Second Personal Project | August 2024

Calculator & Graph

김유진

Go to Site

CONTENTS

01 자료구조 02 – 05p

02 화면설계 06 – 18p

03 코드 19 – 27p

04 Electron 28 – 30p

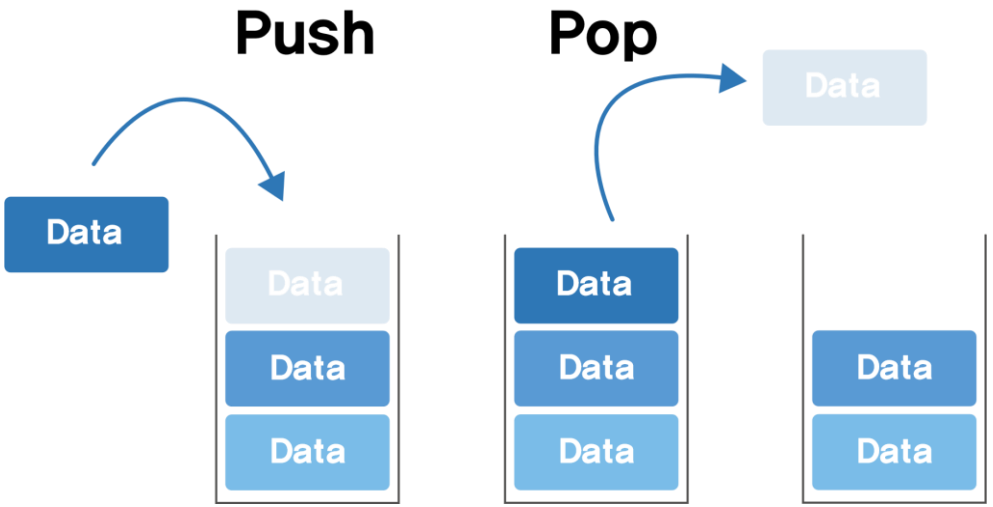
05 후기 31 – 35p

PART 1. 자료구조

Go to Site

Stack

스택 (Stack)

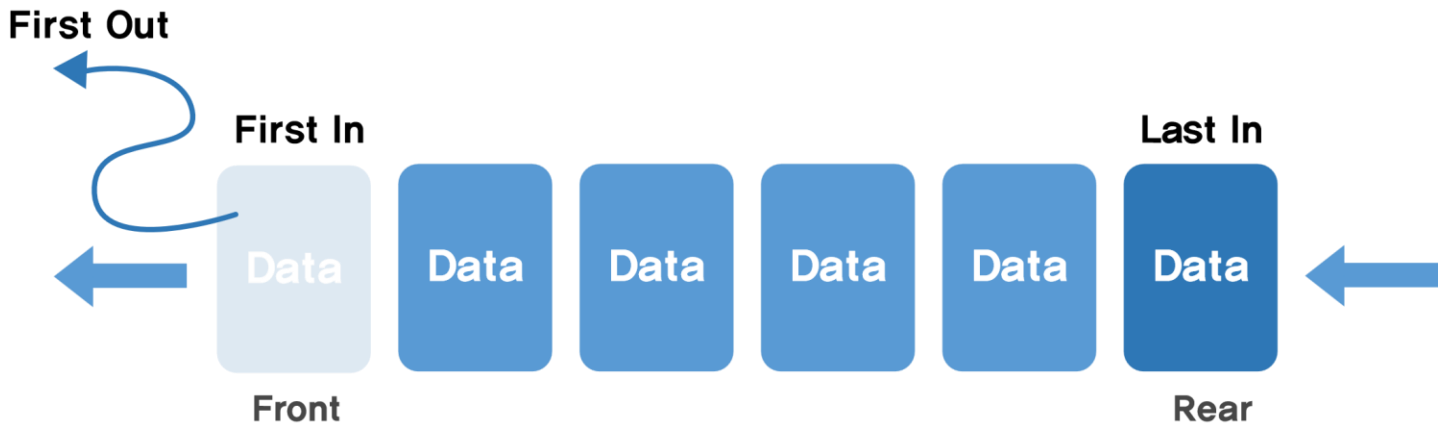


- 가장 나중에 추가된 데이터가 가장 먼저 제거되는 구조
- 후입선출(LIFO, Last In, First Out) 원칙
- 푸시(Push): 데이터를 스택의 최상단에 추가합니다.
- 팝(Pop): 스택의 최상단에 있는 데이터를 제거하고 반환합니다.

Go to Site

Queue

큐 (Queue)

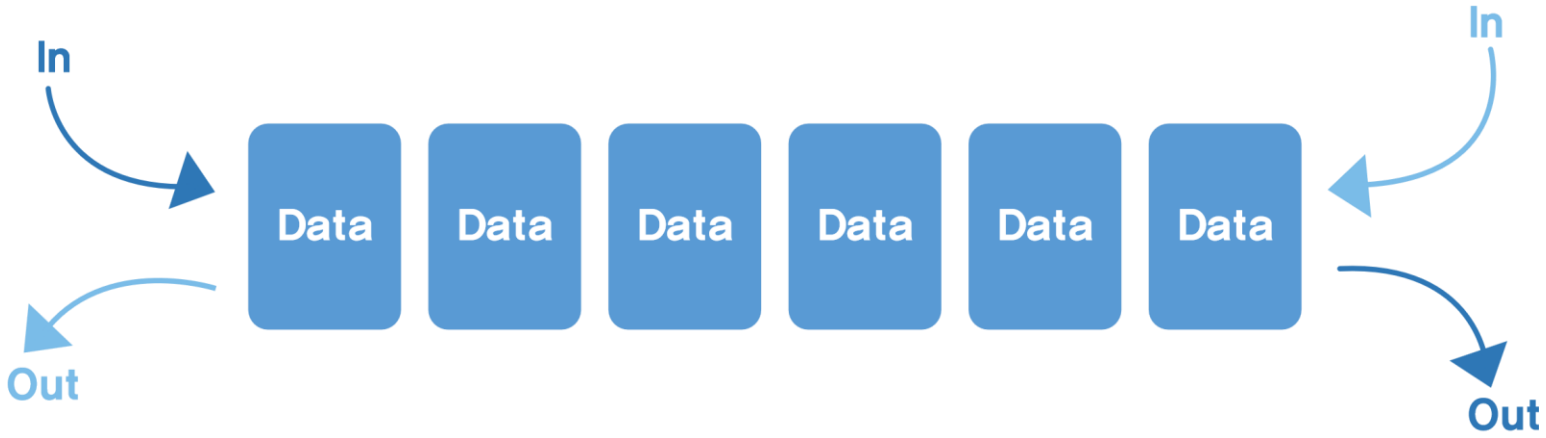


- 가장 먼저 추가된 데이터가 가장 먼저 제거되는 구조
- 선입선출(FIFO, First In, First Out) 원칙
- 푸시(Push): 데이터를 큐의 끝에 추가합니다.
- 시프트(shift): 큐의 앞에서 데이터를 제거하고 반환합니다.

Go to Site

Deque (Double-Ended Queue)

덱 (Deque)

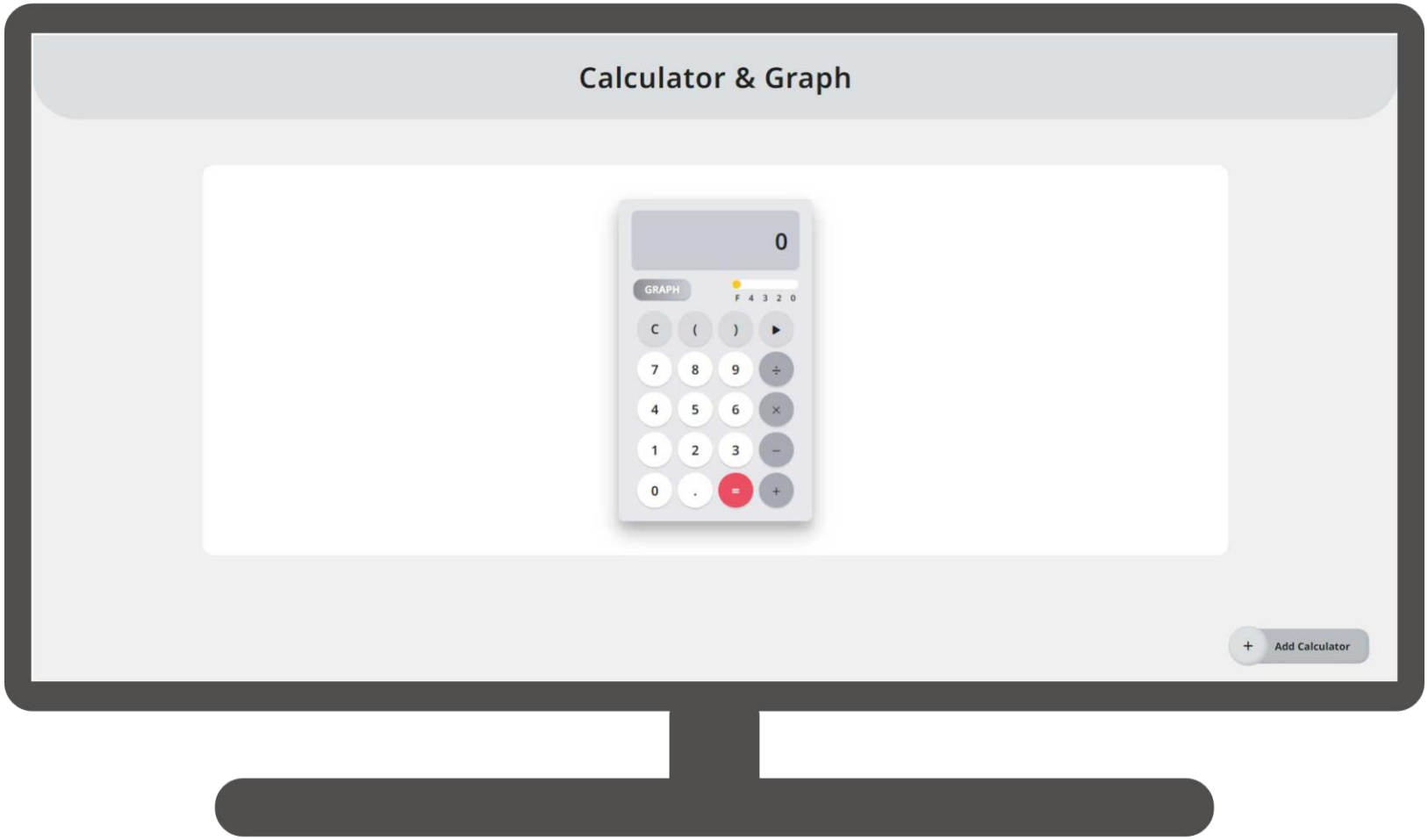


- 양방향에서 삽입과 제거가 가능한 자료구조
- 데이터를 앞쪽에 삽입했으면 뒤쪽에서 제거할 수 있고, 뒤쪽에 삽입했으면 앞쪽에서 제거할 수 있습니다.
- Push: 데이터를 덱의 뒤쪽에 추가합니다.
  Shift: 덱의 앞쪽에서 데이터를 제거하고 반환합니다.
- Unshift: 데이터를 덱의 앞쪽에 추가합니다.
  Pop: 덱의 뒤쪽에서 데이터를 제거하고 반환합니다.

PART 2. 화면설계

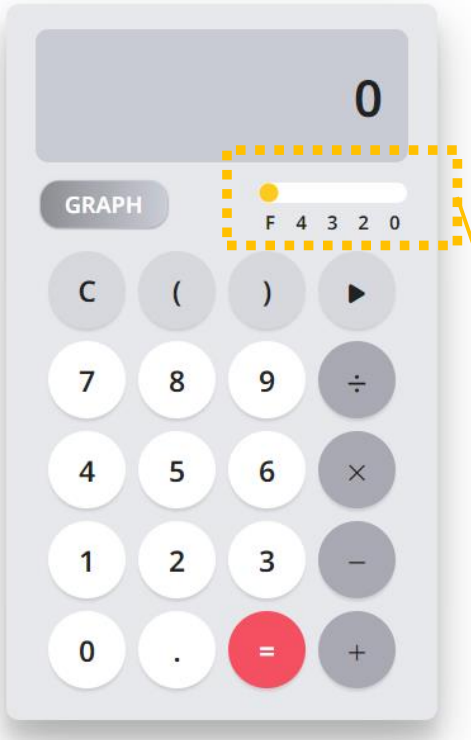
화면구성

Go to Site



□ 계산기 소수점 표시 (Description)

Go to Site



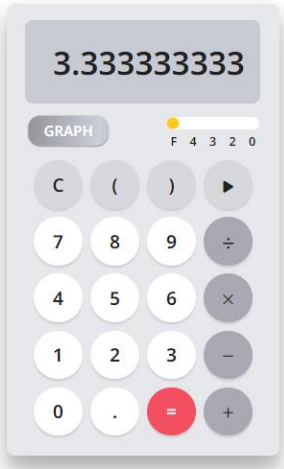
소수점 표시 방법

1. F (기본값): 정수는 소수점 없이 출력되며, 소수는 전체 자릿수만큼 표시됩니다.
2. 4: 정수와 소수 모두 소수점 아래 4자리까지 표시합니다.
3. 3: 정수와 소수 모두 소수점 아래 3자리까지 표시합니다.
4. 2: 정수와 소수 모두 소수점 아래 2자리까지 표시합니다.
5. 0: 정수와 소수 모두 소수점 없이 정수 부분만 출력합니다.

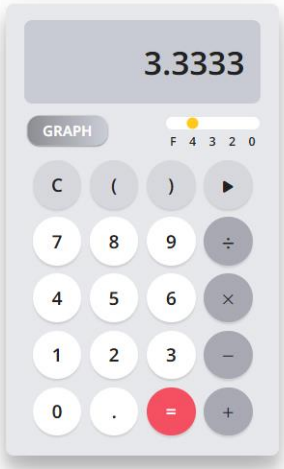
계산기 소수점 표시 (View)

Go to Site

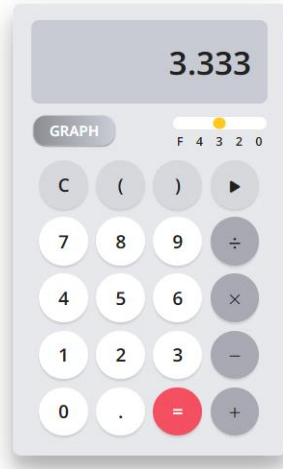
F 설정



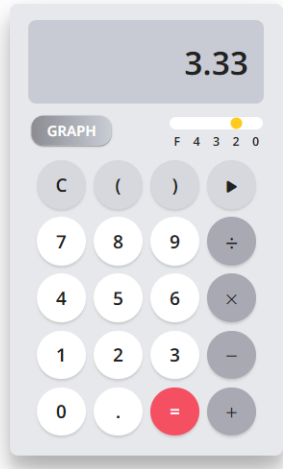
자릿수 4 설정



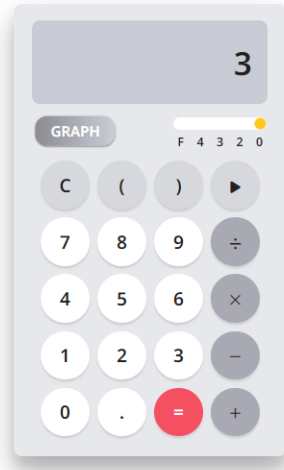
자릿수 3 설정



자릿수 2 설정



자릿수 0 설정



결과 표시 및 가로 스크롤 발생

Go to Site

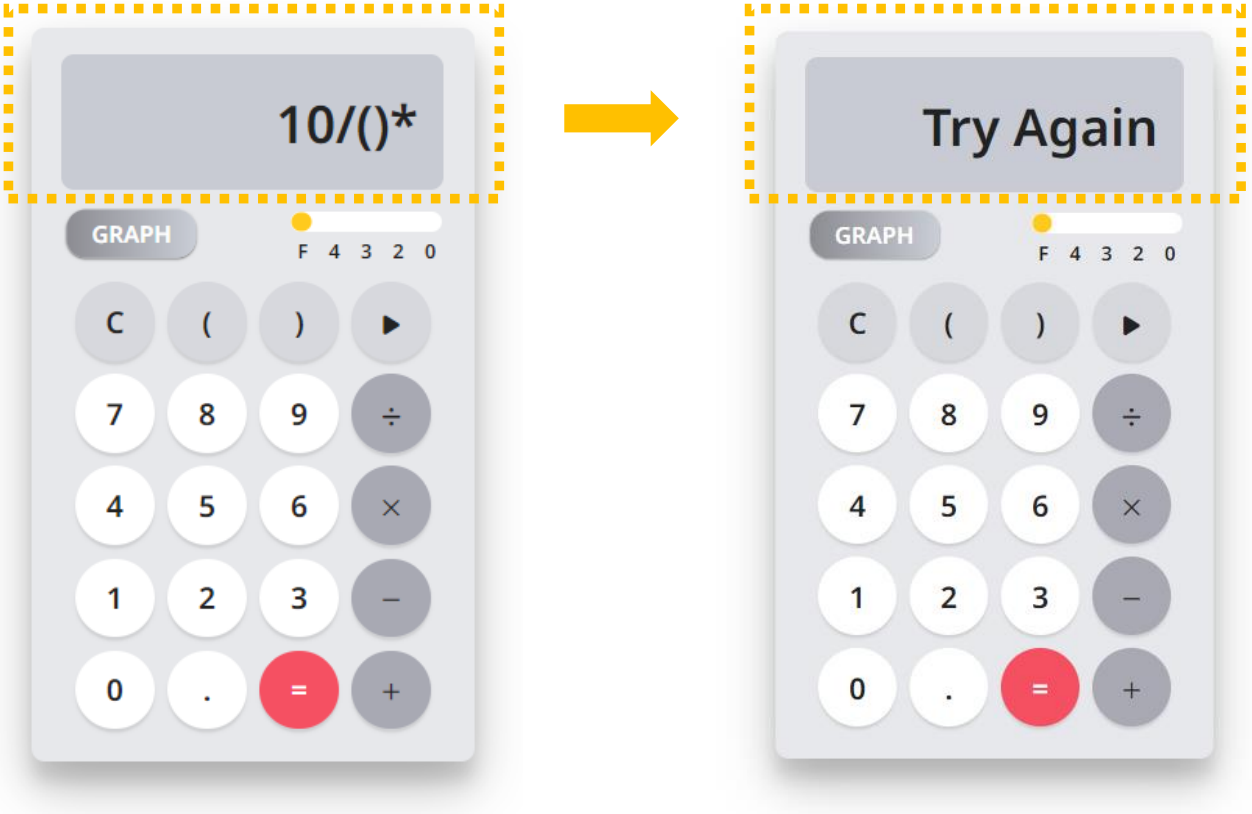
가로 스크롤 발생 조건

1. 입력할 때는 스크롤이 나타나지 않습니다.
2. 계산 결과가 화면 길이를 초과하면 가로 스크롤이 발생합니다.
3. 입력을 수정하면 스크롤이 사라집니다.

Go to Site

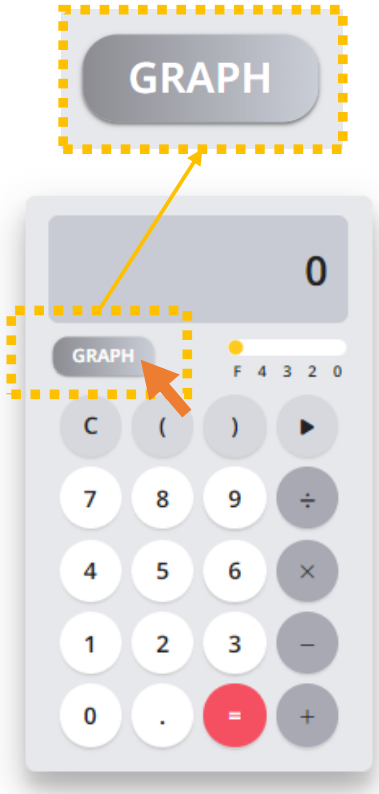
◘ 잘못된 계산식 경고

계산식이 올바르지 않을 경우 결과출력시 “Try Again” 문구가 표시됩니다.

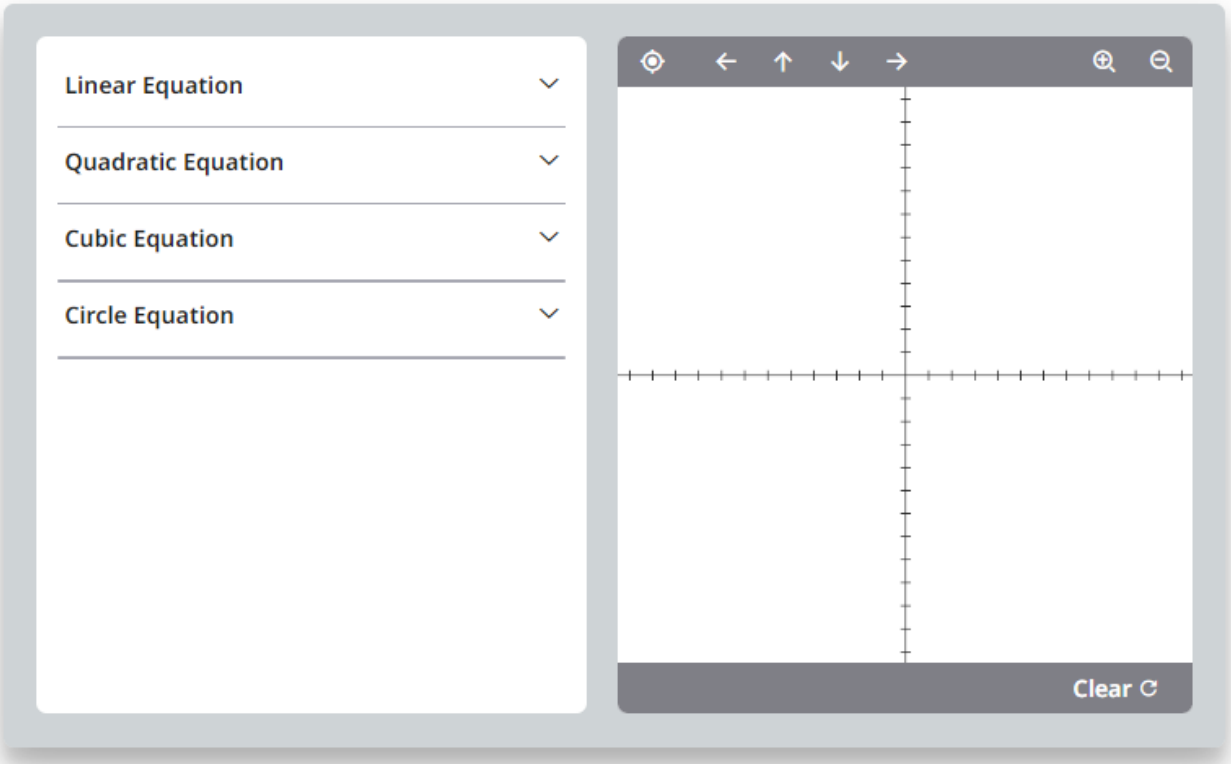


Go to Site

그래핑 계산기 화면 안내



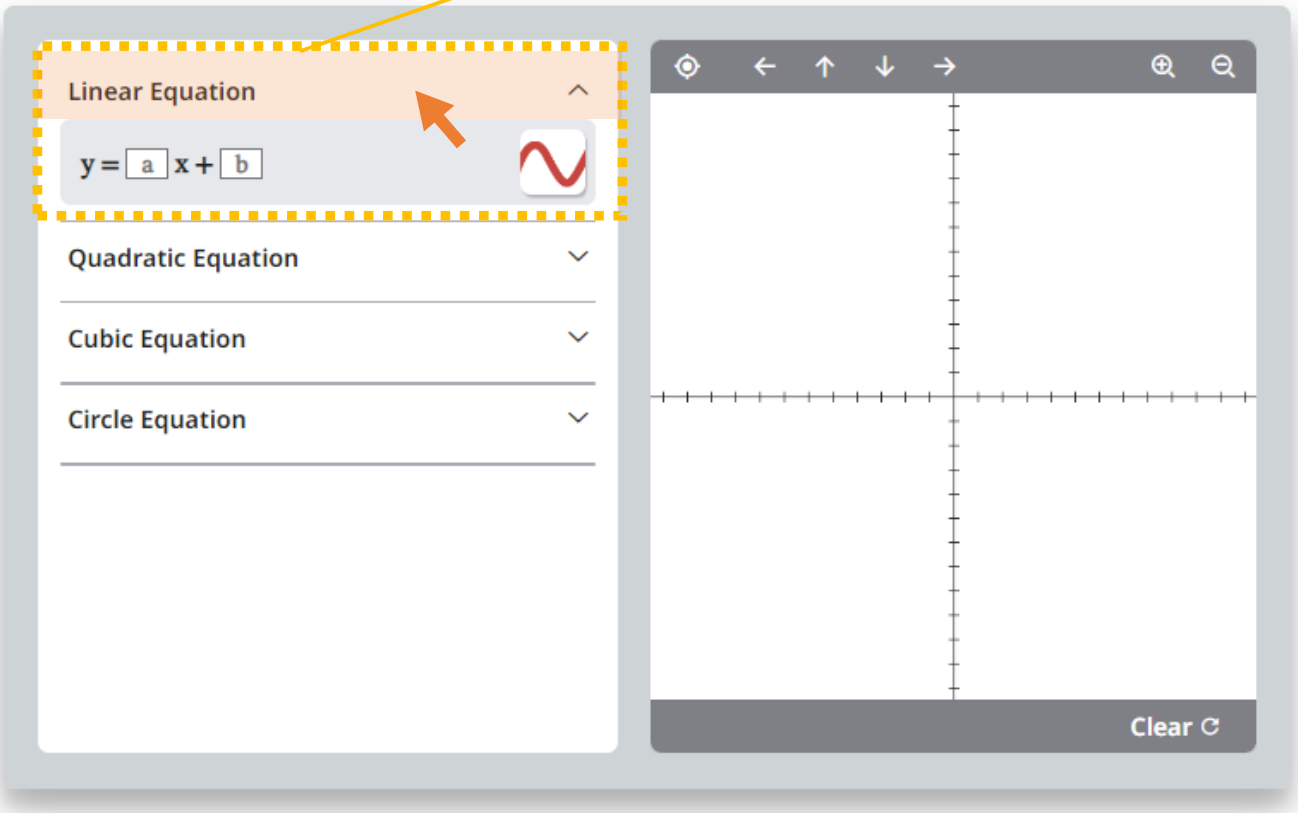
계산기의 "Graph" 버튼을 클릭하면 그래핑계산기 화면이 표시됩니다.



Go to Site

제목 클릭으로 입력란 표시/숨기기

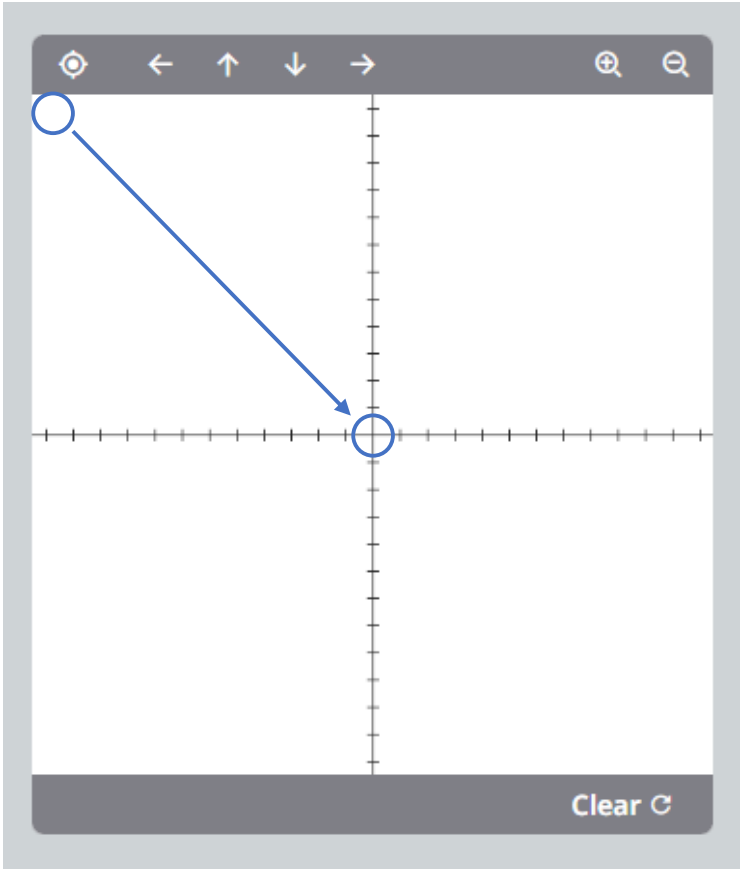
제목 영역을 클릭하면 함수 계산식 입력란이 펼쳐집니다.



Canvas를 활용한 그래프 영역 설계

Go to Site

좌표원점 재배치

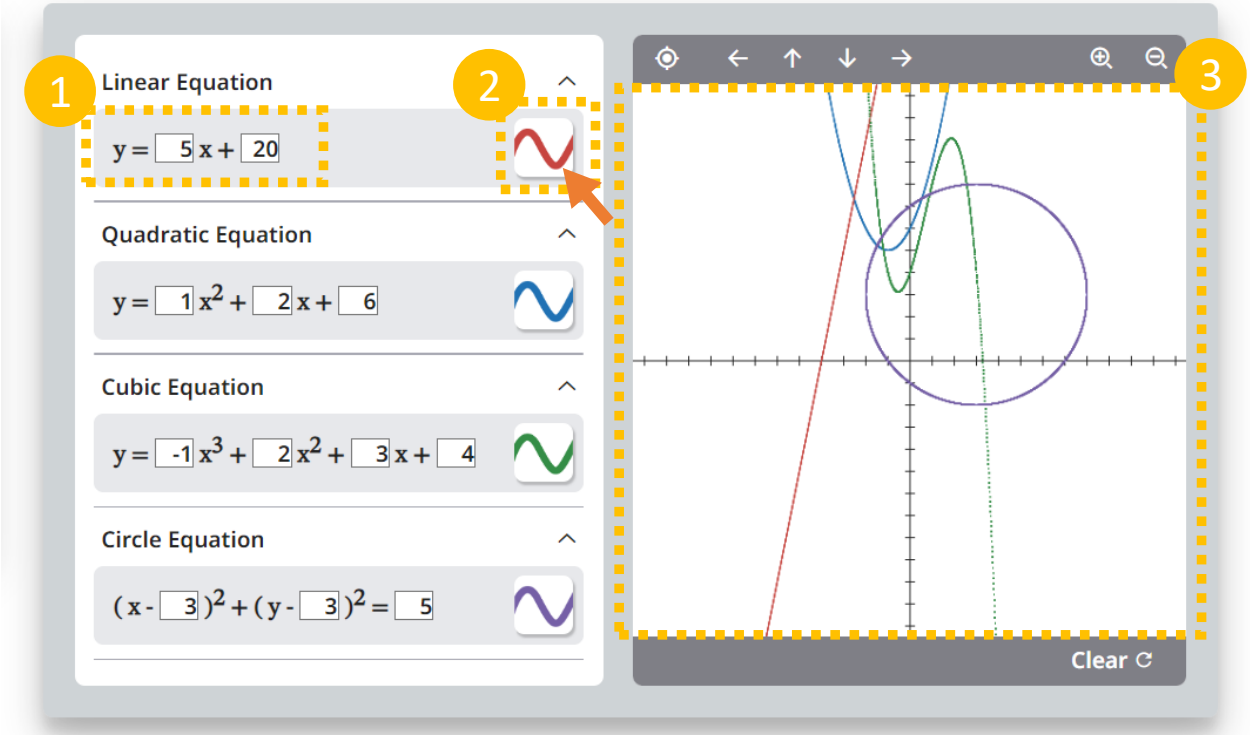


1. 그래프를 표시하는 영역은 Canvas를 사용하여 구현되었습니다.
2. 원점을 Canvas영역의 중앙으로 조정하여, 카데지안 좌표계를 데카르트 좌표계로 변환하였습니다.

그래프 표시 과정

Go to Site

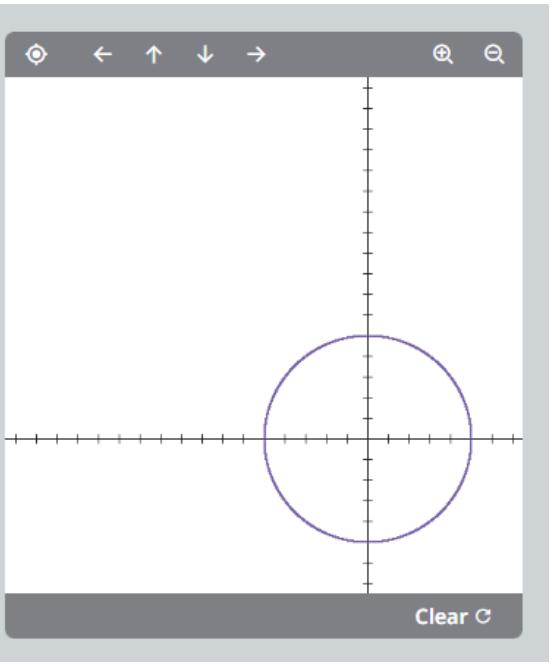
1. Input box에 계산식에 사용할 숫자를 입력
2. 계산식 옆의 버튼 클릭
3. Canvas에 해당 버튼 색상과 일치하는 함수 그래프 표시



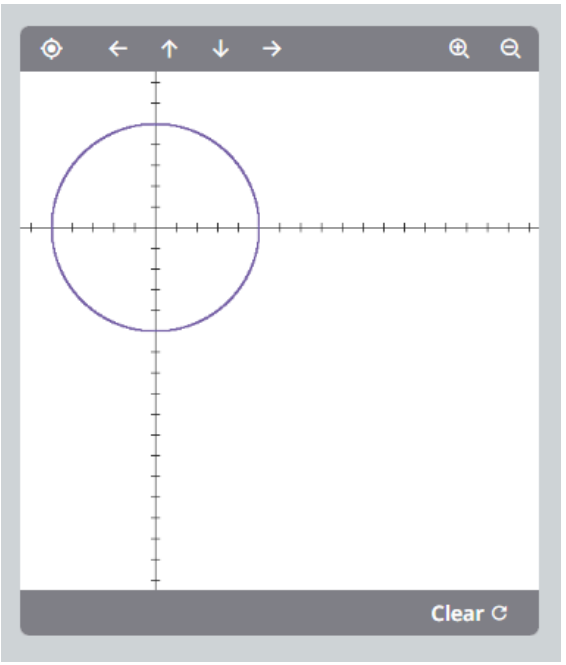
Go to Site

좌표축 이동 및 원점 재배치

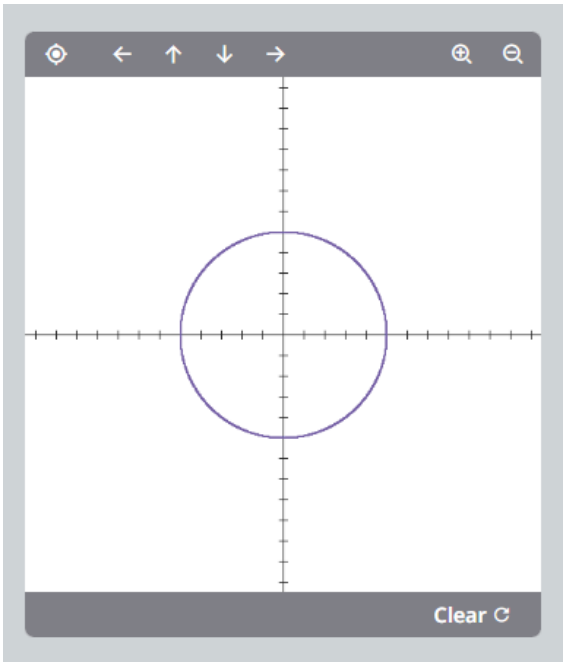
X축을 오른쪽으로 이동합니다.
Y축을 아래쪽으로 이동합니다.



X축을 왼쪽으로 이동합니다.
Y축을 위쪽으로 이동합니다.



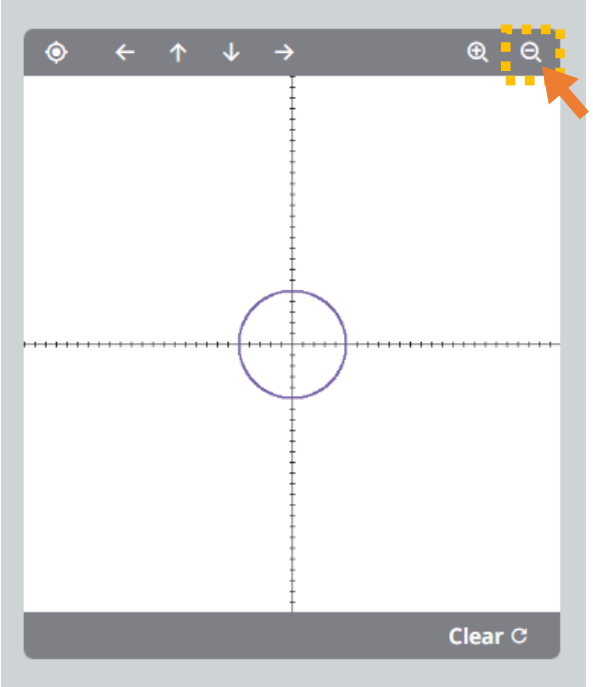
원점으로 재배치합니다.



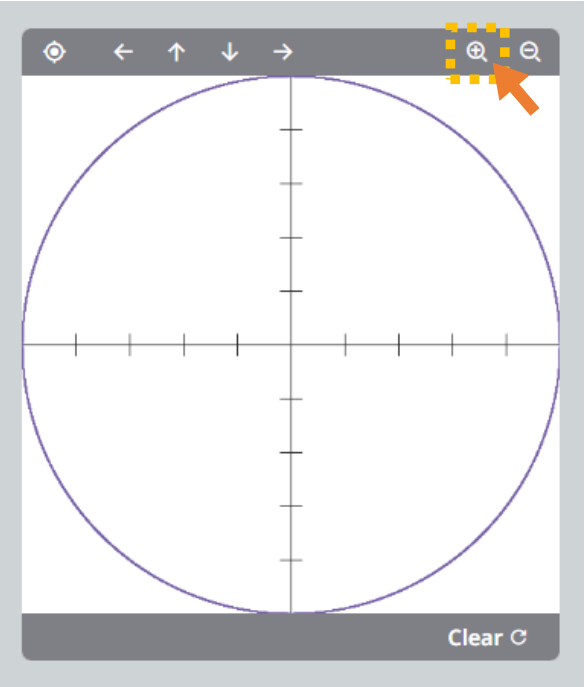
Go to Site

좌표축 확대 및 축소

그래프의 좌표 축을 축소합니다.



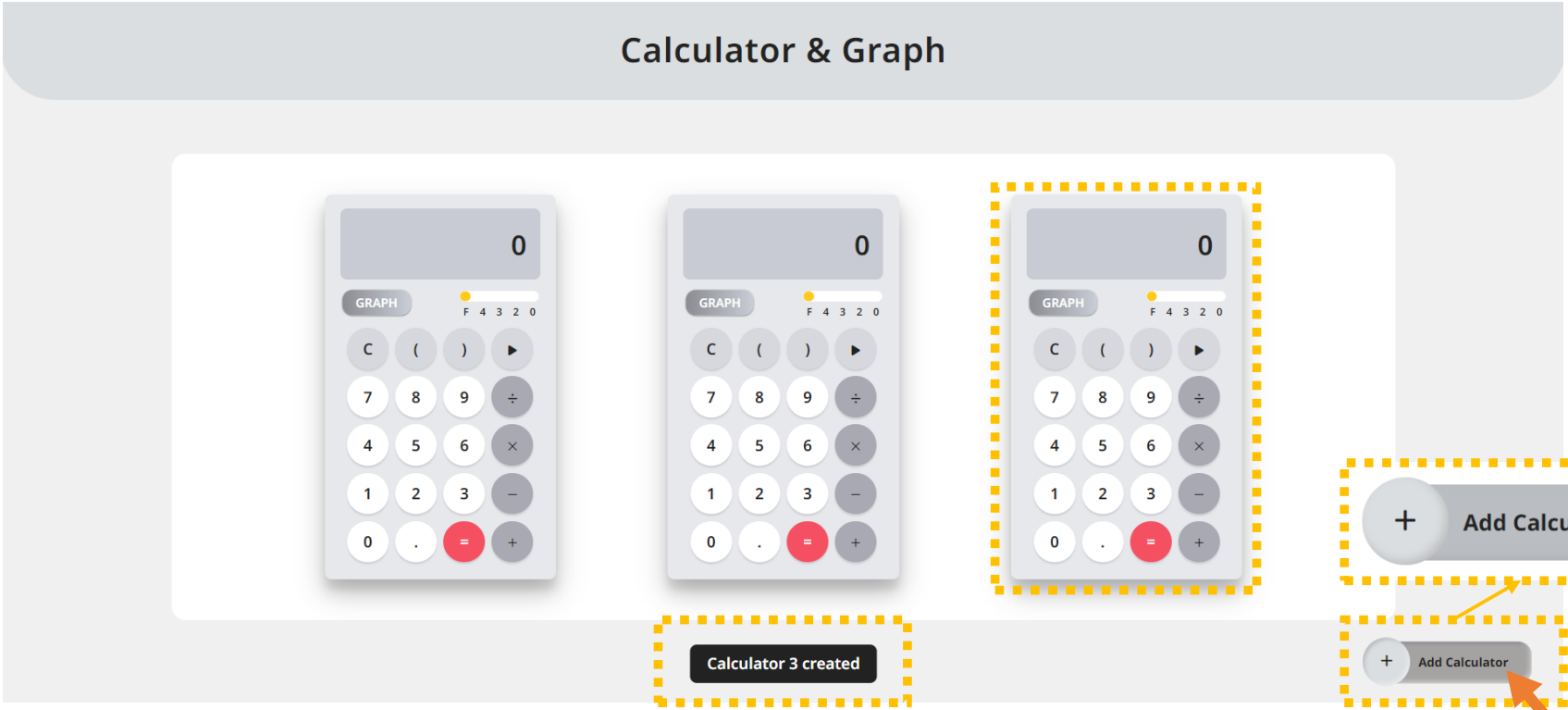
그래프의 좌표 축을 확대합니다.



Go to Site

새로운 계산기 생성 기능

1. "Add Calculator" 버튼을 누르면 새로운 계산기가 생성됩니다.
2. 생성된 계산기의 순번이 안내 문구로 표시됩니다.



PART 3. 코드

웹 컴포넌트 개념을 활용한 클래스 구조화

Go to Site

Javascript

```
// ^Calculator
class Calculator {
  constructor(id) {
    this.id = id;
  }
  initCalc(targetDom) {…
  showData(outputDom, resultContext) {…
  receiveData(outputDom) {…
  graphCalc(targetDom) {…
  graphCalcDisplay() {…
  control(targetDom) {…
}//$Calculator
```

- 웹 컴포넌트 개념을 활용하여 클래스를 구현하였습니다.
- 클래스를 통해 모듈화된 구성 요소를 효과적으로 재사용하고 유지보수할 수 있어서 반복 작업을 줄일 수 있었습니다.

자바스크립트 클래스에서 HTML & CSS 통합 관리

Go to Site

Javascript

```
initCalc(targetDom) {
  const HTMLcomponent = `···
  const CSScomponent = `···
  targetDom.innerHTML += `
  ${HTMLcomponent}
  <style>${CSScomponent}</style>
  `;
}//$initCalc
```

- HTML과 CSS를 자바스크립트 클래스 안에 통합하여 컴포넌트화했습니다.
- 이를 통해 각 컴포넌트는 동일한 구조를 유지하면서도 고유한 기능을 구현할 수 있었습니다.

MVC 모델 적용 데이터 처리, 메서드 내 중첩 클래스 사용

Go to Site

Javascript

```
receiveData(outputDom) {
  class Deck {
    constructor(id) {
      this.id = id;
      this.storage = [];
    }
    pushItem(item) { this.storage.push(item); }
    popItem() { return this.storage.pop(); }
    unshiftItem(item) { this.storage.unshift(item); }
    shiftItem() { return this.storage.shift(); }
    countDeck() { return this.storage.length; }
    refreshDeck() { this.storage = []; }
  } //end_Deck
```

Javascript

```
// ^showData
showData(outputDom, resultContext) {···
// ^receiveData
receiveData(outputDom) {···
```

데이터 수신 영역과
처리 영역을 분리하였습니다.

- 데이터가 전역적으로 수정될 수 있는지 여부를 데이터의 중요성과 의미에 따라 구분하여 MVC 모델을 적용했습니다.
- 클래스 메서드 내에 중첩 클래스를 선언하여 데이터를 처리했습니다.

덱(Deque) 활용 데이터 처리

Go to Site

Javascript

```
document.getElementById(`mainBox${this.id}`).addEventListener("click", (e) => {
  if (e.target.dataset.value != null) {
    switch (e.target.dataset.value) {
      case ">":
        calcDeck.popItem();
        valueUpdate();
        this.showData(outputDom, resultContext);
        break;
      case "=":
        calcDeck.refreshDeck();
        resultContext.isResultChecked = true;
        this.showData(outputDom, resultContext);
        break;
      case "C":
        calcDeck.refreshDeck();
        valueUpdate();
        this.showData(outputDom, resultContext);
        break;
      default:
        calcDeck.pushItem(e.target.dataset.value);
        valueUpdate();
        this.showData(outputDom, resultContext);
    }
  }
});
```

데이터 수신 클래스에서는 전역 속성에 직접 접근하지 않고, 메서드 내에서 덱(Deque) 자료구조를 활용하여 데이터를 처리하고 가공했습니다.

객체를 통한 이벤트 요소 정의

Go to Site

Javascript

```
const inputEventDetails = {
  shapeIds: ['linear', 'quadratic', 'cubic', 'circle'],
  controlsIds: [],
  graphRenderIds: [],
  value: [
    [0, 0],
    [0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0]
  ],
  graphData: []
};
```

해당 객체는 이벤트가 발생하는 요소들을 정의하고, 이들을 매개변수로 넘겨 그래프 렌더링 및 데이터 관리를 수행합니다.

이벤트에 따른 실시간 업데이트 및 데이터 저장

Go to Site

Javascript

```
/// [Event::click] graphRenderIds ///
for (let i = 0; i < inputEventDetails.graphRenderIds.length; i++) {
  const eventId = document.getElementById(inputEventDetails.graphRenderIds[i]);
  eventId.addEventListener("click", () => {
    const eventMode = inputEventDetails.shapeIds[i];
    const eventValue = inputEventDetails.value[i];
    this.renderGraph(eventMode, eventValue); // 그래프그리기 함수 호출
    inputEventDetails.graphData.push({ mode: eventMode, value: eventValue });
  });
}
```

- inputEventDetails.graphRenderIds 배열의 각 ID에 대해 클릭 이벤트 리스너를 추가합니다.
- 사용자가 그래프 요소를 클릭하면, 해당 요소의 모드와 값을 inputEventDetails.shapeIds와 inputEventDetails.value 배열에서 가져옵니다.
- renderGraph 함수를 호출하여 그래프를 업데이트하고, 클릭된 요소의 모드와 값을 inputEventDetails.graphData 배열에 할당하여 관련 이벤트 발생시 참조됩니다.

eventMode에 따른 그래프 렌더링 및 화면 출력

Go to Site

Javascript

```
renderGraph(eventMode, eventValue) {
 const eventValueArr = [...eventValue];
 switch (eventMode) {
  case 'linear':
    this.pen.fillStyle = "#C74540";
    const [linearA, linearB] = eventValueArr;
    for (let x = -(this.cWidth / 2); x <= this.cWidth / 2; x += 0.01) {
      const y = (linearA * x) + linearB;
      this.modiDot(x, y);
    }
    break;
      ◦
      ◦
      ◦
```

- renderGraph(eventMode, eventValue) 함수는 전달받은 매개변수 eventMode와 eventValue를 기반으로 그래프를 화면에 출력합니다.
- eventMode와 일치하는 case부분에서, 해당 함수식의 그래프를 렌더링합니다.
- 계산된 좌표는 modiDot 메서드를 통해 시각적으로 화면에 출력됩니다.

클릭 이벤트로 인해 동적으로 생성되는 계산기 구현

Go to Site

Javascript

```
/// CreateCalc 생성 class ///
class CreateCalc {
  constructor(id) {
    this.id = id;
    this.storage = {};
  }
  receiveData(eventTargetDOM, outputElement, infoMessageDOM) {
    let createCount = 1;
    eventTargetDOM.addEventListener("click", (event) => {
      createCount += 1;
      this.renderView(outputElement, infoMessageDOM, createCount);
    })
  }
  renderView(outputElement, infoMessageDOM, createCount) {···
  getDefaultValue() {···
  control(eventTargetDOM, outputElement, infoMessageDOM) {
    this.receiveData(eventTargetDOM, outputElement, infoMessageDOM);
    this.getDefaultValue();
  }
}//$CreateCalc
```

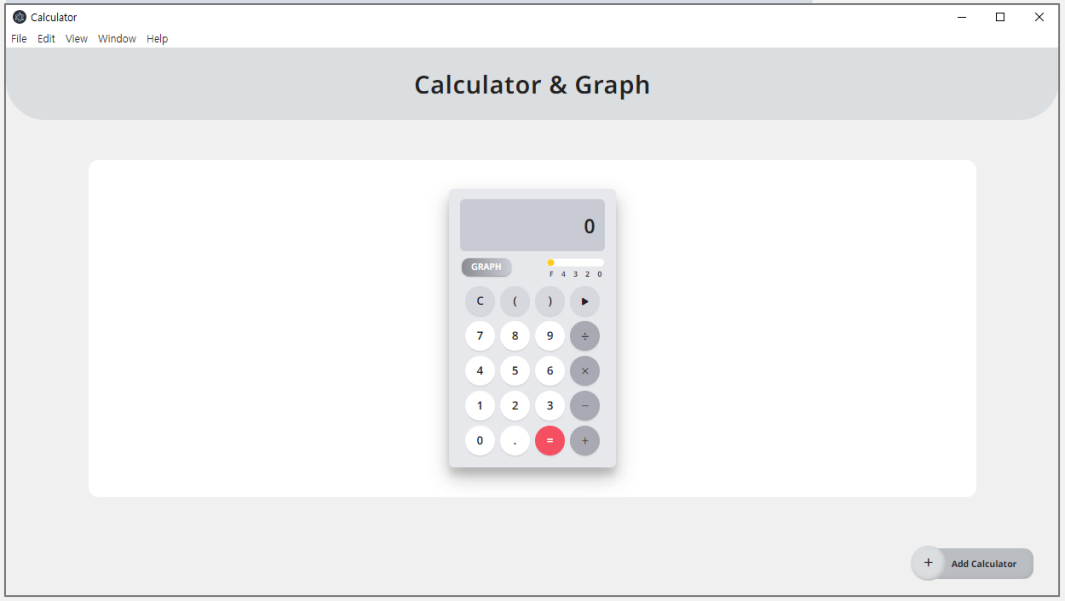
- CreateCalc 클래스는 클릭 시 숫자가 증가하며 새로운 계산기를 생성합니다.
- 생성된 계산기는 화면에 추가되고, 관련 정보를 팝업으로 표시합니다.

PART 4. Electron

Electron을 활용한 서비스 구축 화면

Go to Site

Electron을 사용한 애플리케이션 화면입니다.



앱 생성의 놀라움

간단한 코드로 데스크톱 애플리케이션을 손쉽게 제작할 수 있다는 점이 인상적이었습니다.

확장 가능성 탐색 목표

기본 기능을 개발한 후, Node.js의 다양한 기능을 더욱 깊이 탐구하고 싶어졌습니다. 특히, Electron을 활용해 여러 라이브러리와 연동하여 기능을 확장하는 방법에 큰 관심이 생겼습니다. 앞으로 더 많은 기능을 탐색하고, 이를 통해 실용적인 프로젝트로 발전시키고자 합니다.

Electron 애플리케이션 기본 구조

Go to Site

Node.js에서 실행되는 JavaScript

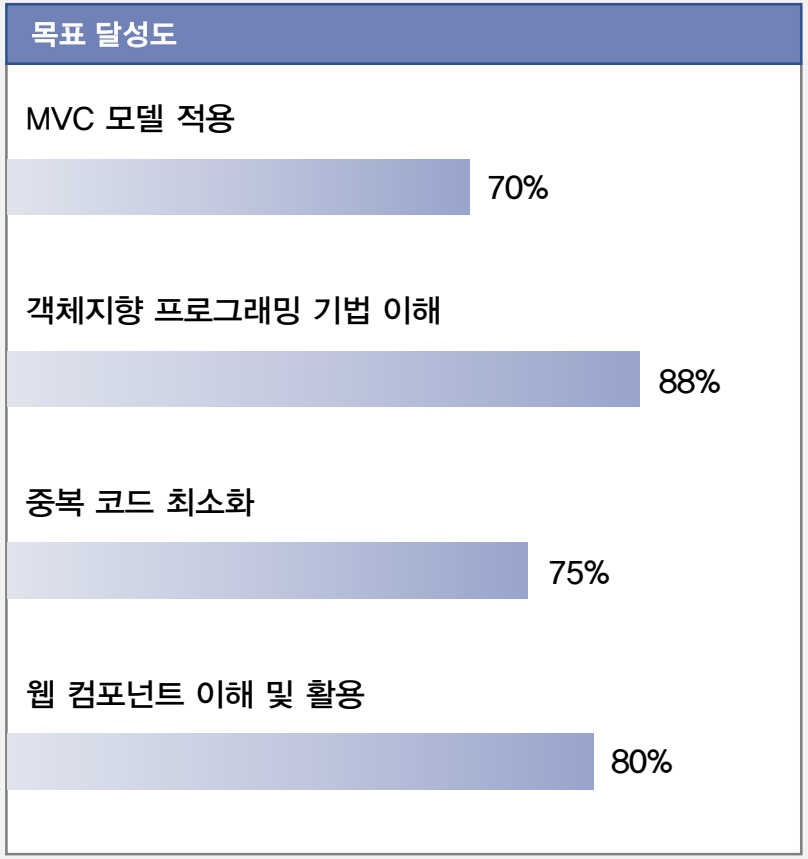
```
const { app, BrowserWindow } = require('electron');
const path = require('path');

const createWindow = () => {
  const win = new BrowserWindow({
    width: 640,
    height: 480,
    webPreferences: { preload: path.join(__dirname, 'preload.js') }
  });
  win.loadFile('index.html');
};

app.whenReady().then(() => {
  createWindow();
  app.on('activate', () => {
    if (BrowserWindow.getAllWindows().length === 0) createWindow();
  });
});

app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') app.quit();
});
```

- 해당 코드를 통해 애플리케이션의 기본 구조를 설정하고 Electron을 활용하여 간단한 데스크탑 앱을 만들었습니다.

PART 5. 후기

Go to Site

프로젝트 목표 달성도



MVC 모델 적용

Model-View-Controller 아키텍처를 도입하여 데이터와 UI를 분리했습니다

객체지향 프로그래밍 기법 이해

클래스와 객체의 개념을 배우고, 자바스크립트로 클래스를 구현해봤습니다. 객체지향 프로그래밍을 적용하면서 코드가 더 깔끔하고 효율적으로 정리되는 것을 느꼈습니다.

중복 코드 최소화

중복 코드를 줄이기 위해 기능별로 모듈화 했습니다. 이를 통해 코드의 재사용성을 높이고 관리하기 쉽게 만들었습니다.

웹 컴포넌트 이해 및 활용

웹 컴포넌트의 기본 개념을 배우고, Shadow DOM과 Custom Elements를 활용해 재사용 가능한 UI 요소를 만들었습니다.

Go to Site

◻ 오류 처리

변경 전

```
outputElement.innerHTML += `
  <div id="calcArea${createCount}" class="calcArea"></div>
`;
```

변경 후

```
const newCalcArea = document.createElement('div');
newCalcArea.id = `calcArea${createCount}`;
newCalcArea.className = 'calcArea';
outputElement.appendChild(newCalcArea);
```

문 제

원래의 코드에서는 innerHTML을 사용하여 새로운 계산기 UI 요소를 추가했습니다.
처음에는 문제없이 작동하는 것처럼 보였습니다. 그러나 실제로는 새롭게 생성된 계산기 하나만 기능이 작동하고, 나머지 계산기들은 전혀 반응하지 않았습니다.

원 인

원인은 이벤트 리스너 소실이었습니다. innerHTML을 사용하여 HTML을 업데이트할 때, 기존의 DOM 요소와 그에 할당된 이벤트가 모두 사라지는 것이 확인되었습니다. 새로운 HTML이 삽입되면서 이전에 설정된 이벤트 리스너들이 제거되었고, 그 결과로 계산기 기능이 작동하지 않는 문제가 발생했습니다.

해 결

문제를 해결하기 위해 createElement와 appendChild를 사용하여 동적으로 요소를 추가하는 방법으로 전환했습니다. 이 방법은 기존의 DOM 요소와 이벤트 리스너를 유지하면서 새로운 요소를 추가할 수 있었습니다.

Go to Site

프로젝트 후기

느낀점

데이터를 수신하는 영역과 처리하는 영역을 명확히 분리하고, 데이터의 성격에 따라 전역으로 설정해야 하는 고유한 값인지, 아니면 서로 전달해야 하는 속성인지 구분하는 것이 어려웠습니다. 이번 프로젝트를 통해 데이터의 의미와 역할을 좀 더 명확히 이해하게 되었고, 단순히 기능을 구현하는 것 이상의 접근이 필요하다는 것을 깨달았습니다. 문법적인 측면을 넘어서서, 데이터가 가진 의미와 그에 맞는 적절한 형식을 고민하는 과정이 중요하다는 것을 배우게 되었습니다.

HTML과 CSS를 DOM에 직접 삽입하지 않고, 자바스크립트 클래스 내에서 웹 컴포넌트화하여 각 컴포넌트가 독립적으로 동작하도록 설계하는 방법을 배웠습니다. 이를 통해 각 컴포넌트가 독립적인 구조와 스타일을 갖추어 다른 컴포넌트와 충돌 없이 기능을 구현할 수 있었습니다.
이 접근 방식 덕분에 코드의 유지 보수와 확장성에 대한 개념이 향상되었고, 모듈화된 코드를 통해 재사용성과 관리가 훨씬 쉬워졌습니다. 이번 경험을 통해 웹 컴포넌트가 가진 장점을 이해할 수 있었습니다.

Go to Site

프로젝트 후기

아쉬웠던 부분

캔버스를 그리는 클래스에서 전역 속성으로 설정된 값들이 기능 구현이 완료된 후에는 매개변수로 전달하는 것이 더 적절했음을 깨달았습니다. 이번 프로젝트를 통해 배운 점을 바탕으로, 다음 프로젝트에서는 데이터의 의미와 중요성을 보다 신중히 고려하여 기능을 구현할 계획입니다. 또한, 계산기 식이 잘못되었을 때 오류 메시지를 반환하는 기능을 구현했지만, 계산식의 오류를 사전에 방지하는 고도화된 기능을 추가하기 위해서는 정규표현식에 대한 심층적인 학습이 필요하다는 것을 깨달았습니다.

```
drawOnCanvas() {
  class Canvas {
    constructor(id) {
      this.id = id;
      this.canvas = null;
      this.pen = null;
      this.cWidth = null;
      this.cHeight = null;
      this.xAxisMove = 0;
      this.yAxisMove = 0;
      this.scaleFactor = 20;
    }//

    initCanvas() {···
```

해당 값은 고유하지 않고, 변동이 있는 값이므로,
메서드에 매개변수로 전달하는 것이 더 의미 있는
데이터 처리 방식이었을 것이라 생각합니다.

Thank you

김유진