Audit Access Expires Apr 20, 2020
You lose all access to this course, including your progress, on Apr 20, 2020.

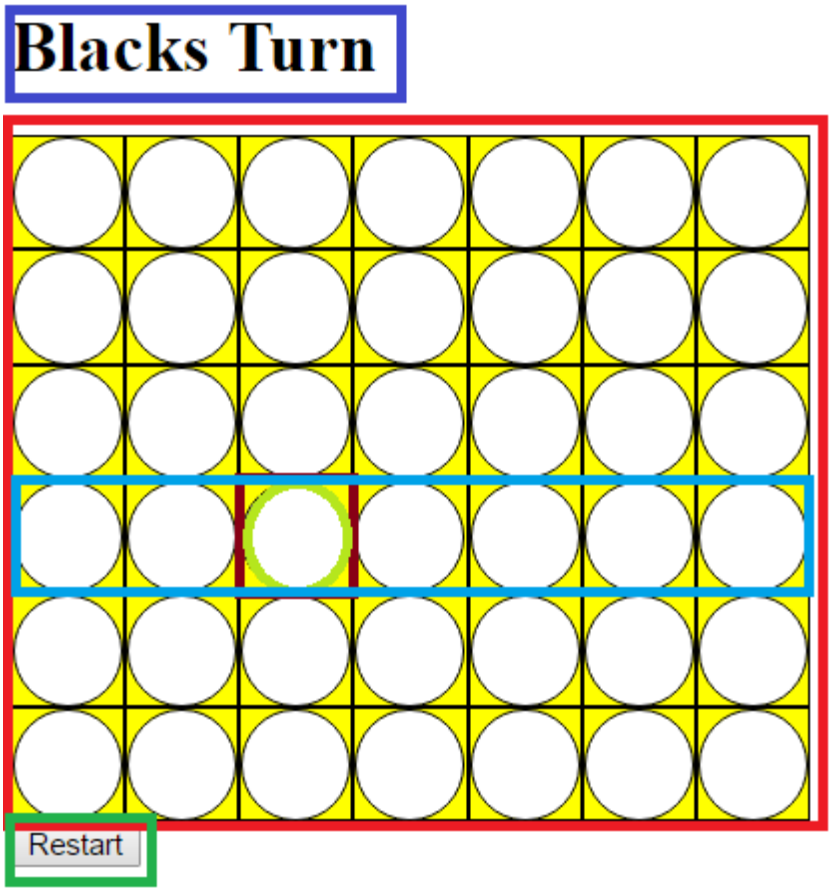
Module 2 Tutorial

Module 2 Tutorial

This tutorial will teach you how to create a Connect 4 game.

Demo of Game/Solution

Step 1: Breaking up the application into components



As the above image shows, we have broken up the application into several sections:

- A component that represents a circle (lime green)
- A component that represents a grid cell (dark red)
- A component that represents a row of cells (teal)
- A component that represents the game board (red)
- A section that displays the game messages (blue)
- A section with a restart button (green)

Step 2: Creating the individual components

To start off lets create a component that represents the circle that will be put inside each grid cell. We can accomplish this by returning a `<div>` tag with some styling to make it into a circle.

```
function Circle(props){
    var style = {
        backgroundColor:"white",
        border: "1px solid black",
        borderRadius: "100%",
        paddingTop: "98%"
    }
    return (
       <div style = {style}></div>
    )
}
```

We can test the component by rendering it to the page, be sure to add a root div in the HTML page:

```
<div id="root"></div>
```

```
ReactDOM.render(
    <Circle/>,
    document.getElementById('root')
)
```

Next, we can create a component that represents the grid cells that make up the game board. We can add some styling to make it into a square and we can embed the Circle component inside of it.

```
function Cell(props){
    var style = {
        height:50,
        width:50,
        border:"1px solid black",
        backgroundColor:"yellow"
    }

    return (
        <div style = {style}>
            <Circle/>
        </div>
    )
}
```

We can test the component by rendering it to the page:

```
ReactDOM.render(
    <Cell/>,
    document.getElementById('root')
)
```

Next, we can create a component that represents a row of grid cells. We can use a for loop to push 7 cells into an array that we will insert in a `<div>` tag. We can add some styling to make the grid cells align horizontally.

```
function Row(props){
    var style = {
          display: "flex"
    }
    var cells = []
    for(let i = 0; i < 7; i++){
        cells.push(<Cell/>)
    }
    return (
        <div style = {style}>
            {cells}
        </div>
    )
}
```

We can test the component by rendering it to the page:

```
ReactDOM.render(
    <Row/>,
    document.getElementById('root')
)
```

Next, we can create a component that represents the game board. We can use a for loop to push 6 rows into an array that we will insert in a `<div>` tag.

```
function Board(props){
    var rows = []
    for(let i = 5; i >= 0; i--){
        rows.push(<Row/>)
    }
    return (
        <div>
            {rows}
        </div>
    )
}
```

We can test the component by rendering it to the page:

```
ReactDOM.render(
    <Board/>,
    document.getElementById('root')
)
```

Lastly, we will create a Game component that encompasses all of the other components. In addition to the Board component, we will add a header for the game messages and a restart button to this component. The header and restart button are simple enough that we do not need to make them into their own components. The Game component will hold all of the application's state so we must make it a class component.

```
class Game extends React.Component{
    constructor(props){
        super(props)
    }
    render(){
        return (
            <div>
                <h1>Blacks Turn</h1>
                <Board/>
                <button>Restart</button>
            </div>
        )
    }
}
```

We can test the component by rendering it to the page:

```
ReactDOM.render(
    <Game/>,
    document.getElementById('root')
)
```

Step 3: Adding game state

The next step is to add state to the Game component. The state should keep track of everything the game needs to know to function.

The state should keep track of the following:

- Which player's turn it is (true for black, false for red)
- Which grid cell's have pieces and what color those pieces should be (0 for empty, 1 for black, 2 for red)
- Which player has won (0 for no one, 1 for black, 2 for red)

The empty grid can be represented by a 2-D array. The 2-D array consists of an array that has 6 indexes that each have 7 indexes.

Set the initial state in the constructor.

```
var cells = []
for(let i = 0; i < 6; i++ ){
    cells.push(new Array(7).fill(0))
}

this.state = {player:false,cells:cells,winner:0}
```

Step 4: Passing State down

The next step is to pass down the cell states all the way down to the individual grid cells. We also need to pass down a click event handler down to the grid cells. Once this is accomplished, we will implement the basic functionality to change a cell circle's color when it is clicked.

First, add a handleClick() method to the Game component:

```
    handleClick(){
        console.log("clicked")
    }
```

Also be sure to bind handleClick to the Game component in the constructor:

```
    this.handleClick = this.handleClick.bind(this)
```

Next, pass in the handleClick method and the this.state.cells attribute into the Board component. Also, pass in the click event handler.

```
    <Board cells = {this.state.cells} handleClick =
{this.handleClick}/>
```

Next, pass in the arrays of length 7 from the this.state.cells 2-D array to the Row components. Also pass in the click event handler.

```
    rows.push(<Row key = {i} row = {i} cells = {props.cells[i]}
handleClick = {props.handleClick}/>)
```

Next, pass in the individual cell states to the Cell components. Also pass in the click event handler.

```
    cells.push(<Cell key = {i} cell = {props.cells[i]} row =
{props.row} col = {i} handleClick = {props.handleClick}/>)
```

Next, pass in the event handler to the onClick attribute of the tag in the Cell component. Also pass in the cell state to the Circle component.

```
    <div style = {style} onClick = {() =>
props.handleClick(props.row,props.col)}>
        <Circle cell = {props.cell}/>
    </div>
```

Next, update the Circle component to change its color based on the cell state. If the cell state is 1 or 2 the circle will change to be black or red respectively.

```
function Circle(props){
    var color = "white"
    if(props.cell == 1){
        color = "black"
    }
    else if(props.cell == 2){
        color = "red"
    }
    var style = {
        backgroundColor:color,
        border: "1px solid black",
        borderRadius: "100%",
        paddingTop: "98%"
    }
    return (
       <div style = {style}></div>
    )
}
```

If you open up the console and click a cell on the grid, you can see that "clicked" will be output to the console log.

Next, modify the handleClick method a bit to get a more detailed idea of which cell is being clicked.

```
handleClick(row,col){
    console.log("row: " + row + " | col: " + col)
}
```

Now when a cell is clicked, the console log will display the row and column of the cell that is clicked.

Now lets modify the handleClick method further to update the state of the cells when they are clicked. The slice() method is used to generate a shallow copy of the inner arrays of the 2-D cells array. These shallow copies are then used to build a new 2-D array named temp. The selected row and column of the new 2-D array is modified and then the cells state is updated to equal the temp 2-D array.

```
handleClick(row,col){
    console.log("row: " + row + " | col: " + col)
    console.log(this.state.cells)
    var temp = [];
    for(let i = 0; i < 6; i++){
      temp.push(this.state.cells[i].slice())
    }
    temp[row][col] = 1;
    this.setState({cells:temp})
}
```

If you click on a grid cell now, the corresponding grid circle will now turn black. When a grid cell is clicked, it calls the handleClick() method which updates the cells state attribute. The state is then passed down until it reaches the Circle component and the circle updates its color to reflect the cell state.

Step 5: Adding functionality to alternate player

The next step is to switch the player everytime a new piece is dropped. The pieces should also alternate colors based on the player that dropped them.

First, edit the handleClick method to alternate the player state everytime a piece is dropped.

```
this.setState({cells:temp, player: !this.state.player})
```

Also, set the value of the temp state cell to equal 1 if *this.player.state* is true and 2 if it is false.

```
temp[row][col] = this.state.player? 1 : 2
```

Next, edit the <h1> tag in the Game component to output whose turn it is based on the player state attribute.

```
<h1>{this.state.player? "Blacks Turn" : "Red Turn"}</h1>
```

Now if you click on a grid cell, the player state will alternate.

The message at the top will display which player's turn it is and the pieces dropped will have their color set based on the player that dropped them.

Step 6: Adding functionality to force pieces to drop all the way down

The next step is to force pieces to drop all the way down until they are on top of another piece or on the bottom row.

To help us accomplish this, we will create a function called findAvailable(col) that will let us know which row a piece should be placed in when it is dropped in a specific column.

The findAvailableRow(col) method loops through the cells state attribute and checks all of the cells in a specified column. It starts at the bottom of the column and checks each grid cell to see if a piece has been placed there. If a grid cell is empty, the method will return the row of that cell. Otherwise, it will return -1.

```
findAvailableRow(col){
  for(var i = 0; i < 6; i++){
    if(this.state.cells[i][col] == 0){
      return i;
    }
  }
  return -1;
}
```

Next, we will modify the handleClick method to update the temp state cell 2-D array based on the row returned by findAvailableRow.

```
var newRow = this.findAvailableRow(col)
temp[newRow][col] = this.state.player? 1 : 2
```

Now if you test the application, the pieces should drop all the way down.

Step 7: Adding victory detection

The next step is to add a way to detect if a player has won.

Add in these methods that help determine whether there are 4 pieces in a row:

```
    checkDiagonal(row,col){
        //find right and left tops
        var c = this.state.cells;
        var val = this.state.player? 2:1;
        var rR = row;
        var cR = col;
        while(rR < 5 && cR < 6){
            rR++;
            cR++;
        }

        while( rR >= 3 && cR >= 3){
            if(c[rR][cR] == val && c[rR-1][cR-1] == val && c[rR-2]
[cR-2] == val && c[rR-3][cR-3] == val){
                return 1
            }
            rR--
            cR--
        }

        var rL = row;
        var cL = col;

        while(rL < 5 && cL > 0){
            rL++
            cL--
        }

        while(rL >= 3 && cL <= 3){
            if(c[rL][cL] == val && c[rL-1][cL+1] == val && c[rL-2]
[cL+2] == val && c[rL-3][cL+3] == val){
                return 1
            }
            rL--
            cL++
        }
        return 0
    }
    checkHorizontal(row,col){
        var c = this.state.cells;
        var i = 6;
```

```
        var val = this.state.player? 2:1;

        while( i >= 3){
            if(c[row][i] == val && c[row][i-1] == val && c[row][i-2]
== val && c[row][i-3] == val){
                return 1
            }
            i--
        }
        return 0
    }
    checkVertical(row,col){
        var c = this.state.cells;
        var i = row;
        var val = this.state.player? 2: 1;

        if(i >= 3){
            if(c[i][col] == val && c[i - 1][col] == val && c[i - 2]
[col] == val && c[i - 3][col] == val){
                return 1
        }
        }
        return 0

    }
    checkVictory(row,col){
        return this.checkVertical(row,col)   ||
this.checkHorizontal(row,col) ||   this.checkDiagonal(row,col)



    }
```

Next, add in a callback to the setState method call.

```
        this.setState({cells:temp, player: !this.state.player}, () =>
{
            if(this.checkVictory(newRow,col) > 0){
                console.log("win")
                this.setState({winner:this.state.player?2:1})
            }


        })
```

Also, add this to the top of the handleClick method to stop the game once someone has won.

```
        if(this.state.winner)
            return
```

Lastly, edit the<h1> tag to display the winner if the game has ended.

```
    <h1>{this.state.winner > 0 ?  this.state.winner == 1? "Black
Wins":"Red Wins": this.state.player? "Blacks Turn" : "Reds Turn"}
</h1>
```

The game should now display the winner of the game and stop the game if someone wins.

Step 8: Adding reset functionality

The last step is to make the Restart button restart the game.

First, create a method called restart() within the Game component. This method will reset the cells, winner and player state attributes.

```
    restart(){
        var cells = [];
        for(let i = 0; i < 6; i++ ){
          cells.push(new Array(7).fill(0));
        }
        this.setState({ player : false, cells : cells, winner:0})
    }
```

Next, edit the button element's onClick attribute to call the restart() method.

```
<button onClick = { () => this.restart()}>Restart</button>
```

Pressing the restart button should now restart the game.