15. 캔버스로
그래픽 요소 다루기

다양한 스타일 지정하기

색상 지정하기

따로 색상을 지정하지 않으면 기본색인 검은색으로 채워지거나 테두리가 그려진다

```
fillStyle = color;      // 채우기 색상
strokeStyle = color;    // 선 색상
```

color를 지정할 때는 색상 이름이나, 16진수 색상값,
rgb/rgba, hsl/hsla 등을 사용할 수 있다

투명도 조절하기

한꺼번에 여러 도형의 불투명도를 조절하려면 globalAlpha 속성 사용

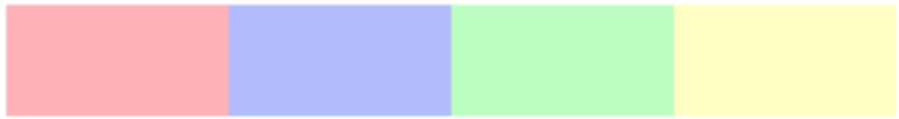
```
globalAlpha = 값
```

사용할 수 있는 값 : 0.0(완전 투명) ~ 1.0(완전 불투명). 기본값은 1.0

globalAlpha 속성으로 투명도를 설정하면 투명도를 설정한 이후에 그려지는 도형에는 같은 투명도가 적용된다

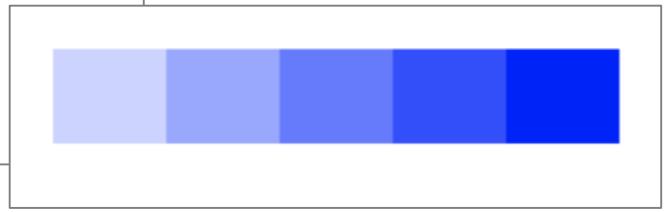
(예) 투명도 조절하기

```
ctx.globalAlpha = 0.3;              // 이후의 모든 도형에 불투명도 적용
ctx.fillStyle = "rgb(255, 0, 0)";
ctx.fillRect(50, 50, 100, 50);
ctx.fillStyle = "rgb(0, 0, 255)";
ctx.fillRect(150, 50, 100, 50);
ctx.fillStyle = "rgb(0, 255, 0)";
ctx.fillRect(250, 50, 100, 50);
ctx.fillStyle = "rgb(255, 255, 0)";
ctx.fillRect(350, 50, 100, 50);
```



(예) rgba() 사용해서 불투명도 조절하기

```
ctx.fillStyle = "rgb(0, 0, 255, .2)";    // 거의 투명하게
ctx.fillRect(50, 50, 60, 50);
ctx.fillStyle = "rgba(0, 0, 255, .4)";
ctx.fillRect(110, 50, 60, 50);
ctx.fillStyle = "rgba(0, 0, 255, .6)";
ctx.fillRect(170, 50, 60, 50);
ctx.fillStyle = "rgb(0, 0, 255, .8)";
ctx.fillRect(230, 50, 60, 50);
ctx.fillStyle = "rgb(0, 0, 255, 1)";     // 불투명하게
ctx.fillRect(290, 50, 60, 50);
```



색상은 같고, 불투명도만 조절해서 다른 느낌을 만들 수 있다.

그러데이션

선형 그러데이션 만들기

1) 도형에 그러데이션을 적용하려면 사용할 **그러데이션 객체부터 만들어야 한다.**

- 선형 그러데이션은 가로 방향이나 세로 방향으로 그러데이션이 바뀐다.
- 시작점(x1, y1)과 끝점(x2, y2)을 지정하여 사각 형태의 그러데이션 영역을 만든다.

```
createLinearGradient(x1, y1, x2, y2)
```

예) 너비값이 없고 높이값만 있기 때문에 위에서 아래로 색상이 변하는 그러데이션이 된다.

```
let grad = ctx.createLinearGradient(0, 0, 0,
100);
```

선형 그러데이션 만들기

2) 그라디언트 객체를 만들었으면 **중지점을 사용해 색상을 지정한다.**

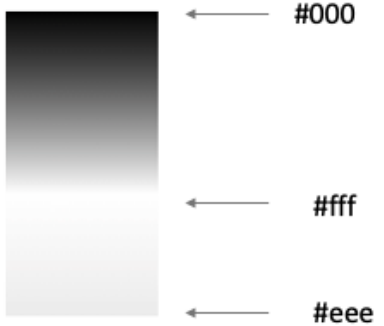
```
addColorStop(position, color)
```

- position : 그러데이션 영역에서의 색상의 위치를 상대적으로 표시.
  0.0~1.0까지의 값을 사용할 수 있습니다. (시작 위치 0.0, 끝 위치 1.0)
- color는 색상 이름이나 색상값 중에서 사용

3) 그러데이션을 만들었으면 **fillStyle**이나 **strokeStyle**을 이용해 스타일에 적용할 수 있습니다.

(예) 선형 그러데이션 만들기

```
const canvas = document.querySelector('canvas');
const ctx = canvas.getContext("2d");

let linGrad = ctx.createLinearGradient(0, 0, 0, 200);
linGrad.addColorStop(0, "#000");          // 시작 위치에 검정색
linGrad.addColorStop(0.6, "#fff");        // 0.6 위치에 흰색
linGrad.addColorStop(1, "#eee");          // 끝나는 위치에 회색

ctx.fillStyle = linGrad;
ctx.fillRect(0, 0, 100, 200);
```



원형 그러데이션 만들기

1) 원형 그러데이션은 색상이 시작되는 원과 색상이 끝나는 원을 지정해서 **그러데이션 객체를 만든다.**

```
createRadialGradient(x1, y1, r1, x2, y2, r2)
```

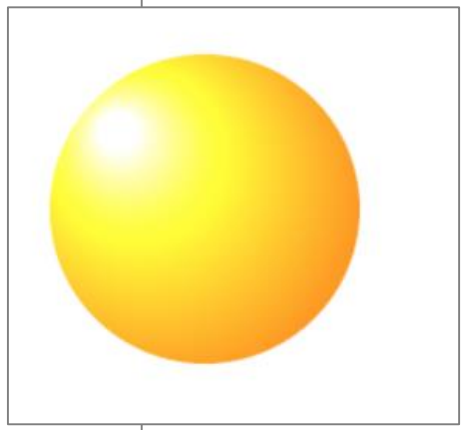
시작 원 : (x1, y1)을 중점으로 하는 반지름 r1인 원, 끝나는 원 : (x2, y2)를 중점으로 하는 반지름 r2인 원

예)
```
let radgrad = ctx.createRadialGradient(55, 60, 10, 80, 80, 100);
```

2) 원형 그러데이션 객체를 만들었으면 addColorStop() 메서드를 사용해서 **중지점에 그러데이션의 색상을 지정한다**.

(예) 원형 그러데이션 만들기

```
const canvas = document.querySelector('canvas');
const ctx = canvas.getContext("2d");

let radGrad = ctx.createRadialGradient(55, 60, 10, 80, 90, 100);
radGrad.addColorStop(0, "white");        // 시작 위치에 흰색 원
radGrad.addColorStop(0.4, "yellow");     // 0.4 위치에 노란색 원
radGrad.addColorStop(1, "orange");       // 끝 위치에 주황색 원
ctx.fillStyle = radGrad;
ctx.beginPath();
ctx.arc(100, 100, 80, 0, Math.PI * 2, false);
ctx.fill();
```



패턴 채우기

패턴을 채울 때에도 패턴 객체를 만든 후 채우기 스타일이나 선 스타일에 패턴 객체를 지정한다.

```
createPattern(image, type)
```

- image: 패턴 이미지 파일의 경로. 이미지 객체를 만든 후 파일을 가져와서 사용한다.
- type: 패턴 이미지의 반복 형태.

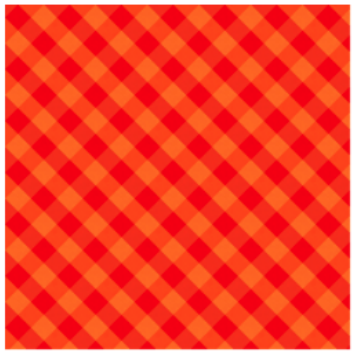
  사용할 수 있는 값은 repeat, repeat-x, repeat-y, no-repeat

(예) 패턴 채우기

```
const canvas = document.querySelector('canvas');
const ctx = canvas.getContext("2d");

let img = new Image();
img.onload = function() {
  let pattern = ctx.createPattern(img, "repeat");    // 패턴 객체
  ctx.fillStyle = pattern;
  ctx.fillRect(0, 0, 200, 200);
}
img.src = "images/pattern.png";
```

원래 이미지

캔버스 패턴



그림자 효과 추가하기

shadowOffsetX `ctx.shadowOffsetX = 거릿값`

- 그림자가 수평 방향으로 얼마나 떨어져 있는지를 나타낸다.
- 속성값이 양수이면 오른쪽 방향으로, 음수이면 왼쪽 방향으로 그림자가 생긴다. 기본값은 0.

shadowOffsetY `ctx.shadowOffsetY = 거릿값`

- 그림자가 수직 방향으로 얼마나 떨어져 있는지를 나타낸다.
- 속성값이 양수이면 아래쪽 방향으로, 음수이면 위쪽 방향으로 그림자가 생긴다. 기본값은 0.

그림자 효과 추가하기

shadowColor `ctx.shadowColor = 색상`

그림자색. 기본 값은 완전히 투명한 검정색

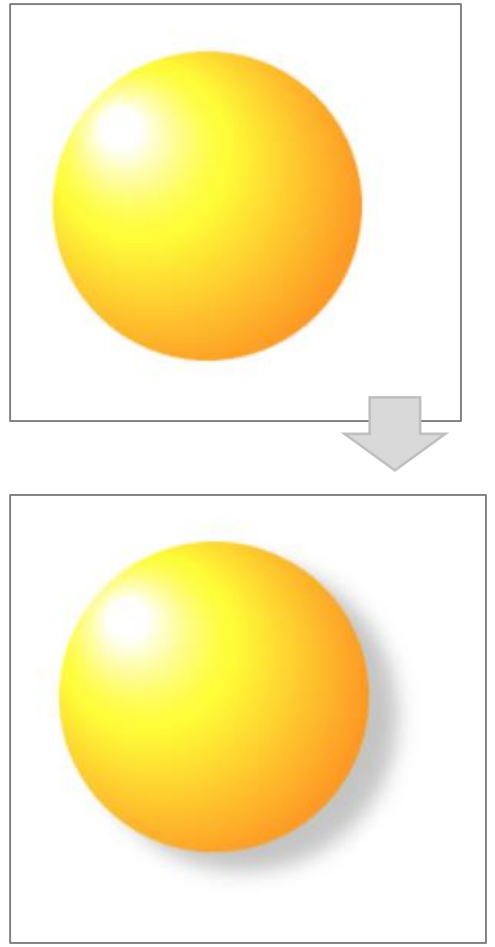
shadowBlur `ctx.shadowBlur = 값`

- 그림자가 얼마나 흐릿한지를 나타낸다.
- 이 속성의 기본값은 0. (0일 때 그림자가 가장 진하고 숫자가 커질수록 그림자는 점점 흐려진다)

(예) 그림자 효과 추가하기

그림자 효과는 도형을 그리기 전에 미리 선언해야 한다.

```
…(중략)
ctx.shadowColor = "#ccc";    // 그림자 색
ctx.shadowOffsetX = 15;      // 그림자 가로 오프셋
ctx.shadowOffsetY = 10;      // 그림자 세로 오프셋
ctx.shadowBlur = 10;         // 그림자 흐림 정도

let radGrad = ctx.createRadialGradient(55, 60, 10, 80, 90,
100);
…(중략)
```



선과 관련된 스타일 속성

선 굵기 `ctx.lineWidth = 값`

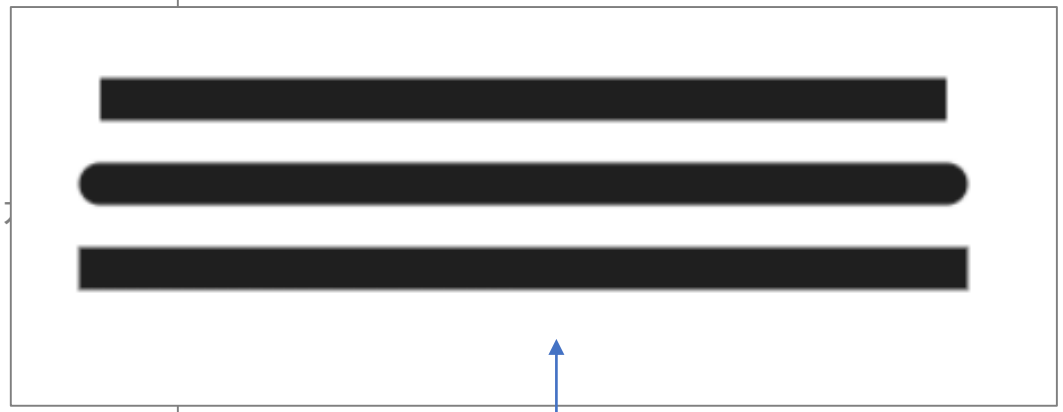
선의 굵기 조절. 기본 값은 1.0이고, 음수는 사용할 수 없다.

선의 끝 모양 `ctx.lineCap = butt 또는 round 또는 square`

- butt: 기본값. 끝부분을 단면으로 처리
- round: 선 너비의 1/2을 반지름으로 하는 반원이 선의 양쪽 끝에 그려진다.
- square: 선의 양쪽 끝에 사각형이 그려지고 높이는 선 너비의 1/2.

(예) 선의 굵기와 끝 모양 지정하기

```
const lineCap = ['butt', 'round', 'square'];

ctx.strokeStyle = '#222';
for(let i = 0; i < lineCap.length; i++) {
  ctx.lineWidth = 15;
  ctx.lineCap = lineCap[i];
  ctx.beginPath();
  ctx.moveTo(50, 50 + i * 30);          // 시작 위
  ctx.lineTo(350, 50 + i * 30);         // 끝 위치
  ctx.stroke();
}
```



lineCap을 round로 설정했을 때와 square로 설정했을 때
끝부분이 선 너비의 1/2만큼씩 확장된

선과 관련된 스타일 속성

선과 선의 만남

```
ctx.lineJoin = bevel || meter || round
```

- bevel: 두 선의 연결 부분을 칼로 자른 듯한 단면으로 만든다.
- miter: 연결한 흔적 없이 마치 처음부터 하나의 선이었던 것처럼 연결한다.
- round: 선과 선이 만나는 부분을 둥글게 처리한다.

선 연결 부분의 잘린 크기

```
ctx.miterLimit = 값
```

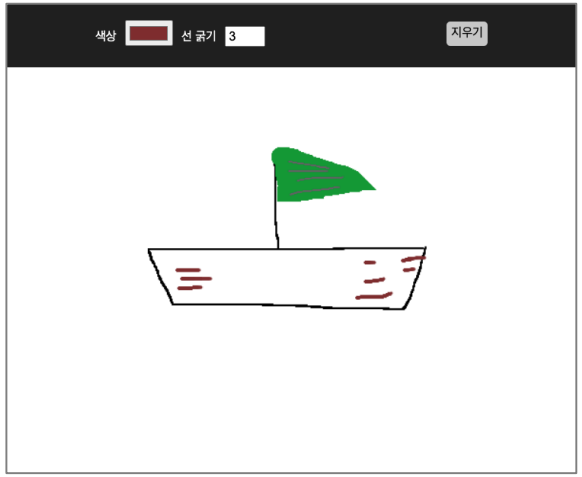
2개의 선이 연결된 부분에는 꼭지점이 생기는데, 이 부분을 얼마나 잘라낼 것인지 결정한다.

기본값은 10.

[실습] 나만의 드로잉 앱 만들기

<생각해 보기>

- 사용자가 선택한 스타일 값을 어떻게 가져올까?
- 마우스를 클릭한 상태에서는 마우스가 움직이는 만큼 선을 그리고, 마우스 버튼에서 손을 떼면 그리기를 멈추려면 어떻게 해야 할까?



문서 구조 파악하기

```
<div id="container">
  <div id="toolbar">
    <div>
      <label for="stroke">색상</label>
      <input type="color" id="stroke" value="#000">
      <label for="lWidth">선 굵기</label>
      <input type="number" id="lWidth" min="1" max="50" value="2">
    </div>
    <button id="reset">지우기</button>
  </div>
  <canvas id="canvas"></canvas>
</div>
```

색상과 굵기 선택 부분

지우기 버튼

툴바 부분

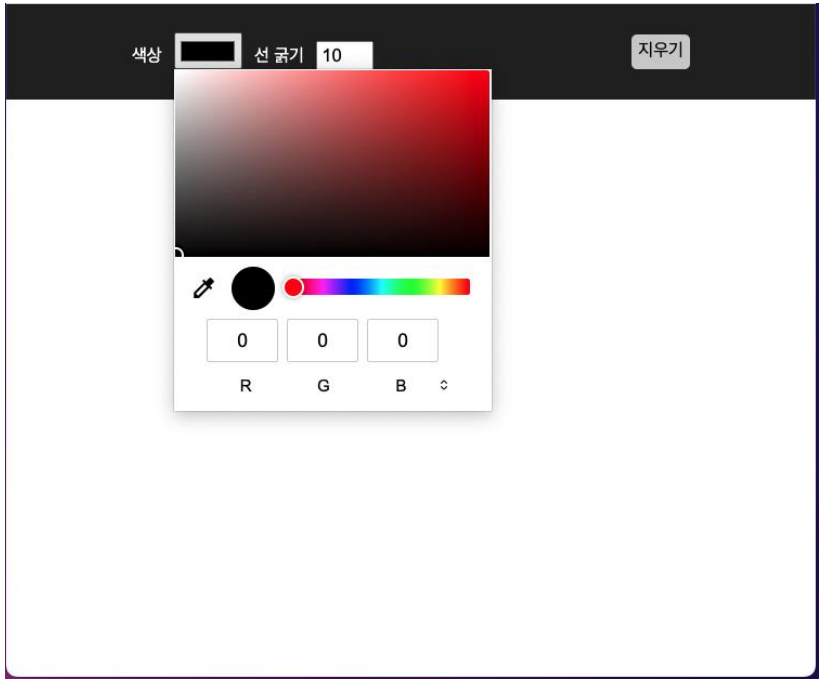
CSS 연결하기

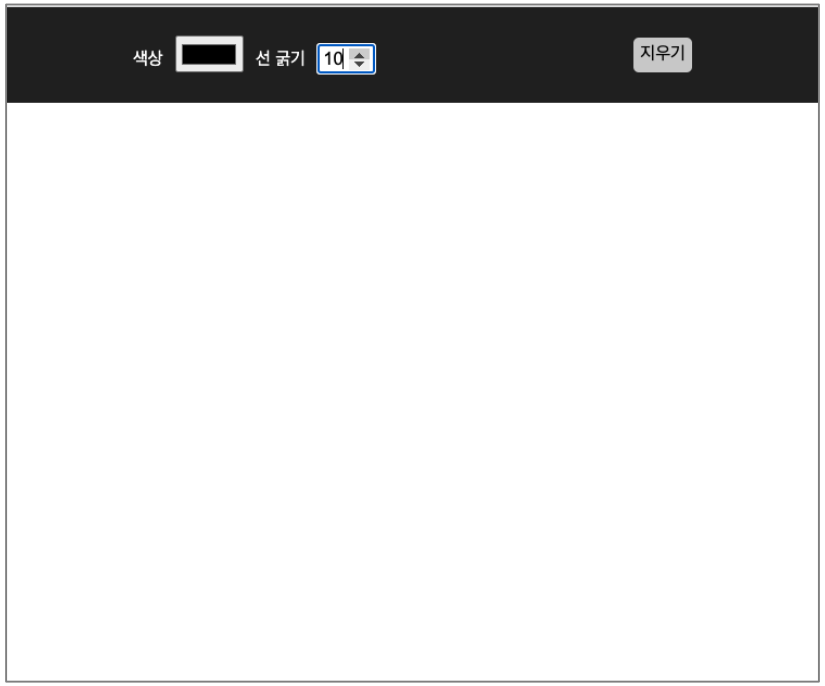
```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
body {
  overflow: hidden;
}
#container {
  width: 100%;
  display:flex;
  flex-direction: column;
  justify-content: left;
  align-items: top;
}
```

```
#toolbar {
  width:100%;
  display:flex;
  flex-direction: row;
  justify-content: space-
around;
  align-items: center;
  height: 70px;
  background-color: #222;
}
#toolbar div * {
  margin-right: 5px;
}
```

```
#toolbar label {
  font-size:0.8rem;
}

#toolbar button {
  background-color:
#cecece;
  border: none;
  border-radius: 4px;
  padding: 5px;
  font-size: 0.8rem;
  cursor: pointer;
}
```

브라우저로 확인하기





캔버스 크기 지정하기

1) drawing.js 파일을 만든 후, drawing.html에 연결

2) 캔버스 영역과 툴바 영역을 가져와서 변수로 저장.

3) 캔버스 크기 지정. 캔버스 높이는 화면 높이에서 툴바 높이를 뺀 만큼.

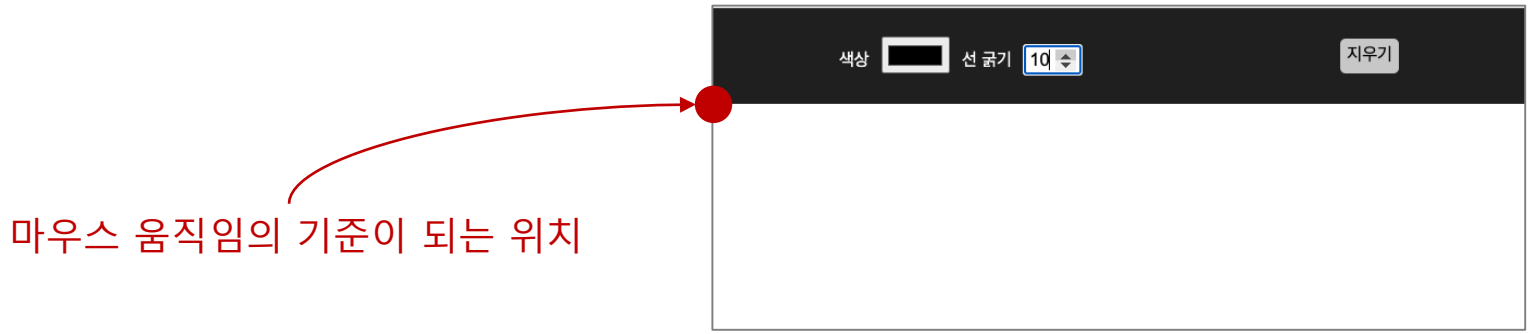
```
const canvas = document.querySelector("#canvas");

const toolbar = document.querySelector("#toolbar");


// 캔버스 너비와 높이. toolbar.offsetHeight는 툴바의 높이.

canvas.width = window.innerWidth;

canvas.height = window.innerHeight - toolbar.offsetHeight;
```

변수 지정하기

- 마우스 커서를 사용해 그리기를 할 때에는 event 객체의 clientX와 clientY 값을 사용해서 마우스 커서의 위치를 알아낸다.
- clientX와 clientY는 브라우저 창의 왼쪽 위 모퉁이를 기준으로 하지만, 여기에서는 브라우저 화면의 툴바 크기를 뺀만큼을 기준으로 해야 한다. 그래서 오프셋 값을 변수로 저장해 둔다.

```
const canvasOffsetX =
canvas.offsetLeft;
```

```
const canvasOffsetY =
canvas.offsetTop;
```



변수 지정하기

- 현재 그리기 상태인지의 여부를 구별하는 isDrawing 변수가 필요 (true / false)
- 선을 그리기 시작할 위치를 좌푯값 startX, startY 변수가 필요하고
- 기본 선 굵기값을 저장할 lineWidth가 필요하다.

```
const ctx = canvas.getContext("2d");


let isDrawing = false;  // 드로잉 상태인지 확인합니다.
let startX;             // 그리기 시작하는 좌표, x
let startY;             // 그리기 시작하는 좌표, y
let lineWidth = 2;      // 선 굵기 기본값
```

툴바에서 값을 변경하거나 버튼을 클릭하면?

- "id=stroke" 부분이 변경되었으면 선 색을 가져오고,
  "id=lwidth" 부분이 변경되었으면 선 굵기를 가져와 값을 할당한다
- [지우기] 버튼을 클릭했다면 캔버스 크기만큼 사각형을 지운다.

```
// 선 색과 선 굵기를 선택했을 때
toolbar.addEventListener("change", e =>
{
  if (e.target.id === "stroke") {
    ctx.strokeStyle = e.target.value;
  }
  if (e.target.id === "lWidth") {
    lineWidth = e.target.value;
  }
});
```

```
// '지우기' 버튼 누르면 캔버스 지우기
reset.addEventListener("click", (e) =>
{
  ctx.clearRect(0, 0, canvas.width,
canvas.height);
});
```

마우스를 클릭하면?

- 마우스 버튼을 클릭하면(mousedown) 그리기 시작한다는 뜻이므로
- `isDrawing`을 `true`로 바꾸고
- 현재 좌푯값(클릭한 위치의 좌푯값)을 시작 좌표 `startX`와 `startY`로 지정한다.

```
canvas.addEventListener("mousedown" , e => {
  isDrawing = true;
  startX = e.clientX;
  startY = e.clientY;
});
```

마우스를 움직이거나 멈추면?

- 마우스를 움직이면 시작 좌표부터 현재 좌푯값까지 선을 그린다.

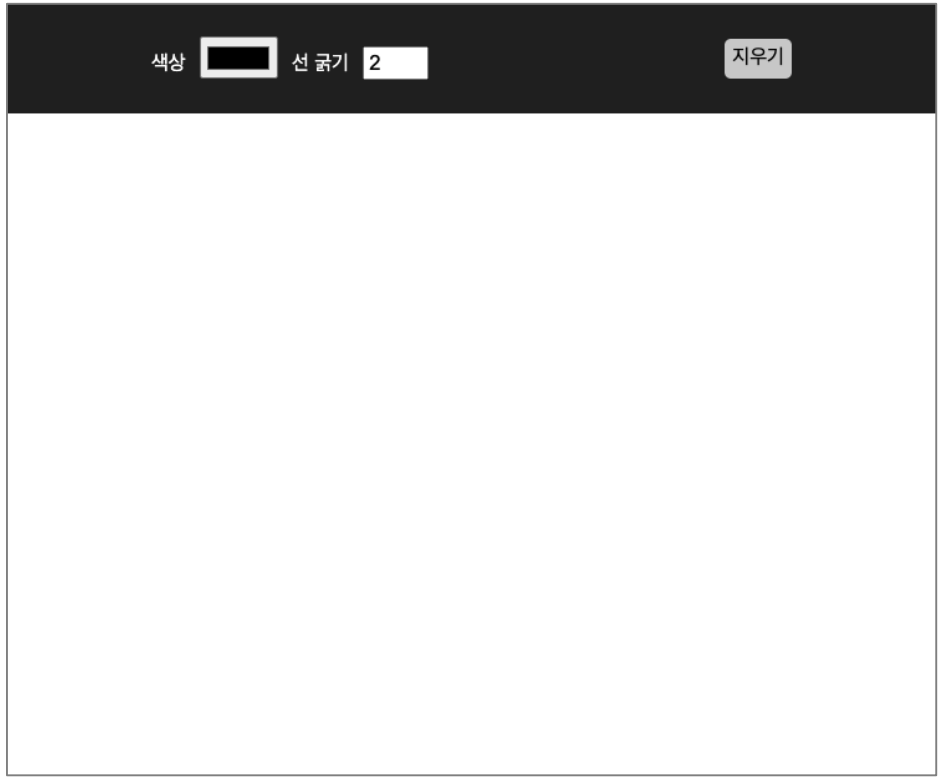
  → 툴바 영역까지 그려지지 않도록 현재 좌표의 y값을 캔버스의 위쪽 오프셋만큼 뺀다.
- 마우스 버튼에서 손을 떼면 그리기를 끝낸다.

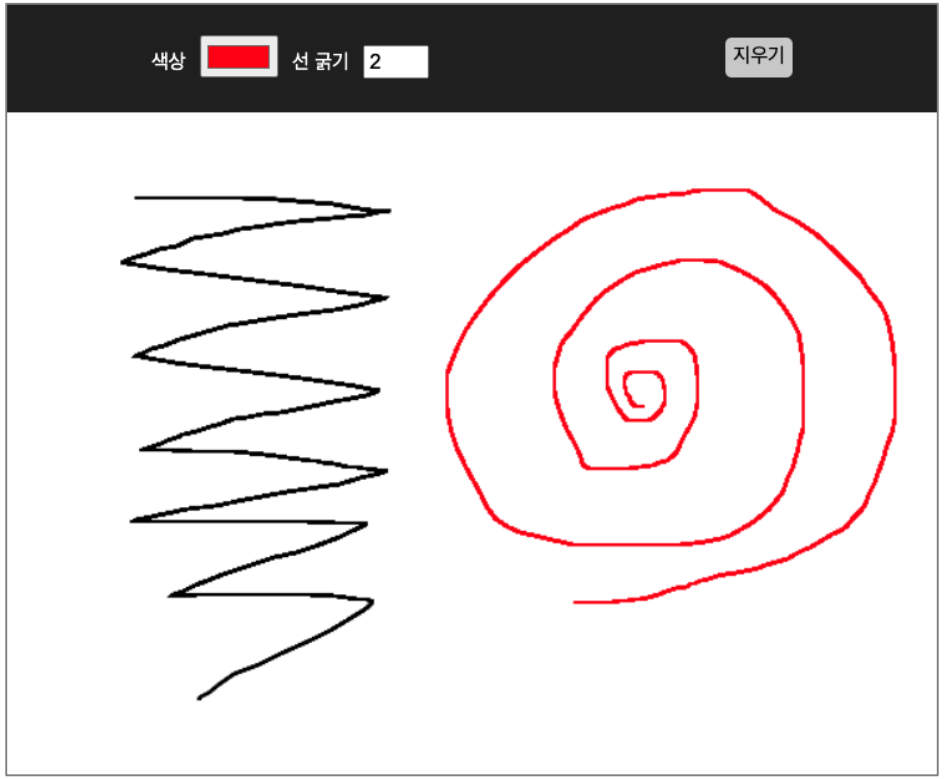
  → `isDrawing`을 `false` 상태로 만들고 다른 경로를 시작할 수 있도록 한다.

```
canvas.addEventListener("mousemove", (e) => {
  if (!isDrawing) return;
  ctx.lineWidth = lineWidth;
  ctx.lineCap = "round";
  ctx.lineTo(e.clientX, e.clientY - canvasOffsetY);
  ctx.stroke();
});
canvas.addEventListener("mouseup", () => {
  isDrawing = false;
  ctx.beginPath();
});
```

브라우저에서 확인하기

25\result\js\drawing.js





그래픽 요소 변형하기

콘텍스트 상태 저장하기

- 캔버스에서 이동시키거나 회전시키는 동작은 캔버스 위에 있는 그래픽 요소를 변형시키는 것이 아니라 2D 콘텍스트 자체를 옮기거나 회전시키는 것
- 그래픽 요소를 변형시키려면 콘텍스트 상태를 저장한 후 변형하고 다시 원래 상태로 복구시키는 과정이 필요하다.

```
save()                    // 현재 상태 저장
restore()                 // 저장한 상태 복구
```

어떤 정보가 저장될까?

캔버스에 적용된 변형, strokeStyle과 fillStyle, globalAlpha, lineWidth, lineCap, lineJoin, shadowOffsetX, shadowOffsetY, shadowBlur, shadowColor 속성, 클리핑 경로 등이 저장됩니다.

위치 옮기기

- 캔버스에 있는 도형이나 이미지는 그 좌표를 정할 때 캔버스의 원점을 기준으로 한다.
  → 캔버스의 원점을 옮기면 캔버스와 관련된 모든 요소도 그에 맞춰 위치가 바뀐다.

- translate() 메서드를 사용하면 원점의 위치를 옮겨서 그래픽 요소의 위치를 바꿀 수 있다.
- translate(x, y)는 원점의 위치를 (x, y)로 옮긴다.

```
translate(x, y)
```

  - x: 왼쪽이나 오른쪽으로 옮길 크기
  - y: 위나 아래로 옮길 크기

(예) 위치 옮기기

1) 회색 사각형 그리기

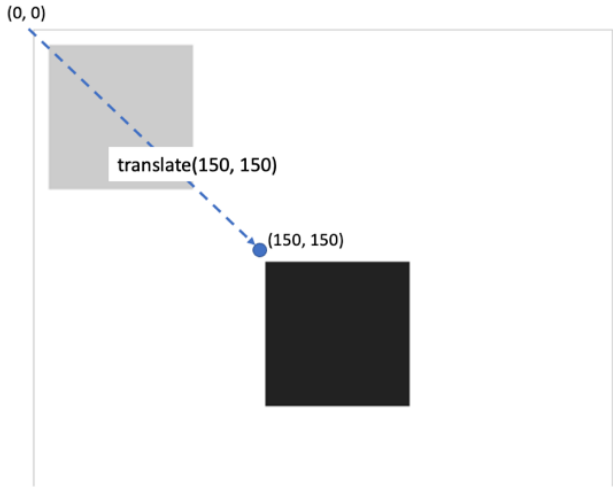
```
ctx.fillStyle = "#ccc";
ctx.fillRect(10, 10, 100, 100);
```

2) 현재 상태 저장 & 원점 옮기기

```
ctx.save();                    // 현재 드로잉 상태 저장
ctx.translate(150, 150);       // 원점 이동
```

3) 검정 사각형 그리기 & 빨간 사각형 그리기 & 상태 복구

```
ctx.fillStyle = "#222";
ctx.fillRect(10, 10, 100, 100);
ctx.fillStyle = "red";
ctx.fillRect(50, 50, 20, 80);
ctx.restore();                 // 드로잉 상태 복구
```

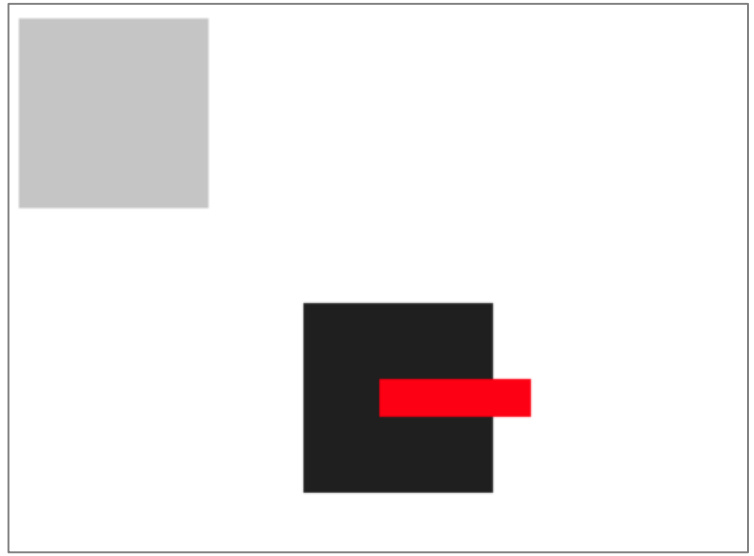


2) 현재 상태 저장 & 원점 옮기기

```
ctx.save();                    // 현재 드로잉 상태 저장
ctx.translate(150, 150);       // 원점 이동
```

3) 검정 사각형 그리기 & 빨간 사각형 그리기 & 상태 복구

```
ctx.fillStyle = "#222";
ctx.fillRect(10, 10, 100, 100);
ctx.fillStyle = "red";
ctx.fillRect(50, 50, 20, 80);
ctx.restore();                 // 드로잉 상태 복구
```



회전시키기

rotate() 메서드를 사용하면 그래픽 요소를 회전시킬 수 있다.

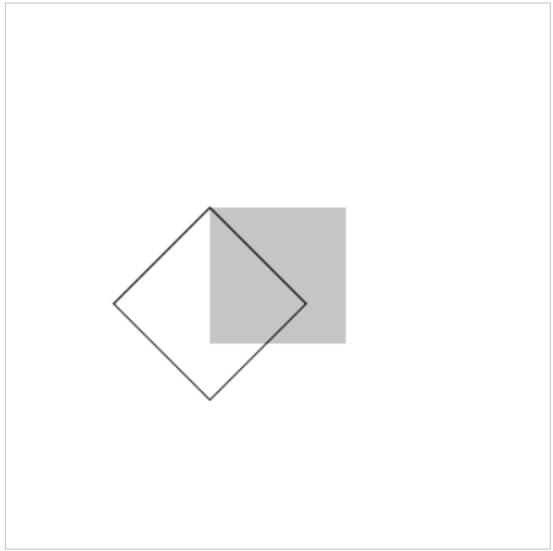
rotate() 메서드에 있는 매개변수 '각도'는 라디안값이고 시계 방향으로 회전한다.

그래픽 요소를 회전할 때는 캔버스의 원점을 기준으로 회전시킨다는 점에 주의해야 한다.

```
rotate(각도)
```

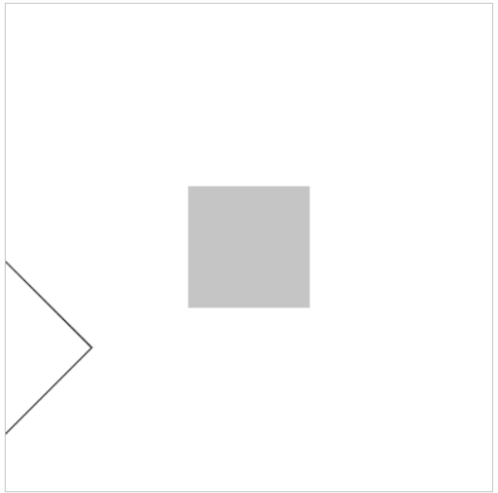
(150, 150) 위치에서 회색 사각형을 그렸을 때,

사각형을 회전시켜서 마름모 도형을 선으로 표시하려면 어떻게 해야 할까?



(예) 45도 회전시키키

```
// 회색 사각형
ctx.fillStyle = "#ccc";
ctx.fillRect(150, 150, 100, 100);

// 마름모 사각형
ctx.rotate(45 * Math.PI / 180);      // 45도 회전
ctx.strokeRect(150, 150, 100, 100);  // 선으로 그리기
```



왜 엉뚱한 결과가 나왔을까?

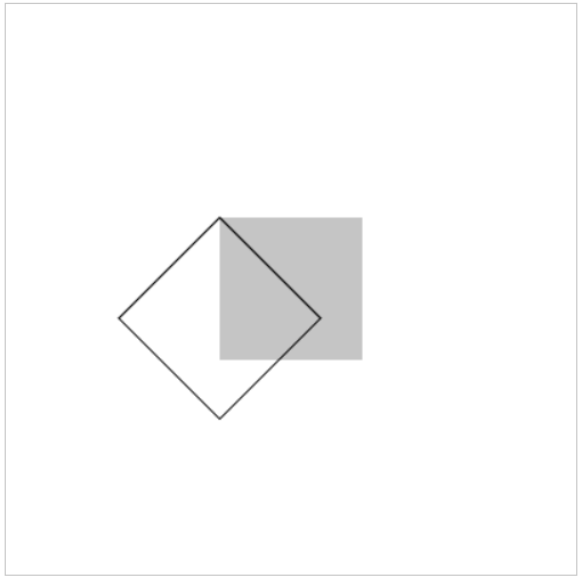
45° 회전시키면 현재의 원점, 즉 (0, 0)을 기준으로 회전시킨다.
이 상태에서 strokeRect(150, 150, 100, 100)으로 지정했기 때문에 위 화면처럼 표시된다.

그래픽 요소를 회전시킬 때 가장 먼저 할 일은?

→ 회전의 기준점이 되는 위치로 원점을 이동시키는 것

앞의 소스에, 원점을 이동하는 소스와 원래 상태로 되돌리는 소스를 추가합니다.

```
// 회색 사각형
ctx.fillStyle = "#ccc";
ctx.fillRect(150, 150, 100, 100);

// 마름모 사각형
ctx.translate(150, 150);          // 원점을 이동합니다.
ctx.rotate(45 * Math.PI / 180);   // 45도 회전합니다.
ctx.strokeRect(0, 0, 100, 100);
ctx.translate(-150, -150);        // 원점으로 복귀합니다.
```



크기 조절하기

scale() 메서드는 캔버스에서 도형이나 이미지를 크게, 또는 작게 표시할 수 있다.

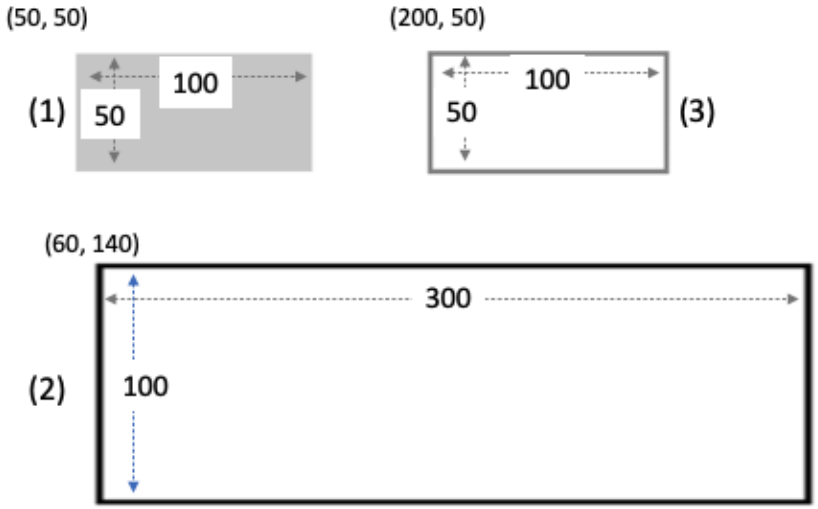
```
scale(x, y)
```

- 배율이 1보다 크면 확대되고 작으면 축소.
- 음수값을 사용하면 방향이 반대로 바뀐다.

<참고> 캔버스에서 크기 단위는 1픽셀.
scale(2, 2)로 지정해서 사각형을 그리면 이 사각형의 크기가 바뀌는 것이 아니라
크기를 나타내는 단위가 1픽셀에서 2픽셀로 바뀌면서 두 배로 표시된다.

[실습] 크기 조절하기

```
// 기본 사각형
ctx.fillStyle = "#ccc";
ctx.fillRect(50, 50, 100, 50);          // 1)
ctx.save();


// 확대한 사각형
ctx.scale(3, 2);
ctx.strokeRect(20, 70, 100, 50);)       //
2)


ctx.restore();
ctx.strokeRect(200, 50, 100, 50);       //
3)
```



그래픽 요소 합성하기

globalCompositeOperation 속성

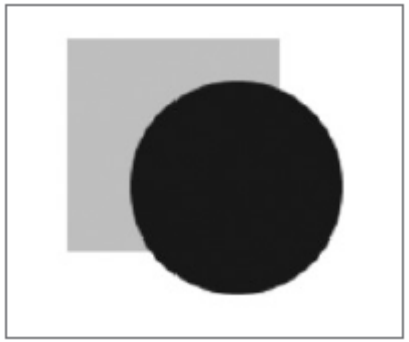
- 2개 이상의 캔버스 그래픽을 겹쳐 그리면 소스에서 나중에 그린 그래픽이 이전 그래픽 위에 그려진다.
- 둘 이상의 그래픽 요소가 겹쳐져 있으면 `globalCompositeOperation` 속성으로 그래픽 요소를 여러 형태로 합성해서 표시할 수 있다

```
globalCompositeOperation = type
```

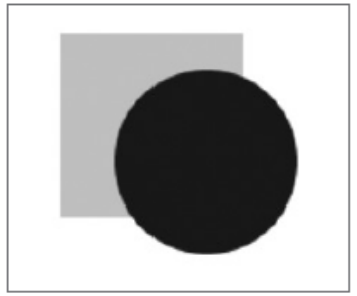
  - type에 사용할 수 있는 값은 아주 많다.
  - type 값 중에서 destination은 먼저 그린 도형을, source는 나중에 그린 도형을 가리킨다.

```
// destination
ctx.fillStyle = "#eee";
ctx.fillRect(100, 50, 100, 100);
ctx.strokeRect(100, 50, 100, 100);
ctx.globalCompositeOperation   =   "source-
over";
// source
ctx.fillStyle = "#222";
ctx.fillRect(130, 80, 100, 100);
```

부분을 바꿔 가면서 결과를 확인해 보세요

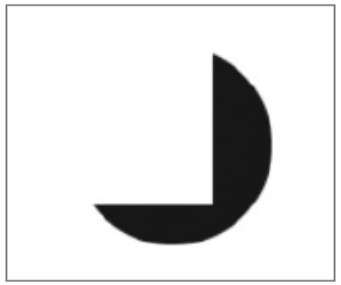


- source-over : source를 기준으로 destination 위에 그린다.



- source-in : source를 기준으로, destinatio과 겹쳐진 부분만 그린

- source-out : source를 기준으로 destinaion과 겹쳐지지 않는 부분만 그린다.



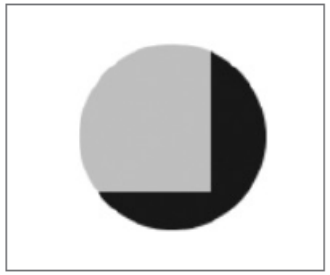
- source-atop : destination과 겹쳐진 부분을 그리고 destination은 불투명하게 처리한다

- destination-over : destination을 기준으로 source 위에 그린다.

- destination-in : destination 기준으로, source와 겹쳐진 부분만 그린다.

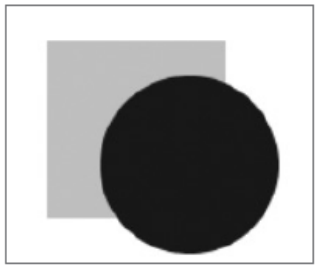
- destination-out : destination 기준으로 source와 겹쳐지지 않는 부분만 그린다.

- destination-atop : source 와 겹쳐진 부분을 그리고 source 부분은 불투명하게 처리한다.

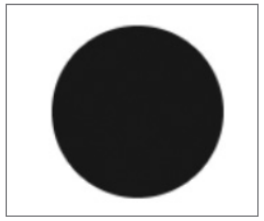


- lighter : 두 개의 그래픽을 모두 그리고, 겹쳐진 부분은 색상값을 합쳐서 결정한다.

- darken : 두 개의 그래픽을 모두 그리고, 겹쳐진 부분은 색상 차이로 결정한다.



- copy : 나중에 그린 그래픽만 표시한다.



- xor : 두 개의 그래픽을 모두 그리고, 겹쳐진 부분은 투명하게 처리한다