(슬라이드 1) 안녕하십니까 오늘 memoization에 대해 발표하게 된 소프트웨어학과 22학번 이이슬입니다.

(슬라이드 2) 먼저 메모이제이션이란 컴퓨터 프로그램이 동일한 계산을 반복해야 할 때, 이전에 계산한 값을 메모리에 저장함으로써 동일한 계산의 반복 수행을 제거하여 프로그램 실행 속도를 빠르게 하는 기술입니다.

(슬라이드 3) 리액트에서 메모이제이션을 하는 대표적인 방법으로는 React.memo, useMemo, useCallback 이와 같이 3가지가 있고 오늘 이에 대해 알아보도록 하겠습니다.

(슬라이드 4) 먼저 리액트 메모의 사용이유에 대해 보자면, Student라는 컴포넌트가 있다고 가정해보겠습니다. 이 컴포넌트는 이름, 나이, 주소를 props로 전달받아 props에 따라 다른 학생의 정보를 보여주는 컴포넌트입니다. 그리고 오른쪽에는 Student컴포넌트를 자식으로 갖고 있는 School이라는 부모 컴포넌트가 있습니다.  
리액트에서는 기본적으로 부모 컴포넌트가 렌더링 되면 모든 자식 컴포넌트들도 자동적으로 렌더링이 되는데요. 이 말은 student props가 홍길동, 20, 우리집과 같이 고정되어 있어 같은 결과를 출력함에도 불구하고 school컴포넌트가 렌더링 되면 student컴포넌트도 렌더링 되는 것인데 이는 매우 비효율적인 것을 알 수 있죠.   
그렇다면 props값이 다를 때 렌더링 되도록 하면 좀 더 효율적이겠죠. 이때 사용하는 것이 리액트 메모입니다.

(슬라이드 5) 리액트 메모는 리액트에서 제공하는 고차 컴포넌트인데요. 영어로 Higher Order Component로 줄여서 HOC라고 부릅니다. 고차 컴포넌트는 어떤 컴포넌트를 인자로 받아서 새로운 컴포넌트를 반환해 주는 것인데요.   
이와 같이 리액트 메모에 컴포넌트를 인자로 받아 좀 더 최적화된 컴포넌트를 반환해 줍니다.   
이렇게 최적화된 컴포넌트는 렌더링이 되어야 할 상황에 놓일 때마다 props check를 통해 props에 변화여부를 확인하는데 변화가 있다면 렌더링을 하고, 변화가 없다면 기존에 렌더링이 된 내용을 재사용 합니다.

(슬라이드 6) 리액트 메모에서 메모는 메모이제이션을 의미하는데, 바로 앞에서 렌더링을 재사용한다고 하였는데 렌더링 재사용이란 말은 이전에 렌더링 한 컴포넌트를 메모리에 저장해놓고 새로 렌더링 하는 대신 꺼내서 재사용하는 것이며 이는 리액트 메모의 메모이제이션 기법이죠.  
그림을 참고해 보자면 홍길동이라는 props로 렌더링을 처음 하였을 때 메모리에 저장해 놓고, 똑같은 홍길동이라는 props라면 새로 렌더링 하는 대신 메모리에서 꺼내서 재사용 하는 것입니다.

(슬라이드 7) 다음으로는 useMemo입니다. 시작하기 전에 함수형 컴포넌트에 대해 알아야 하는데 간략히 설명하면, 함수형 컴포넌트가 렌더링이 된다는 것은 그 함수가 호출된다는 것이고, 함수는 호출될 때마다 함수 내부에 정의되어 있는 모든 변수들이 초기화 됩니다.  
왼쪽에 있는 컴포넌트를 한번 보면 컴포넌트 안에는 value라는 변수가 하나 있는데 이 변수는 calculate라는 함수로부터 값을 받아오죠.  
컴포넌트가 렌더링이 될 때마다 value라는 변수가 초기화 되기 때문에 calculate함수는 반복적으로 호출되겠죠. calculate함수가 지금은 간단히 10이라는 값을 리턴 하지만 무거운 일을 한다면 굉장히 비효율적일 겁니다. 이 경우 useMemo를 사용해 이런 상황을 해결할 수 있습니다.

(슬라이드 8) useMemo는 처음에 계산된 결과값을 메모리에 저장해서 컴포넌트가 반복적으로 렌더링 되어도 calculate를 다시 호출하지 않고, 이전에 이미 계산된 결과 값을 메모리에서 꺼내서 재사용할 수 있게 해줍니다.

(슬라이드 9) 다음으로는 useMemo의 구조에 대해서 간단하게 살펴보겠습니다. useMemo는 이와 같이 2개의 인자를 받습니다. 첫번째 인자로는 콜백함수, 두번째 인자로는 배열을 받습니다.  
첫번째 인자인 콜백함수는 우리가 메모이제이션 해줄 값을 계산해서 리턴해주는 함수이며, 이 콜백함수가 리턴 하는 값이 바로 useMemo의 리턴 값이 됩니다.  
두번째 인자인 배열은 의존성 배열이라고도 불립니다. useMemo는 배열 안에 요소의 값이 업데이트 될 때만 콜백함수를 다시 호출해서 메모이제이션 된 값을 업데이트해서 다시 메모이제이션 합니다.

(슬라이드 10) 이제 다음으로 useCallback을 보겠습니다. 아까와 비슷한데요 왼쪽의 예제를 보면 calculate 변수 안에 제가 표시해 둔 것과 같이 함수 객체가 할당되어 있는 것을 볼 수 있는데요.  
렌더링 될 때마다 변수가 초기화 되기 때문에 함수 객체도 반복적으로 할당 받는 다는 것을 알 수 있습니다.

(슬라이드 11) 이 때, useCallback을 사용한다면 컴포넌트가 다시 렌더링 되어도 calculate가 초기화 되는 것을 막을 수 있습니다.  
컴포넌트가 처음 렌더링 될 때만 함수 객체를 만들어서 초기화 해주고, 이후 렌더링 될 때는 새로운 함수 객체를 다시 할당 받지 않고, 이전에 할당 받는 함수 객체를 계속해 갖고 있으면서 재사용하게 됩니다.

(슬라이드 12) 다음으로 useCallback의 구조에 대해 보자면 useMemo와 동일하게 2개의 인자를 받습니다. 첫번째 인자로는 메모이제이션 해줄 콜백함수, 두번째 인자로는 의존성 배열을 받습니다. Calculate 변수는 메모이제이션 된 함수를 가지고, 이 메모이제이션 된 함수는 의존성 배열 내부에 있는 값이 변경되지 않는 이상 다시 초기화 되지 않습니다. 이렇게 보시면 useMemo와 동일하지 않나 싶으실텐데 useMemo는 값을 메모이제이션하고, useCallback은 함수를 메모이제이션 한다고 이해하면 좋을 것 같습니다.

(슬라이드 13) 이제 실습을 하겠습니다. 먼저 리액트 메모 실습을 하겠습니다. 리액트 만들었다고 가정하고, App.js에 슬라이드 13에 있는 코드를 복사 붙여넣기 해주세요.

(슬라이드 14) 다음으로 Parent.js, Child.js 파일을 만들고, 14슬라이드에 있는 코드를 복사 붙여넣기 해주세요!!  
이 코드는 파일명과 같이 child를 자식 컴포넌트로 가지는 Parent 부모 컴포넌트를 입니다.  
실행시켜보면 이와 같이 부모 나이 증가, 자녀 나이 증가 버튼이 있고 이를 누르게 되면 숫자가 증가하는 것을 볼 수 있죠,   
F12를 눌러 개발자 창으로 가서 새로 고침 하고, 콘솔을 확인해보면 처음 렌더링 되는 것이기 때문에 부모, 자식 컴포넌트가 렌더링 된 것을 볼 수 있고, 자녀 나이를 증가시키면 자식 컴포넌트가 렌더링 되어 당연하게 부모 컴포넌트도 렌더링 되죠.   
그리고 부모 나이 증가 버튼을 누르는데 자녀 컴포넌트도 같이 렌더링 되는 것을 볼 수 있습니다. 이와 같이 자녀 나이에는 변화가 없는데 같이 렌더링 되는 것을 볼 수 있고, 이는 매우 비효율적이죠.  
따라서 리액트 메모를 사용해 자녀 컴포넌트를 최적화시켜 보겠습니다. Child.js로 이동해주시고 제가 주석 처리했던 문장과 동일하게 변경하여 저장합니다.   
그리고 다시 새로 고침 하여 확인해 보면 처음에 같이 렌더링이 되고, 부모 나이 증가 버튼을 누를 때 자녀 컴포넌트는 렌더링 되지 않는 것을 확인할 수 있습니다.

(슬라이드 15) 다음으로 useMemo실습을 해보겠습니다. App.js에 가서 실습1은 주석처리 실습2는 주석처리를 풀어줍니다! 그리고 useMemo.js파일을 만들어 주시고 ppt 코드를 복사 붙여넣기 해주시고 저장해주세요.  
이 코드는 시간이 조금 걸리는 어려운 계산기와 바로 계산이 되는 쉬운 계산기 함수를 사용하는 코드입니다.  
실행하여 보면 어려운 계산기는 버튼 클릭을 하면 1초쯤 뒤에 변합니다. 하지만 쉬운 계산기는 딜레이 없이 바로 실행되어야 하는데 딜레이가 발생하는 것을 볼 수 있습니다. 이 이유는 함수를 렌더링 하면 모든 변수가 초기화 되어 어려운 계산도 같이 실행되기 때문이고, 개발자 창에서 이와 같이 확인할 수 있습니다.  
이를 해결하기 위해 useMemo를 사용해 어려운 계산을 최적화해 주겠습니다. useMemo.js에 들어가 주시고, 제가 주석 처리한 것처럼 변경해 주시면 됩니다.  
다시 새로 고침 해보면, 처음에는 어려운 계산 쉬운 계산이 초기화 되어 같이 실행되는 것을 볼 수 있고, 어려운 계산기는 이전과 동일하게 딜레이가 있고 쉬운 계산은 딜레이가 없어지고 쉬운 계산을 할 때 어려운 계산 함수가 호출되지 않는 것을 확인할 수 있습니다.

(슬라이드 16) 마지막으로 useCallback실습을 해보겠습니다. 전 실습과 같이 App.js에서 실습2를 주석처리 해주시고, 실습3의 주석처리를 풀어줍니다. 그리고 useCallback.js파일을 만들어 주시고 ppt 코드를 복사 붙여넣기 해주세요.  
실행시켜 주시면 이와 같이 수 증가 및 감소 가능하고, 밑의 버튼을 눌러 주시면 콘솔창에 이와 같이 숫자에 대해 나오게 됩니다.  
다시 코드로 돌아가서 useEffect의 주석처리를 풀어주세요. 이는 someFunction이 변경될 때 렌더링 되고 콘솔창에 출력하는 것입니다.   
이제 실행시켜 보시면 숫자를 변경시키고, 함수 버튼을 안 눌러도 계속 변경되었다고 하는 것을 알 수 있습니다. Number가 변경될 때마다 변수가 초기화 되어, 함수 객체가 새로 생겨, useEffect는 함수 객체의 주소가 변경되어 버튼을 누르지 않아도 변경되었다고 인식해 출력한 것입니다.  
이를 해결하기 위해서 useCallback을 사용하여 함수 객체를 메모이제이션 하겠습니다. 코드로 돌아가 제가 주석한 것과 동일하게 변경해 주시고 새로 고침 해줍니다.  
그리고 숫자를 변경해 주면, 이전과는 다르게 useEffect가 실행되지 않는 것을 볼 수 있습니다. 하지만 버튼을 눌러도 변경되었다는 출력이 없고 현재 숫자와 다르게 출력되는 것을 확인할 수 있는데요. 이는 처음 메모이제이션 해줬을 당시에 number state가 0이였고, 그 다음 렌더링 부터는 이미 메모이제이션 해준 함수를 재사용 하기 때문에 number state가 바뀌어도 메모이제이션 해준 함수에는 number에 0이 들어 있기 때문입니다.  
이를 해결해보겠습니다. 코드로 돌아가 2번째 인자인 의존성 배열에 number라고 적어 주시고 저장해 주세요. 이는 number state가 변경되면 함수객체를 다시 생성하여 메모이제이션 하라는 의미입니다.  
이제 새로 고침하고 숫자 변경을 해보면 Number가 바뀌었기 때문에 함수 객체가 새로 생성되어 useEffect가 계속 실행되는 것을 볼 수 있습니다. 그리고 버튼을 클릭하면 현재 숫자에 맞게 나오는 것도 확인할 수 있습니다.

(슬라이드 17) 이상으로 발표를 마치겠습니다.