

チュートリアル： React の導入

このチュートリアルは React の事前知識ゼロでも読み進められます。

チュートリアルを始める前に

このチュートリアルでは小さなゲームを作成します。自分はゲームを作りたいのではないから、と飛ばしたくなるかもしれませんが、是非目を通してみてください

い。このチュートリアルで学ぶ技法はどのような React のアプリにおいても基本的なものであり、マスターすることで React への深い理解が得られます。

ヒント

このチュートリアルは実際に手を動かして学びたい人向けに構成されています。コンセプトを順番に学んでいきたい人は一段階ずつ学べるガイドを参照してください。このチュートリアルとガイドは互いに相補的なものです。

このチュートリアルは複数のセクションに分割されています。

- チュートリアルの準備：以下のチュートリアルを進めるにあたっての開始地点です。
- 概要：コンポーネントや props、state といった**基礎概念**について学びます。

- ゲームを完成させる：React での開発における**非常によくある技法**について学びます。
- タイムトラベル機能の追加：React 独自の利点について**深い洞察**が得られます。

このチュートリアルから価値を得るために全セクションを一度に終わらせる必要はありません。セクション 1 つや 2 つ分でも構いませんので、できるところまで進めましょう。

これから作るもの

このチュートリアルでは、インタラクティブな三目並べゲーム (tic-tac-toe) の作り方をお見せします。

最終的な結果をここで確認することができます：**最終結果**。まだコードが理解できなくても、あるいはよく知らない構文があっても、心配は要りません。このチュートリアルの目的は、React とその構文について学ぶお手伝いをすることです。

チュートリアルを進める前に三目並べゲームで遊んでみることをお勧めします。いくつか機能があります

が、ひとつには、盤面の右側に番号付きリストがあることに気づかれるでしょう。このリストにはゲーム内で起きたすべての着手 (move) のリストが表示され、ゲームが進むにつれて更新されていきます。

どんな物か分かったら三目並べゲームを閉じて構いません。もっと小さな雛形から始めましょう。次のステップはゲームの構築ができるようにするための環境のセットアップ作業です。

前提知識

HTML と JavaScript に多少慣れていることを想定していますが、他のプログラミング言語を使ってきた人でも進めていくことはできるはずです。また、関数、オブジェクト、配列、あるいは（相対的には重要ではありませんが）クラスといったプログラミングにおける概念について、馴染みがあることを想定しています。

JavaScript を復習する必要がある場合は、このガイドを読むことをお勧めします。また ES6 という

JavaScript の最近のバージョンからいくつかの機能を

使用していることにも注意してください。このチュートリアルでは、アロー関数、クラス、`let` および `const` ステートメントを使用しています。Babel REPL を使って ES6 のコードがどのようにコンパイルされるのか確認することができます。

チュートリアルの準備

このチュートリアルを最後まで進めるための方法が 2 種類あります。ブラウザでコードを書くか、マシン上にローカルな開発環境をセットアップするかのどちらかです。

オプション 1: ブラウザでコードを書く

始めるのに一番手っ取り早い方法です！

まず、この**スターターコード**を新しいタブで開いてください。空の三目並べの盤面と React のコードが表示

されるはずですが。このチュートリアルではこのコードを編集していくことになります。

次のオプションはスキップして、直接概要へと進んで React の全体像を学びましょう。

オプション 2: ローカル開発環境

これは完全にオプションであり、このチュートリアルを進めるのに必須ではありません！

▶ オプション：好きなテキストエディタを使ってローカルでチュートリアルを進める方法

助けて、ハマった！

もし問題にはまったら、コミュニティーによるサポート情報をチェックしてみてください。特に Reactiflux Chat は迅速なヘルプが得られます。良い回答が得られなかった場合や問題が解決しない場合は、issue を作成して下さい。私たちがお手伝いします。

概要

準備が完了したので、React の概要を学びましょう！

React とは？

React はユーザインターフェイスを構築するための、宣言型で効率的で柔軟な JavaScript ライブラリです。複雑な UI を、「コンポーネント」と呼ばれる小さく独立した部品から組み立てることができます。

React にはいくつか異なる種類のコンポーネントがあるのですが、ここでは `React.Component` のサブクラスから始めましょう：

```
class ShoppingList extends React.Component {  
  render() {  
    return (  
      <div className="shopping-list">  
        <h1>Shopping List for  
{this.props.name}</h1>  
        <ul>
```

```
        <li>Instagram</li>
        <li>WhatsApp</li>
        <li>Oculus</li>
    </ul>
</div>
);
}
}

// Example usage: <ShoppingList name="Mark"
/>
```

この妙な XML のようなタグについてはすぐ後で説明します。コンポーネントは、React に何を描画したいかを伝えます。データが変更されると、React はコンポーネントを効率よく更新して再レンダーします。

ここで ShoppingList は **React コンポーネントクラス**、もしくは **React コンポーネント型**です。コンポーネントは props (“properties” の略) と呼ばれるパラメータを受け取り、render メソッドを通じて、表示するビューの階層構造を返します。

render メソッドが返すのはあなたが画面上に表示したいものの**説明書き**です。React はその説明書きを受

け取って画面に描画します。具体的には、`render` は、描画すべきものの軽量な記述形式である **React 要素** というものを返します。たいていの React 開発者は、これらの構造を簡単に記述できる "JSX" と呼ばれる構文を使用しています。`<div />` という構文は、ビルド時に `React.createElement('div')` に変換されます。上記の例は以下のコードと同等です：

```
return React.createElement('div',
  {className: 'shopping-list'},
  React.createElement('h1', /* ... h1
children ... */),
  React.createElement('ul', /* ... ul
children ... */)
);
```

全体バージョンを参照する。

興味があれば、`createElement()` は API リファレンスでより詳細に説明されていますが、このチュートリアルでは直接この関数を使用することはありません。代わりに JSX を使い続けます。

JSX では JavaScript のすべての能力を使うことができます。**どのような** JavaScript の式も JSX 内で中括弧に囲んで記入することができます。各 React 要素は、変数に格納したりプログラム内で受け渡ししたりできる、JavaScript のオブジェクトです。

上記の ShoppingList コンポーネントは `<div />` や `` といった組み込みの DOM コンポーネントのみをレンダーしていますが、自分で書いたカスタム React コンポーネントを組み合わせることも可能です。例えば、`<ShoppingList />` と書いてショッピングリスト全体を指し示すことができます。それぞれの React のコンポーネントはカプセル化されており独立して動作します。これにより単純なコンポーネントから複雑な UI を作成することができます。

スターターコードの中身を確認する

ブラウザでチュートリアルを進めている場合、このスターターコードを新しいタブで開いてください。**ローカル**でチュートリアルを進めている場合は、代わりにプロジェクトフォルダにある `src/index.js` を開い

てください（セットアップ時にこのファイルを既に触ったはずです）。

このスターターコードが我々が作ろうとしているもののベースになります。CSS によるスタイルは既に含まれていますので、React と三目並べのプログラミングに集中できるようになっています。

コードを見てみると、3 つの React コンポーネントがあることが分かります。

- Square（正方形のマス目）
- Board（盤面）
- Game

Square（マス目）コンポーネントは1つの `<button>` をレンダーし、Board（盤面）が9個のマスをレンダーしています。Game コンポーネントは盤面と、後ほど埋めることになるプレイスホルダーを描画しています。この時点ではインタラクティブなコンポーネントはありません。

データを Props 経由で渡す

では手始めに、Board コンポーネントから Square コンポーネントにデータを渡してみましょう。

チュートリアルを進めるにあたって、コードをコピー・ペーストしないで、手でタイプすることをお勧めします。そうすれば手が動きを覚えるとともに、理解も進むようになるでしょう。

Board の renderSquare メソッド内で、props として value という名前の値を Square に渡すようにコードを変更します：

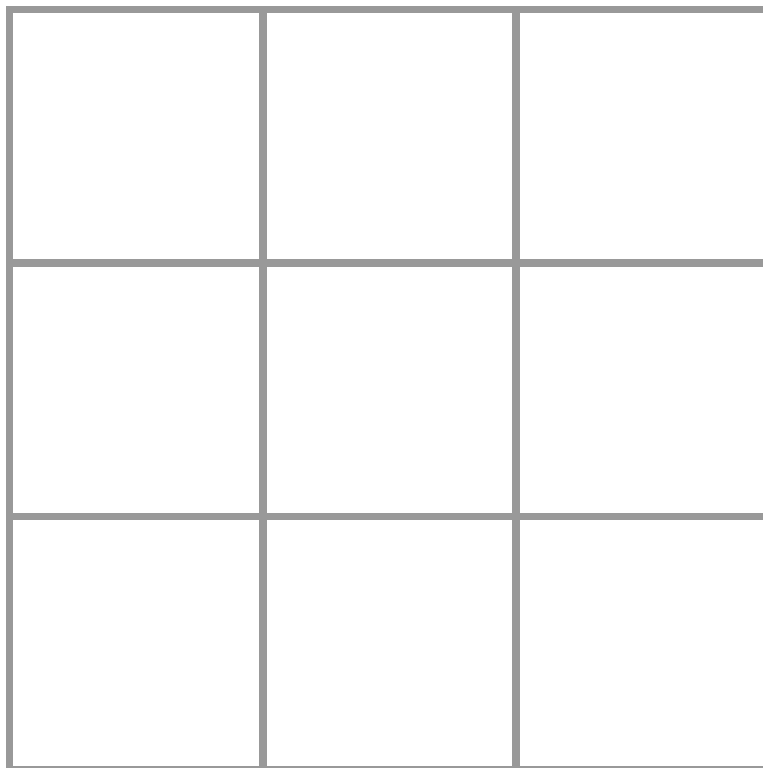
```
class Board extends React.Component {  
  renderSquare(i) {  
    return <Square value={i} />;  
  }  
}
```

そして Square の render メソッドで、渡された値を表示するように、`{/* TODO */}` を `{this.props.value}` に書き換えます。

```
class Square extends React.Component {  
  render() {  
    return (  
      <button className="square">  
        {this.props.value}  
      </button>  
    );  
  }  
}
```

变更前：

Next player: X



変更後：描画された出力の正方形のそれぞれに番号があるはずです。

Next player: X

0	1	2
3	4	5
6	7	8

この時点でのコード全体を見る

おめでとうございます！これで親である Board コンポーネントから子である Square コンポーネントに

「props を渡す」ことができました。React では、親から子へと props を渡すことで、アプリ内を情報が流れていきます。

インタラクティブなコンポーネントを作る

Square コンポーネントがクリックされた場合に "X" と表示されるようにしましょう。まず、Square コンポーネントの `render()` 関数から返されているボタンタグを、以下のように変更してみましょう：

```
class Square extends React.Component {  
  render() {  
    return (  
      <button className="square" onClick=  
{function() { console.log('click'); }}>  
        {this.props.value}  
      </button>  
    );  
  }  
}
```

ここで Square をクリックすると、ブラウザの開発者コンソールに 'click' と表示されるはずです。

補足

タイプ量を減らして this の混乱しやすい挙動を回避するため、この例以降ではアロー関数構文をつかってイベントハンドラを記述します。

```
class Square extends React.Component
{
  render() {
    return (
      <button className="square"
onClick={() => console.log('click')}>
        {this.props.value}
      </button>
    );
  }
}
```

onClick={() => console.log('click')} と記載したときに onClick プロパティに渡しているのは**関数**であることに注意してください。

React はクリックされるまでこの関数を実行しません。() => を書くのを忘れて onClick={console.log('click')} と書いてしまうのは

よくある間違いであり、こうするとコンポーネントが再レンダーされるたびにログが表示されてしまいます。

次のステップとして、Square コンポーネントに自分がクリックされたことを「覚えさせ」て、“X” マークでマス埋めるようにさせます。コンポーネントが何かを「覚える」ためには、**state** というものを使います。

React コンポーネントはコンストラクタで `this.state` を設定することで、状態を持つことができるようになります。 `this.state` はそれが定義されているコンポーネント内でプライベートと見なすべきものです。現在の Square の状態を `this.state` に保存して、マス目がクリックされた時にそれを変更するようにしましょう。

まず、クラスにコンストラクタを追加して `state` を初期化します：

```
class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }

  render() {
    return (
      <button className="square" onClick={()
=> console.log('click')}>
        {this.props.value}
      </button>
    );
  }
}
```

補足

JavaScript のクラスでは、サブクラスのコンストラクタを定義する際は常に `super` を呼ぶ必要があります。constructor を持つ React のクラスコンポーネントでは、すべてコンストラクタを

`super(props)` の呼び出しから始めるべきです。

次に `Square` の `render` メソッドを書き換えて、クリックされた時に `state` の現在値を表示するようにします。

- `<button>` タグ内の `this.props.value` を `this.state.value` に置き換える。
- `onClick={...}` というイベントハンドラを `onClick={() => this.setState({value: 'X'})}` に書き換える。
- 読みやすくするため、`className` と `onClick` の両プロパティをそれぞれ独立した行に配置する。

これらの書き換えの後、`Square` の `render` メソッドから返される `<button>` タグは以下のようになります。

```
class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }

  render() {
    return (
      <button
        className="square"
        onClick={() => this.setState({value:
'X'})})
      >
        {this.state.value}
      </button>
    );
  }
}
```

Square の render メソッド内に書かれた onClick ハンドラ内で this.setState を呼び出すことで、React に <button> がクリックされたら常に再レンダーするよう伝えることができます。データ更新のあと、この Square の this.state.value は 'X' にな

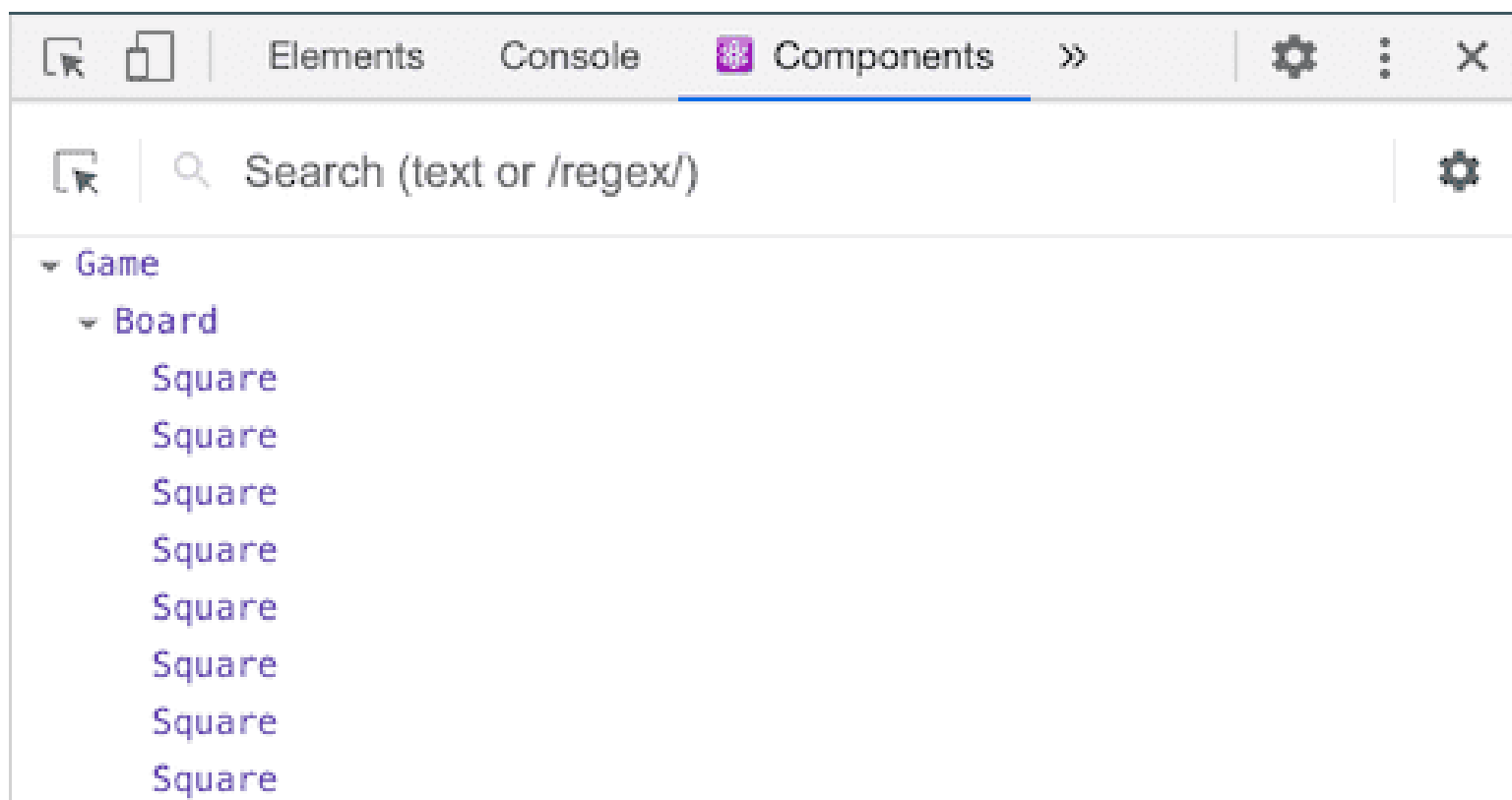
っていますので、盤面に X と表示されることになります。どのマス目をクリックしても X が出てくるはずで
す。

useState をコンポーネント内で呼び出すと、React
はその内部の子コンポーネントも自動的に更新しま
す。




この時点でのコード全体を見る

Developer Tools

Chrome と Firefox 用の React Devtools 拡張機能によ
り、ブラウザの開発ツールで React のコンポーネント
ツリーを調べることができます。



React DevTools を使えば React コンポーネントの props と state を確認できます。

インストールした後は、ページ上の任意の要素で右クリックして、“Inspect” をクリックして開発者向けツールを開くと、React タブ (“ Components” and “ Profiler”) が右端のタブとして表示されます。“ Components” を利用して、コンポーネントツリーを検証します。

ただし、開発者向けツールを CodePen で動作させるには追加のステップが必要です：

1. ログインまたは登録してメールを認証（スパム防止に必要です）。
2. “Fork” ボタンをクリック。
3. “Change View” をクリックして “Debug mode” を選択。
4. 新しいタブを開けば、開発者向けツール内に React タブが現れるようになっているはずです。

ゲームを完成させる

ここまでで三目並べゲームの基本的な部品が揃いました。完全に動作するゲームにするためには、盤面に "X" と "O" を交互に置けるようにすることと、どちらのプレーヤが勝利したか判定できるようにすることが必要です。

State のリフトアップ

現時点では、それぞれの Square コンポーネントがゲームの状態を保持しています。どちらが勝利したかチェックするために、9 個のマス目の値を 1 カ所で管理するようにします。

Board が各 Square に、現時点の state がどうなっているか問い合わせればよいだけでは、と思うかもしれませんが、React でそれをする 것도 可能ですが、コードが分かりにくく、より壊れやすく、リファクタリングしづらいものになるのでお勧めしません。ここでのべ

ストの解決策はそうではなく、ゲームの状態を各 Square の代わりに親の Board コンポーネントで保持することです。Board コンポーネントはそれぞれの Square に props を渡すことで、何を表示すべきかを伝えられます。以前にそれぞれの Square に番号を表示させた時と同じです。

複数の子要素からデータを集めたい、または 2 つの子コンポーネントに互いにやりとりさせたいと思った場合は、代わりに親コンポーネント内で共有の state を宣言する必要があります。親コンポーネントは props を使うことで子に情報を返すことができます。こうすることで、子コンポーネントが兄弟同士、あるいは親との間で常に同期されるようになります。

このように state を親コンポーネントにリフトアップ (lift up) することは React コンポーネントのリファクタリングでよくあることですので、この機会に挑戦してみましょう。

Board にコンストラクタを追加し、初期 state として 9 個のマス目に対応する 9 個の null 値をセットします。


```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
    };
  }

  renderSquare(i) {
    return <Square value={i} />;
  }
}
```

後で盤面が埋まっていくと、`this.state.squares` 配列はこのような見た目になるでしょう：

```
[
  'O', null, 'X',
  'X', 'X', 'O',
  'O', null, null,
]
```

Board の `renderSquare` メソッドは現在以下のようになっています：

```
renderSquare(i) {  
  return <Square value={i} />;  
}
```

まず Board から value プロパティを渡して 0 から 8 までの数字をそれぞれのマス目に表示させたのでしたね。その後のステップで Square 自身の state を使って "X" マークを表示させるようにしたのでした。なので、今のところは Board から渡されている value プロパティの値は無視されています。

改めて、props を渡すメカニズムを使うようにしましょう。Board を書き換えて、それぞれの個別の Square に現在の値（'X'、'0' または null）を伝えるようにします。squares という配列は Board のコンストラクタで定義していますので、Board の renderSquare がそこから値を読み込むように書き換えましょう。

```
renderSquare(i) {  
  return <Square value=  
    {this.state.squares[i]} />;  
}
```

}

この時点でのコード全体を見る

これでそれぞれの Square が value プロパティ（'X'、'O'、または空のマス目の場合は null）を受け取るようになります。

次に、マス目がクリックされた時の挙動を変更しましょう。現在、どのマス目に何が入っているのかを管理しているのは Board です。Square が Board の state を更新できるようにする必要があります。state はそれを定義しているコンポーネント内でプライベートなものですので、Square から Board の state を直接書き換えることはできません。

代わりに、Board から Square に関数を渡すことにして、マス目がクリックされた時に Square にその関数を呼んでもらうようにしましょう。renderSquare メソッドを以下のように書き換えましょう：

```
renderSquare(i) {  
  return (
```

```
<Square
  value={this.state.squares[i]}
  onClick={() => this.handleClick(i)}
/>
);
}
```

補足

読みやすさのために return される要素を複数行に分割しています。また JavaScript が return の後にセミコロンを挿入するのを防ぐため、カッコを付け加えています。

現在、Board から Square には props として 2 つの値を渡しています。value と onClick です。onClick プロパティはマス目がクリックされた時に Square が呼び出すためのものです。Square に以下のような変更を加えましょう。

- Square の render メソッド内の
this.state.value を this.props.value に書き換える
- Square の render メソッド内の
this.setState() を this.props.onClick() に書き換える
- Square はもはやゲームの状態を管理しなくなったので、Square の constructor を削除する

これらの変更のあと、Square コンポーネントは以下のようになります。

```
class Square extends React.Component {  
  render() {  
    return (  
      <button  
        className="square"  
        onClick={() => this.props.onClick()}  
      >  
        {this.props.value}  
      </button>  
    );  
  }  
}
```

Square がクリックされると、Board から渡された `onClick` 関数がコールされます。どのようになっているのかおさらいしましょう。

1. 組み込みの DOM コンポーネントである `<button>` に `onClick` プロパティが設定されているため React がクリックに対するイベントリスナを設定します。
2. ボタンがクリックされると、React は Square の `render()` メソッド内に定義されている `onClick` のイベントハンドラをコールします。
3. このイベントハンドラが `this.props.onClick()` をコールします。Square の `onClick` プロパティは Board から渡されているものです。
4. Board は Square に `onClick={() => this.handleClick(i)}` を渡していたので、Square はクリックされたときに Board の `handleClick(i)` を呼び出します。
5. まだ `handleClick()` は定義していないので、コードがクラッシュします。Square をクリックする

と、“this.handleClick is not a function” といった赤いエラー画面が表示されるはずです。

補足

DOM 要素である `<button>` は組み込みコンポーネントなので、`onClick` 属性は React にとって特別な意味を持っています。Square のようなカスタムコンポーネントでは、名前の付け方はあなたの自由です。Square の `onClick` プロパティや Board の `handleClick` メソッドについては別の名前を付けたとしても同じように動作します。React では、イベントを表す props には `on[Event]` という名前、イベントを処理するメソッドには `handle[Event]` という名前を付けるのが慣習となっています。

まだ `handleClick` を定義していないので、マスをクリックしようとするときエラーが出るはずです。この `handleClick` を Board クラスに加えましょう。

```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
    };
  }

  handleClick(i) {
    const squares =
this.state.squares.slice();
    squares[i] = 'X';
    this.setState({squares: squares});
  }

  renderSquare(i) {
    return (
      <Square
        value={this.state.squares[i]}
        onClick={() => this.handleClick(i)}
      />
    );
  }

  render() {
    const status = 'Next player: X';

    return (
```



```

    <div>
      <div className="status">{status}</div>
    </div>

    <div className="board-row">
      {this.renderSquare(0)}
      {this.renderSquare(1)}
      {this.renderSquare(2)}
    </div>
    <div className="board-row">
      {this.renderSquare(3)}
      {this.renderSquare(4)}
      {this.renderSquare(5)}
    </div>
    <div className="board-row">
      {this.renderSquare(6)}
      {this.renderSquare(7)}
      {this.renderSquare(8)}
    </div>
  </div>
);
}
}

```

この時点でのコード全体を見る

これらの変更を加えれば、再びマス目をクリックすると値が書き込まれるようになります。しかし今や、状

態は個々の Square コンポーネントではなく Board コンポーネント内に保存されています。Board の state が変更されると、個々の Square コンポーネントも自動的に再レンダーされます。全てのマス目の状態を Board コンポーネント内で保持するようにしたことで、この後でどちらが勝者か判定できるようになります。

Square コンポーネントはもう自分で state を管理しないようになったので、Board コンポーネントから値を受け取って、クリックされた時はそのことを Board コンポーネントに伝えるだけになりました。React 用語でいうと、Square コンポーネントは**制御されたコンポーネント** (controlled component) になったということです。Board が Square コンポーネントを全面的に制御しています。

handleClick 内では、squares を直接変更する代わりに、`.slice()` を呼んで配列のコピーを作成していることに注意してください。次のセクションで、なぜ squares 配列のコピーを作成しているのか説明します。

イミュータビリティは何故重要なのか

上記のコード例において、現在の配列を直接変更する代わりに、`.slice()` メソッドを使って `square` 配列のコピーを作成することをお勧めしました。ここでイミュータビリティ (immutability; 不変性) について解説し、それがなぜ重要なのかについて説明します。

一般的に、変化するデータに対しては 2 種類のアプローチがあります。1 番目のアプローチはデータの値を直接いじってデータを**ミューテート (mutate; 書き換え)** することです。2 番目のアプローチは、望む変更を加えた新しいデータのコピーで古いデータを置き換えることです。

ミューテートを伴うデータの変化

```
var player = {score: 1, name: 'Jeff'};
player.score = 2;
// Now player is {score: 2, name: 'Jeff'}
```

ミューテートを伴わないデータの変化

```
var player = {score: 1, name: 'Jeff'};

var newPlayer = Object.assign({}, player,
{score: 2});
// Now player is unchanged, but newPlayer is
{score: 2, name: 'Jeff'}

// Or if you are using object spread syntax
proposal, you can write:
// var newPlayer = {...player, score: 2};
```

最終的な結果は同じですが、直接データのミューテート（すなわち内部データの書き換え）をしないことで、以下に述べるようないくつかの利点を得られます。

複雑な機能が簡単に実装できる

イミュータビリティにより、複雑な機能の実装がとても簡単になります。このチュートリアルの後の部分で、三目並べの着手の履歴を振り返って以前の着手まで「巻き戻し」ができる「タイムトラベル」機能を実装します。このような機能はゲーム特有のものではありません。直接的なデータのミューテートを避けることで、ゲームの以前のヒストリをそのまま保って後で再利用することができるようになります。

変更の検出

ミュータブル (mutable) なオブジェクトは中身が直接書き換えられるため、変更があったかどうかの検出が困難です。ミュータブルなオブジェクト変更の検出のためには、以前のコピーと比較してオブジェクトツリーの全体を走査する必要があります。

イミュータブルなオブジェクトでの変更の検出はとても簡単です。参照しているイミュータブルなオブジェクトが前と別のものであれば、変更があったということです。

React の再レンダータイミングの決定

イミュータビリティの主な利点は、React で *pure component* を構築しやすくなるということです。イミュータブルなデータは変更があったかどうか簡単に分かるため、コンポーネントをいつ再レンダーすべきなのか決定しやすくなります。

`shouldComponentUpdate()` および *pure component* をどのように作成するのかについては、パフォーマンス最適化のページで説明しています。

関数コンポーネント

ここで Square を関数コンポーネントに書き換えましょう。

React における関数コンポーネントとは、`render` メソッドだけを有して自分の `state` を持たないコンポーネントを、よりシンプルに書くための方法です。

`React.Component` を継承するクラスを定義する代わりに、`props` を入力として受け取り表示すべき内容を

返す関数を定義します。関数コンポーネントはクラスよりも書くのが楽であり、多くのコンポーネントはこれで書くことができます。

Square クラスを以下の関数で書き換えましょう。

```
function Square(props) {  
  return (  
    <button className="square" onClick=  
      {props.onClick}>  
      {props.value}  
    </button>  
  );  
}
```

2 箇所に出てくる `this.props` は、`props` に書き換えました。

この時点でのコード全体を見る

補足

Square を関数コンポーネントに変えた際、
`onClick={() => this.props.onClick()}` を
より短い `onClick={props.onClick}` に書き換
えました（**両側**でカッコが消えています）。

手番の処理

さて次に、この三目並べの明らかな欠点、すなわち
"O" がまだ盤面に出てこないという問題を修正しまし
よう。

デフォルトでは、先手を "X" にします。Board のコン
ストラクタで state の初期値を変えればこのデフォル
ト値は変更可能です。

```
class Board extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      squares: Array(9).fill(null),  
      xIsNext: true,  
    };  
  }  
}
```



```
};  
}
```

プレーヤが着手するたびに、どちらのプレーヤの手番なのかを決める `xIsNext`（真偽値）が反転され、ゲームの状態が保存されます。Board の `handleClick` 関数を書き換えて `xIsNext` の値を反転させるようにします。

```
handleClick(i) {  
  const squares =  
this.state.squares.slice();  
  squares[i] = this.state.xIsNext ? 'X' :  
'O';  
  this.setState({  
    squares: squares,  
    xIsNext: !this.state.xIsNext,  
  });  
}
```

この変更により、“X” 側と “O” 側が交互に着手できるようになります。試してみてください！

Board の render 内にある "status" テキストも変更して、どちらのプレーヤの手番なのかを表示するようにしましょう。

```
render() {  
  const status = 'Next player: ' +  
(this.state.xIsNext ? 'X' : 'O');  
  
  return (  
    // the rest has not changed
```

ここまでの変更により、Board コンポーネントは以下のようにになっているはずです。

```
class Board extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      squares: Array(9).fill(null),  
      xIsNext: true,  
    };  
  }  
}
```

```
handleClick(i) {
  const squares =
this.state.squares.slice();
  squares[i] = this.state.xIsNext ? 'X' :
'0';
  this.setState({
    squares: squares,
    xIsNext: !this.state.xIsNext,
  });
}

renderSquare(i) {
  return (
    <Square
      value={this.state.squares[i]}
      onClick={() => this.handleClick(i)}
    />
  );
}

render() {
  const status = 'Next player: ' +
(this.state.xIsNext ? 'X' : '0');

  return (
    <div>
      <div className="status">{status}
```

```
</div>
    <div className="board-row">
      {this.renderSquare(0)}
      {this.renderSquare(1)}
      {this.renderSquare(2)}
    </div>
    <div className="board-row">
      {this.renderSquare(3)}
      {this.renderSquare(4)}
      {this.renderSquare(5)}
    </div>
    <div className="board-row">
      {this.renderSquare(6)}
      {this.renderSquare(7)}
      {this.renderSquare(8)}
    </div>
  </div>
);
}
}
```

この時点でのコード全体を見る

ゲーム勝者の判定

どちらの手番なのかを表示できたので、次にやることはゲームが決着して次の手番がなくなった時にそれを表示することです。ファイル末尾に以下のヘルパー関数をコピーして貼り付けてください。

```
function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6],
  ];
  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] ===
squares[b] && squares[a] === squares[c]) {
      return squares[a];
    }
  }
  return null;
}
```

9 つの square の配列が与えられると、この関数は勝者がいるか適切に確認し、 'X' か 'O'、あるいは null を返します。

Board の render 関数内で

calculateWinner(squares) を呼び出して、いずれかのプレーヤが勝利したかどうか判定します。決着がついた場合は "Winner: X" あるいは "Winner: O" のようなテキストを表示するとよいでしょう。Board の render 関数の status 宣言を以下のコードで置き換えましょう。

```
render() {  
  const winner =  
calculateWinner(this.state.squares);  
  let status;  
  if (winner) {  
    status = 'Winner: ' + winner;  
  } else {  
    status = 'Next player: ' +  
(this.state.xIsNext ? 'X' : 'O');  
  }  
  
  return (  

```

```
// the rest has not changed
```

Board の handleClick を書き換えて、ゲームの決着が既についている場合やクリックされたマス目が既に埋まっている場合に早期に return するようにします。

```
handleClick(i) {
  const squares =
this.state.squares.slice();
  if (calculateWinner(squares) ||
squares[i]) {
    return;
  }
  squares[i] = this.state.xIsNext ? 'X' :
'0';
  this.setState({
    squares: squares,
    xIsNext: !this.state.xIsNext,
  });
}
```

この時点でのコード全体を見る

おめでとうございます！これで動作する三目並べゲームができました。そして React の基本についても学ぶことができました。このゲームの真の勝者は**あなた**かもしれませんね。

タイムトラベル機能の追加

最後の練習として、以前の着手まで「時間を巻き戻す」ことができるようにしましょう。

着手の履歴の保存

`squares` の配列をミューテートしていたとすれば、タイムトラベルの実装はとても難しかったでしょう。

しかし我々は着手があるたびに `squares` のコピーを作り、この配列をイミュータブルなものとして扱っていました。このため、`squares` の過去のバージョンをすべて保存しておいて、過去の手番をさかのぼることができるようになります。

過去の squares の配列を、history という名前の別の配列に保存しましょう。この history 配列は初手から最後まで盤面の全ての状態を表現しており、以下のような構造を持っています。

```
history = [  
  // Before first move  
  {  
    squares: [  
      null, null, null,  
      null, null, null,  
      null, null, null,  
    ],  
  },  
  // After first move  
  {  
    squares: [  
      null, null, null,  
      null, 'X', null,  
      null, null, null,  
    ],  
  },  
  // After second move  
  {  
    squares: [  
      null, null, null,
```

```
        null, 'X', null,  
        null, null, 'O',  
    ]  
},  
// ...  
]
```

ここで、この `history` の状態をどのコンポーネントが保持すべきか考える必要があります。

State のリフトアップ、再び

トップレベルの `Game` コンポーネント内で過去の着手の履歴を表示したいと思います。そのためには `Game` コンポーネントが `history` にアクセスできる必要がありますので、`history` という `state` はトップレベルの `Game` コンポーネントに置くようにしましょう。

`history state` を `Game` コンポーネント内に置くことで、`squares` の `state` を、子である `Board` コンポーネントから取り除くことができます。Square コンポーネントにあった「`state` をリフトアップ」して `Board` コンポーネントに移動したときと全く同様にして、今度

は Board にある state をトップレベルの Game コンポーネントにリフトアップしましょう。これにより Game コンポーネントは Board のデータを完全に制御することになり、history 内の過去の手番のデータを Board にレンダーさせることができるようになります。

まず、Game コンポーネントの初期 state をコンストラクタ内でセットします。

```
class Game extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      history: [{
        squares: Array(9).fill(null),
      }],
      xIsNext: true,
    };
  }

  render() {
    return (
      <div className="game">
```

```

    <div className="game-board">
      <Board />
    </div>
    <div className="game-info">
      <div>{/* status */}</div>
      <ol>{/* TODO */}</ol>
    </div>
  </div>
);
}
}

```

次に、Board コンポーネントが squares と onClick プロパティを Game コンポーネントから受け取るようにします。Board 内には多数のマス目に対応するクリックハンドラが1つだけあるので、Square の位置を onClick ハンドラに渡してどのマス目がクリックされたのかを伝えるようにします。以下の手順で Board コンポーネントを書き換えます。

- Board の constructor を削除する。
- Board の renderSquare にある `this.state.squares[i]` を

`this.props.squares[i]` に置き換える。

- Board の `renderSquare` にある

`this.handleClick(i)` を

`this.props.onClick(i)` に置き換える。

Board コンポーネントは現時点で以下のようになります。

```
class Board extends React.Component {
  handleClick(i) {
    const squares =
this.state.squares.slice();
    if (calculateWinner(squares) ||
squares[i]) {
      return;
    }
    squares[i] = this.state.xIsNext ? 'X' :
'0';
    this.setState({
      squares: squares,
      xIsNext: !this.state.xIsNext,
    });
  }

  renderSquare(i) {
```

```
    return (  
      <Square  
        value={this.props.squares[i]}  
        onClick={() =>  
this.props.onClick(i)}  
      />  
    );  
  }  
}
```

```
render() {  
  const winner =  
calculateWinner(this.state.squares);  
  let status;  
  if (winner) {  
    status = 'Winner: ' + winner;  
  } else {  
    status = 'Next player: ' +  
(this.state.xIsNext ? 'X' : 'O');  
  }  
}
```

```
    return (  
      <div>  
        <div className="status">{status}</div>  
        <div className="board-row">  
          {this.renderSquare(0)}  
          {this.renderSquare(1)}  
          {this.renderSquare(2)}  
        </div>  
      </div>  
    );  
  }  
}
```

```

    <div className="board-row">
      {this.renderSquare(3)}
      {this.renderSquare(4)}
      {this.renderSquare(5)}
    </div>
    <div className="board-row">
      {this.renderSquare(6)}
      {this.renderSquare(7)}
      {this.renderSquare(8)}
    </div>
  </div>
);
}
}

```

Game コンポーネントの render 関数を更新して、ゲームのステータステキストの決定や表示の際に最新の履歴が使われるようにします。

```

render() {
  const history = this.state.history;
  const current = history[history.length -
1];
  const winner =
calculateWinner(current.squares);

```

```
let status;
if (winner) {
  status = 'Winner: ' + winner;
} else {
  status = 'Next player: ' +
(this.state.xIsNext ? 'X' : 'O');
}

return (
  <div className="game">
    <div className="game-board">
      <Board
        squares={current.squares}
        onClick={(i) =>
this.handleClick(i)}
      />
    </div>
    <div className="game-info">
      <div>{status}</div>
      <ol>{/* TODO */}</ol>
    </div>
  </div>
);
}
```


Game コンポーネントがゲームのステータステキストを表示するようになったので、対応するコードは Board 内の render メソッドからは削除できます。このリファクタリングの後で、Board の render 関数は以下のようになります。

```
render() {  
  return (  
    <div>  
      <div className="board-row">  
        {this.renderSquare(0)}  
        {this.renderSquare(1)}  
        {this.renderSquare(2)}  
      </div>  
      <div className="board-row">  
        {this.renderSquare(3)}  
        {this.renderSquare(4)}  
        {this.renderSquare(5)}  
      </div>  
      <div className="board-row">  
        {this.renderSquare(6)}  
        {this.renderSquare(7)}  
        {this.renderSquare(8)}  
      </div>  
    </div>  
  )  
}
```

```
);  
}
```

最後に handleClick メソッドを Board コンポーネントから Game コンポーネントに移動します。また Game コンポーネントの state は異なる形で構成されていますので、handleClick の中身も修正する必要があります。Game 内の handleClick メソッドで、新しい履歴エントリを history に追加します。

```
handleClick(i) {  
  const history = this.state.history;  
  const current = history[history.length -  
1];  
  const squares = current.squares.slice();  
  if (calculateWinner(squares) ||  
squares[i]) {  
    return;  
  }  
  squares[i] = this.state.xIsNext ? 'X' :  
'O';  
  this.setState({  
    history: history.concat([  
      squares: squares,
```

```
    }]),  
    xIsNext: !this.state.xIsNext,  
  });  
}
```

補足

`push()` メソッドの方に慣れているかもしれませんが、それと違って `concat()` は元の配列をミューテートしないため、こちらを利用します。

現時点で Board コンポーネントに必要なのは `renderSquare` と `render` メソッドだけです。ゲームの状態と `handleClick` メソッドは Game コンポーネント内にあります。

この時点でのコード全体を見る

過去の着手の表示

三目並べの履歴を記録しているので、これを過去の着手のリストとしてプレーヤに表示することが可能です。

以前、React 要素は第一級の JavaScript オブジェクトであり、それらをアプリケーション内で受け渡しできるということを学びました。React で複数の要素を描画するには、React 要素の配列を使うことができます。

JavaScript では、配列には `map()` メソッドが存在しており、これはデータを別のデータにマップするのによく利用されます。例えば：

```
const numbers = [1, 2, 3];  
const doubled = numbers.map(x => x * 2); //  
[2, 4, 6]
```

`map` メソッドを使うことで、着手履歴の配列をマップして画面上のボタンを表現する React 要素を作りだし、過去の手番に「ジャンプ」するためのボタンの一覧を表示できます。

Game の render メソッド内で history に map を作用させてみましょう。

```
render() {
  const history = this.state.history;
  const current = history[history.length -
1];
  const winner =
calculateWinner(current.squares);

  const moves = history.map((step, move)
=> {
    const desc = move ?
      'Go to move #' + move :
      'Go to game start';
    return (
      <li>
        <button onClick={() =>
this.jumpTo(move)}>{desc}</button>
      </li>
    );
  });

  let status;
  if (winner) {
    status = 'Winner: ' + winner;
```

```

    } else {
        status = 'Next player: ' +
        (this.state.xIsNext ? 'X' : 'O');
    }

    return (
        <div className="game">
            <div className="game-board">
                <Board
                    squares={current.squares}
                    onClick={(i) =>
this.handleClick(i)}
                />
            </div>
            <div className="game-info">
                <div>{status}</div>
                <ol>{moves}</ol>
            </div>
        </div>
    );
}

```

この時点でのコード全体を見る

history 配列をループ処理する部分では、step という変数が history 内の現在の要素を参照し、move という変数が現在の要素のインデックスを参照してい

ます。ここでは `step` はその後何にも割り当てないので、`move` の方にのみ興味があります。

ゲームの履歴内にある三目並べのそれぞれの着手に対応して、ボタン `<button>` を有するリストアイテム `` を作ります。ボタンには `onClick` ハンドラがあり、それは `this.jumpTo()` というメソッドを呼び出します。まだ `jumpTo()` は実装していません。ひとまずこのコードにより、ゲーム内で行われた着手のリストが表示されるようになりましたが、同時に開発者ツールのコンソール内に以下の警告も出力されているはずです：

Warning: Each child in an array or iterator should have a unique "key" prop. Check the render method of "Game".

この警告が何を意味するのかについて説明しましょう。

key を選ぶ

リストをレンダーする際、リストの項目それぞれについて、React はとある情報を保持します。リストが変更になった場合、React はどのアイテムが変更になったのかを知る必要があります。リストのアイテムは追加された可能性も、削除された可能性も、並び替えられた可能性も、中身自体が変更になった可能性もあります。

例えば以下のツリーから：

```
<li>Alexa: 7 tasks left</li>  
<li>Ben: 5 tasks left</li>
```

以下のツリーへ遷移する場合を想像してみてください：

```
<li>Ben: 9 tasks left</li>  
<li>Claudia: 8 tasks left</li>  
<li>Alexa: 5 tasks left</li>
```


タスクの数も変わっていますが、これを人間が見た場合、おそらく Alexa と Ben の順番が変わって、その 2 人の間に Claudia が挿入されている、と考えるでしょう。しかし React は単なるコンピュータプログラムなので、あなたが意図するところを理解しません。React は我々の意図までは理解しないので、リストの項目それぞれに対して *key* プロパティを与えることで、兄弟要素の中でそのアイテムが区別できるようにしてあげる必要があります。このケースでは、`alex`、`ben`、`claudia` の文字列を使う方法があります。データベースからのデータを表示している場合は、`Alexa`、`Ben`、`Claudia` のデータベース内での ID を *key* として使うこともできるでしょう。

```
<li key={user.id}>{user.name}:  
  {user.taskCount} tasks left</li>
```

リストが再レンダーされる際、React はそれぞれのリスト項目の *key* について、前回のリスト項目内に同じ *key* を持つものがないか探します。もし以前になかった

た key がリストに含まれていれば、React はコンポーネントを作成します。もし以前のリストにあった key が新しいリストに含まれていなければ、React は以前のコンポーネントを破棄します。もし 2 つの key がマッチした場合、対応するコンポーネントは移動されます。key はそれぞれのコンポーネントの同一性に関する情報を React に与え、それにより React は再レンダ一間で state を保持できるようになります。もしコンポーネントの key が変化していれば、コンポーネントは破棄されて新しい state で再作成されます。

key は特別なプロパティであり React によって予約されています（より応用的な機能である ref も同様です）。要素が作成される際、React は key プロパティを引き抜いて、返される要素に直接その key を格納します。key は props の一部のようにも思えますが、`this.props.key` で参照できません。React はどの子要素を更新すべきかを決定する際に、key を自動的に使用します。コンポーネントが自身の key について確認する方法はありません。

動的なリストを構築する場合は正しい key を割り当てることが強く推奨されます。適切な key がない場合は、データ構造を再構成してそのような key が存在するようにするべきかもしれません。

key が指定されなかった場合、React は警告を表示し、デフォルトで key として配列のインデックスを使用します。配列のインデックスを key として使うことは、項目を並び替えたり挿入/削除する際に問題の原因となります。明示的に `key={i}` と渡すことで警告を消すことはできますが、配列のインデックスを使う場合と同様な問題が生じるためほとんどの場合は推奨されません。

key はグローバルに一意である必要はありません。コンポーネントとその兄弟の間で一意であれば十分です。

タイムトラベルの実装

三目並べゲームの履歴内においては、すべての着手にはそれに関連付けられた一意な ID が存在します。すな

わち着手順の連番数字のことです。着手はゲームの最中に並び変わったり削除されたり挿入されたりすることはありませんから、着手のインデックスを key として使うのは安全です。

Game コンポーネントの render メソッド内で、key は `<li key={move}>` のようにして加えることができ、これで React の key に関する警告は表示されなくなります。

```
const moves = history.map((step, move)
=> {
  const desc = move ?
    'Go to move #' + move :
    'Go to game start';
  return (
    <li key={move}>
      <button onClick={() =>
this.jumpTo(move)}>{desc}</button>
    </li>
  );
});
```

この時点でのコード全体を見る

まだ `jumpTo` メソッドが未定義なので、このリスト項目内のボタンをクリックするとエラーが発生します。
`jumpTo` を実装する前に、`Game` コンポーネントの `state` に `stepNumber` という値を加えます。これは、いま何手目の状態を見ているのかを表すのに使います。

まず、`Game` の `constructor` 内で、`state` の初期値として `stepNumber: 0` を加えます。

```
class Game extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      history: [{  
        squares: Array(9).fill(null),  
      }],  
      stepNumber: 0,  
      xIsNext: true,  
    };  
  }  
}
```

次に Game 内に jumpTo メソッドを定義してその stepNumber が更新されるようにします。また更新しようとしている stepNumber の値が偶数だった場合は xIsNext を true に設定します。

```
handleClick(i) {  
  // this method has not changed  
}  
  
jumpTo(step) {  
  this.setState({  
    stepNumber: step,  
    xIsNext: (step % 2) === 0,  
  });  
}  
  
render() {  
  // this method has not changed  
}
```

jumpTo メソッド内では state の history プロパティは更新していないことに注意してください。これは、state の更新はマージされるから、より簡単に言うと、

React は `setState` で直接指定されたプロパティのみを更新しほかの `state` はそのまま残すからです。詳しくは**ドキュメントを参照**。

では次に、マス目をクリックしたときに実行される `Game` の `handleClick` メソッドに、いくつかの変更を加えます。

今加えた `state` である `stepNumber` は現在ユーザに見せている着手を反映しています。新しい着手が発生した場合は、`this.setState` の引数の一部として `stepNumber: history.length` を加えることで、`stepNumber` を更新する必要があります。

また `this.state.history` から読み取っているところを `this.state.history.slice(0, this.state.stepNumber + 1)` に書き換えます。これにより、「時間の巻き戻し」をしてからその時点で新しい着手を起こした場合に、そこから見て「将来」にある履歴（もはや正しくなくなったもの）を確実に捨て去ることができます。

```
handleClick(i) {
```

```
    const history =
this.state.history.slice(0,
this.state.stepNumber + 1);
    const current = history[history.length -
1];
    const squares = current.squares.slice();
    if (calculateWinner(squares) ||
squares[i]) {
        return;
    }
    squares[i] = this.state.xIsNext ? 'X' :
'0';
    this.setState({
        history: history.concat([{
            squares: squares
        }]),
        stepNumber: history.length,
        xIsNext: !this.state.xIsNext,
    });
}
```

最後に、Game コンポーネントの render を書き換えて、常に最後の着手後の状態をレンダーするのではなく stepNumber によって現在選択されている着手をレンダーするようにします。


```
render() {  
    const history = this.state.history;  
    const current =  
history[this.state.stepNumber];  
    const winner =  
calculateWinner(current.squares);  
  
    // the rest has not changed
```

ゲーム履歴内のどの手番をクリックした場合でも、三目並べの盤面は、該当の着手が発生した直後の状態を表示するように更新されるはずです。

この時点でのコード全体を見る

まとめ

おめでとうございます! これで以下のような機能を有する三目並べゲームの完成です。

- 三目並べが遊べる
- 決着がついたときに表示ができる
- ゲーム進行にあわせて履歴が保存される

- 着手の履歴の見直しや盤面の以前の状態の参照ができる

よくできました！ あなたが React がどのように動作するのか、ちゃんと理解できたと感じていることを願っています。

最終的な結果は、ここで確認することができます：**最終結果**

まだ時間がある場合や、今回身につけた新しいスキルを練習してみたい場合に、あなたが挑戦できる改良のアイデアを以下にリストアップしています。後ろの方ほど難易度が上がります：

1. 履歴内のそれぞれの着手の位置を (col, row) というフォーマットで表示する。
2. 着手履歴のリスト中で現在選択されているアイテムを太字にする。
3. Board でマス目を並べる部分を、ハードコーディングではなく 2 つのループを使用するように書き換える。

4. 着手履歴のリストを昇順・降順いずれでも並べかえられるよう、トグルボタンを追加する。
5. どちらかが勝利した際に、勝利につながった3つのマス目をハイライトする。
6. どちらも勝利しなかった場合、結果が引き分けになったというメッセージを表示する。

このチュートリアルを通じて、要素、コンポーネント、props、state といった React の概念に触れてきました。これらのトピックについての更に掘り下げた説明は、ドキュメントの続きをご覧ください。コンポーネントの作成方法についてより詳細に学ぶには、`React.Component` API リファレンスを参照してください。