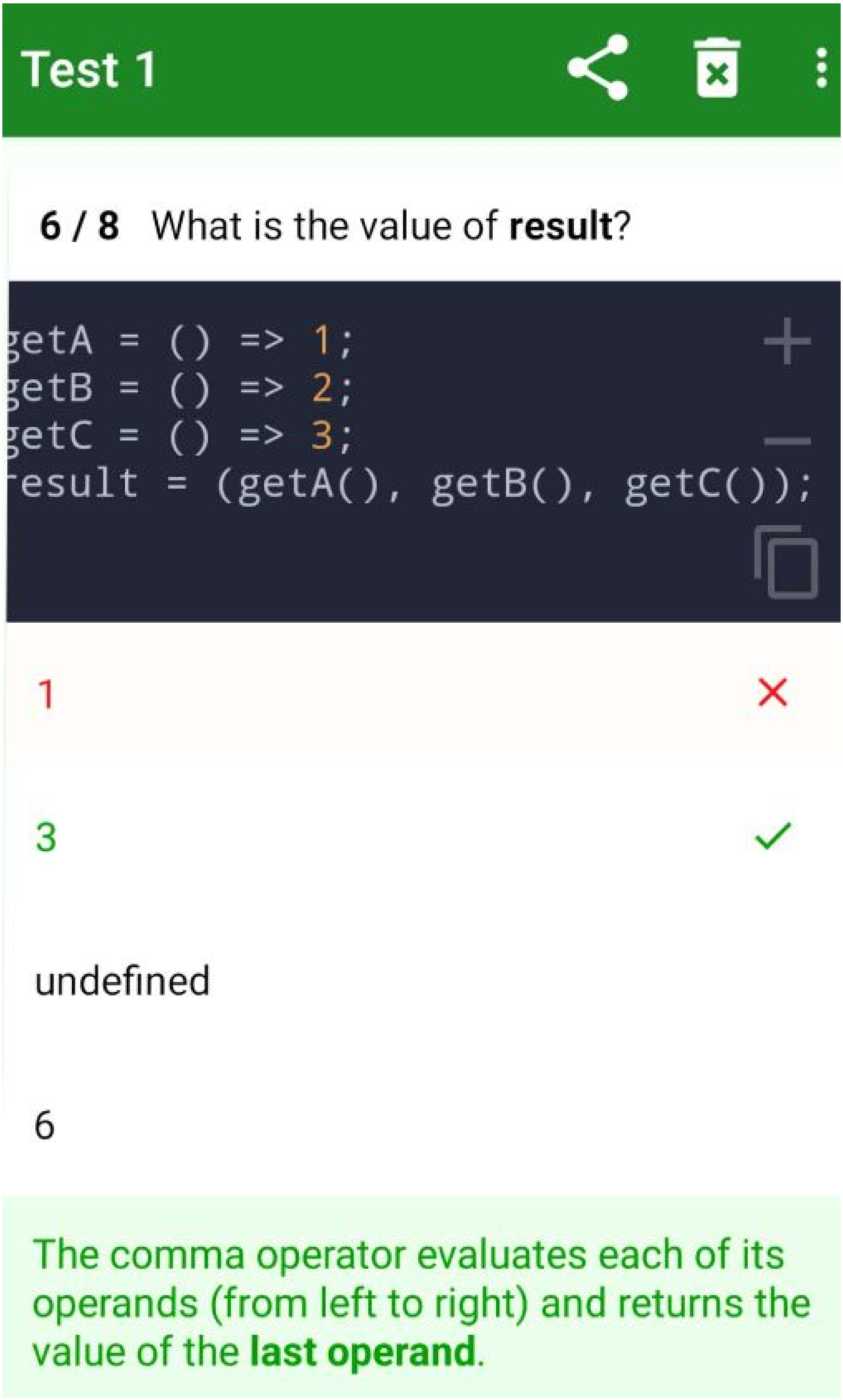
123

14

[✓ 149]

1,2,3

The **Array.from()** method creates a new, shallow-copied Array instance from an array-like or iterable object. Second parameter is a map function to call on every element of the array.



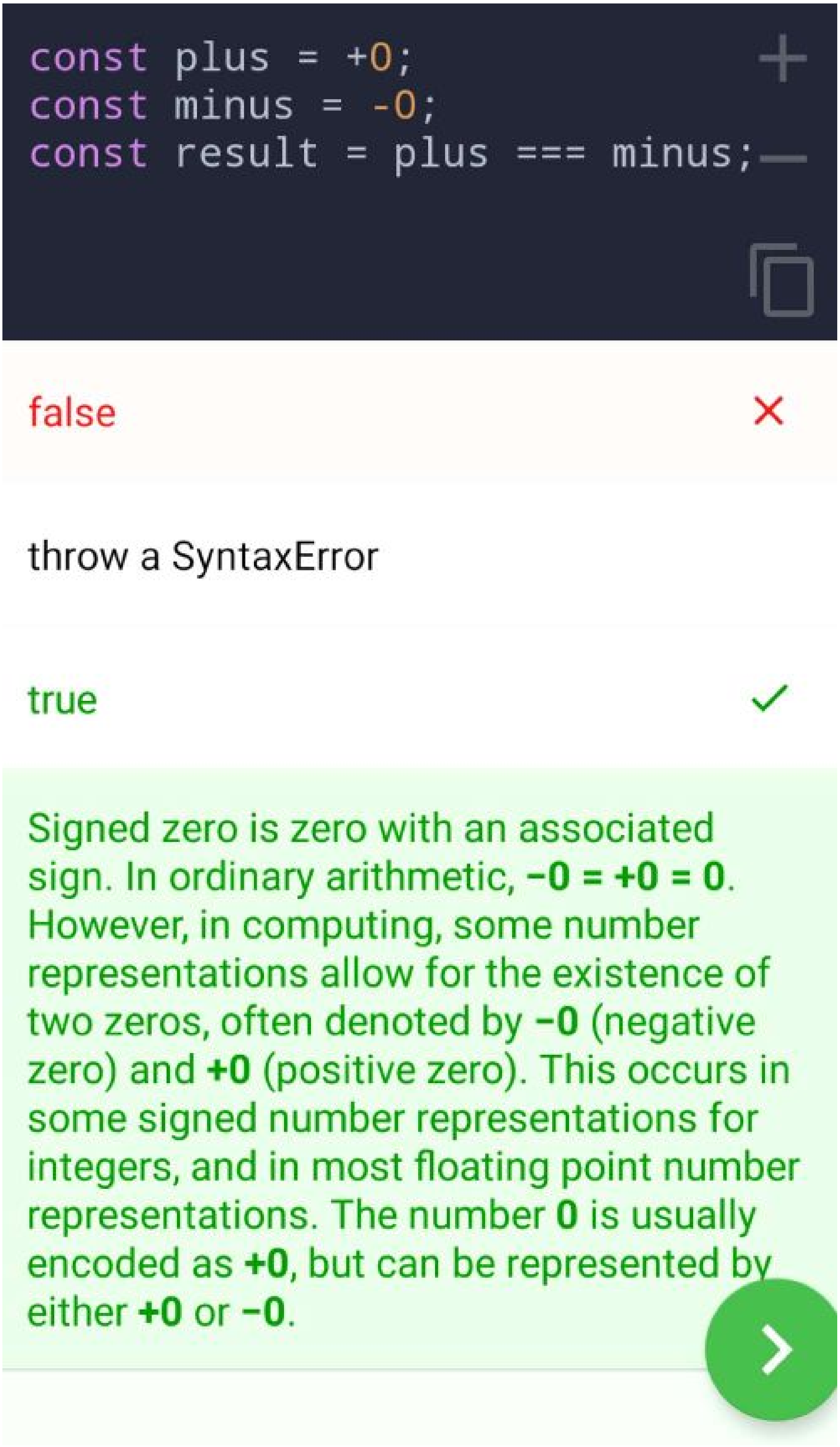
Test 1

6 / 8 What is the value of **result**?

```
getA = () => 1;
getB = () => 2;
getC = () => 3;
result = (getA(), getB(), getC());
```

- 1 ✗
- 3 ✓
- undefined
- 6

The comma operator evaluates each of its operands (from left to right) and returns the value of the **last operand**.



```
const plus = +0;
const minus = -0;
const result = plus === minus;
```

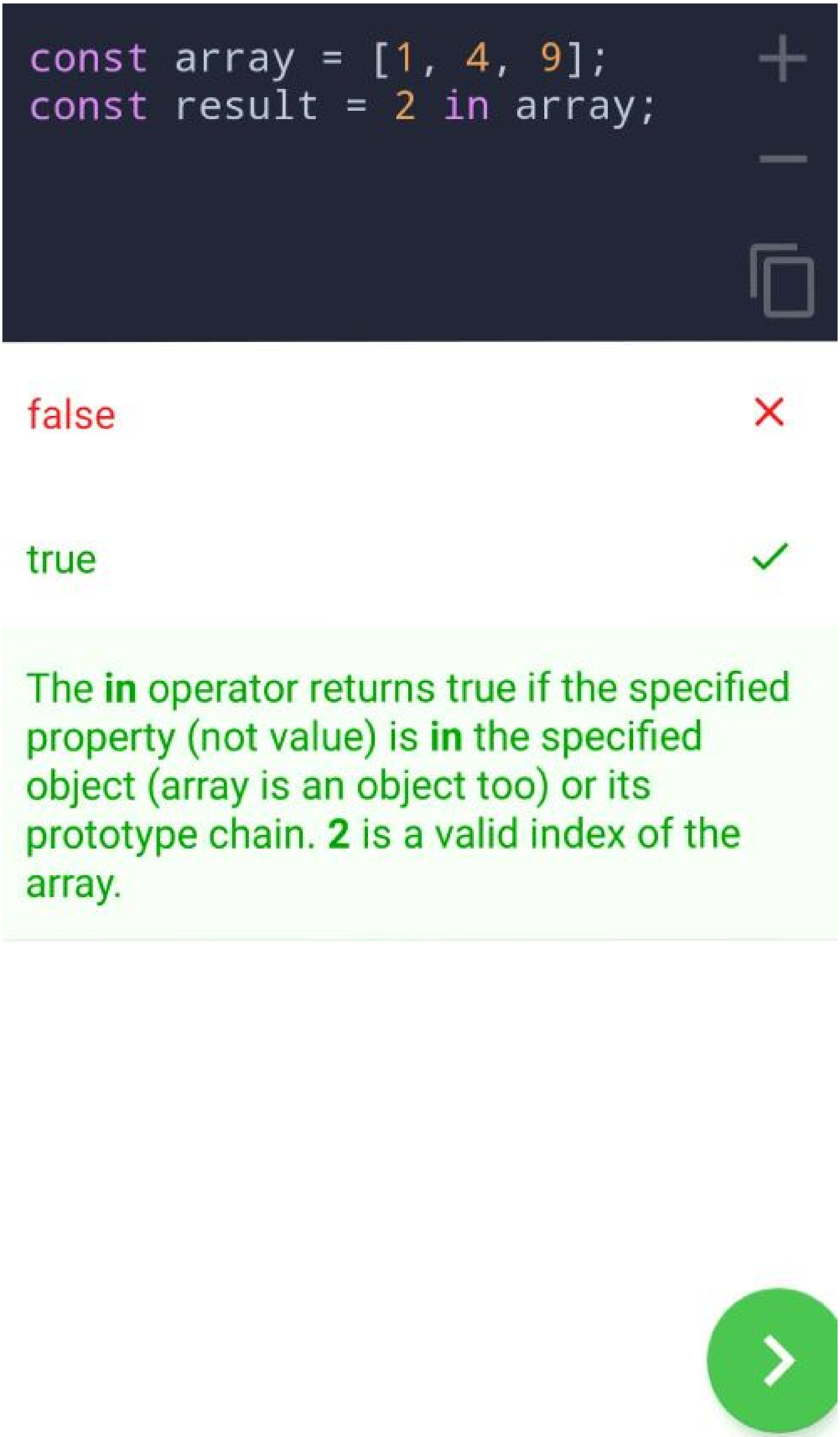
false ✗

throw a SyntaxError

true ✓

Signed zero is zero with an associated sign. In ordinary arithmetic, **−0 = +0 = 0**. However, in computing, some number representations allow for the existence of two zeros, often denoted by **−0** (negative zero) and **+0** (positive zero). This occurs in some signed number representations for integers, and in most floating point number representations. The number **0** is usually encoded as **+0**, but can be represented by either **+0** or **−0**.

Scanned by CamScanner

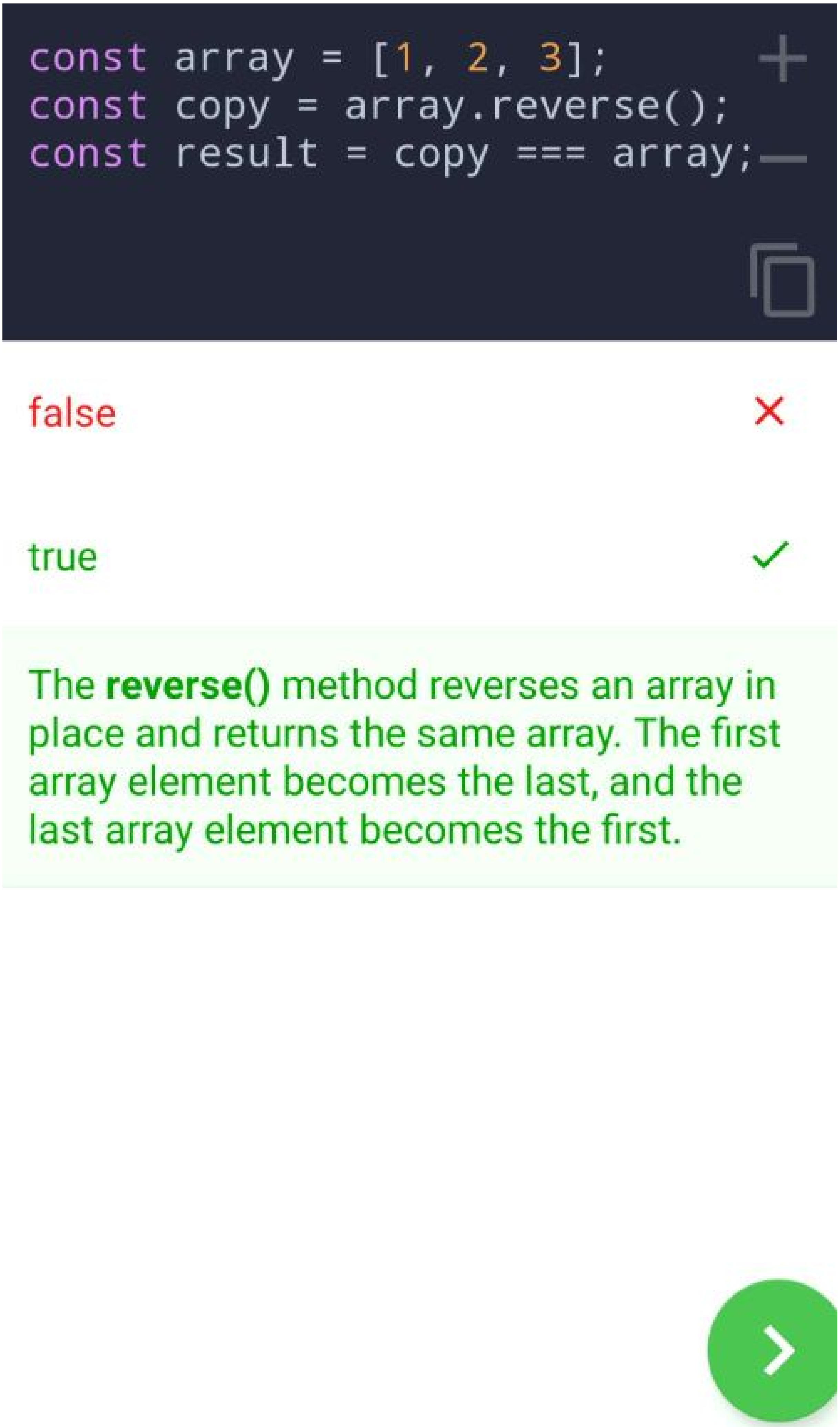


```
const array = [1, 4, 9];
const result = 2 in array;
```

false ✗

true ✓

The **in** operator returns true if the specified property (not value) is **in** the specified object (array is an object too) or its prototype chain. **2** is a valid index of the array.

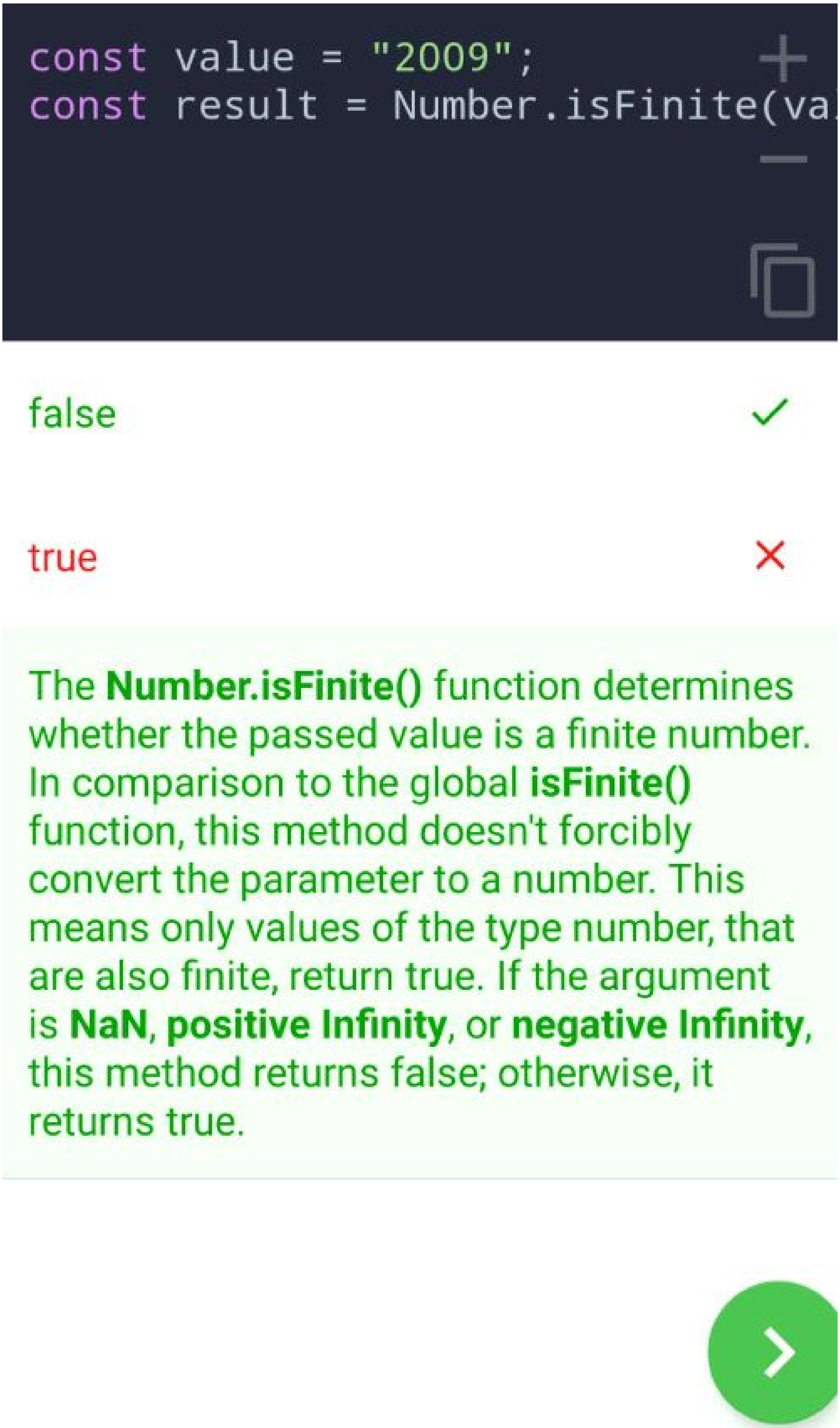


```
const array = [1, 2, 3];
const copy = array.reverse();
const result = copy === array;
```

false ✗

true ✓

The **reverse()** method reverses an array in place and returns the same array. The first array element becomes the last, and the last array element becomes the first.

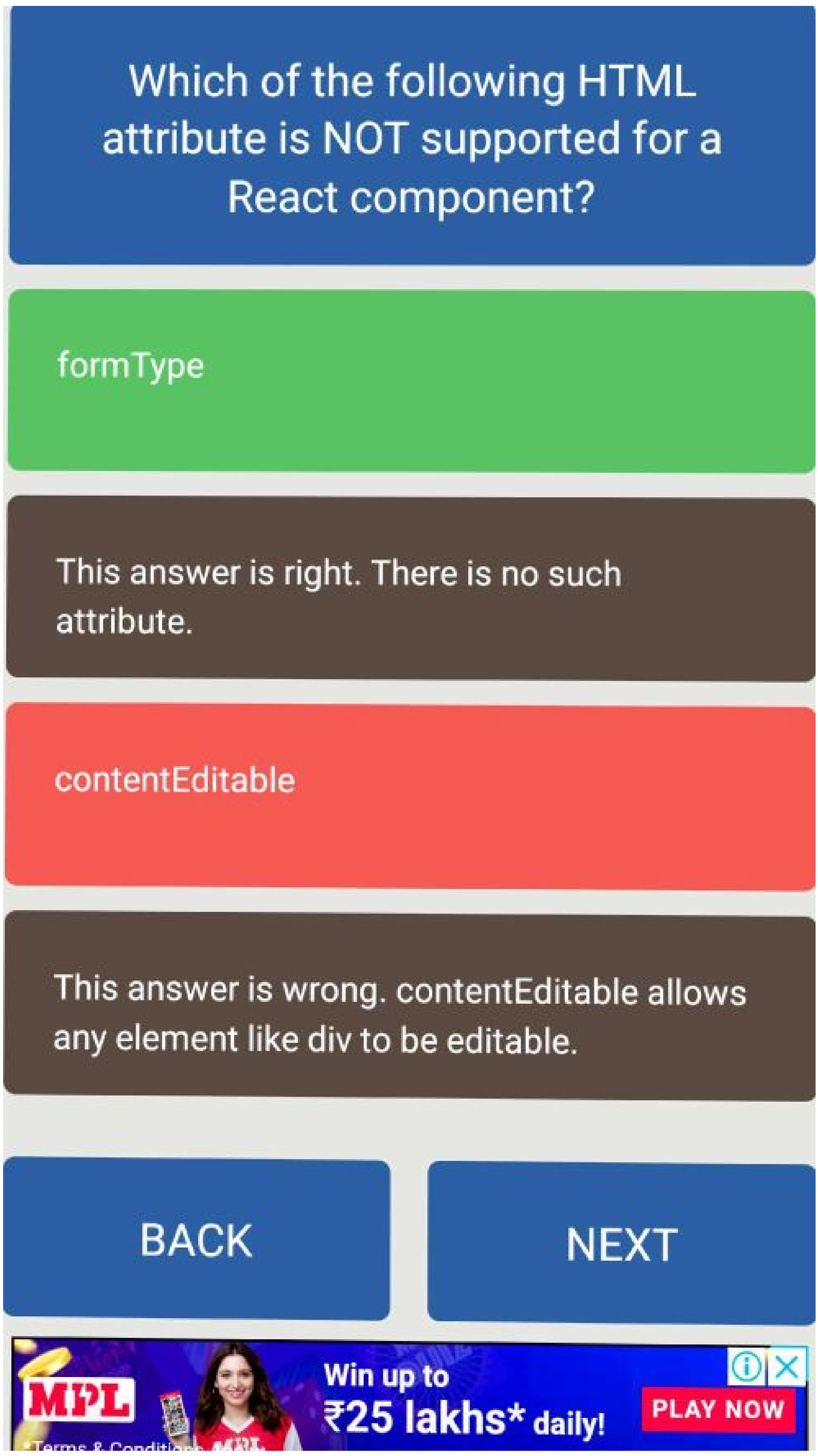


```
const value = "2009";
const result = Number.isFinite(va
```

false ✓

true ✗

The **Number.isFinite()** function determines whether the passed value is a finite number. In comparison to the global **isFinite()** function, this method doesn't forcibly convert the parameter to a number. This means only values of the type number, that are also finite, return true. If the argument is **NaN**, **positive Infinity**, or **negative Infinity**, this method returns false; otherwise, it returns true.



Which of the following HTML attribute is NOT supported for a React component?

formType

This answer is right. There is no such attribute.

contentEditable

This answer is wrong. contentEditable allows any element like div to be editable.

BACK

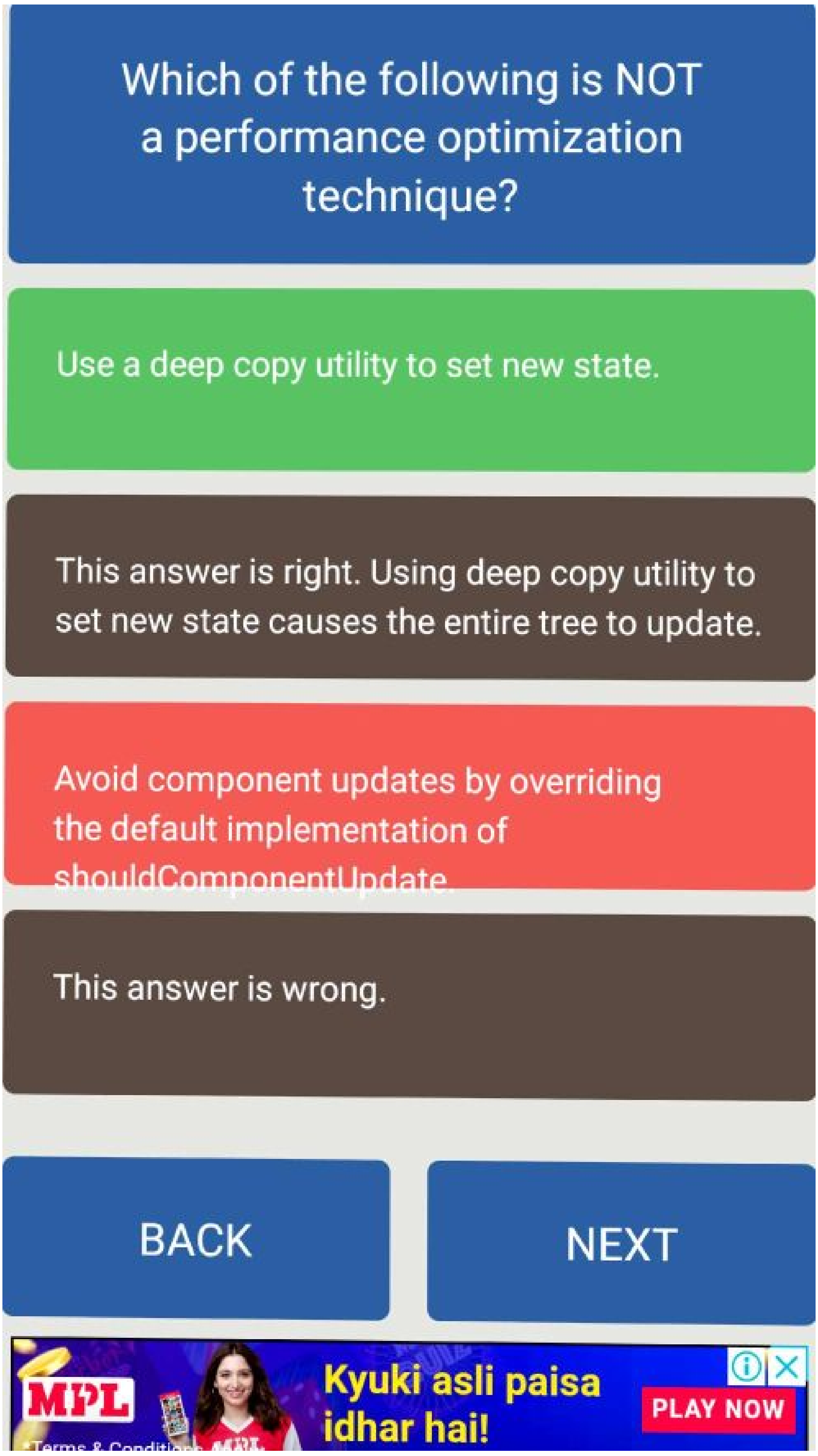
NEXT

MPL

Win up to ₹25 lakhs* daily!

PLAY NOW

*Terms & Conditions



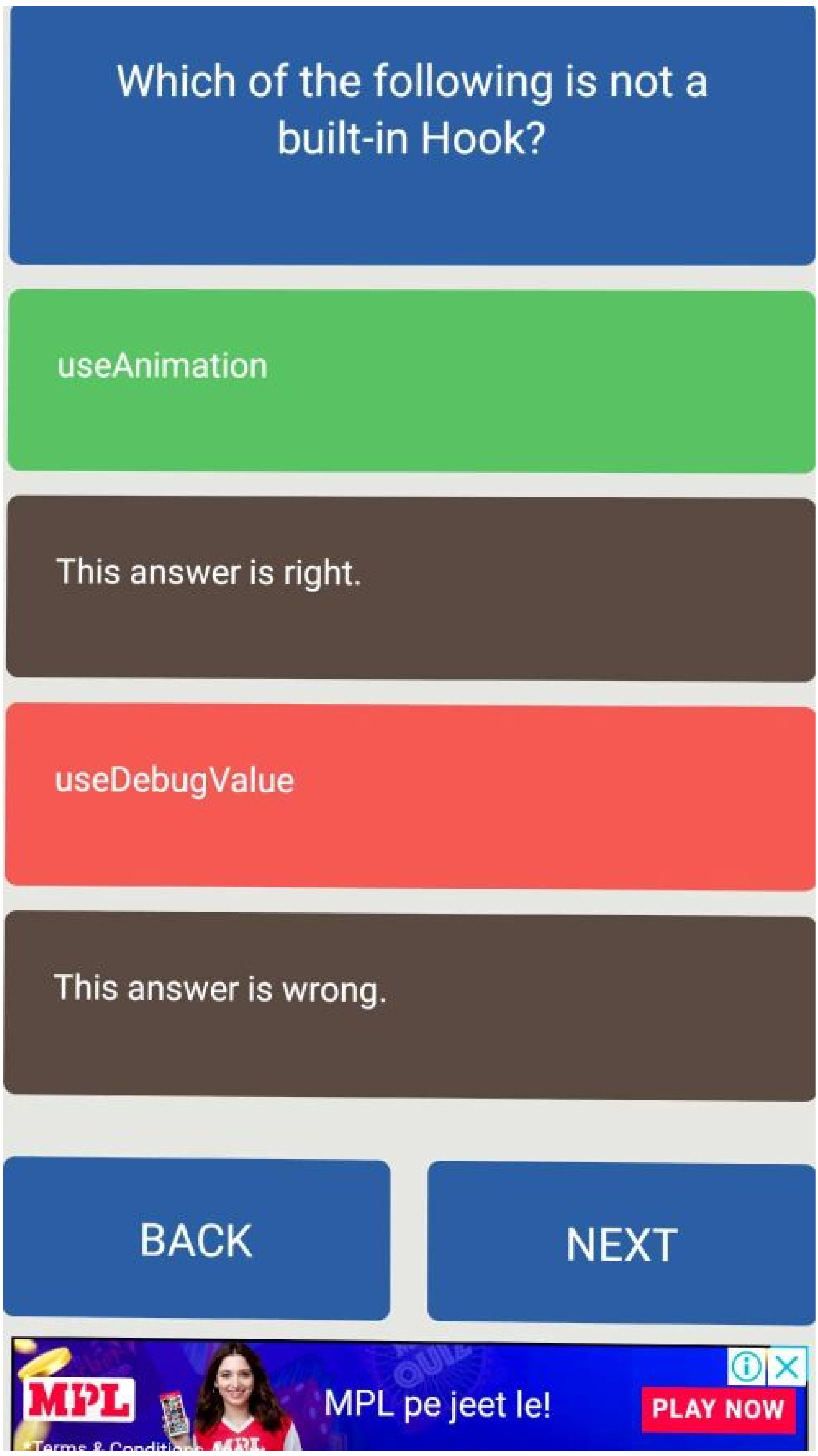
Which of the following is NOT a performance optimization technique?

Use a deep copy utility to set new state.

This answer is right. Using deep copy utility to set new state causes the entire tree to update.

Avoid component updates by overriding the default implementation of shouldComponentUpdate.

This answer is wrong.



Which of the following is not a built-in Hook?

useAnimation

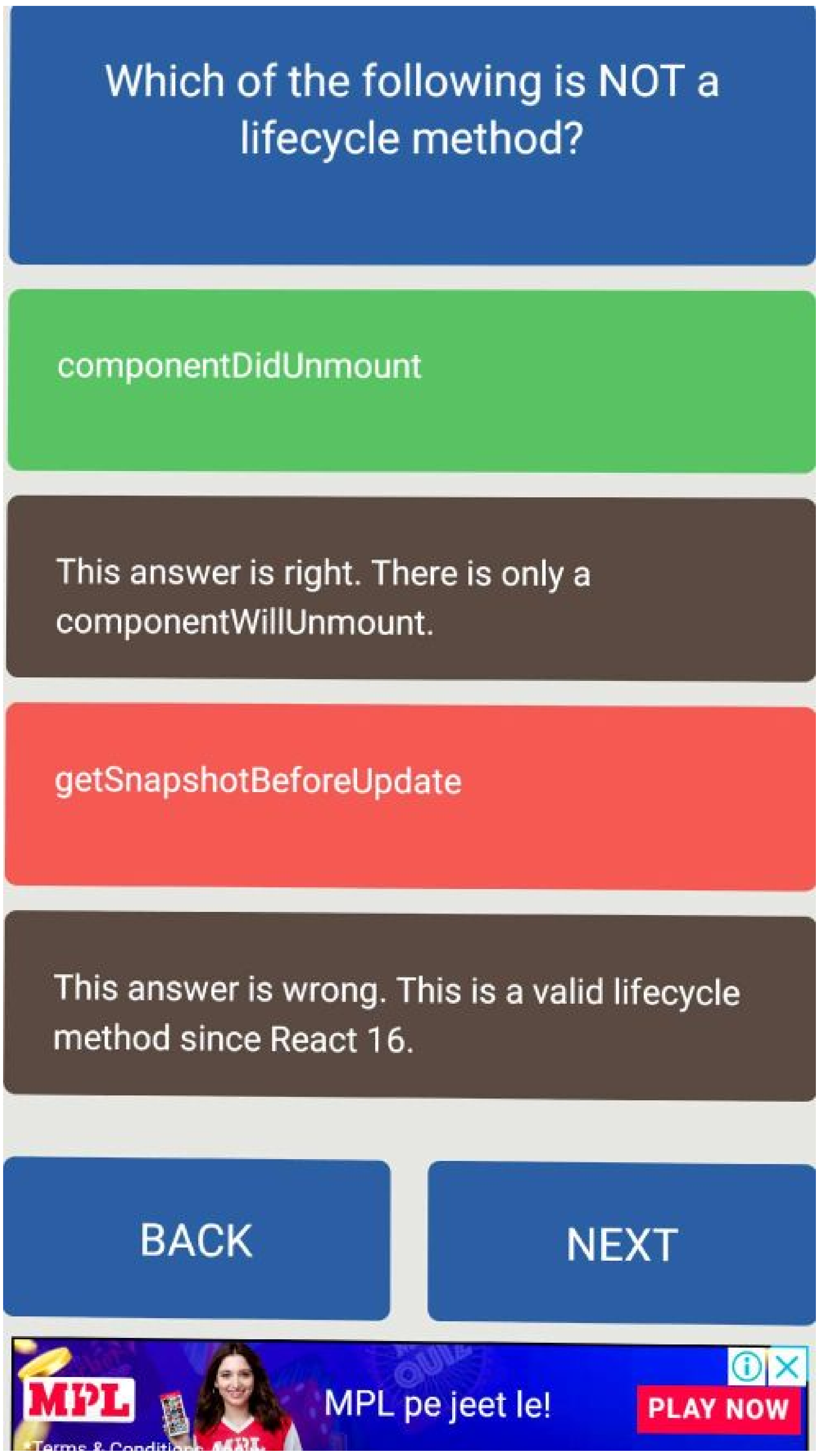
This answer is right.

useDebugValue

This answer is wrong.

BACK

NEXT



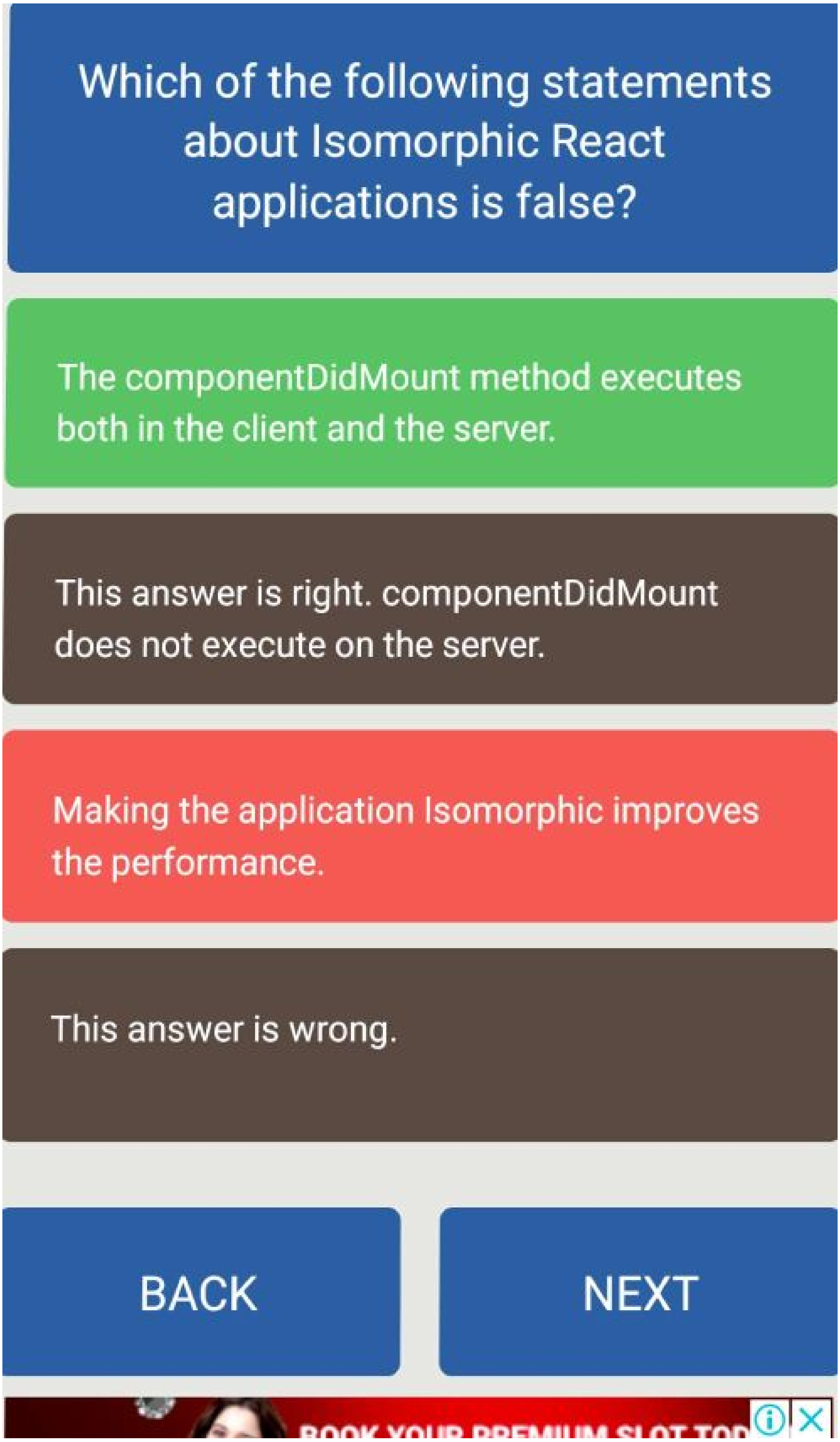
Which of the following is NOT a lifecycle method?

componentDidUnmount

This answer is right. There is only a componentWillUnmount.

getSnapshotBeforeUpdate

This answer is wrong. This is a valid lifecycle method since React 16.



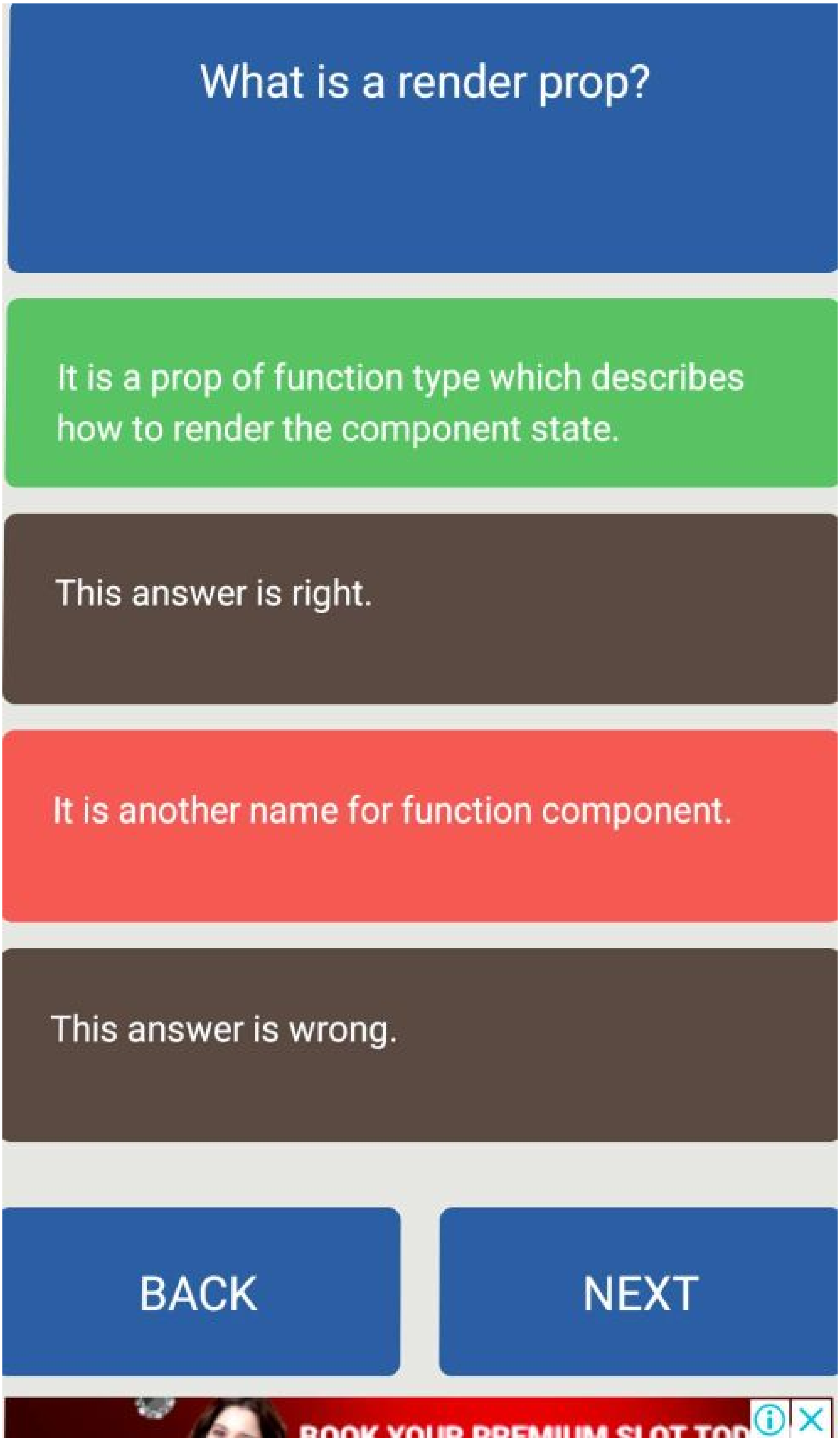
Which of the following statements about Isomorphic React applications is false?

The componentDidMount method executes both in the client and the server.

This answer is right. componentDidMount does not execute on the server.

Making the application Isomorphic improves the performance.

This answer is wrong.



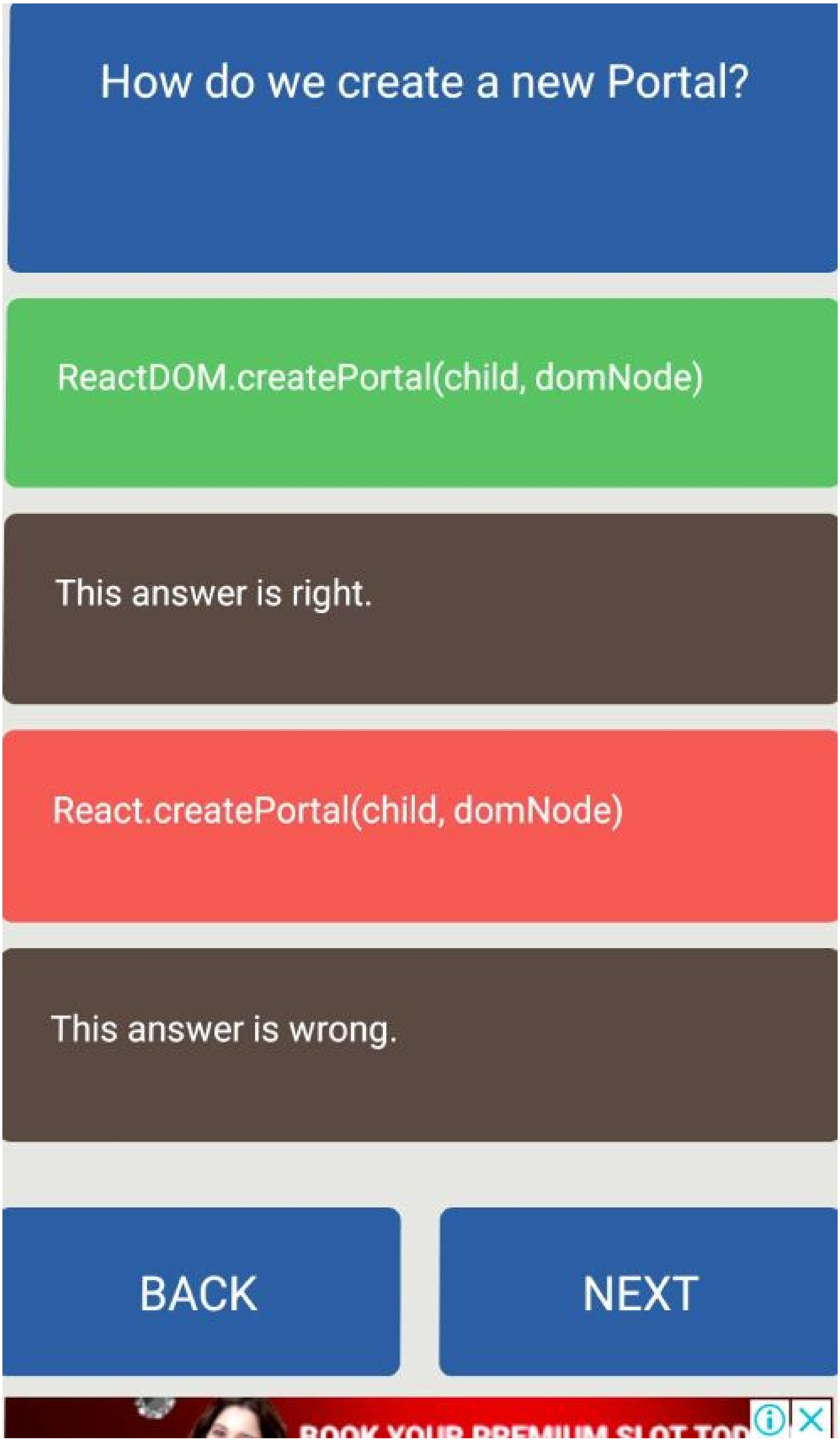
What is a render prop?

It is a prop of function type which describes how to render the component state.

This answer is right.

It is another name for function component.

This answer is wrong.



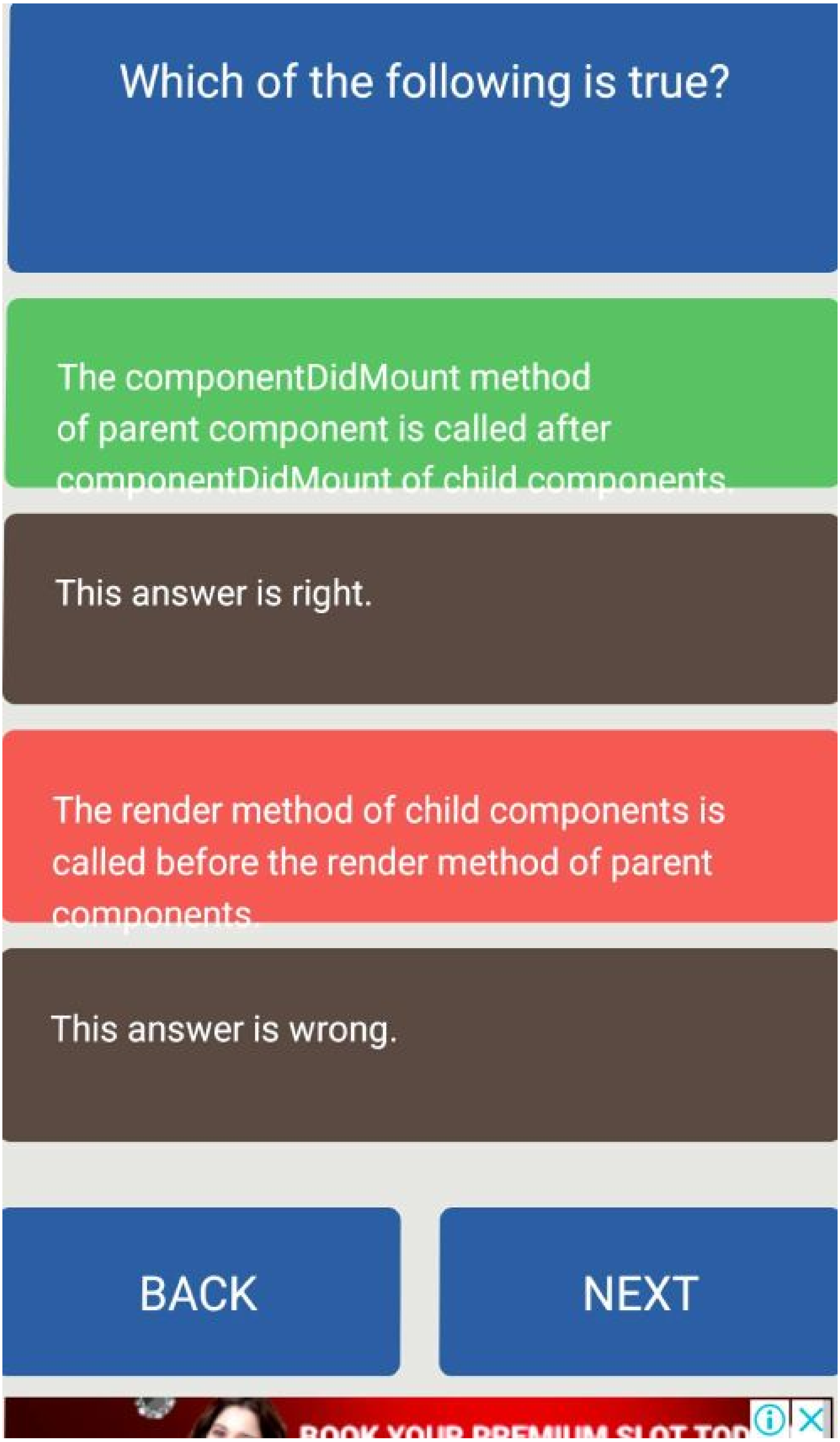
How do we create a new Portal?

ReactDOM.createPortal(child, domNode)

This answer is right.

React.createPortal(child, domNode)

This answer is wrong.

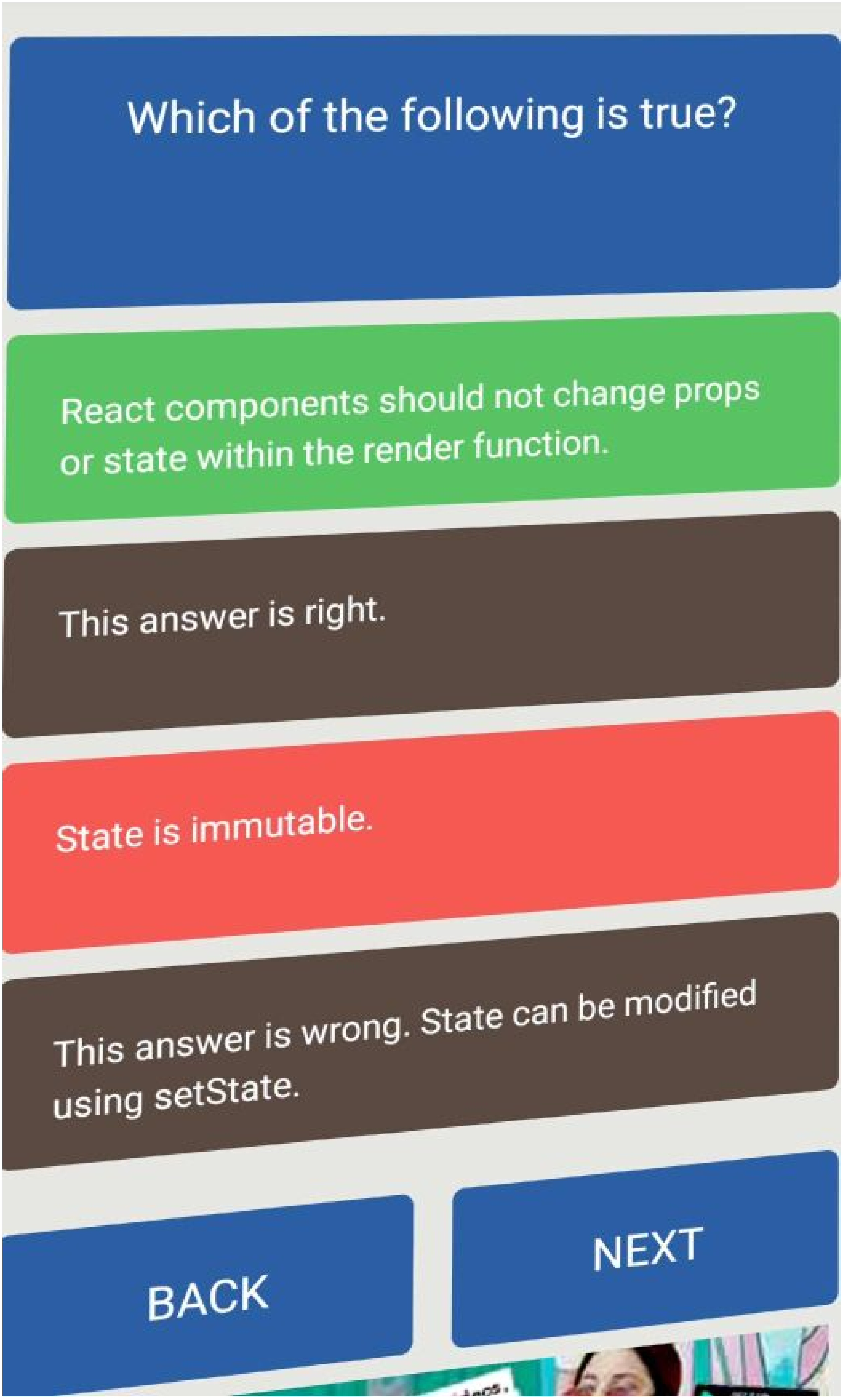

Which of the following is true?

The componentDidMount method of parent component is called after componentDidMount of child components.

This answer is right.

The render method of child components is called before the render method of parent components.

This answer is wrong.



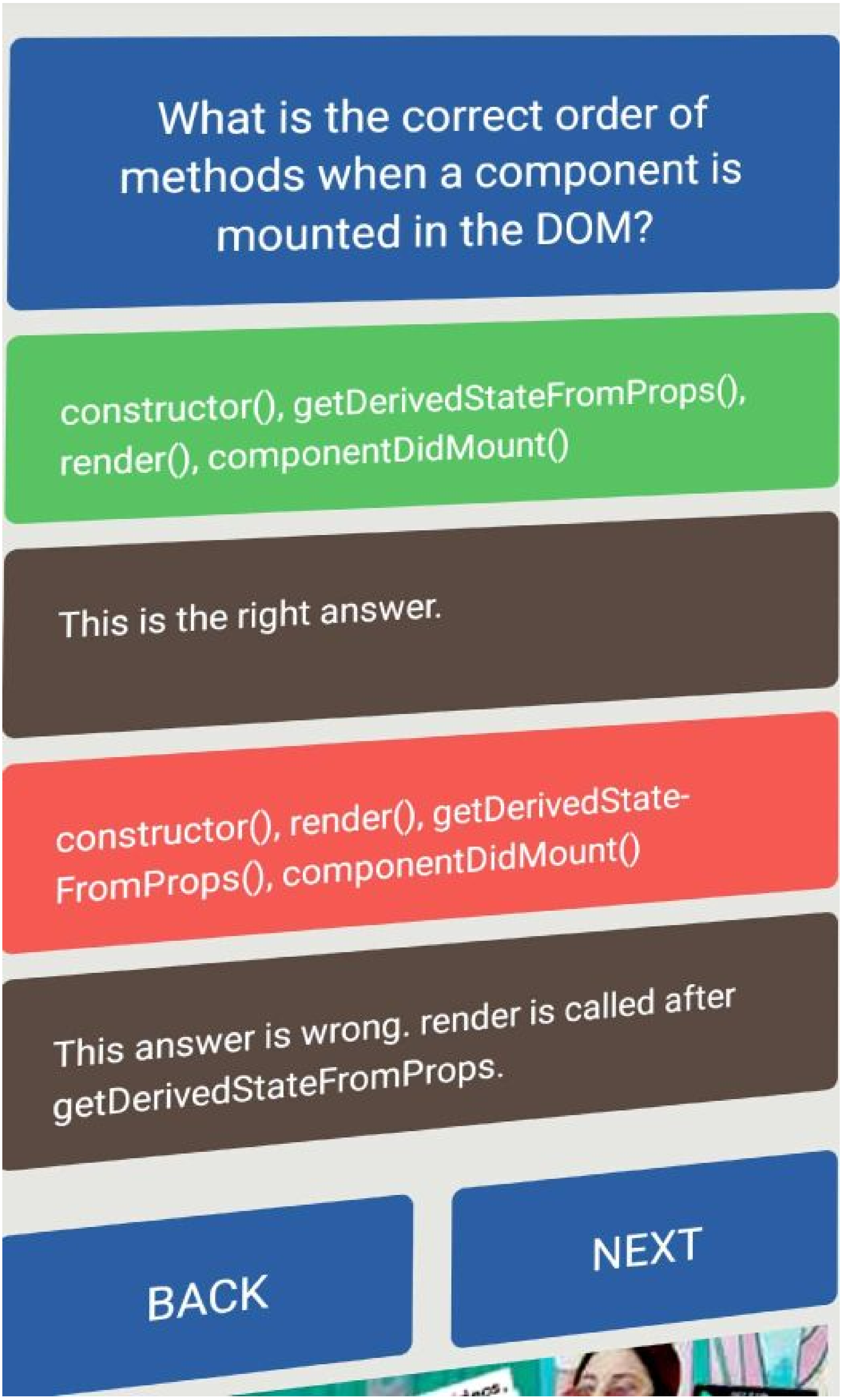
Which of the following is true?

React components should not change props or state within the render function.

This answer is right.

State is immutable.

This answer is wrong. State can be modified using setState.



What is the correct order of methods when a component is mounted in the DOM?

constructor(), getDerivedStateFromProps(), render(), componentDidMount()

This is the right answer.

constructor(), render(), getDerivedState-FromProps(), componentDidMount()

This answer is wrong. render is called after getDerivedStateFromProps.