

“Tytułem wstępu”

Processing to język programowania podobny do Javy, ale znacznie prostszy, wyposażony jednak w biblioteki graficzne, dźwiękowe, filmowe etc. oraz bardzo proste środowisko programowania. Wszystko to stworzono z myślą o artystach, jednak przydać się może też innym użytkownikom, którzy nie chcą zostać profesjonalnymi programistami, ale w ich pracy lub hobby programowanie może być użyteczne.

PRZYDA SIĘ TEŻ OSOBOM CHCĄCYM W NIESTRESUJĄCY SPOSÓB NAUCZYĆ SIĘ PODSTAW PROGRAMOWANIA

Tekst powstał na podstawie bloga “[Processing w edukacji](#)”, który [działa w powiązaniu ze stroną na Facebooku o tym samym tytule](#).

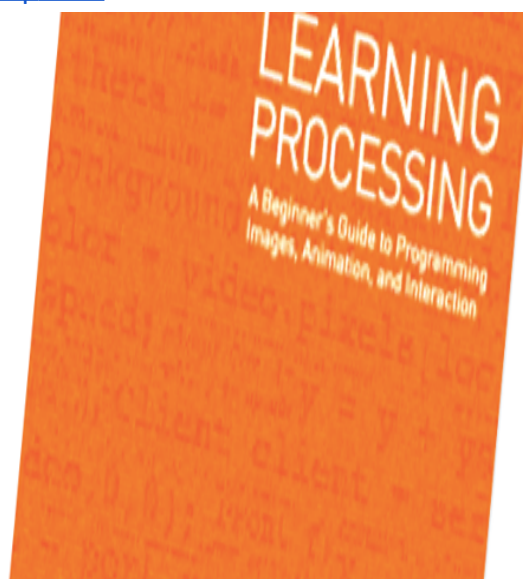
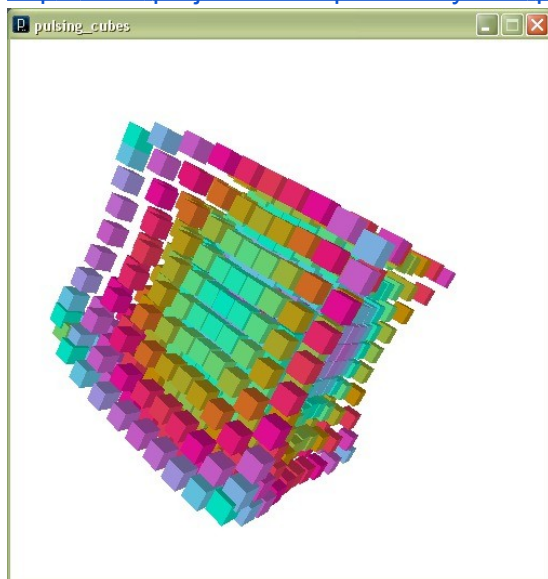
Poniżej lista ważnych linków:

- <https://processing.org>
- <https://www.facebook.com/page.processing/>
- <https://processing.org/download/>
- <http://www.codingforart.com/category/technical/>
- <https://web.facebook.com/ProcessingWEdukacji/>

Cała książka jest zarówno kursem Processingu, jak też podstaw symulacji komputerowych, ale łatwo znajdziecie inne proste kursy Processingu, nastawione bardziej na sztukę, lub nawet obróbkę obrazu.

Na przykład ten krótki kurs dla niecierpliwych:

<http://www.projektowanieparametryczne.pl/?p=137>



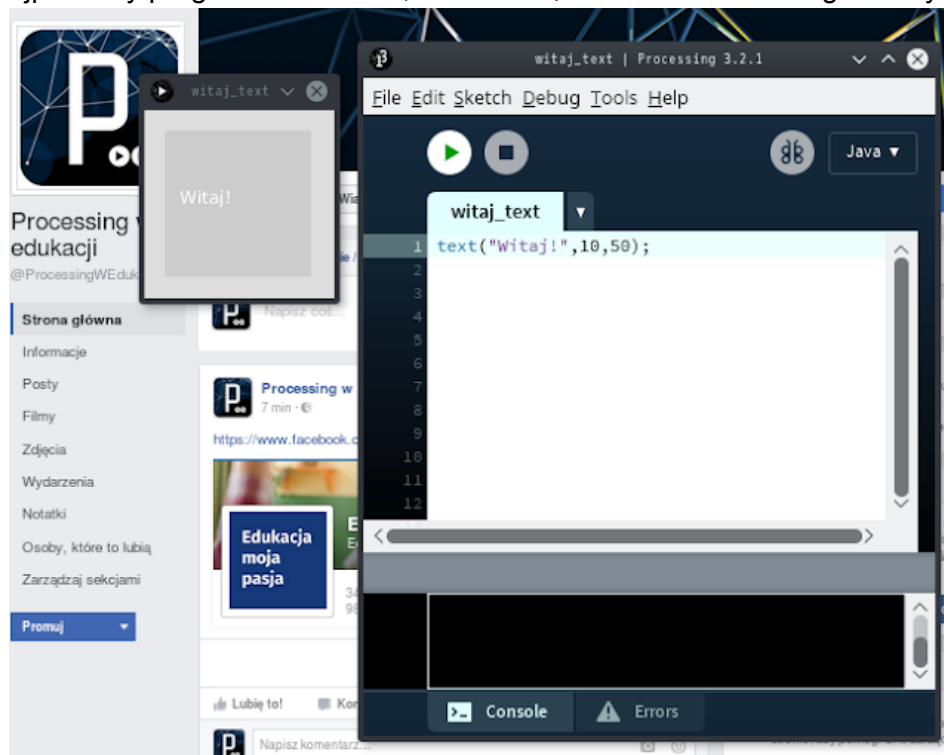
Albo całkiem rozbudowany podręcznik LEARNING PROCESSING 2ND EDITION:

<http://learningprocessing.com/>

Sam w sobie ciekawy, ale jeszcze ciekawsze są przykłady do niego
<http://learningprocessing.com/examples/> , które świetnie obywają się bez podręcznika :-)
Autor **Holger Lippman** to artysta z [Wandlitz](#) tworzący w Processingu. Można z nim
zawrzeć bliższą znajomość odwiedzając stronę:
<https://www.facebook.com/holger.lippmann>

Hello World!

Chyba najprostszy program na świecie, zwłaszcza, że od razu w oknie graficznym:



Jak widać niewiele trzeba, żeby uruchomić program w Processingu. Wystarczy JEDNA linia kodu i już mamy okno z efektem jej wykonania.

Tak prosto to było tylko kiedyś w BASICU na ZX Spectrum :-D

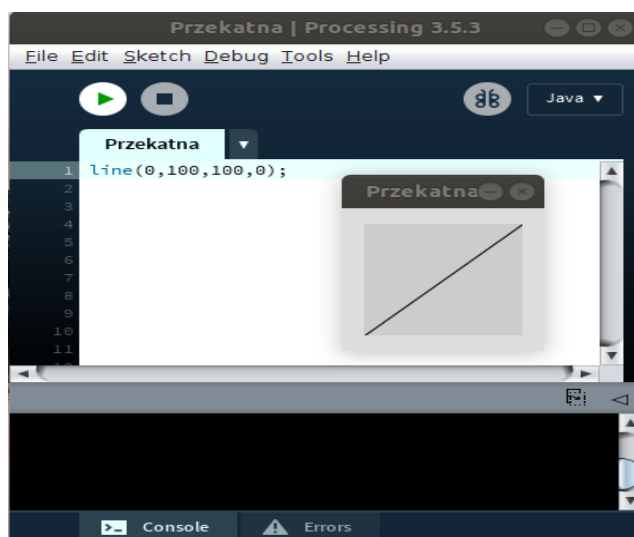
W tym wypadku używamy “instrukcji” wypisania na ekran zadanego tekstu (`text()`) zaczynając od punktu ekranu o współrzędnych $x=10$, $y=50$.

Dla uproszczenia umówimy się na razie, że takie słowa jak `text`, które środowisko Processingu wyświetla na niebiesko będziemy nazywać “*komendami*”.

Większość “komend” ma jakieś parametry, które podajemy w nawiasach. Tutaj jest to tekst do wyświetlenia oraz współrzędne punktu w oknie od którego należy rozpocząć wypisywanie tekstu.

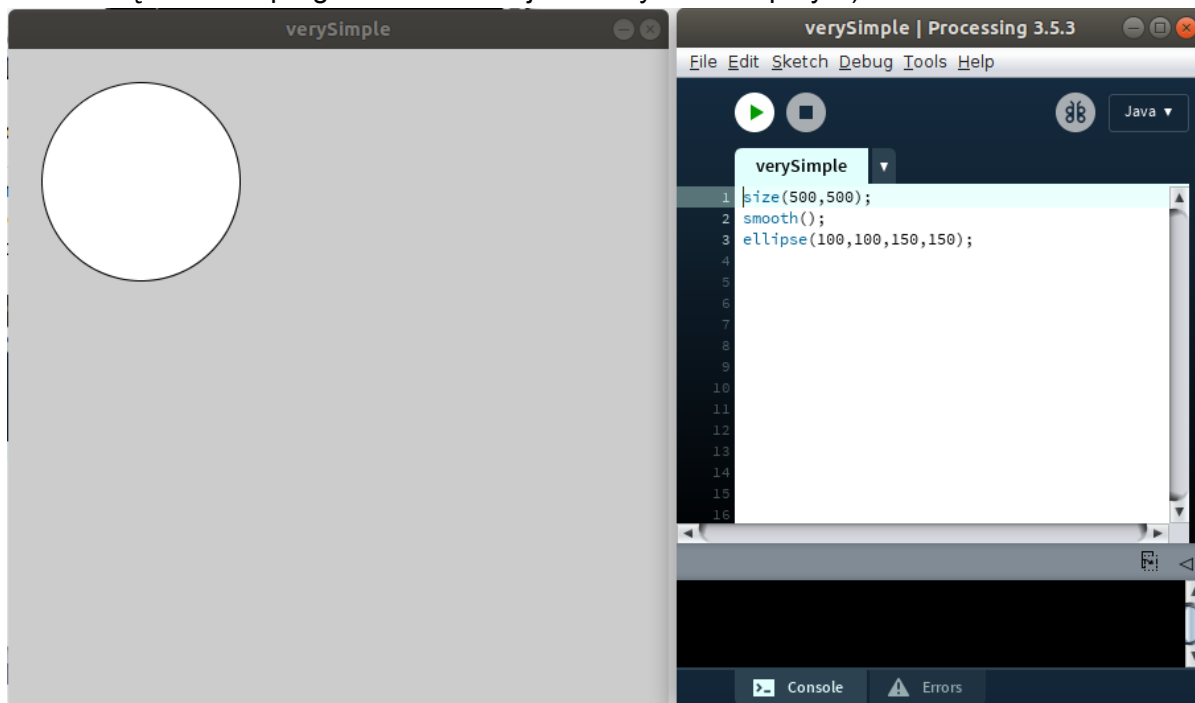
Kolejność elementów podanych w nawiasie jest zawsze istotna. Ważna jest też wielkość liter, chyba że mamy do czynienia z “*literalem*” tekstowym, czyli tekstem zamkniętym w cudzysłów. Tam możemy sobie pisać co nam się żywnie podoba, choć zdarzają się ciągi znaków, które mają znaczenie specjalne. Np. `\n`

Poniżej jeszcze kilka prostych ćwiczeń. Np. linia będąca przekątną “obszaru roboczego” okna:

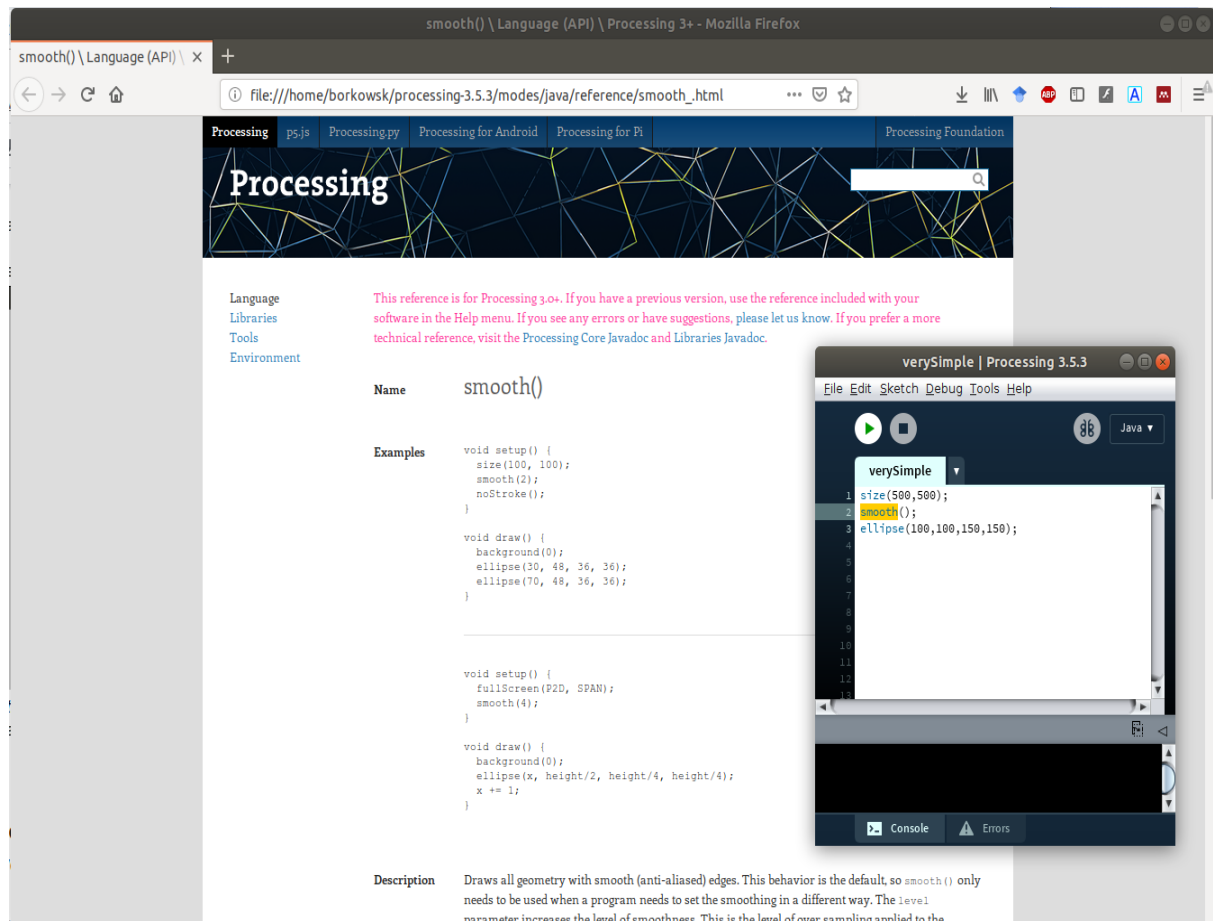


Okno domyślnie ma rozmiar 100 na 100 [pikseli](#) (jeśli przypadkiem nie wiecie co to “pixel” to tu odeślę was do Wikipedii), a liczby to współrzędne początku i końca linii.

Teraz trzy instrukcje “umiejące” w nieco większym oknie (`size(500,500)`) namalować elipsę. Oczywiście wykonają się jedna po drugiej. Ważne żeby `size` było pierwszą komendą w kodzie programiku - inaczej możemy mieć kłopoty :-)



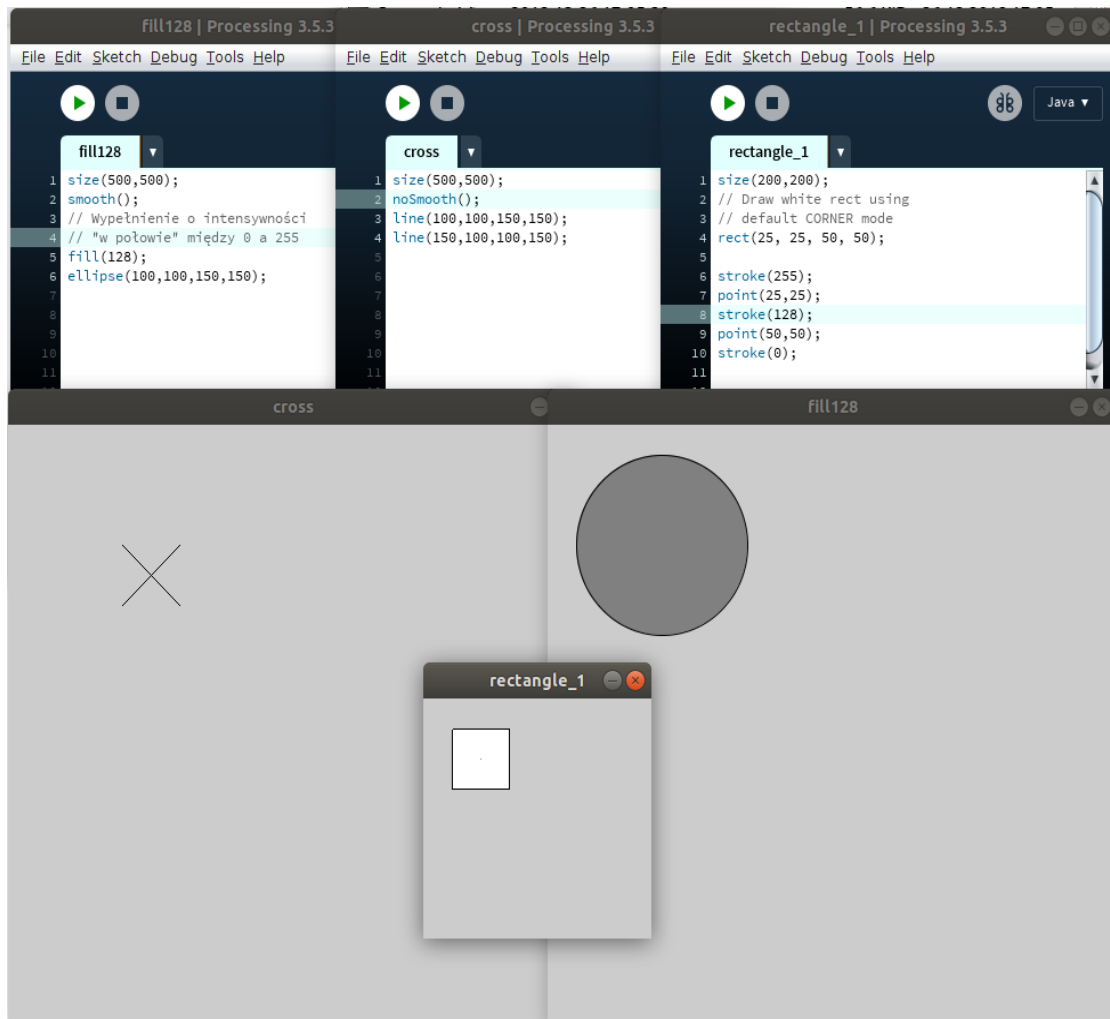
Tylko po co to `smooth()` ? A to łatwo sprawdzić. W podręcznym menu wyświetlanym przez edytor dla tego słowa widnieje opcja “Find in reference” otwierająca w domyślnej przeglądarce internetowej stronę z opisem tej “komendy” (w rzeczywistości to wywołanie funkcji bibliotecznej, ale nie komplikujmy przedwcześnie ;-)



I jeszcze ze trzy ćwiczenia “w ciemno”...

Powinniście trochę poeksperymentować z liczbami, co was może intuicyjnie naprowadzi

na coś co was za chwilę może nieco zaskoczyć.



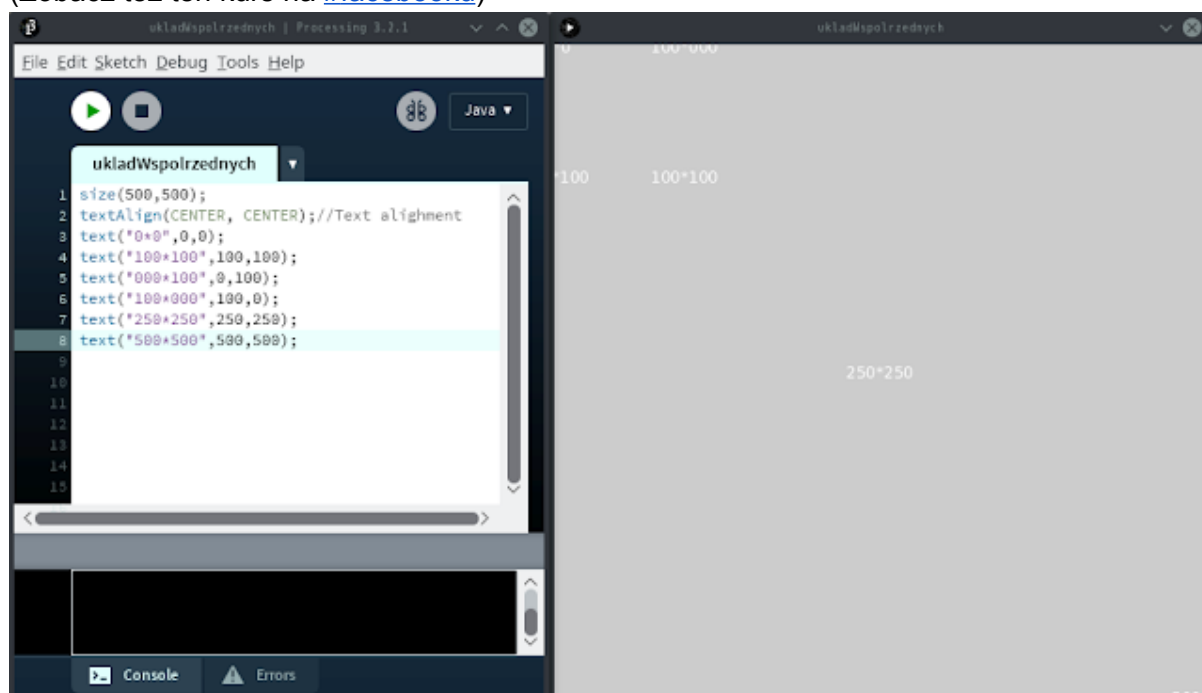
#text #size #smooth #noSmooth #fill #stroke #point #line #rect #ellipse

Układ współrzędnych ekranowych (2D - w wersji dwuwymiarowej)

Komputerowy układ współrzędnych nie różni się w Processingu od typowych rozwiązań grafiki 2D (dwuwymiarowej), ale różni się od układu przyjętego w matematyce.

Tak nawiasem, mam nadzieję że geometria analityczna nie była waszą najbardziej znienawidzoną dziedziną matematyki w szkole...

Poniższy program pomaga w przyswojeniu tej konwencji.
(Zobacz też ten kurs na [#facebooku](#))



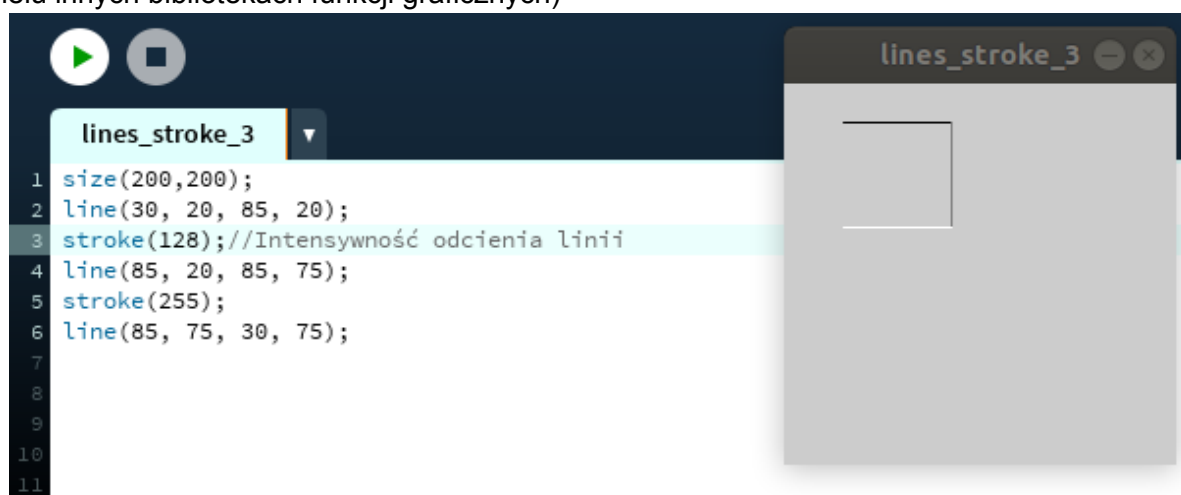
#textAlign

Taki dziwny układ współrzędnych wynika z przyczyn historycznych - pierwsza komputerowa grafika 2D powstawała przez sterowanie wiązką elektronów na ekranie [CRT \(kineskop\)](#). Ta wiązka omiata ekran od góry do dołu, z lewej do prawej. A to z kolei wynika po prostu z układu tekstu w europejskich książkach, który stosujemy już od czasów starożytnej Grecji, czyli jakieś 2500 lat!

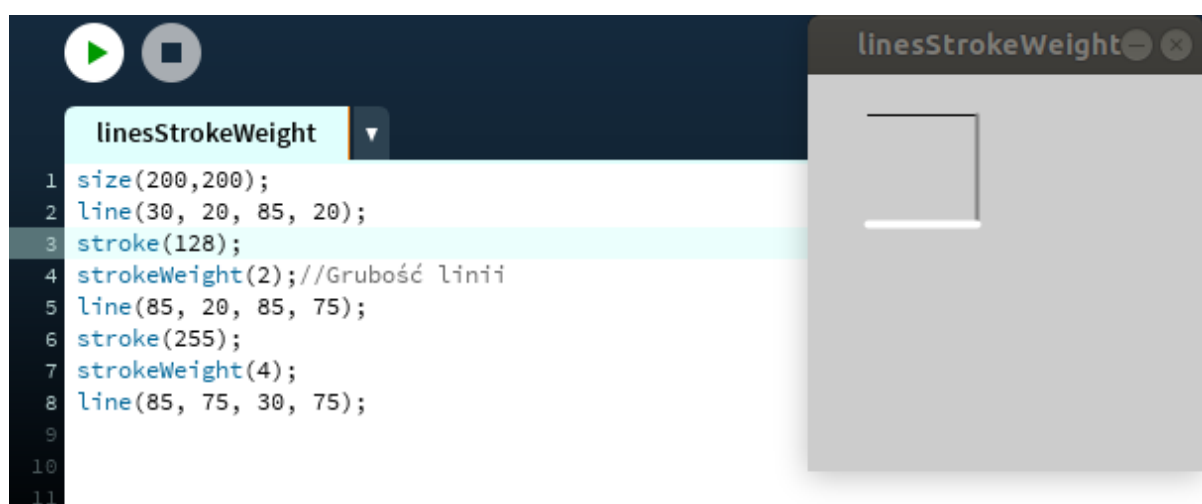
Teraz trochę przećwiczymy użycie współrzędnych (oczywiście jak najbardziej zachęcam do własnych kombinacji liczb, jak już się uda uruchomić przykład w wersji podstawowej).

Sprawdzajcie zawsze w dokumentacji znaczenie parametrów używanych funkcji!

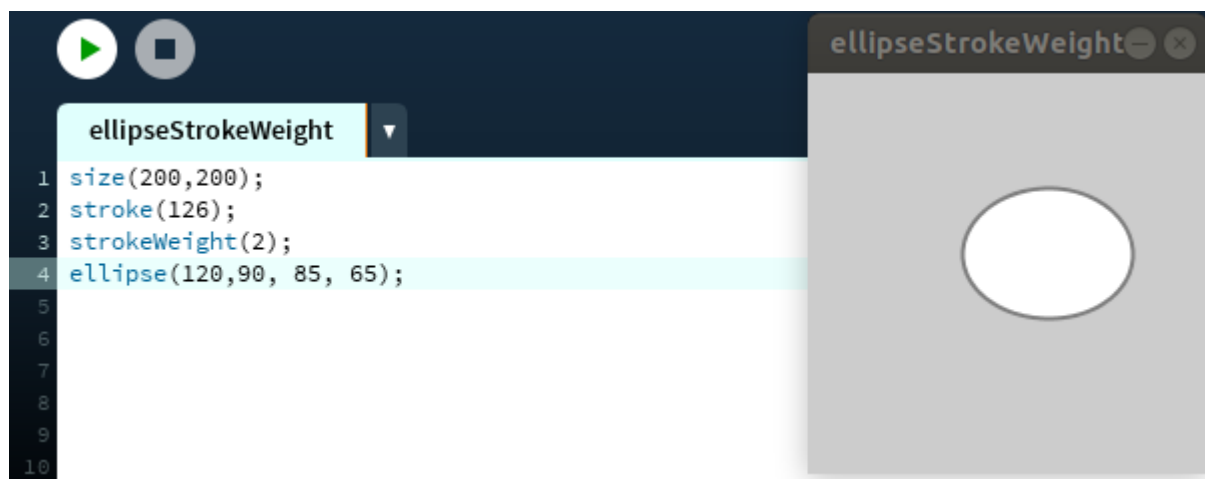
Np. dla `line(...)` w Processingu pierwszy parametr to **x1**, drugi **y1**, trzeci **x2** i czwarty **y2**, ale dla `ellipse(...)` pierwszy parametr to **x** środka, drugi to **y** środka, trzeci to wysokość elipsy, a czwarty to jej szerokość (a nie półosie, jakby się można spodziewać, i jak jest w wielu innych bibliotekach funkcji graficznych)



Dzięki użyciu trzech różnych odcieni linii rysujemy coś co od biedy może uchodzić za przycisk. Efekt będzie jeszcze lepszy gdy linie będą grubsze:

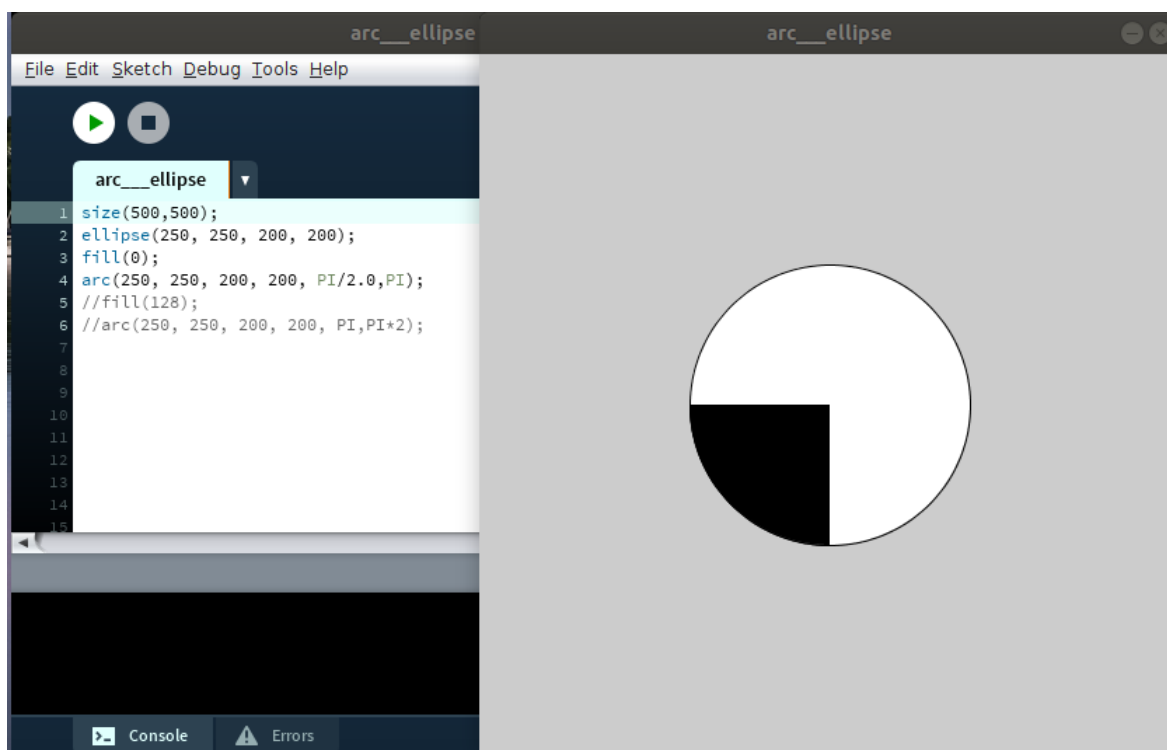


Tych samych “instrukcji” do określania odcienia i grubości linii możemy użyć do innych kształtów. Np. do elipsy:



Została nam jeszcze jedna kwestia dotycząca układu współrzędnych. Niestety zahacza ona o inną dziedzinę geometrii, do której możecie mieć uraz - trygonometrię.

Czasem w instrukcji rysowania musimy podać KĄT. Np. żeby narysować łuk albo oparty na łuku segment elipsy:



W językach programowania kąty prawie zawsze podaje się w RADIANACH. To dziwna miara bo kąt pełny jest przy jej użyciu liczbą niewymierną! Jest to bowiem 2 razy π ... Na szczęście komputery potrafią sobie same wyliczyć liczbę π z wystarczającą dla nich dokładnością (w Processingu wystarczy napisać π), a my musimy tylko pamiętać, że kąt półpełny to całe π , kąt prosty to pół π ($\pi / 2$), kąt 45 stopni to ćwierć π ($\pi / 4$) a potem sobie dodawać.

Przećwiczcie to na powyższym programie zmieniając parametry kąta startowego i końcowego.

#strokeWeight #arc

ZMIENNA!

Co prawda można sobie wyobrazić programowanie bez zapamiętywania różnych wartości, ale to byłoby trudne. Zazwyczaj potrzebujemy takich arbitralnie NAZWANYCH “skrytek”, do których wkładamy wartości, a potem używamy ich w miarę potrzeb, wydobywając je dzięki tej nazwie. Te “skrytki” to właśnie ZMIENNE.

W Processingu każda zmienna poza nazwą ma TYP. Np. poniższa zmienna o nazwie “a” jest typu “int” czyli może przyjąć jedynie wartość będącą LICZBĄ CAŁKOWITĄ.

Taka instrukcja jak poniżej nazywana jest INICJALIZACJĄ ZMIENNEJ.

```
int a=20;
```

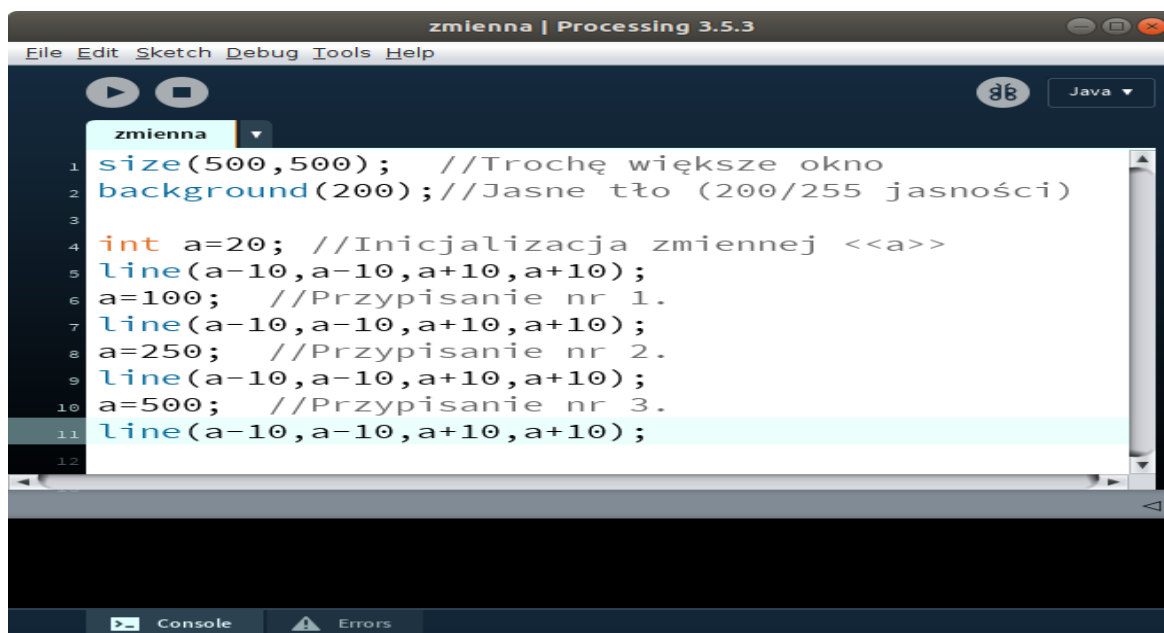
Po inicjalizacji można już zmienną używać w “działaniach”, które w programowaniu nazywane są WYRAŻENIAMI. W poniższej linii kodu tych wyrażeń jest cztery, ale każde potrzebuje wartości zmiennej “a”.

```
        // Przypominamy:  
line(a-10,a-10,a+10,a+10); //Pierwszy parametr to x1, drugi y1,  
        //trzeci x2 i czwarty y2
```

Zmienną można użyć ponownie nadając jej inną wartość. Taką instrukcję nazywamy PRZYPISANIEM.

```
a=100;
```

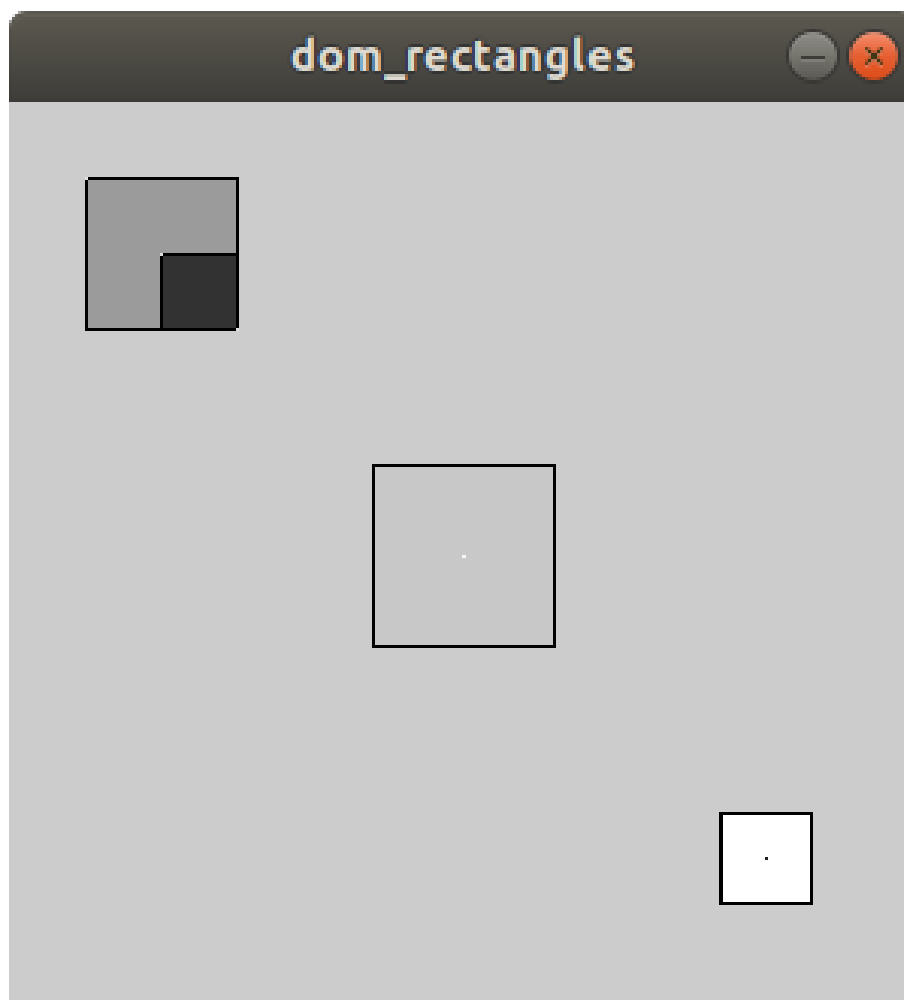
Po przypisaniu w skrypcie jest inna wartość i powtórzenie obliczenia wyrażenia da inny



rezultat. Jaki? A to już sprawdźcie sami. A może potraficie zgadnąć bez sprawdzania?

Zadanie domowe

Za pomocą samego przepisywania cudzego kodu, nawet z modyfikacjami jeszcze nikt się programowania nie nauczył. Trzeba rozwiązywać też zadania od początku do końca. To pierwsze zadanie dla was - uzyskanie takiego mniej więcej efektu jak na rysunku poniżej. Można użyć zmiennych, to pewnie ułatwi sprawę, ale można też nie użyć. Dokładne rozmiary okna też nie są istotne. Ważne są proporcje i kontrasty, czyli różnice odcieni.



Przyda się użyte przed chwilą `#background` oraz kilka innych wcześniej wprowadzonych “komend” czyli funkcjonalności Processingu.

Przykłady prostych pętli

Pętla – czyli wielokrotne powtarzanie jakiejś czynności to coś, co jest *de facto* sednem programowania. To w cierpliwym, wielokrotnym i szybkim powtarzaniu obliczeń komputery są od nas faktycznie dużo lepsze.

Pętli zwykle wyróżniamy kilka rodzajów. Najprostsza wykonuje się dopóki nie spełni się jakiś warunek, np.:

“Piecz aż ciasto będzie suche w środku (sprawdź nakłuwając patyczkiem)”

Inna wykonuje się właśnie dopóki warunek jest spełniony, np.:

“Jeśli warzywa są jeszcze twarde dalej gotuj”

Ale najczęściej używane w programowaniu są pętle, w których liczba powtórzeń jest od początku ściśle określona, np.:

“Odkrój od schabu 5 plasterków”

W grafice komputerowej da się znaleźć niezliczoną liczbę przykładów użycia takiej właśnie pętli. Dwóch pierwszych typów też kiedyś użyjemy, ale nie dziś...

```
for(int i=0;i<256;i++) //POWTARZAJ 256 razy
    line(i*2,i*2,0,500); //Ta instrukcja się powtarza
// Wyszliśmy z pętli
...
```

To odliczania kolejnych nawrotów pętli służy zmienna *i*, której wartość inicjujemy, w tym wypadku, na 0, a potem powiększamy o 1 w każdym nawrocie pętli (*i++*) dopóki jest mniejsza od 256. Dla każdego nawrotu wykonujemy instrukcję, w razie potrzeby korzystając ze zmiennej sterującej, żeby zróżnicować efekty wykonania każdego nawrotu (analogicznie jak w przykładzie ze zmienną ‘a’ w poprzednim rozdziale).

Gdy ‘a’ osiągnie tę wartość, warunek wejścia do wnętrza pętli (*i<256*) przestaje być spełniony i program z pętli wychodzi. Możemy wykonać dalsze instrukcję.

Oczywiście jest kłopot gdy pomylimy warunki lub instrukcje zmiany i pętla nie skończy się nigdy. Np. gdyby zamiast instrukcji powiększania i o 1 (*i++*) daliśmy instrukcję POMNIEJSZANIA i o 1 (*i--*). Wtedy warunek *i<256* spełniony byłby “niemal zawsze” – liczba zapisana w zmiennej ‘i’ malała by “aż do komputerowej minus nieskończoności”... I może nawet byśmy się nie doczekali końca programu, choć ze względu na specyficzne właściwości arytmetyki komputerowej kiedyś by się ten program jednak skończył.

Zobaczmy teraz cały program:

```
line_every_i ▼
1 size(500,500);
2 noSmooth();//Bez wygładzania linii ("antyaliasingu")
3 for(int i=0;i<100;i++) //POWTARZAJ
4   line(i,i,0,500);
```

Proste? Ale jakbyśmy chcieli nadać inną INTENSYWNOŚĆ każdej z linii?

A to już nie takie proste :-). Musimy jakoś przekazać informacje, że w pętli jest więcej niż jedna linia kodu czyli tzw. *blok kodu*. Służą do tego "klamerki" - { }

```
line_every_i_stroke ▼
1 size(500,500);
2 noSmooth();//Bez wygładzania linii ("antyaliasingu")
3 for(int i=0;i<256;i++) //POWTARZAJ
4 {
5   stroke(i);//Zmienia się jasność
6   line(i,0,128,500);//zmienia się tylko x, y=0
7 }
```

Nie pokażę tu efektu, ale gwarantuję, że wam się spodoba :-)

Manipulując liczbami w tym programie można uzyskiwać różne figury. Np. prostokąt zamiast trójkąta.

Ale teraz zrobimy coś więcej. Znamy już instrukcję/operację przypisania do zmiennej, oznaczaną znakiem = . Potrafimy powiększyć zmienną o 1 (operacja ++). Jest też specjalna instrukcja do powiększania zmiennej o dowolną liczbę całkowitą:

+=

Czyli pisząc: `i+=2` uzyskujemy skoki zmiennej 'i' co dwa, np. 0,2,4,6,8 ... etc.

Jeśli napiszemy `i+=10` to będzie to skok o 10: 0,10,20,30 ... etc.

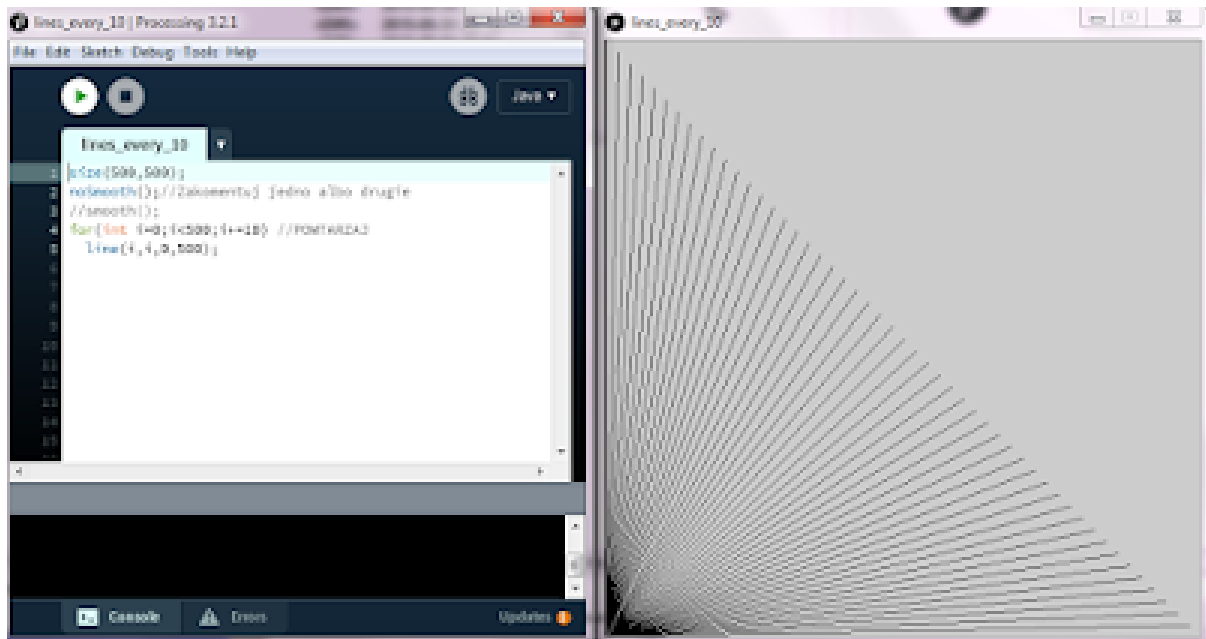
Ale jak napiszemy `i+=1` to będzie to inny sposób zapisu powiększania o jeden czyli `i++`. Spróbujmy na razie narysować linie, których punkty końcowe są oddalone o 2 piksele:

```
line_every_2 ▼
1 size(500,500);
2 smooth();//Z wygładzaniem linii ("antyaliasingiem")
3 for(int i=0;i<200;i+=2) //POWTARZAJ CO DRUGI!
4   line(i,i,0,500);
```

Takie proste programy demonstrujące działanie pętli “for” bardzo przypominają moje pierwsze w życiu programy w języku [BASIC](#) na komputerze [ZX Spectrum](#) ([Sinclair BASIC](#)). Miałem wtedy 16 lat, a komputer kolega pożyczał na weekendy z LX Liceum Eksperymentalnego w Warszawie (<http://liceum.gorski.edu.pl/o-szkole/historia/>). To chyba wtedy zaraziłem się bakcylem programowania :-)

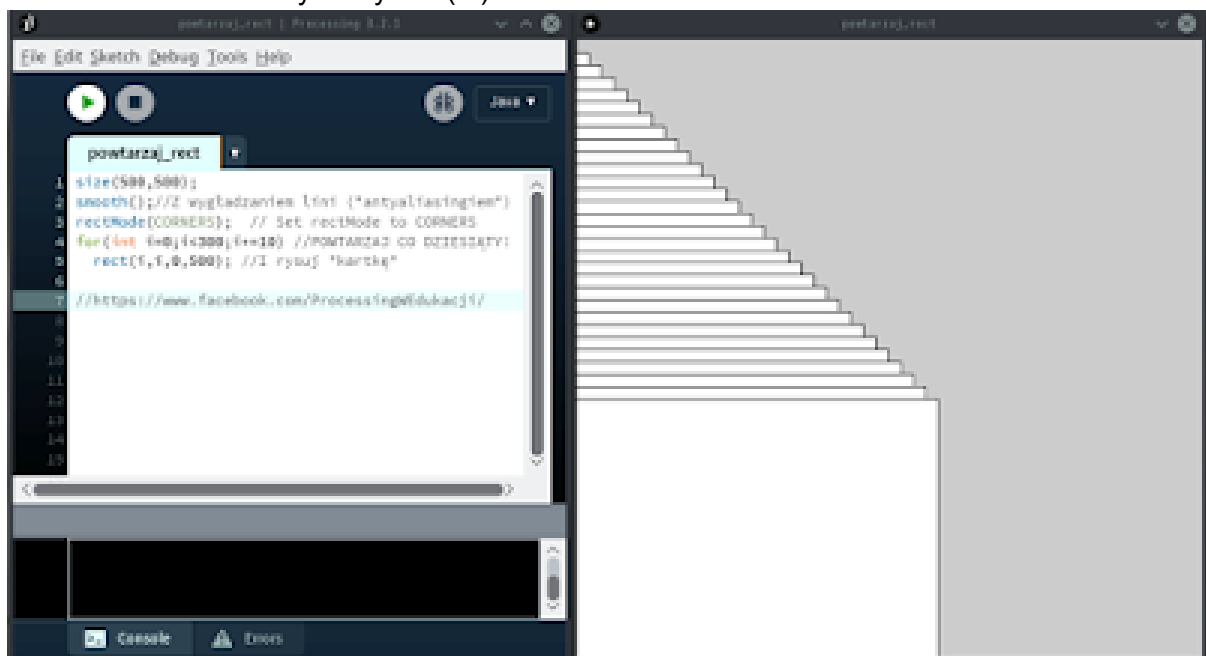


W BASICU też [krzywa uczenia](#) języka była bardzo łagodna... Rysując sobie na ekranie telewizora różne figury geometryczne i gradienty szarości lub kolorów szybko opanowałem idee pętli i komputerowy układ współrzędnych. Potem doszły kolejne instrukcje i skończyło się na próbie pisania prostej gry w zagadki...

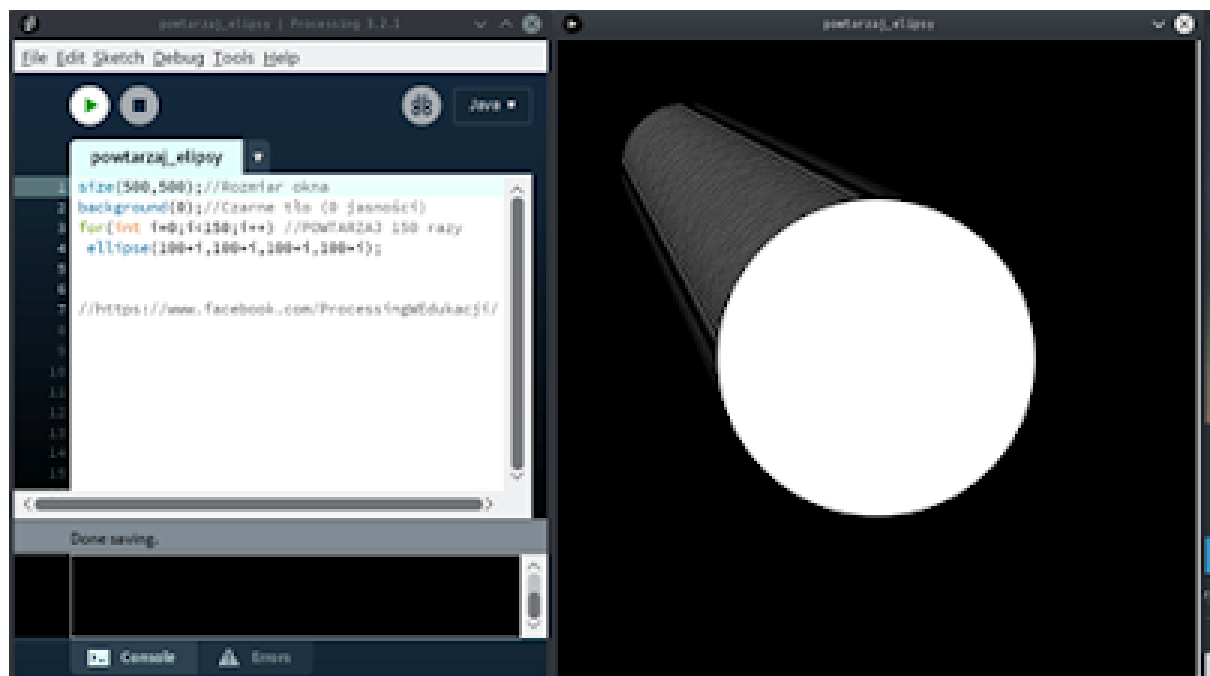


Powyżej program używający instrukcji `i+=10`.

A teraz zamiast linii używamy `rect(...)`



Aż wreszcie używamy elipsy. Tu już można dużo pokombinować... Np. przed dodaniem mnożyć `i` przez 2 lub 3.



#for #warunki #operatory

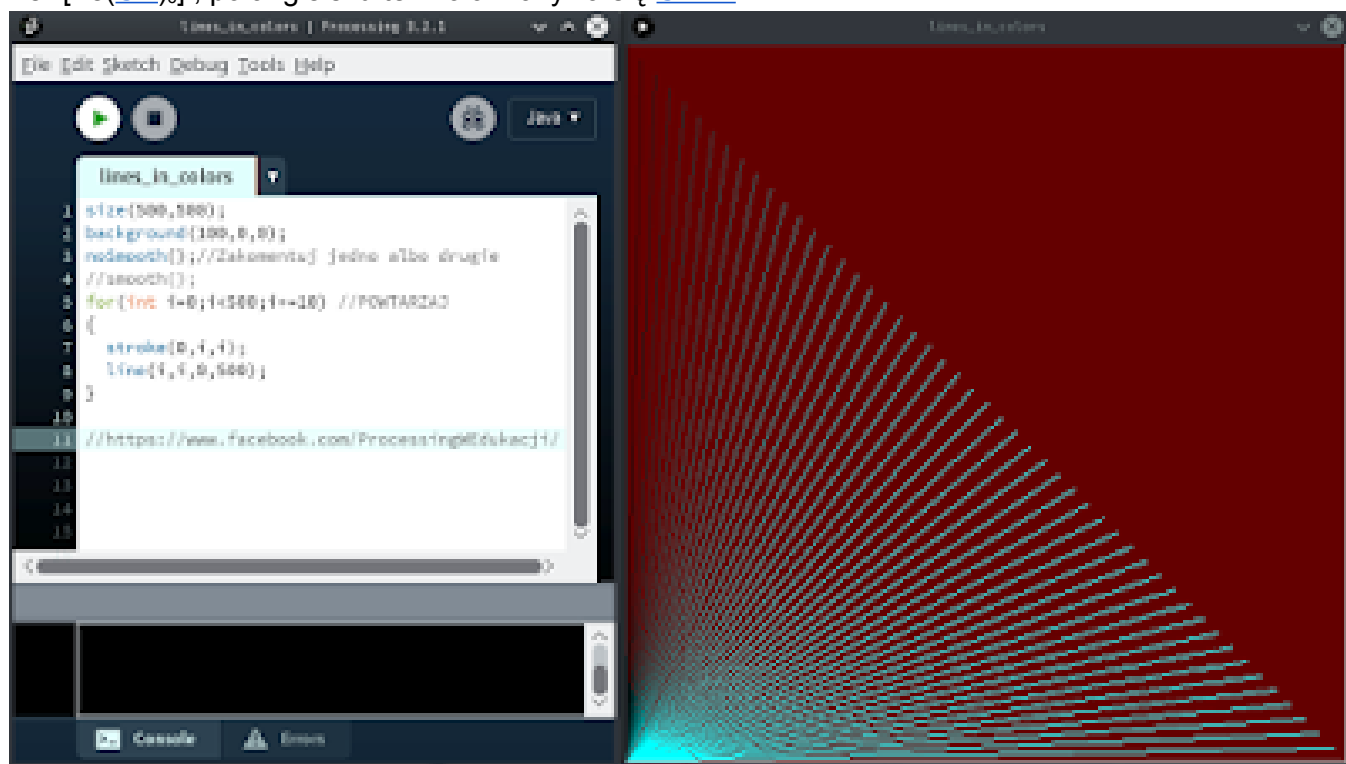
Pętle RGB

Na deser dodamy do tego kolory!

Tak naprawdę różnica polega na alternatywnych wersjach komend `background()` i `stroke()`.

Zamiast podawać jedną wartość - JASNOŚĆ od 0..255 podajemy tym komendom trzy wartości - INTENSYWNOŚĆ czy też JASNOŚĆ każdej ze składowych światła białego: czerwonej (RED), zielonej (GREEN), niebieskiej (BLUE) - stąd nazwa "kolor RGB".
Małutkie kropki właśnie w takich kolorach zobaczycie jeśli przyjrzyjecie się z bardzo bliska monitorowi komputera, albo jeszcze lepiej ekranowi telewizora, który ma zazwyczaj większe piksele (sorry jeśli nudzę, pewnie wam już o tym mówili kiedyś w szkole).

Czyli trójka 100,0,0 daje nam kolor ciemnoczerwony, natomiast 255,0,0 to kolor maksymalnie jasno czerwony. 0,200,0 daje dosyć jasno zielony. Z kolei 0,255,255 daje jasne połączenie zielonego z niebieskim czyli coś w rodzaju błękitu. Trochę oszukany ten błękit... Od składu chemicznego barwnika, który przez małe kilkadziesiąt lat był używany do malowania nieba, tzw. błękitu pruskiego, zawierającego reszty kwasu cyjanowodorowego $\text{Fe}[\text{Fe}(\text{CN})_6]$, po angielsku ten kolor nazywa się CYAN.



Zauważcie, że pętla przechodzi od 0 do 500, a intensywność koloru nie może być większa niż 255. Funkcja `stroke` jakoś sobie z tym radzi. Po prostu od pewnego momentu linie już nie zmieniają koloru. W innych ćwiczeniach zobaczymy jeszcze w jaki sposób funkcja `stroke(...)`, a także inne funkcje reagują na niepoprawne parametry.

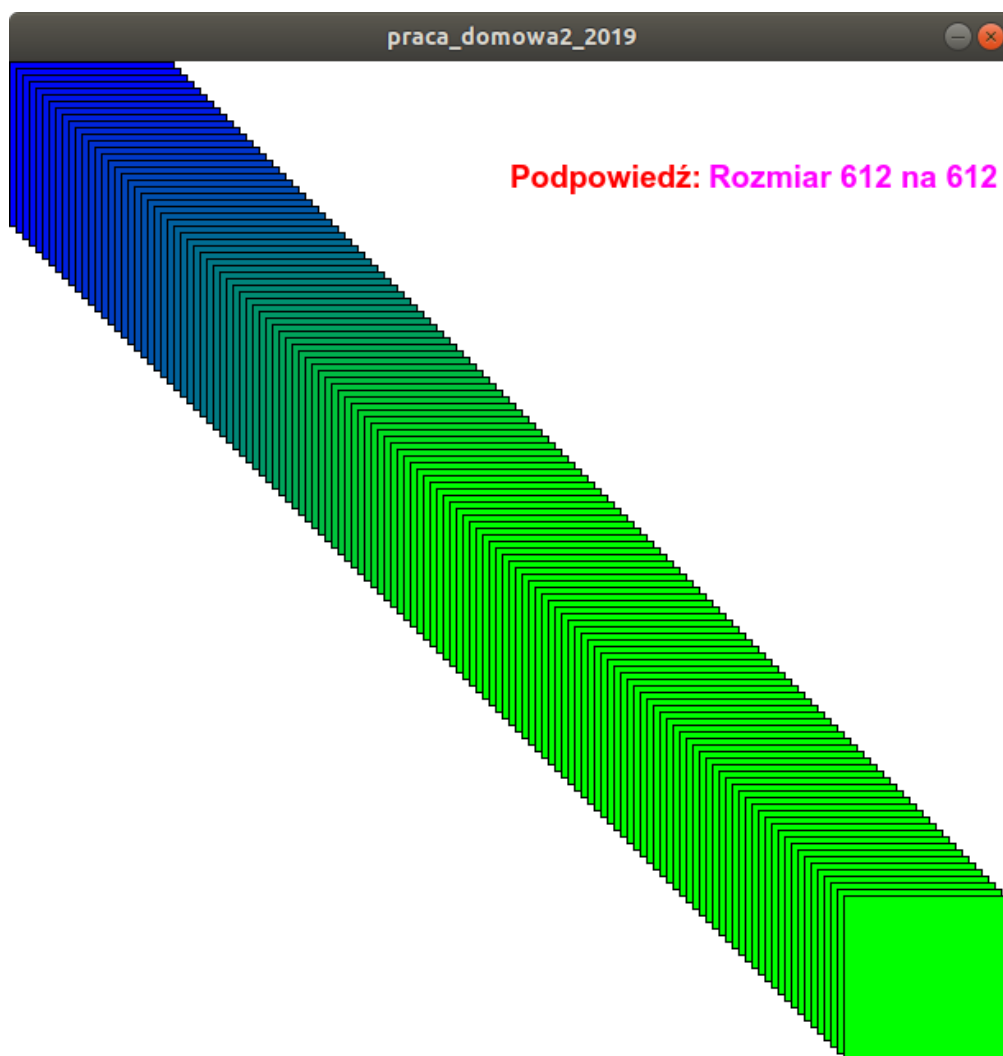
Z kolei złożenie czerwonego z niebieskim (np. 100,0,100) daje kolor MAGENTA, którego nazwa pochodzi od bitwy pod Magenta, która odbyła się w roku (1859), czyli w roku

wynalezienia tego barwnika [anilinowego](#). Nietrudno będzie zmienić kolor tła na jakiś odcień “magenta”.

Możecie też zmienić kolor tła na czarny: `background(0)` (albo `background(0,0,0)`, co da ten sam efekt) i użyć różnych intensywności tej barwy w obrębie pętli, analogicznie jak poprzednio użyliśmy “cyanu”.

A co się stanie jeśli zamiast `stroke(i,0,i)` napiszemy `stroke(i,0,255-i)` ?

Pora na pracę domową:



Poprzednie ćwiczenia na pewno się przydadzą:

`#RGB #size #for #operatory`

Oczywiście tekstowej podpowiedzi z rysunku nie musicie

[implementować](#) :-)

Generator liczb losowych w pętli ...

Zacznijmy od prostego programiku:

```
float a=random(1.0); //Liczba losowa z zakres 0..1
                        //Oczywiście zakres można zmienić
println(a); //Wypisana na konsole
```

Który uruchomicie kilka razy...

Co widać?

Zamiast `random(1.0)`; może być też `random(2.0)`; albo `random(10.0)`;

Idea pozostaje ta sama. Początkiem zakresu będzie 0, a końcem podana liczba.

Komenda `random()` jest pierwszym w tym kursie przykładem “funkcji”, wyróżnia się tym, iż ma nie tylko parametry, ale zwraca też jakąś wartość, którą możemy zapamiętać na zmiennej i coś z nią potem zrobić. Podobnie działa funkcja `sin(...)`, licząca *sinus* kąta, ale ona dla danego parametru zawsze zwraca ten sam wynik.

Funkcji `random()` można też podać początek i koniec zakresu, wtedy “wywołanie funkcji” będzie wyglądać tak:

```
float a=random(1.0,5.0); //Liczba losowa z zakres 1..5
```

A co się stanie gdy podamy zakres -5.0 do -1.0?

...

A od 5 do 1 ?

.....

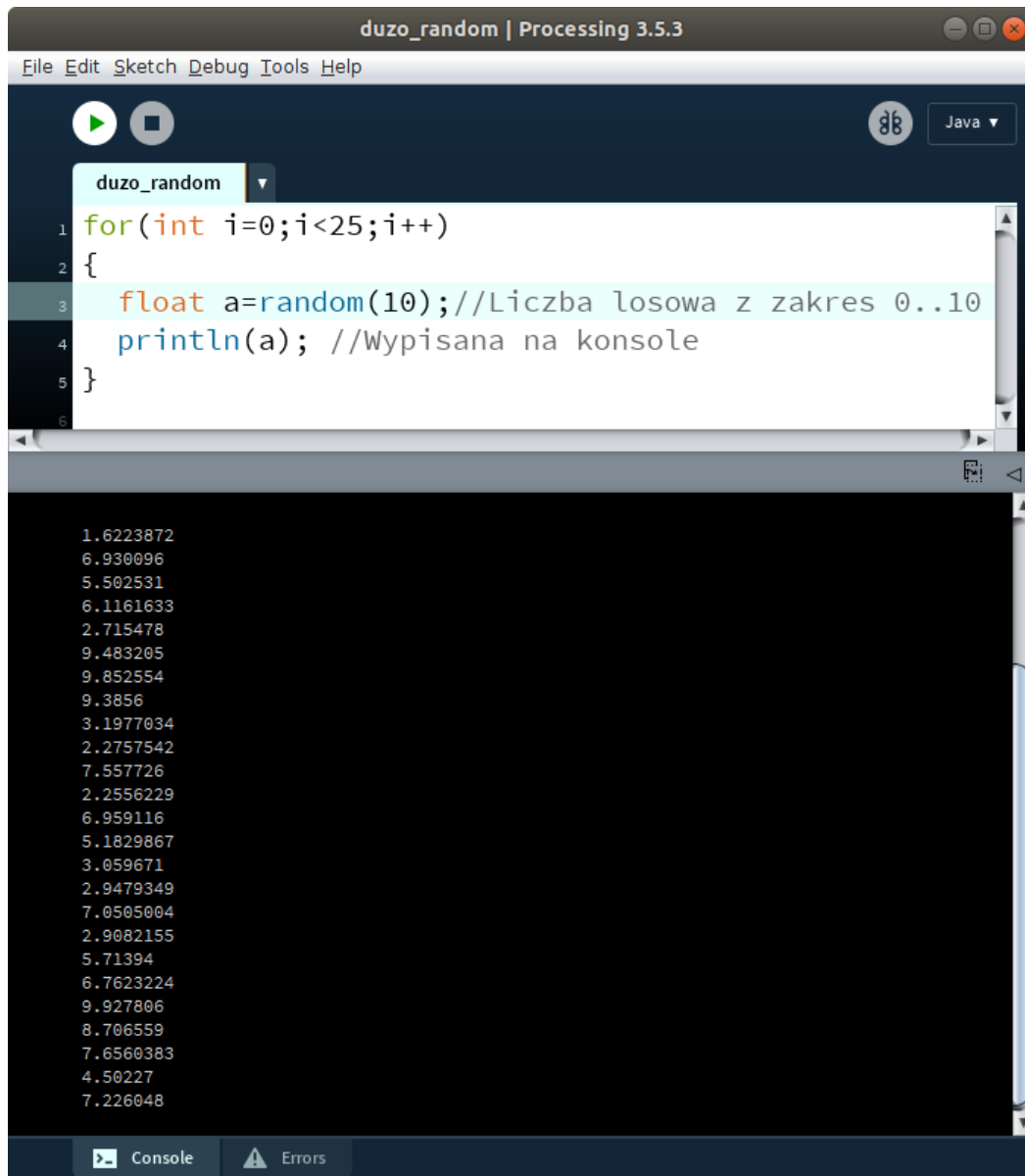
No tak :-)

Ale warto było sprawdzić :-D

Funkcja `random()` służy właśnie do tego, aby uzyskać w każdym wywołaniu inną liczbę losową. No tak właściwie to taką “prawie losową”, albo mówiąc w języku informatyki “*pseudolosową*”.

Generator liczb pseudolosowych (ang. *pseudo-random number generator* lub PRNG) – program lub podprogram, który na podstawie niewielkiej ilości informacji (ziarno, zarodek, ang. *seed*) generuje deterministycznie ciąg bitów, który pod pewnymi względami jest nieodróżnialny od ciągu uzyskanego z prawdziwie losowego źródła ([Wikipedia](#))

W rzeczywistości taki ciąg liczb generowanych przez funkcję powtarza się po odpowiednio dużej liczbie losowań. Jak na razie nie musimy się tym przejmować. Nie będziemy jej “wołać” znowu aż tak wiele razy:



```
duzo_random | Processing 3.5.3
File Edit Sketch Debug Tools Help

duzo_random
1 for(int i=0;i<25;i++)
2 {
3   float a=random(10); //Liczba losowa z zakres 0..10
4   println(a); //Wypisana na konsole
5 }
6

1.6223872
6.930096
5.502531
6.1161633
2.715478
9.483205
9.852554
9.3856
3.1977034
2.2757542
7.557726
2.2556229
6.959116
5.1829867
3.059671
2.9479349
7.0505004
2.9082155
5.71394
6.7623224
9.927806
8.706559
7.6560383
4.50227
7.226048

Console Errors
```

No to teraz przekształćmy to na grafikę. Zaprezentujemy liczby losowe jako długość linii. Żeby było lepiej widać co jest co, użyjemy asymetrycznego okna 200,300.

```
graficznie_random
1 size(200,300);
2 for(int i=0;i<300;i++)
3 {
4   float a=random(200);//Liczba z zakresu 0..200
5   line(0,i,a,i);
6 }
```

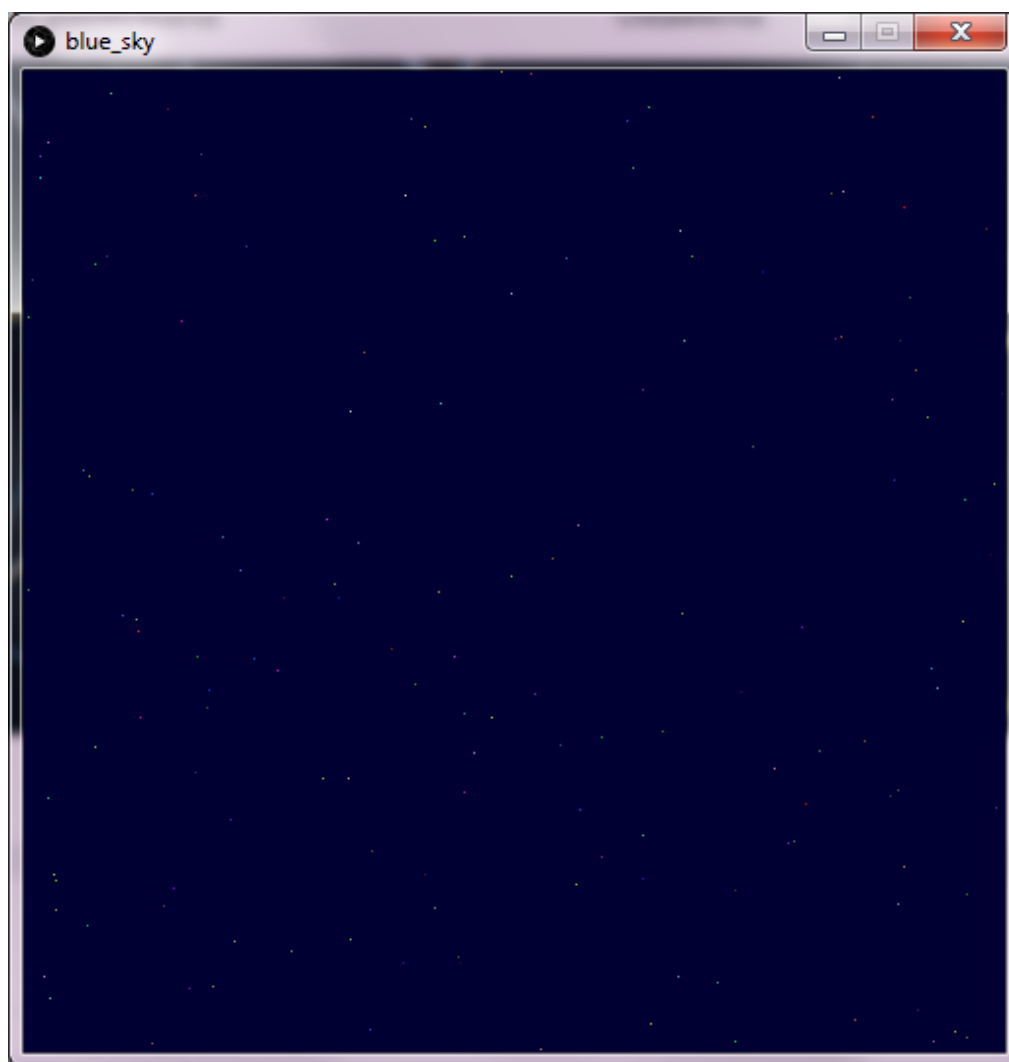
A teraz używając liczb losowych ustawmy sobie kolor każdej z linii:

```
stroke(random(256),random(256),random(256));
```

I zmieńmy orientację linii z poziomej na pionową:



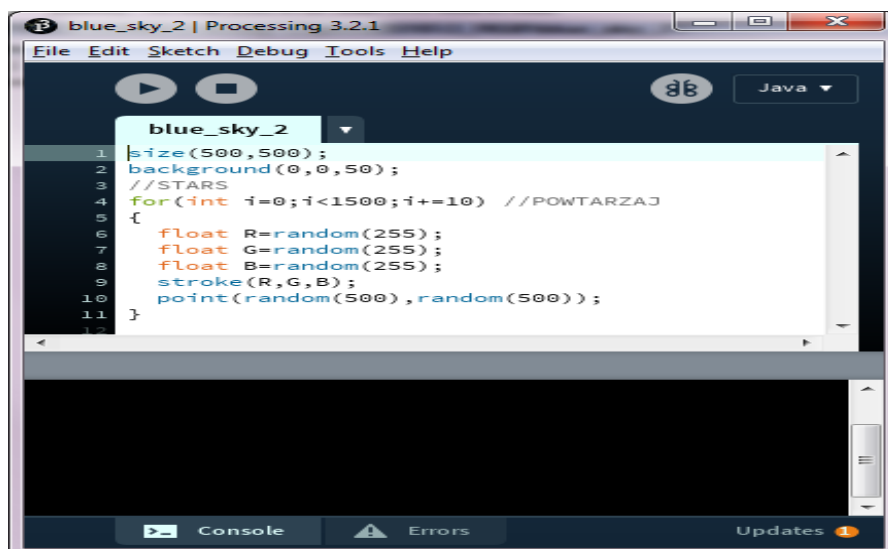
A jakby użyć punktów czyli `point(x,y)` i innego koloru tła...



Tak wygląda rozgwieżdżone niebo w Processingu. Program jest bardzo prosty, może sami go napiszecie? Podpowiadam tylko słowa kluczowe:

`#for #stroke #point #random` i jeszcze oczywiście `#size #background`

Oto kod jednej z możliwych wersji programiku rysującego to rozgwieżdżone niebo:



To o ile zwiększamy w pętli *i* jest sposobem na regulację liczby gwiazd. Oczywiście można to zrobić prościej :-). Po prostu odliczać co jeden, a liczba gwiazd będzie po prostu taka jak liczba “obrotów” pętli.

Zobacz wersje na blogu → [Pętla, funkcja random, oraz kolorowe punkty czyli "rozgwieżdżone niebo"](#)

Pętla w pętli

Często zdarza się tak, że w ramach jednej powtarzalnej czynności musimy wykonać inną powtarzalną czynność. Np. w ramach kolejnych zajęć danego dnia przeczytać listę obecności uczniów czy studentów. Albo każdemu pacjentowi na oddziale przygotować fiolki z lekarstwami na rano, południe i wieczór...

Składnia takiej “*struktury sterowania*” podana jest poniżej. Jest logiczna :-), ale na wszelki wypadek dodano “drukowanie” na konsoli zmiennych sterujących *i* i *j*, żeby można było lepiej zrozumieć kolejność:


```
petla_w_petli
1 for(int i=0;i<10;i++)
2 {
3   for(int j=0;j<10;j++)
4   {
5     println(i,j);
6   }
7 }
```

```
0 0
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
0 9
1 0
1 1
1 2
1 3
1 4
1 5
1 6
1 7
1 8
1 9
2 0
2 1
2 2
2 3
2 4
2 5
2 6
2 7
2 8
2 9
3 0
```

Jak widać pierwsza liczba zmienia się rzadziej niż druga. I słusznie.
Przecież druga to *j* czyli zmienna sterująca pętlą wewnętrzną.

Gdybyśmy chcieli namalować sobie taki obrazek jak poniżej, to wykonanie go pojedynczą pętlą byłoby dosyć trudne, choć możliwe. Za pomocą “pętli zagnieżdżonych” powinno być to względnie proste. Możecie nawet sami spróbować.

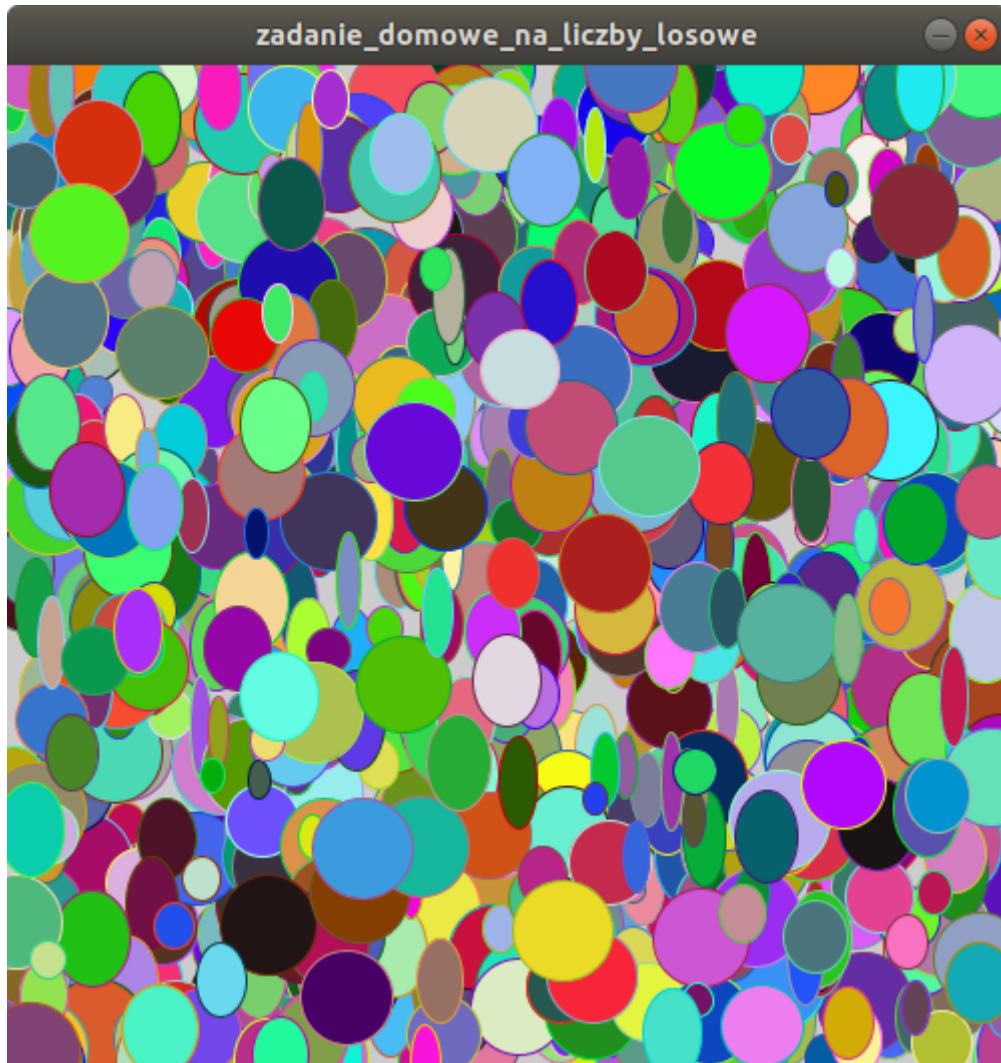


Używając pętli mającej we wnętrzu drugą pętlę dajemy sobie z tym zadaniem radę w 9 liniach prostego (?) kodu, który rysuje nam 100 różnokolorowych kwadratów. Oczywiście rozmiar kwadratu trzeba dopasować do rozmiaru okna, żeby wszystkie mogły się zmieścić.

```
random_petla_w_petli
1 size(500,500);
2 for(int i=0;i<10;i++)
3 {
4   for(int j=0;j<10;j++)
5   {
6     fill(random(256),random(256),random(256));
7     rect(i*50,j*50,50,50);
8   }
9 }
```

Spróbujcie trochę z tym programem poeksperymentować. Np. podmieńcie mnożniki przy *i* oraz *j*, albo kwadraty zamieńcie na elipsy. Do zagadnienia jeszcze nie raz wrócimy w drugiej części kursu.

Zadanie domowe na liczby losowe:



Zauważcie że elipsy mają zróżnicowane zarówno wypełnienie jak i obrzeża, żadna nie jest bardzo mała, a także nie są zbyt duże w stosunku do okna.

I jest jeszcze jedno ograniczenie w tym zadaniu, które powinniście odgadnąć uważnie patrząc na ten zbiór elips.

```
#size #random #ellipse #RGB #fill #stroke
```

PROCEDURY I FUNKCJE

Procedura - to ustalony sposób prowadzenia działania lub procesu. Jest to również sposób postępowania, przepis obejmujący tryb i sposób przeprowadzania oraz załatwiania jakiejś sprawy (np. badania/oceny np. towaru). Procedurą możemy nazwać również ustalony sposób postępowania w jakiejś dziedzinie obejmujący mniej lub bardziej ściśle określone kolejne kroki¹.

Procedura w programowaniu to inaczej **NAZWANY podprogram** czyli wydzielona część programu wykonująca jakieś operacje, możliwa do wykonania podczas wykonywania programu. Podprogramy stosuje się, aby uprościć program główny i zwiększyć czytelność kodu.

Podprogram możemy “wywołać” w dowolnym miejscu programu głównego. Oznacza to wykonanie zadania zdefiniowanego w podprogramie i powrót do programu głównego w miejsce tuż za wywołaniem:

Instrukcja1

Instrukcja2

Wywołanie podprogramu ---> Instrukcja1

Instrukcja2

Instrukcja3

... •

InstrukcjaN-ta

← Powrót

Instrukcja3

Instrukcja4

Co więcej możemy takie wywołanie powtarzać dowolną liczbę razy w różnych miejscach programu i INNYCH PODPROGRAMÓW.

W języku maszynowym i pierwotnych językach programowania (także w BASIC’u na komputery 8-bitowe) podprogram był po prostu sekwencją instrukcji kończącą się instrukcją powrotu, a wywołanie polegało na skoku do pierwszej, albo DALSZEJ instrukcji podprogramu (w BASIC’u była to instrukcja GOSUB)

Dosyć szybko jednak wymyślono żeby podprogramy nazywać i podzielono je na dwie kategorie - takie które służą obliczeniu jakiejś wartości i takie które mają zmieniać stan procesu: odpowiednio są to właśnie **funkcje** i **procedury**. Ich wywołania są jednak wszędzie podobne.

Używa się nazwy procedury i listy wartości dla parametrów wywołania, np.:

¹ Internetowa “Encyklopedia Zarządzania” <https://mfiles.pl/pl/index.php/Procedura>

```
narysuj_linie(0,0,100,100);  
y=pierwiastek_kwadratowy(x);  
z=sinus(x+Pi)-1;
```

Reasumując. Funkcja jest nazwanym rodzajem podprogramu którego zadaniem jest coś policzyć, np. funkcja licząca pierwiastek kwadratowy, sinus czy tangens z danej liczby albo poznana poprzednio funkcja zwracająca za każdym razem inną liczbę losową. Funkcja nie powinna robić nic innego, a jeśli robi to puryści informatyczni mówią o niej z obrzydzeniem że ma “efekty uboczne”.

Procedure odróżnia od funkcji to, że nie zwraca żadnej wartości, zamiast tego wykonuje pewne “prace” - jej działanie objawia się przez zmianę stanu programu - np. wartości zmiennych czy wyglądu ekranu/okna (np. znane wam już procedury rysujące linię albo prostokąt).

We wszystkich językach programowania istnieje pewien zestaw procedur i funkcji “standardowych”, czyli zawsze dostępnych (choć nie każdy język ma formalny standard przemysłowy!)

W Processingu już z podprogramów korzystaliśmy. Większość “instrukcji” we wszystkich poprzednich programach to były wywołania funkcji i procedur standardowych dla języka Processing.

Procedury obowiązkowe

W niektórych językach pojawia się jeszcze jedna specjalna kategoria podprogramów. Są to podprogramy, które użytkownik sam musi koniecznie zdefiniować, żeby program działał. W języku C i C++ jest to funkcja **main()**, a czasem wariacje na jej temat, np **wmain()**.

W Processingu, jako języku nastawionym na tworzenie grafiki animowanej na ekranie taką obowiązkową procedurą jest **draw()**.

Instrukcje zawarte w tym podprogramie są wykonywane wielokrotnie w ciągu działania programu z zadaną częstotliwością. Dzięki temu powstaje animacja.

Zazwyczaj zanim zaczniemy animację, potrzebujemy przygotować dla niej środowisko - np. ustalić rozmiar okna (**size()**), kolor tła (**background()**), sposób rysowania (**noSmooth()**).

Procedura, która na za zadanie to zrobić też ma z góry ustaloną nazwę i nazywa się **setup()**. W tej procedurze można też coś narysować, i de facto wszystkie nasze poprzednie programy po prostu w sposób niejawni definiowały zawartość procedury **setup()** - Processing przyjmuje, że jak w naszym programie nie ma żadnych definicji procedur i funkcji to cały kod jest właśnie zawartością **setup**’u.

Jeśli już zdecydujemy się na definiowanie procedur, to musimy zdefiniować przynajmniej **draw()**.

Proste lecz (pseudo-)losowe

No to najwyższa pora na przykład. To chyba najprostszy program w Processingu mający procedurę `draw()`:

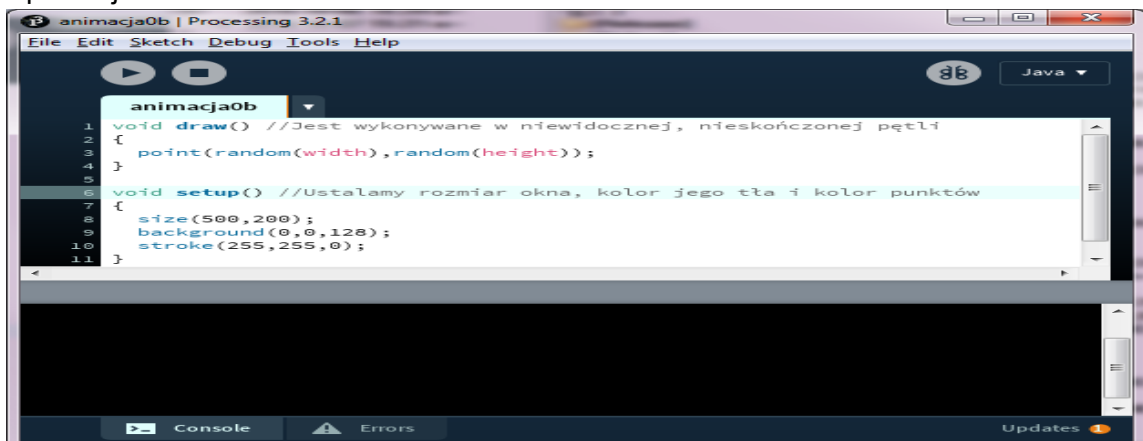
```
void draw()
{
    point( random(width) , random(height) );
}
```

Wykorzystujemy poznaną już procedurę `point` zapalającą pojedynczy piksel w oknie, funkcję `random(val)` losującą liczbę z zakresu `<0,val)` oraz dwie **domyślne zmienne** zawierające aktualną wysokość (`height`) i szerokość (`width`) obszaru rysowania w oknie.

To już zadziała...

Efekt przypomina trochę to co w było w poprzednim rozdziale, tyle że domyślne są: wielkość okna, kolor tła, kolor punktów. (--> [Losowe punkty. c.d.](#))

Żeby podobieństwo było pełne musimy dodać procedurę `setup()` i jeszcze coś, czego nie ma poniżej...



Uzupełniając poprzedni programik o funkcję `setup` możemy ustalić rozmiary okna (`size()`), kolor jego tła (`background()`), i kolor gwiazd na niebie (funkcją `stroke()`)

#setup #draw #size #background #stroke

Przykłady prostych animacji.



Kolejny przykład prostej animacji. Zauważcie że skoro funkcja draw() wykonuje się wielokrotnie możemy ją potraktować jako coś w rodzaju niejawnej pętli. I wprowadzić zmienną i oraz znaną nam z przykładów dla **for(...;...;...)** instrukcję powiększenia zmiennej o jakąś wartość.

```
i+=10;
```

Dodatkowo testujemy funkcję **noSmooth()** wyłączając antyaliasing w rysowaniu, oraz funkcję **arc()** rysującą sektor koła od podanego kąta początkowego do końcowego. Ponieważ kąty podaje się w radianach to do przeliczenia że bardziej naturalnej dla większości ludzi miary w stopniach używamy funkcji **radians()**. Efekt końcowy jest dynamiczny więc musicie sami zobaczyć ;-)
Jeszcze ciekawiej będzie, gdy wywołanie **fill** nieco zmodyfikujemy:

```
fill( 0, i % 256 , 0 ); //Jeśli na i użyjemy reszty z dzielenia  
                        // czyli operacji modulo  
//Teraz gdy i przekroczy 255 to wartość parametru  
//wróci do 0 i tak w nieskończoność  
#nosmooth #arc #radians
```


To jeszcze przykład, w którym już mamy złudzenie ruchu obiektu:

```
animLine0 ▼
3 void setup() {
4   frameRate(16);
5   size(300,100);
6 }
7
8 int pos = 0;
9
10 void draw()
11 {
12   background(204);
13   pos++;
14   line(pos, 20, pos, 80);
15 }
```

Po raz pierwszy użyliśmy **frameRate(liczba klatek)**. W ten sposób ustalamy ile razy w ciągu sekundy Processing ma wywołać procedurę **draw()**.

Im więcej tych wywołań na sekundę tym szybszy będzie ruch linii.

Póki zadania w **draw()** nie są czasochłonne, możemy dawać dosyć duże wartości liczby klatek. Jednak gdy nie uda się wykonać zadanej liczby wywołań draw, to zostanie wykonane tyle ile się uda. Nie dowiedziecie się tego, chyba że spytacie (ale o tym innym razem). Fakt faktem że już samo czyszczenie okna jest dosyć czasochłonne. WYgląda niewinnie, ale trzeba przecież WSZYSTKIE piksele okna ustawić na zadany kolor.

Ile w tym wypadku?

.....

Może was też zdziwić wywołanie procedury **background** powtarzana w każdym wywołaniu procedury **draw()**. Nie wystarczy ustalić koloru tła raz w procedurze **setup()**?

Nic prostszego jak sprawdzić co się stanie gdy ją do **setup**'u przeniesiecie ;-)

Szybkie zadanie: Przerobić program tak by linia przesuwiała się z prawej do lewej strony albo z góry na dół.