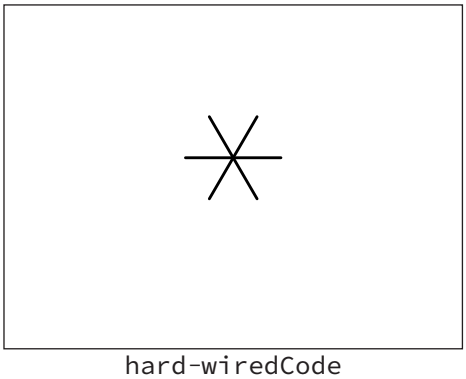
04
함수와 반복문

04.01
함수

프로그램의 여러 부분에서 비슷한 코드가 반복된다면 이를 함수로 묶어 캡슐화하는 것이 좋습니다. 가루약은 먹기 힘들지만 같은 가루약이라도 콘택600처럼 캡슐로 감싸면 보관하기도 복용하기도 쉬운 것과 마찬가지입니다.

함수없는 코드

```
void setup() {
    size(480, 360);
    smooth();
}

void draw() {
    strokeWeight(3);
    line(265, 203, 215, 117);
    line(215, 203, 265, 117);
    line(190, 160, 290, 160);
}
```



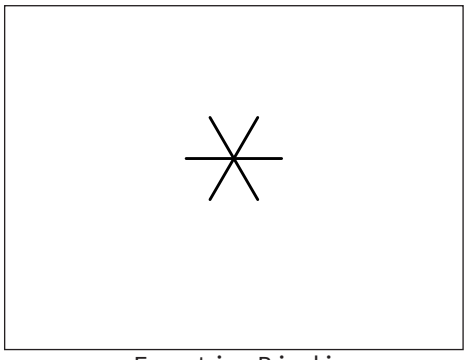
hard-wiredCode

별모양을 그릴 때마다 위의 코드처럼 숫자를 써넣은 것은 너무나 번거롭습니다. 그렇다고 기본도형이 지원하는 것도 아니니 차라리 우리가 사용할 함수를 우리가 정의하는 것이 좋겠습니다. 프로세싱은 사용자가 정의하는 함수를

지원합니다. 다만 컴퓨터가 이해할 수 있도록 특별한 문법 규칙을 따라야 합니다. 실제 예를 보면서 이야기를 계속해봅시다.

함수만들기 첫번째: 묶기

```
void setup() {
    size(480, 360);
    smooth();
}

void draw() {
    drawStar();
}

void drawStar() {
    strokeWeight(3);
    line(265, 203, 215, 117);
    line(215, 203, 265, 117);
    line(190, 160, 290, 160);
}
```



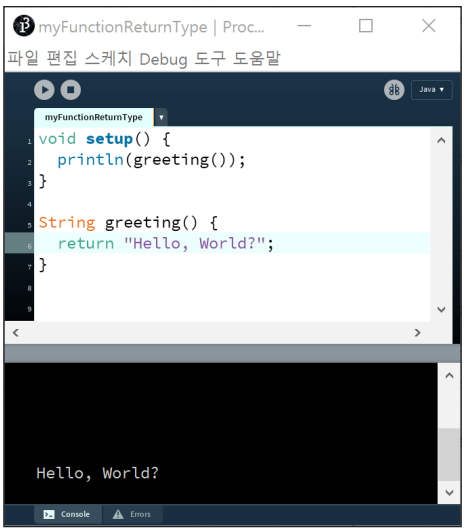
myFunctionBinding

draw() 함수 안에 있던 내용을 drawStar() 함수 안으로 이동시켰습니다. draw() 함수 안에서 대신 drawStar() 함수를 한 번 실행시킵니다.

drawStar() 앞에 void라는 말은 이 함수가 아무 것도 반환하지 않는다는 말입니다. 함수는 반환값을 가질 수 있는데 함수를 정의할 때는 반드시 반환값의 타입을 적어두어야 프로세싱 컴파일러가 불평을 하지 않습니다.

예를 들어 실행할 때마다 문자열을 반환하는 함수를 정의해 봅시다.

```
void setup() {
    println(greeting());
}

String greeting() {
    return "Hello, World?";
}
```



myFunctionReturnType

함수이름 greeting 앞에 문자열을 반환한다는 의미로 String을 붙입니다. 함수가 반환하는 값 앞에 return을 붙여 어떤 값을 반환하는지 명확하게 밝힙니다.

함수 만들기 두번째: 입력값 받기

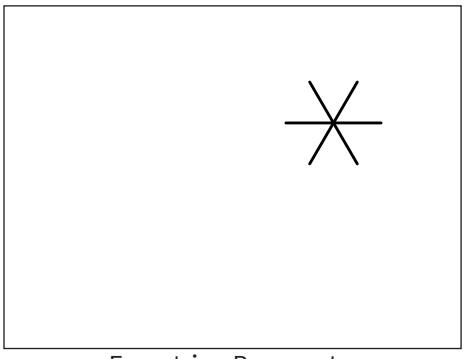
drawStar() 함수는 여전히 부족한 부분이 있습니다.

drawStar() 함수를 실행시키면 별은 언제나 화면 한가운데 그려집니다. 만약 다른 위치에 별을 그리고 싶으면 어떻게 해야 할까요? 기본도형처럼 입력값(혹은 매개변수라고도 불립니다)을 받을 수 있게 함수를 수정할 수 있습니다.

```
void setup() {
    size(480, 360);
    smooth();
}

void draw() {
    drawStar(width / 2, height / 2);
}

void drawStar(int posX, int posY) {
    strokeWeight(3);
    line(posX + 25, posY + 43, posX - 25,
posY - 43);
    line(posX - 25, posY  + 43, posX + 25,
posY - 43);
    line(posX - 50, posY, posX + 50,
posY);
}
```

이렇게 drawStar() 함수를 정의해두면 원하는 장소에 별을 찍을 수 있습니다.

```
void setup() {
    size(480, 360);
    smooth();
}

void draw() {
    background(255);
    drawStar(mouseX, mouseY);
}

void drawStar(int posX, int posY) {
    strokeWeight(3);
    line(posX + 25, posY + 43, posX - 25,
posY - 43);
    line(posX - 25, posY  + 43, posX + 25,
posY - 43);
    line(posX - 50, posY, posX + 50,
posY);
}
```



myFunctionParameters

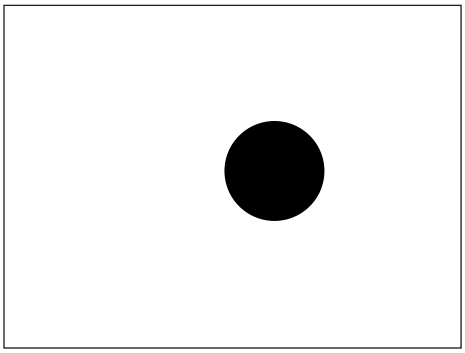
이벤트 핸들러: 특별한 함수

프로세싱은 이벤트 핸들러(Event Handler)라고 불리

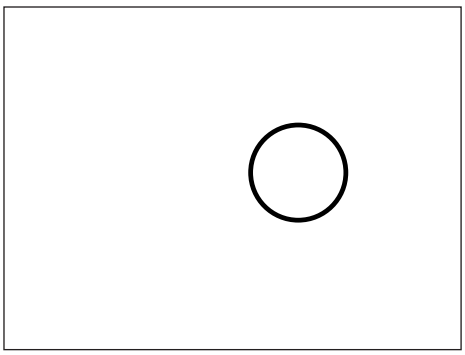
는 특별한 함수를 가지고 있습니다. 마우스나 키보드에 어떤 변화가 있을 때(어떤 이벤트가 생길 때) 특정한 동작을 할 수 있도록 지정할 수 있습니다.

예를 들어 마우스 버튼을 눌렀을 때 도형의 색을 바꾸려고 합니다. 마우스 버튼이 클릭되는 것이 이벤트이고 이 이벤트가 발생하면 mouseClicked() 함수 내부의 명령이 실행됩니다.

```
color toggle = 0;

void setup() {
    size(480, 360);
}

void draw() {
    background(255);
    strokeWeight(5);
    fill(toggle);
    ellipse(mouseX, mouseY, 100, 100);
}

void mouseClicked() {
    if (toggle == 0) {
          toggle = 255;
    }
    else {
          toggle = 0;
    }
}
```



myFunctionEventHandler01



myFunctionEventHandler02

mouseClicked() 함수 이외에도 mousePressed(), mouseMoved(), keyPressed(), keyReleased() 등의 이벤트 핸들러가 있습니다. 필요에 따라 적당한 함수를 사용하세요.

함수 레퍼런스

새로운 내용이 등장하면 아는 것이 늘어난다는 기쁨보다 모르는 것이 많았구나라는 자괴감이 더 커지기가 쉽습니다. 그렇게 생각하지 마세요. 모든 것을 다 알 필요도, 다 알 방법도 없습니다. 다만 필요한 만큼 알면 충분합니다.

어디서 필요한 내용을 찾을 수 있을까요?

가장 처음 가봄 직한 곳은 프로세싱 공식 사이트 레퍼런스(https://processing.org/reference/)입니

다. 프로세싱의 주요 변수와 함수에 대한 자세한 설명을 찾을 수 있는 곳입니다.

다른 사람의 프로세싱 작품을 감상하면서 영감을 얻을 수도 있습니다. OpenProcessing(http://www.openprocessing.org/)에 방문해 다른 사람들의 프로세싱 작품을 감상하고 코드를 분석해봅시다. 여러분의 작품도 사이트에 전시해봅시다. 특별한 작품만 사이트에 올릴 수 있는 것은 아닙니다. 둘어보면 뭐 그렇고 그런 프로그램도 많이 있으니 부담가지지 말고 자신의 작업을 자랑해 봅시다.

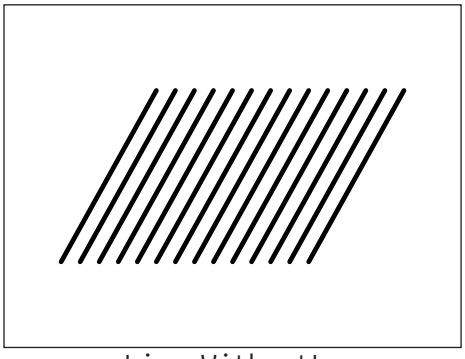
nature of code(http://natureofcode.com/)도 유명한 사이트입니다. 단순히 프로그래밍 언어로 프로세싱을 다루는데 그치지 않고 주변의 물리 현상을 시뮬레이션하는 과정을 차근차근 설명하고 있는 사이트입니다. 한국어로 번역되어 있으니 한 번 읽어 보세요. 한국어 번역도 꽤 잘 되어 있습니다.

얇은 책이 보고 싶다면 손에 잡히는 프로세싱(http://www.yes24.com/24/viewer/preview/4740458?PID=121879)을 권합니다. 구성이 좋고 부담없는 책입니다.

04.02
반복문

비슷한 도형을 여러 번 그려야 할 때가 있습니다. 함수로 묶을 수도 있지만 함수로 만들기에도 부담이 될 정도로 여러번 그려야 한다면 어떻게 해야 할까요?

```
void setup() {
    size(480, 360);
    smooth();
}
void draw() {
    strokeWeight(5);
    background(255);
    line(160, 90, 60, 270);
    line(180, 90, 80, 270);
    line(200, 90, 100, 270);
    line(220, 90, 120, 270);
    line(240, 90, 140, 270);
    line(260, 90, 160, 270);
    line(280, 90, 180, 270);
    line(300, 90, 200, 270);
    line(320, 90, 220, 270);
    line(340, 90, 240, 270);
    line(360, 90, 260, 270);
    line(380, 90, 280, 270);
```



myLinesWithoutLoop

```
    line(400, 90, 300, 270);
    line(420, 90, 320, 270);
}
```

손으로 일일이 코드를 찍고 있으니 몸 속에서 사리가 생기는 기분입니다. 이것은 좋은 방법이 아니라는 것을 알겠습니다. 더 좋은 방법이 없을까요?

반복문이 여러분의 고민을 해결해 줄 수 있습니다. 반복문은 프로그램 내부에서 특정 명령을 여러 번 실행할 때 사용합니다.

두가지 반복문이 있습니다. 하나씩 차례로 살펴봅시다.

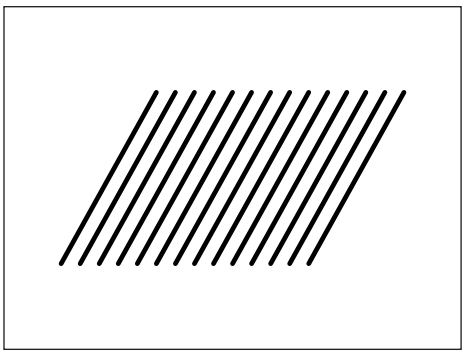
for 반복문(for loop)

for 반복문은 반복하는 횟수를 알고 있을 때나 반복문 내부에서 카운터를 활용하는 경우에 주로 사용합니다.

위의 손코딩을 for 반복문으로 고쳐봅시다.

for 반복문을 이용하려고 할 때 가장 먼저 해야 할 일은 반복되는 규칙을 파악하는 일입니다. 우선 14개의 선을 그어야 합니다. 그리고 마지막에 그린 성을 기준으로 오른쪽으로 평행하게 20픽셀 이동시켜 다음 선을 그어야 합니다. 이제 이 조건에 맞게 for 반복문을 만들어 봅시다.

```
void setup() {
    size(480, 360);
    smooth();
}

void draw() {
    background(255);
    strokeWeight(5);
    for (int i = 0; i < 14; ++i) {
       line(160 + i * 20, 90, 60 + i * 20,
270);
    }
}
```



myForLoop

for 반복문이 잘 동작하는 것을 확인했습니다. 이제 대체 무슨 일이 일어났는지 하나씩 따져봅시다.

for 반복문 실행 순서

for 반복문은 아래와 같은 기본 구조를 가지고 있습니다.

```
for (초기화; 조건식; 갱신) {
구문
...
}
```

초기화

반복문이 실행되면 가장 먼저 초기화(initialization)가 실행됩니다. 위의 예제에서는 반복문 안에서만 존재하는 int 타입 변수 i가 생성되고 0으로 초기화됩니다.

```
i의 값: 0
```

조건식

다음으로는 조건식이 실행됩니다. i < 14 를 만족하는지 여부를 테스트합니다. 방금 초기화를 마친 변수 i의 값은 0이므로 i < 14 조건식은 참(true)를 반환합니다.

```
i의 값: 0
조건식: 참
```

구문

조건식이 참(true)이면 for 반복문의 {...} 사이 구문이 실행됩니다. 구문에 사용된 변수 i의 변수값은 0입니다.

```
line(160 + 20 * 0, 90, 60 + 20 * 0, 270);
```

구문이 실행되어 0번째 선을 그립니다.

```
i의 값: 0
```

갱신

구문이 실행되고 난 다음 초기화 과정에서 선언한 변수를 업데이트 합니다. 단항연산자 ++i는 i의 값에 1을 더합니다.

```
i의 값: 1
```

갱신이 되고 난 다음 이제 반복문은 조건식으로 돌아갑니다. i값이 14보다 작으니 구문이 실행되고 다시 갱신이

일어나고 다시 조건식으로 다시 다시... 이렇게 반복됩니다. 반복이 될 때마다 i번째 선이 화면에 그려집니다. 이 반복문은 언제 종료될까요?

반복문의 종료

13번째 선(0부터 시작해서)을 그리고 i가 다시 갱신이 됩니다. i는 14가 되고 다시 조건식을 확인받게 됩니다. 그런데 14 〈 14는 false입니다. 이제 for 반복문을 벗어나 반복문 다음 구문으로 진행됩니다.

while 반복문

while 반복문은 초기화와 갱신부분이 없고 단지 조건식을 테스트 하는 부분만 있습니다. while 반복문의 구조는 아래와 같습니다. 조건문이 true일 때 실행할 구문이 {...} 블록 사이에 들어갑니다.

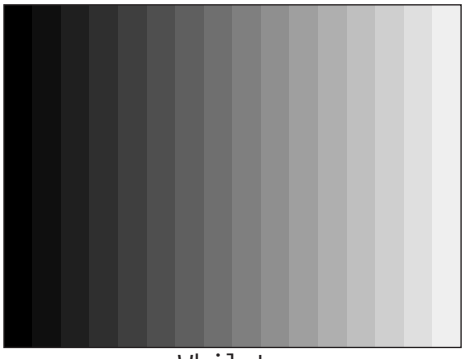
```
while (조건문) {
구문
...
}
```

초기화와 갱신부분이 따로 없기 때문에 while 반복문 내부에서 사용할 변수를 미리 선언해야 하고 변수의 갱신도 {...} 블록 안에서 사용자가 선언해주어야 합니다.

while 반복문 예

화면에 사각형을 가득 채우겠습니다. 사각형을 구분하기 위해 사각형의 위치에 따라 채우기를 다르게 하겠습니다.

```
void setup() {
    size(480, 360);
    smooth();
}

void draw() {
    int i = 0;
    float c;

    while (i 〈 width) {
           c = map(i, 0, width, 0, 255);
           fill(i);
           noStroke();
           rect(i, 0, 30, height);
           i = i + 30;
    }
}
```



myWhileLoop

앞서 설명한 while 반복문과 for 반복문과의 차이점을 확인하셨나요? while 반복문의 경우 카운터로 사용할 변수 i를 while 반복문 전에 선언하고 갱신도 {...} 블록 안에서 따로 해주었습니다.

while 반복문을 for 반복문으로 고치기

위의 while 반복문을 for 반복문으로 고쳐봅시다. 반복문 부분만 적어보겠습니다.

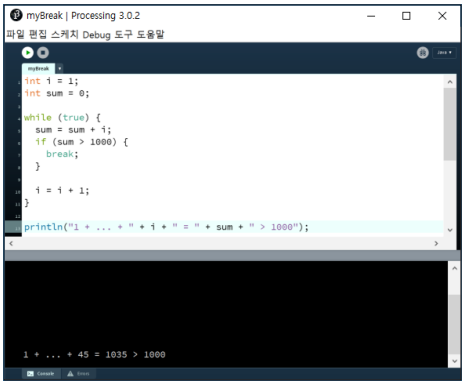
```
float c;

for (int i = 0; i < width ; i = i + 30) {
    c = map(i, 0, width, 0, 255);
    fill(c);
    rect(i, 30, 30, 30);
}
```

break

break는 보통 조건문과 함께 사용됩니다. 조건을 만족하면 break가 실행되고 이런 경우 가장 가까운 반복문을 벗어납니다.

예를 들어 1부터 숫자를 더해간다고 합시다. 1 + 2 + 3 + 4 + ... 1에서 얼마까지 더해야 1000보다 큰 수가 될까요? 이런 문제를 풀 때 break를 사용하면 편합니다.

```
int i = 1;
int sum = 0;

while (true) {
    sum = sum + i;
    if (sum > 1000) {
          break;
    }

    i = i + 1;
}

println("1 + ... + " + i + " = " + sum " > 1000");
```



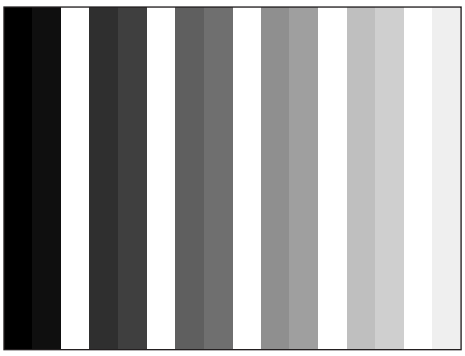
myBreak

continue

continue는 반복문 안에서만 사용되고 보통 조건문과 함께 사용됩니다. continue가 실행되면 continue 다음에 위치한 구문은 생략되고 for 반복문인 경우는 갱신으로, while 반복문인 경우는 조건식으로 이동합니다.

예를 들어 도형을 그릴 때 세번째 도형을 생략하고 싶습니다. 어떻게 해야 할까요?

```
void setup() {
 size(480, 360);
 smooth();
 noStroke();
 background(255);
}

void draw() {
    float c;

    for (int i = 0; i < width; i = i + 30) {
    if(60 == i % 90) {
          continue;
    }

    c = map(i, 0, width, 0, 255);
    fill(c);
    rect(i, 0, 30, height);
    }
}
```



myContinue

이제 반복문이 전혀 생소하지 않습니다.

과제

반복문을 이용해 오른쪽 그림을 그려보세요. 프로그램을 구상하고 빈 칸을 채워보세요.

이번 과제는 투명한 색을 도형에 사용합니다. 이에 대한 설명을 조금 덧붙입니다.

`fill()` 함수는 도형의 내부를 색으로 채울 때 사용하는 명령입니다. 지금까지는 `fill()` 함수에 값을 하나만 전달하는 것을 보았을 것입니다. 이제 값을 두 개 전달할 것입니다. 이렇게 하면 도형내부의 색 뿐만 아니라 투명도도 정할 수 있습니다.

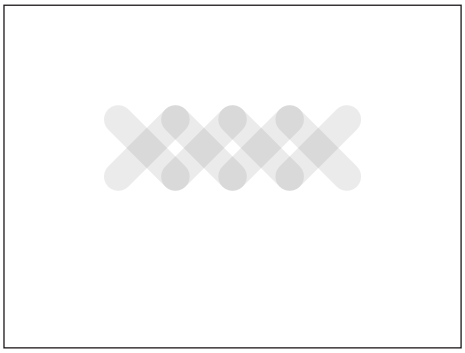
예를 들어 `fill(0, 128)`은 검은색으로 투명도가 절반정도인 도형을 그리는 방법입니다. 투명한 도형이 서로 겹치면 아래 도형의 색이 위로 비치고 색이 진해집니다.

noLoop() 함수

`noLoop()` 함수도 처음 등장했습니다. `noLoop()` 함수는 `draw()` 함수가 반복해서 실행되는 것을 막는 함수입니다. 투명값을 사용하는 경우 `noLoop()`를 사용할 필

요가 있는지 생각할 필요가 있습니다. 반복해서 같은 도형이 그려져 도형이 겹치게 되면 색이 점점 진해져 투명도를 설정한 이유가 없어지기 때문입니다.

```
void setup() {
    size(480, 360);
    smooth();
    background(255);
    strokeWeight(30);
    noLoop();
}

void draw() {
    for (int ___ ; i < ___ ; ___) {
          line(120 + ___, 120, 120 +
___, 120 + 60);
          line(120 + ___, 120 + 60, ___,
120);
    }
}
```



myLadder