ANY JS value can be a state variable

- Strings
- Numbers
- Booleans
- ARRAYS!
- OBJECTS!

What do you save as state?

- Anything that can change during run of app
    - Unless based solely on other state
        - Known as **derived state**
        - Ex: A class name based on a state value
        - Don't store class name as state
        - Derive it from the state when needed

Immutable State

- Objects and Arrays have special fact
  - They can **mutate**
  - Change contents without changing identity
- State is used to make rendering decisions
  - Compares "previous" state to "current" state
  - Should elements recreate from components?
- State MUST be **immutable**
  - State Objects/Arrays should never mutate
  - Replace with copies that have changes

Changing Arrays in State

YES:

```
const [names, setNames] = useState( ['Jorts', 'Jean'] );
//...
function addName(newName) {
  setNames( [...names, newName ] ); // set to a new array!
}
```

NO:

```
const [names, setNames] = useState( ['Jorts', 'Jean'] );
//...
function addName(newName) {
  names.push(newName); // BAD! mututes existing array!
  setNames(names); // new array and old array are the same!
}
```

More Complex Array Changes in State

- What if you have more complicated changes?
- Create a new array that is copy of original
- Change that new array in normal way
- Set state to new array

```
const [names, setNames] = useState( ['Jorts', 'Jean'] );
//...
function modifyNames(newName) {
  const newNames = [ ...names ]; // New array that is copy
  newNames.splice(1, 0, 'Nyancat'); // Modify new array
  setNames( newNames ); // set state to a new array!
}
```

Remember "Shallow" vs "Deep"

- A copy using `...` is "shallow"
- If your collection has a collection inside...
    - and you copy it...
    - Inside collection is SAME collection, not copy!

```
const cats = [
  'test',
  [ 'Jorts', 'Nyancat'],
  [ 'Jean'],
];

const copy = [ ...cats ];
copy[0] = 'changed'; // Changing "shallow" value
copy[1][0] = 'Grumpy'; // Changing "deep" array

console.log(cats); // Jorts gone! test still test!
console.log(copy); // Jorts gone! test changed!
```

Changing Objects in State

YES:

```
const [cat, setCat] = useState( { name: 'Jorts' });

function updateCat(age) {
  setCat( { ...cat, age } ); // set to new object!
}
```

NO:

```
const [cat, setCat] = useState( { name: 'Jorts' });

function updateCat(age) {
  cat.age = age; // BAD! Mutates existing object!
  setCat( cat ); // new object and old object are the same!
}
```

More Complex Object Changes in State

- What if you have more complicated changes?
- Create a new object that is copy of original
- Change that new object in normal way
- Set state to new object

```
const [cat, setCat] = useState( { name: 'Jorts', age: 3 } );
//...
function removeAge(cat) {
  const newCat = { ...cat };
  delete newCat.age; // Removes age from object
  setCat( newCat );
}
```

"Shallow" vs "Deep" is true for objects too

- A copy using `...` is "shallow"
- If your collection has a collection inside...
    - and you copy it...
    - Inside collection is SAME collection, not copy!

```
const cat = {
  name: 'Jorts',
  age: 3,
  traits: { orange: true, mode: 'friendly', toebeans: 4 },
};

const copy = { ...cat };
copy.age = 4; // Changing "shallow" value
copy.traits.mode = 'classic'; // Changing "deep" array

console.log(cat); // Classic! age still 3!
console.log(copy); // Classic! age is 4!
```

Remember the timing of setting State

- Incredibly common mistake!

```
const [count, setCount] = useState(1);

return (
  <button
    onClick={ () => {
      setCount( count + 1 );
      console.log(count); // What does this output?
    }}
  > {count} </button>
);
```

- Calling `setCount()` does NOT change `count`!
- Changes `count` from `useState()` next render
    - Which will happen because you called setter
    - But hasn't happened yet

How to get new value of state before next render?

- You literally just set it, you have the value!

Original:

```
onClick={ () => {
  setCount( count + 1 );
  console.log(count); // What does this output?
}}
```

Knowing the next value:

```
onClick={ () => {
  const nextCount = count + 1;
  setCount( nextCount );
  console.log( nextCount ); // What does this output?
}}
```

Setting explicit vs relative values

- If a setter does not have a set time it takes
  - Can result in out-of-sync behavior

```
const [count, setCount] = useState(1);

return (
  <button
    onClick={ () => {
      const delay = Math.floor(Math.random()*2000);
      // Called after 1-2 seconds
      setTimeout( () => setCount( count + 1 ), delay );
    }}
  > {count} </button>
);
```

- `count` in `count + 1` can be an old value

Resolving out of sync state

- Can pass a callback function to setter
  - Callback passed CURRENT state when called
  - Return value is what state is set to

```
const [count, setCount] = useState(1);

return (
  <button
    onClick={ () => {
      const delay = Math.floor(Math.random()*2000);
      // Called after 1-2 seconds
      setTimeout(() => setCount( prev => prev + 1 ), delay);
    }}
  > {count} </button>
);
```