Moving from vanilla JavaScript to React can be a difficult mental lift for most beginner programmers. However, a lot of the mystique can be chocked up to syntactical differences, especially when it comes to the concept of props. Keep these two things in mind if you get thrown by React's props:

1. It's just a parameter. React's syntax may obscure that, but it's always just a parameter.
2. It's just an object. And like all other JavaScript objects, it consists of key-value pairs. React has you build that object in an unconventional way, but it's always just an object.

To help demystify React's props, I'm going to compare a vanilla JavaScript function with a React component. I'll demonstrate how they're built and invoked to emphasize their similarities and differences. Each main function will take a dog object as a parameter, then display something about that dog. And each main function will then be called in another function that will display information about many dogs. I'll also provide a mocked up data file with info about a few dogs, which will be used in both scenarios. For simplicity, assume all files are in the same folder, and that in both scenarios, there exists a main app file that imports and invokes the parent function to finally render the info. Try to ignore the `document.createElement`s in the vanilla functions (they're only included to make the examples more parallel) and instead focus on the props.

Data

Creating a bunch of dogs (in `data.js` file):

```
const firstDog = {
  name: 'Lassie',
  breed: 'collie',
  age: 12
```

```
}

const secondDog = {
  name: 'Bobo',
  breed: 'terrier',
  age: 4
}

const thirdDog = {
  name: 'Sunny',
  breed: 'shepherd',
  age: 7
}

const dogs = [firstDog, secondDog, thirdDog]

export default dogs
```

Vanilla JavaScript Function

Creating the function (in dog.js file):

```
function normalishDogFunction(props) {
  // Actually using the props object
  const statement = `Hello! I'm a ${props.breed}. My name is ${props.name}, and I am
${props.age} years old. Bark!`

  // Creating a DOM node and adding text content to it the old-fashioned way, so this function
can display something to the screen
  const item = document.createElement('li')
  item.textContent = statement

  // Returning DOM element to display, just like in React
  return item
}

export default normalishDogFunction
```

Using the function (in dogList.js file):

```
import dogs from './data.js'
import normalishDogFunction from './dog.js'
```

```
function normalishDogListFunction() {
  // Create a DOM node the old-fashioned way
  const list = document.createElement('ul')

  dogs.forEach(dog => {
    // Invoke the main function by passing it one of the imported objects; using forEach instead of
map for simplicity's sake
    const dogItem = normalishDogFunction(dog)
    list.append(dogItem)
  })

  return list
}

export default normalishDogListFunction
```

React Functional Component

Creating the component (in `Dog.jsx` file):

```
function ReactishDogComponent(props) {
  // Actually using the props object, and in the exact same way as with the vanilla version
  const statement = `Hello! I'm a ${props.breed}. My name is ${props.name}, and I am
${props.age} years old. Bark!`

  // Use JSX syntax to return a virtual node without interacting with the actual document
  return (
    <li>{statement}</li>
  )
}

export default ReactishDogComponent
```

Using the component (in `DogList.jsx` file):

```
import dogs from './data.js'
import ReactishDogComponent from './Dog'

function ReactishDogListComponent() {
  const dogList = dogs.map(dog => {
    // Invoke the main component by passing it one of the imported objects
    return <ReactishDogComponent
      key={dog.name}
```

```
      name={dog.name}
      age={dog.age}
      breed={dog.breed}
    />
  })

  // Again, use JSX syntax to return a virtual node without interacting with the actual document
  return (
    <ul>{dogList}</ul>
  )
}

export default ReactishDogListComponent
```

Analysis

When it comes to the props, here are some key similarities:

- In both cases, the main function just takes a single argument (props)
- In both cases, the main function accesses values on the props object by using dot notation
- In both cases, the main function must be called in another function, in which it receives the props it will end up using

Now, the key difference for the props object:

- In vanilla JavaScript, you pass the entire object as a single entity to the main function
- In React, you individually pass each of the properties that will exist on the props object to the main component; in essence, you build the props object within the main component's invocation

Advanced

The biggest trick with props is a direct consequence of the previously noted difference. Since you are passing the props object directly with vanilla JavaScript, the keys will end up exactly matching. Since you are creating the props object afresh with each invocation of the React component, you can make the keys differ if desired.

If you choose that approach, just remember that the key (what's before the equals sign in the invocation within the parent) must match the name of the property on the props object (what comes after `props.` in the component), while the value (what comes after the equals sign) must match something within the scope of the parent component.

For instance, you could have the following parent component:

```
function SomeOtherParent() {
  const emotions = ['happy', 'sad', 'angry']

  const list = emotions.map(singleEmotion => {
    return <SomeOtherChild
      key={singleEmotion}
      currentEmotion={singleEmotion}
    />
  })

  return (
    <ul>{list}</ul>
  )
}
```

And pair it with the following child component:

```
function SomeOtherChild(props) {
  const isHappy = props.currentEmotion === 'happy'
  const statement = `I'm ${isHappy ? '' : 'not'} happy!`

  return (
```

```
    <li>{statement}</li>
  )
}
```

This works since the key is established as `currentEmotion` in the invocation of the child component within the parent component, and it is then referenced via `props.currentEmotion` in the child component. Similarly, the value is established as `singleEmotion` in the invocation of the child component, which is a value accessible to that invocation at that level of scope within the parent component.

Note that if you tried to use `props.singleEmotion` or `props.emotion` (or any other permutation of that) in the child component, you would get an error because those keys do not exist on the component's props object.

Hopefully that helps to demystify props, but feel free to let me know about any confusion that persists, and I'll try to update this post to further clarify.