

Minimalne kompendium wiedzy

Minimalna ilość wiedzy jaką dobrze jest mieć przed przystąpieniem do kursu i rozwiązywaniem zadań.

aplikacja a program

W tych materiałach zamiennie jest używane określenie aplikacja i program. W systemie App Inventor tworzymy aplikacje na Androida ale tym samym tworzymy program na androida. Ponieważ jest to system graficzny polegający na układaniu gotowych bloków w logiczną i zadana całość nie ma tu jako takiego kodu źródłowego a jego rolę pełni element systemu AI2 jakim jest okno bloków.

kompilator i kompilacja

Środowisko programistyczne i program do pisania programów, który sprawdza poprawność napisanego przez nas kodu i zamienia go na plik wykonywalny inaczej mówiąc kompiluje go.

wielkość liter

Jak w systemie Linux taki i w wielu językach programowania jest ważna i rozróżniana wielkość liter we wszelkich nazwach i poleceniach. App Inventor rozróżnia wielkość liter w nazwach zmiennych.

zmienna

miejsce w pamięci, "pojemnik" czy kontener na nasze dane.

typ danych

W zależności od tego jakich danych potrzebujemy możemy używać liczb całkowitych, rzeczywistych (czyli takich z przecinkiem), znaków, ciągów znakowych (tekstów), macierzy (tablic) i innych. Wiele języków programowania nie ma typów danych - co wpisujemy do zmiennej czy tekst czy liczbę zostanie przyjęte. App Inventor nie ma ściśle określonych typów danych tworzonych zmiennych.

stała

Dana, której nie można zmienić w trakcie działania programu, coś jak kontener na dane ale bez możliwości zmiany wartości w trakcie działania programu.

instrukcja

podstawowy element większości języków programowania wykonujący pewną czynność. App Inventor posługuje się blokami i one w pewnym sensie są instrukcjami tworzonego programu inaczej mówiąc aplikacji.

pętla (instrukcja iteracyjna)

Pewne fragmenty kodu należy wiele razy powtórzyć z np innym parametrem. Przykładowo obliczając potęgę danej liczby mnożymy ją tyle razy przez siebie na ile wskazuje wykładnik. Pętla służy zatem do powtarzania fragmentów kodu.

warunek (instrukcja warunkowa)

Jak w życiu tak i w programie występuje wiele miejsc, gdzie trzeba dokonać wyboru i zapytać: jeśli warunek jest spełniony to wykonujemy daną czynność a jeśli nie jest spełniony wykonujemy coś innego. Przykładowo: jeśli na skrzyżowaniu widzisz światło czerwone to stój a jeśli zielone to jedź itp.

wejście i wyjście

Sposoby wyprowadzania informacji na ekran oraz pobierania danych od użytkownika do zmiennych. Mogą to być funkcje, strumienie, pola tekstowe, etykiety tekstowe, opisy przycisków i inne.

funkcja

Blok kodu, zestaw logicznie powiązanych instrukcji, które często wykorzystujemy w programie. Działa tak jak funkcja znana z matematyki $y=f(x)$: funkcja przyjmuje argument (parametr), w jakiś sposób go przetwarza i zwraca wynik. Zamiast wielokrotnie pisać ten sam fragment kodu piszemy go raz a potem wiele razy używamy. Np funkcja potęga, pierwiastek, sinus itp.

komponent

Element graficznych języków (raczej systemów) programowania będący składową interfejsu programu. Komponentem może być przycisk, pole tekstowe itp.

1. Języki i programy do tworzenia programów

Języki i systemy tworzenia programów przeszły długą drogę. W niniejszym kursie omawiamy system App Inventor za pomocą którego można utworzyć program nie pisząc ani jednej linii kodu. Tak więc programy tworzymy w specjalnych programach do ich tworzenia :) Pewne zasady i elementy obowiązują niezależnie czy piszemy kod z palca czy przeciągamy po ekranie "klocki" z jakich budujemy naszą aplikację. Większość z nich podano w poniższych punktach.

2. Zmienne i ich właściwości (nazwa, wartość, typ)

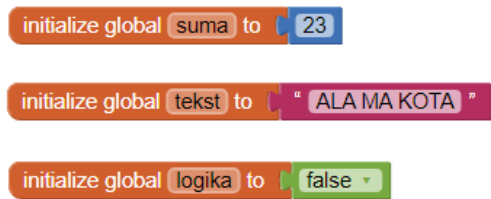
Zmienne służą do przechowywania naszych danych podczas działania programu. Przykładowo pisząc program dokonujący pomiaru temperatury procesora potrzebujemy zapewne zmiennej na temperaturę. Inaczej mówiąc szufladki na nasze dane. W języku C zdefiniujemy zmienną temperatura zdolną do przechowywania liczb rzeczywistych jak poniżej:

```
float temperatura;
```

Nazwy zmiennych nie mogą się powtarzać, zazwyczaj istnieje ograniczenie długości 16 czy 32 znaki, zmienne nie powinny zawierać (w niektórych systemach nie mogą) polskich znaków, spacji, nie mogą też zaczynać się od cyfry.

W systemie AI2 mamy zmienne globalne i lokalne. Zmienna globalna jest dostępna w całym programie, lokalna tylko w procedurze w jakiej została zdefiniowana.

Powyżej widok inicjalizacji zmiennej globalnej o nazwie "suma", należy teraz oczywiście nadać jej wartość za pomocą dołączenia bloku np jakąś liczbą lub bloku z wyrażeniem. W AI2 nie ma ściśle określonych typów zmiennych - co do takiej zmiennej wstawimy będzie poprawnie zinterpretowane. Możemy zmiennej suma przypisać wartość 32, TRUE czy "ala ma kota". Poniżej przykład inicjalizacji trzech zmiennych globalnych:



3. Zasięg zmiennych

Zmienne dzielimy na globalne widoczne w całym programie i lokalne widoczne w danym zakresie. Zakres może wyznaczyć procedura czy funkcja, blok kodu najczęściej oznaczony nawiasami klamrowymi. Po co to? Aby uelastyczyć sposoby kodowania i pisania programów i dać większe możliwości programiście.

4. Stałe

Jak nazwa wskazuje stała ma stałą wartość podczas działania programu. Aby zmienić wartość stałej należy dokonać tego w kodzie programu i ponownie go skompilować lub poddać interpretacji (w zależności od języka w jakim on powstaje).

5. Typy danych

Typy danych nadmienione w punkcie 2 służą do konkretnego i precyzyjnego określenia ile komórek pamięci zajmują nasza zmienna. Determinuje to zakres wartości jakie może zmienna przyjmować. W wielu językach programowanie nie ma ściśle określonych typów danych. Język taki sam "dopasuje" to co podamy do odpowiedniego typu i dokona właściwych konwersji. Przykładowo w języku C/C++ jeśli zadeklarujemy zmienną o typie "ciąg znaków" to nie możemy później jej w prosty sposób potraktować jako liczbę nawet jeśli do takiej zmiennej wpisujemy tekst 234 co sugerowało by wartość numeryczną. W języku PHP czy w AppInventor jest to możliwe. W systemie AI2 nie ma jasno określonych typów danych. Zmienna może być tekstem, liczbą, wartością logiczną.

6 Operatory

Operatory służą do wykonywania różnych operacji na danych (zmiennych i stałych). Poniżej kilka przykładów:

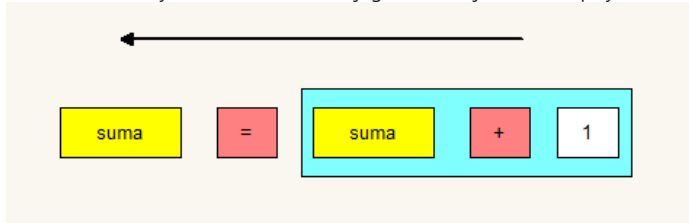
= operator przyrównania:

```
int suma;  
suma = 32;  
suma = suma + 1;
```

Tutaj zdefiniowano zmienną o nazwie suma a kolejno przypisano jej wartość 32.

Należy wspomnieć iż przepływ danych jest od prawej do lewej. Najpierw jest opracowywane wyrażenie po prawej operatora (w podanym przypadku operatora przyrównania) i kiedy zostanie wyznaczone jest przypisywane zmiennej po lewej. Widać to w trzeciej linii, gdzie cała operacja przebiega tak:

- pobranie do zmiennej tymczasowej (gdzieś w pamięci) wartości zmiennej suma
- dodanie do tej zmiennej tymczasowej wartości 1
- tak obliczone wyrażenie a konkretnie jego wartość jest teraz zapisywana do zmiennej suma. Poniższy rysunek obrazuje opisany proces.



W niektórych językach programowania operator przyrównania wygląda tak:

:=

w innych po prostu tak

=

Istnieją różne operatory, a działanie większości z nich jest z godne z tym co wiemy ze szkoły podstawowej z matematyki.

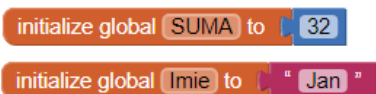
Operatory arytmetyczne + - * /

Operatory porównania == > < >= <= !=

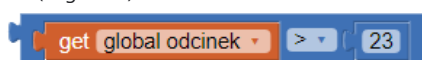
Przykłady używania operatorów:

```
a = 32;  
a = z;  
if (a >= 15)  
suma = suma + 23 * (a+b/2);
```

Pamiętać należy o priorytecie operatorów a tym samym kolejności wykonywania operacji w tak złożonych instrukcjach. W App Inventor ponieważ nie piszemy kodu z palca do dyspozycji mamy bloki. Poniżej przykład: zdefiniowano zmienną SUMA i kolejno wpisano do niej wartość numeryczną 32 oraz drugą zmienną Imie i wpisano do niej wartość tekstową "Jan".



Poniżej kilka bloków z AI2 których sens można zinterpretować: "czy zmienna odcinek jest większa od 23?". W odpowiedzi otrzymamy prawdę (ang. true) lub fałsz (ang. false).



Poniżej kolejny przykład: "jeśli pole tekstowe tbxA jest równe zero ORAZ pole tekstowe tbxB jest równe zero".



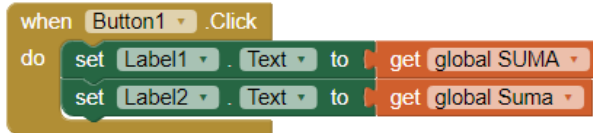
7. Instrukcje

Przez instrukcje rozumiemy poszczególne polecenia danego języka. Instrukcje przyrównania, porównania, warunkowe, pętli i wiele innych. Przykładowo:

```
a = b+32;
```

```
if (a>10) wykonaj_cos;
```

W AI2 nie mamy poleceń pisanych z palca ale bloki realizujące takowe. Poniżej przykład instrukcji, którą można zinterpretować tak: kiedy zostanie naciśnięty przycisk Button1 wykonaj polecenia z sekcji *do* (w tym wypadku etykieta Label1 wyświetli zawartość zmiennej SUMA a etykieta Label2 zawartość zmiennej Suma - uwaga na wielkość liter w AI2).



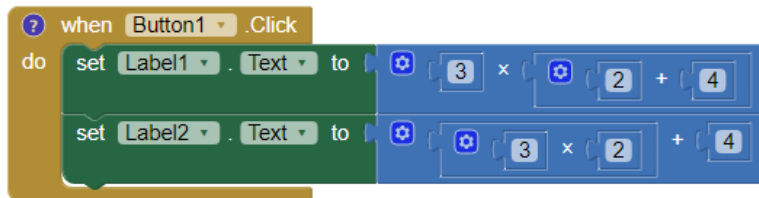
8. Opracowywanie instrukcji i wyrażeń

Instrukcje opracowywane są w zdefiniowanej kolejności. Wpływ na nią mogą mieć zasady matematyczne, przyjęty standard języka, reguły obowiązujące w danym języku, powiązanie operatorów czy ich kontekst w jakim zostały umieszczone czy zastosowanie nawiasów. Przykład - jaki jest wynik poniższych operacji?:

```
a = 12/3+1 / 5
```

```
a = 12(3+1) / 3
```

Poniżej przykład opracowywania wyrażeń w AI2:



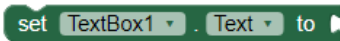
Jaki będzie wynik pierwszego a jaki drugiego wyrażenia? Pierwsze wyrażenie możemy zapisać jako: $3 \times (2+4)$ drugie jako $(3 \times 2) + 4$ - różnica jest istotna.

9. Ustawianie wartości zmiennych

Nadawanie zmiennym wartości zaprezentowano już w tym module kilka razy, generalnie w takim wypadku zmienna znajduje się po lewej stronie znaku przypisania wartości (w większości języków znak ten to równa się =). Kierunek przepływu danych z prawej do lewej. Przykład:

```
suma = 32;
```

W systemie AI2 aby ustawić wartość danej właściwości komponentu lub zmiennej używamy bloku "ustawiacza" (ang. setter): danych, przykładowo:



Blok ustawia wartość pola tekstowego komponentu TextBox (z prawej strony trzeba dołączyć blok z konkretną wartością numeryczną lub tekstową).

10. Pobieranie wartości zmiennych

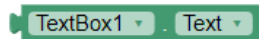
W tym wypadku zmienna, której wartość chcemy pozyskać znajduje zazwyczaj się po prawej stronie znaku przypisania wartości lub innego operatora. Kierunek przepływu danych z prawej do lewej.

```
temperatura = tablica[2];
```

```
cout << suma;
```

Pierwsza instrukcja zmiennej *temperatura* przypisuje wartość ze zmiennej tablicowej czy wektorowej (konkretnie z pola o indeksie 2), druga wyświetla wartość zmiennej *suma*.

W systemie AI2 aby pobrać wartość danej właściwości komponentu używamy bloku "pobieracza" (ang. getter): danych, przykładowo:



Blok pobiera wartość tekstu (Text) z komponentu TextBox (pole edycyjne) i wstawia je do bloku jaki znajdzie się po lewej stronie. Przykład:



Do zmiennej o nazwie SUMA zostanie wstawiona wartość jaka znajduje się w atrybucie Text komponentu TextBox o nazwie TextBox1.

11. Instrukcje warunkowe (decyzyjne)

Decyzje podejmujemy wielokrotnie każdego dnia. Generalnie można je opisać następująco: jeśli jakiś warunek jest spełniony to wykonaj coś, jeśli nie jest spełniony to wykonaj coś innego. Bardziej formalnie:

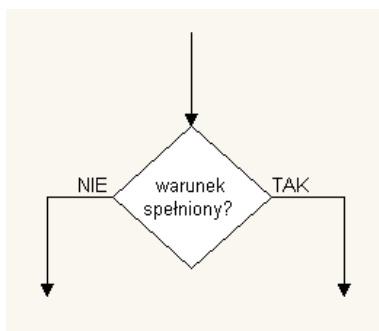
```
jeśli warunek=prawda wykonaj1
```

```
w przeciwnym razie wykonaj2
```

```
lub nieco prościej:
```

```
jeśli warunek=prawda wykonaj_cos
```

Zrozumienie jak warunki działają i jakie mają zastosowanie jest jednym z najważniejszych elementów związanych z programowaniem w ogólności. Poniższy rysunek ilustruje podejmowanie decyzji. Najpierw testujemy (sprawdzamy) warunek. Kolejno podejmujemy na bazie tej wiedzy decyzje co robimy, którą ścieżką podążamy.



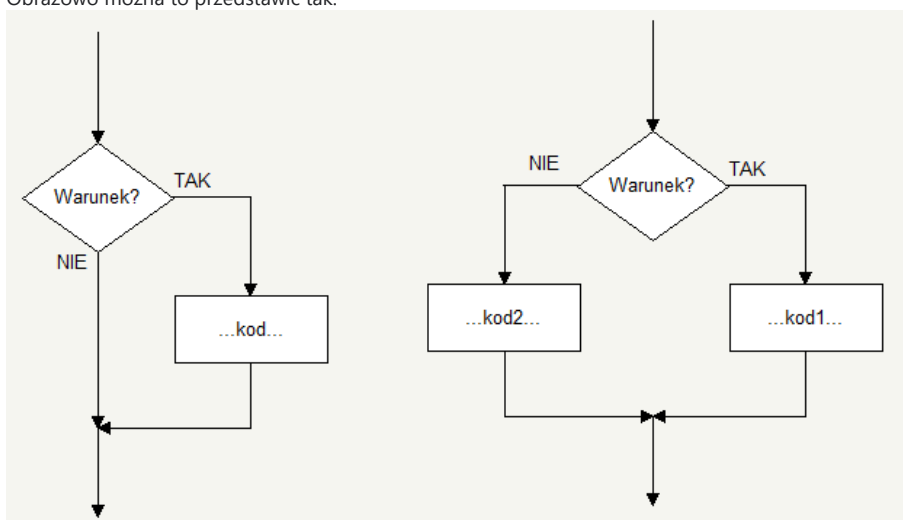
Warunki zazwyczaj pojawiają w dwóch postaciach:

jeżeli warunek_spełniony **to** instrukcje

lub

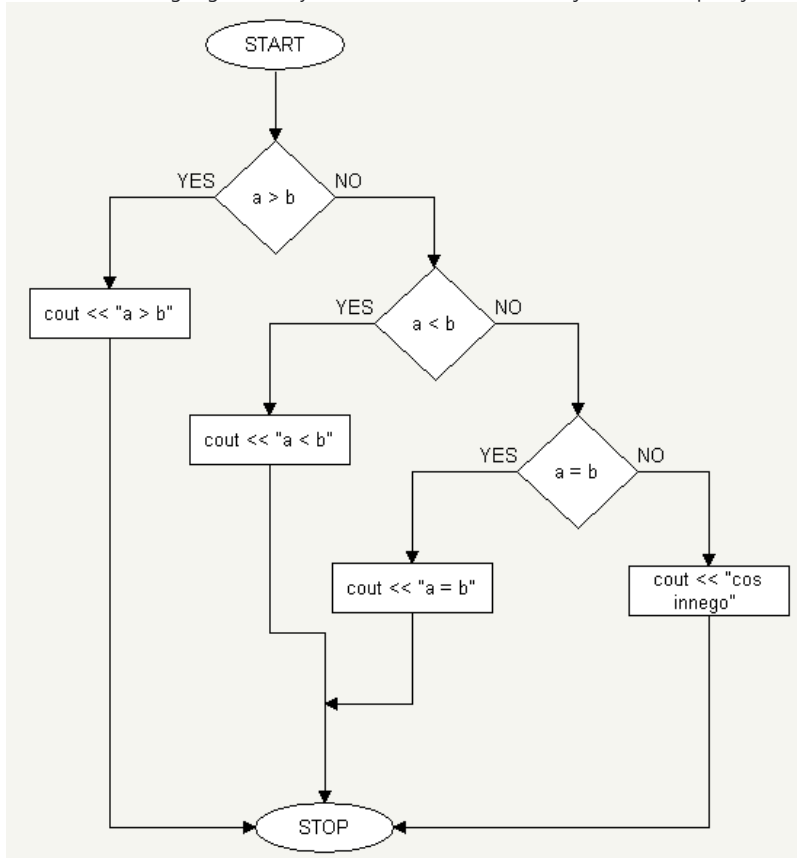
jeżeli warunek_spełniony **to** instrukcje1
w przeciwnym razie instrukcje2

Obrazowo można to przedstawić tak:



W każdym przypadku po przejściu przez warunek i ewentualny fragment kodu następuje kontynuacja programu.

Może istnieć szereg zagnieżdżonych warunków do testowania i jednocześnie podejmowania decyzji, obrazowo można to przedstawić jak niżej:



Przy okazji warunków należy wspomnieć o operacji porównania (nie przyrównania omówionego wcześniej) ponieważ w różnych językach może on wyglądać nieco inaczej. W Pascalu wyrażenie warunkowe (omówione dalej) z operatorem porównania wygląda tak:

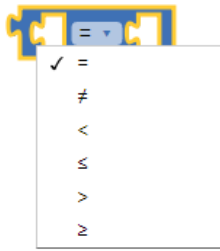
if (a = b) then zrób_cos;

W języku C/C++ i pochodnych operatorem porównania są dwa znaki `=` (równa się):

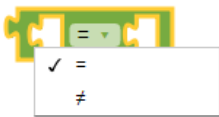
`if (a == b) zrób_cos;`

Należy zwrócić na to baczną uwagę ponieważ podczas porównań pojawia się częsty błąd gdzie piszący program zamiast `==` wstawia w warunku `=` co całkowicie zmienia sens takiej instrukcji. Powyższy schemat przedstawia instrukcje warunkowe zagnieżdżone. AI2 udostępnia bloki operacji logicznych. Umieszczone są w gałęzi *Logic* oraz *Math* sekcji *Blocks* okna *Blocks*. Najważniejsze z nich:

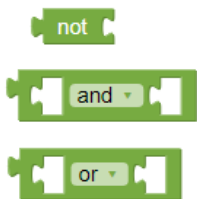
Blok porównań z sekcji *Math* (kolor niebieski), w wyniku operacji daje `true` lub `false`.



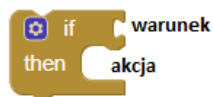
Blok porównań równy lub różny z sekcji *Logic* (kolor zielony), w wyniku operacji daje `true` lub `false`.



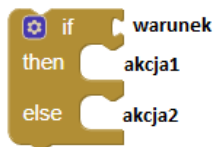
W sekcji *Logic* mamy jeszcze bardzo przydatne operacje logiczne takie jak negacja (ang. *not*), koniunkcja (ang. *and*) i alternatywa (ang. *or*):



Jeśli chodzi o możliwość podejmowania decyzji to w systemie AI2 mamy kilka bloków warunkowych:



interpretacja: jeśli *warunek* jest spełniony to wykonaj *akcja*
(kolejno przejdź do dalszej części programu o ile istnieje)



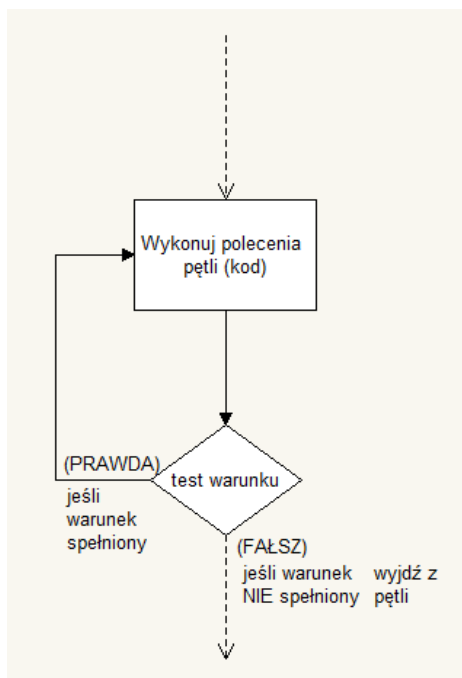
interpretacja: jeśli *warunek* jest spełniony wykonaj *akcja1* w przeciwnym razie (gdy nie jest spełniony) wykonaj *akcja2*
(kolejno przejdź do dalszej części programu o ile istnieje)

12. Pętle

Jak nazwa wskazuje są to konstrukcje pozwalające powtarzać fragmenty kodu. Powtarzanie jest bardzo istotnym elementem programowania, pozwala sekwencyjnie przeglądać listy, odczytywać kolejne wiersze z pliku czy rekordy z bazy itp. Najczęściej spotkamy się z trzema rodzajami pętli:

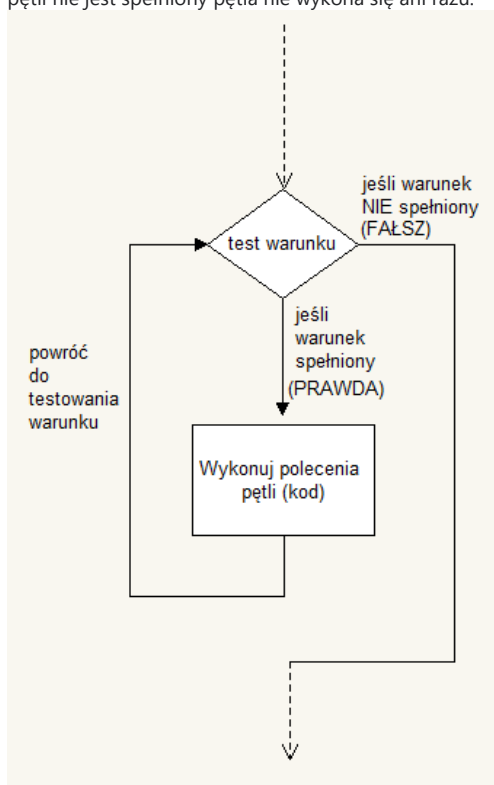
do-while czyli rób coś tak długo jak spełniony jest warunek

W tej wersji pętli najpierw wykonywany jest blok kodu przeznaczony do powtarzania a po jego wykonaniu sprawdzany jest warunek. Jeśli warunek jest prawdą (warunek jest spełniony) pętla jest wykonywana kolejny raz, jeśli warunek przestaje być prawdą (warunek jest nie spełniony). Należy zauważyć, że w tym wypadku blok kodu zostanie wykonany zawsze minimum jeden raz ponieważ testowanie warunku odbywa się już po jego wykonaniu.

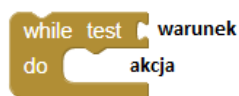


while-do czyli tak długo jak spełniony jest warunek rób coś

W tej wersji pętli najpierw jest test testowany warunek i jeśli jest prawdą (jest spełniony) blok kodu zostanie wykonany. Jeśli warunek od pierwszego przebiegu pętli nie jest spełniony pętla nie wykona się ani razu.



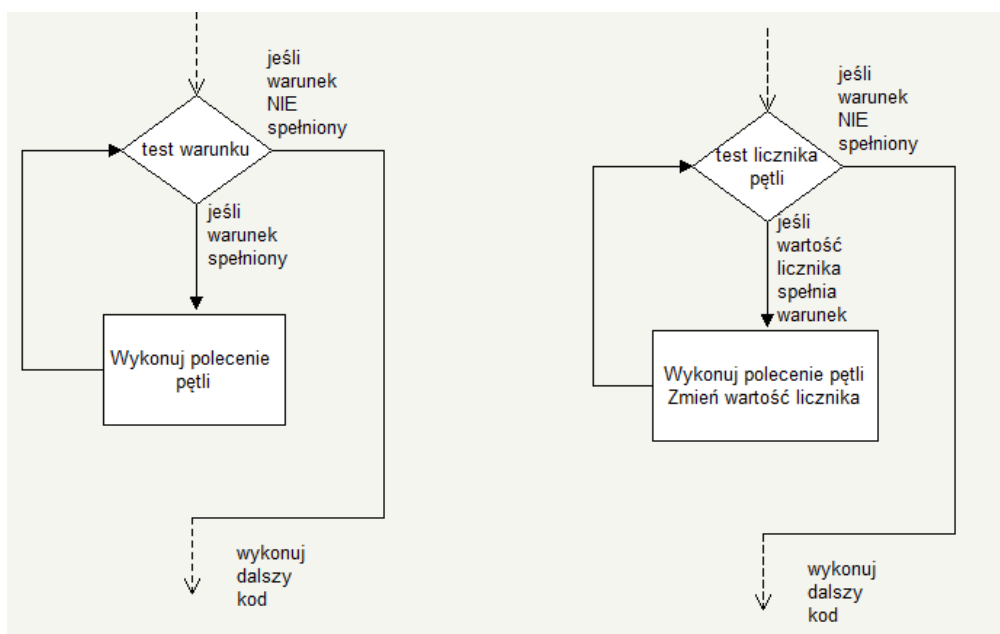
Pętla *while..do...* w systemie AI2:



Tak długo jak *warunek* jest prawdą (jest spełniony) wykonuj *akcja*.

Pętla *for n to m* do pętla z licznikiem czyli wykonuj pętlę ściśle określoną ilość razy

W większości języków programowania pętla *for* jest najbardziej złożoną pętlą i może być skonstruowana na wiele sposobów. Dwa z nich pokazana na poniższym rysunku.



W przykładzie po lewej pętla sprawdza warunek i o ile jest on prawdą przystępuje do wykonania kodu zawartego w pętli. Aby pętla zakończyła działanie warunek musi zostać zmieniony w kodzie pętli tak aby nie był spełniony.

W przykładzie po prawej mamy coś takiego jak licznik pętli. Licznik jest sprawdzany w warunku przed każdym wykonaniem pętli. Jeśli warunek z udziałem licznika jest spełniony pętla wykona zaplanowany kod. Zmiana licznika może być dokonana w owym kodzie ale bardziej elegancko jest zrobić to w nagłówku pętli do czego jest przeznaczony. Poniżej mały przykład w C.

```
#include <iostream>
using namespace std;

int main()
{
    int licznik;
    for (licznik=1; licznik <= 10; licznik++)
        cout << "licznik = " << licznik << endl;

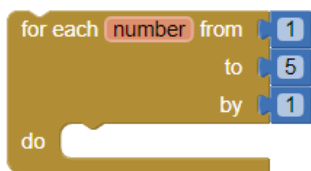
    for (licznik=1; licznik <= 10; )
    {
        cout << "licznik = " << licznik << endl;
        licznik++;
    }

    return 0;
}
```

```
F:\wsti\ppr\zadania\m
licznik = 1
licznik = 2
licznik = 3
licznik = 4
licznik = 5
licznik = 6
licznik = 7
licznik = 8
licznik = 9
licznik = 10
licznik = 1
licznik = 2
licznik = 3
licznik = 4
licznik = 5
licznik = 6
licznik = 7
licznik = 8
licznik = 9
licznik = 10
```

Obie pętle działają tak samo. Zasada działania pętli jest prawie identyczna w każdym języku programowania.

W systemie AI2 pętla `for` ma taką reprezentację:



W powyższym przykładzie pętla rusza (liczy) od 1, liczy do 5 z krokiem 1 czyli mamy tu pięć powtórzeń kodu z sekcji `do`, inaczej mówiąc sekcja `do` zostanie wykonana 5 razy. Pętla w swoim zakresie udostępnia zmienną lokalną `number` - jest to licznik pętli przy każdym jej przebiegu zwiększany o wartość z pola `by`.

13. Bloki kodu

Blok kodu, w większości języków symbolizowany parą nawiasów klamrowych `{ }` (tak jak to robiliśmy przy tworzeniu pseudokodów) wydziela z całości pewien zestaw, powiązanych w jakiś sposób ze sobą, instrukcji programu. Bloki kodu mają zastosowanie do pętli, warunków, procedur, funkcji i wiele innych. Poniżej przykłady bloków kodu w języku C/C++.

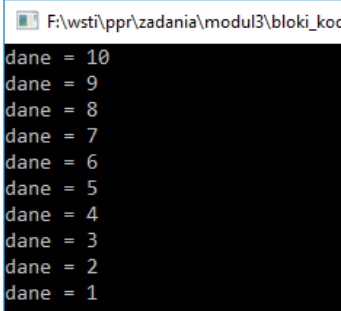
```
#include <iostream>

using namespace std;

int main()
{
    int dane = 10;

    while (dane > 0)
    {
        cout << "dane = " << dane << endl;
        dane--;
    }

    return 0;
}
```



W powyższym przypadku mamy pętlę wyświetlającą wartość zmiennej *dane* tak długo jak ona jest większa od 0. Przy każdym przebiegu (poprawnie mówiąc iteracji) wykonywane są dwie instrukcje: wyświetlenie wartości zmiennej *dane* oraz zmniejszenie wartości tejże zmiennej o jeden. Druga operacja pełni rolę licznika i jest podstawą poprawnego zadziałania pętli. Jeśli tej instrukcji się pozbedziemy pętla będzie wykonywać się w nieskończoność ponieważ wartość zmiennej *dane* zawsze będzie równa 10 czyli większa od zera co spowoduje każdorazowe spełnienie warunku. Dla czego więc przy tak prostej pętli trzeba zastosować blok kodu? Nie można tego zapisać tak:

```
#include <iostream>

using namespace std;

int main()
{
    int dane = 10;

    while (dane > 0)
        cout << "dane = " << dane << endl;
        dane--;

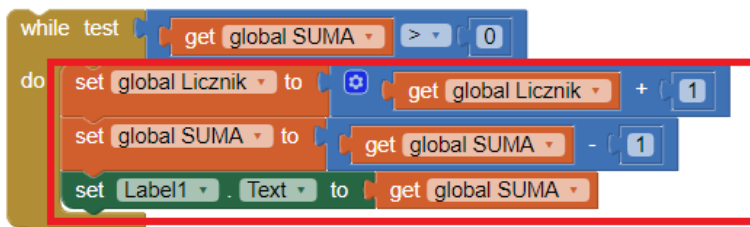
    return 0;
}
```

Wygląda prawie tak samo ale w tym wypadku (bez bloku kodu) mimo, że mamy zmniejszanie wartości zmiennej *dane* nie zostanie to nigdy przez pętlę zauważone ponieważ pętla dotyczy tutaj tylko jednej, pierwszej zaraz za pętlą instrukcji czyli "cout << ...".

W instrukcjach warunkowych, pętlach i innych blok kodu stosuje się zaraz za definicją pętli czy warunku aby pętla czy warunek obejmował wiele wierszy zgromadzonych w tymże bloku a nie tylko pierwszy wiersz za warunkiem czy pętlą.

Z blokiem kodu związany jest także zasięg obowiązywania zmiennej. Jeśli zmienna jest zdefiniowana w bloku kodu to można tylko tam ją używać. Po za tym blokiem jej po prostu nie widać. Upraszcza to budowę złożonych programów.

W AI2 funkcję bloków kodu pełni odpowiednie ułożenie bloków. Poniżej przykład: blokiem kodu jest wszystko co znajduje się w sekcji *do* ponieważ będzie to wykonane w przypadku spełnienia warunku *test*.



Najlepszym sposobem tworzenia bloków kodu jest stosowanie procedur.

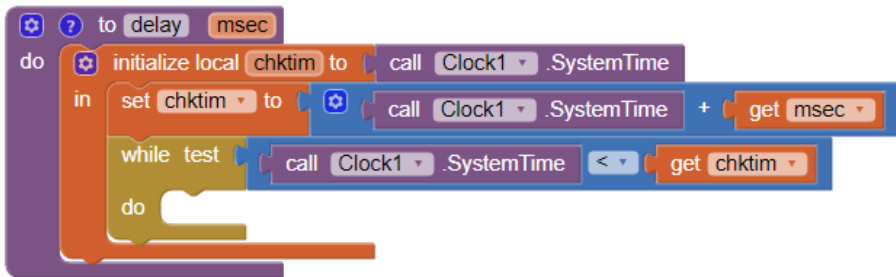
14. Procedury i funkcje

Procedura czy funkcja to wydzielony fragment kodu, do wielokrotnego użytku wykonujący pewne operacje. Przykładowo bardzo często w programie wykonujemy zestaw identycznych akcji:

- na środku ekranu wyświetl okienko
- w jego opisie wpisz: BŁĄD!
- u dołu okienka wstaw dwa przyciski: POPRAW, IGNORUJ

Taki fragment kodu można oczywiście skopiować i w powiedzmy 25 miejscach programu po prostu wkleić. Ale co w sytuacji gdy trzeba coś w wyglądzie okna lub działaniu przycisków zmienić? Ano trzeba by 25 razy dokonać tej zmiany. Z pomocą przychodzi procedura czy funkcja. Nazwa może być używana zamiennie choć w niektórych językach programowania procedura mogła tylko coś robić a funkcja (jak w matematyce) dodatkowo mogła zwracać wartość. W powyższym przypadku można zastosować funkcję i będzie ona zwracać np 0 gdy zostanie naciśnięty przycisk POPRAW oraz 1 gdy zostanie naciśnięty przycisk IGNORUJ. W C/C++ występuje tylko pojęcie funkcji a w ApplInventorze tylko procedury ale za to jest ono rozgraniczone na dwa bloki. Procedura, która może tylko coś wykonać (nie może zwracać wartości) oraz procedura z elementem *result*, która po za tym, że coś wykonuje może także zwracać wartość. Dodatkowo

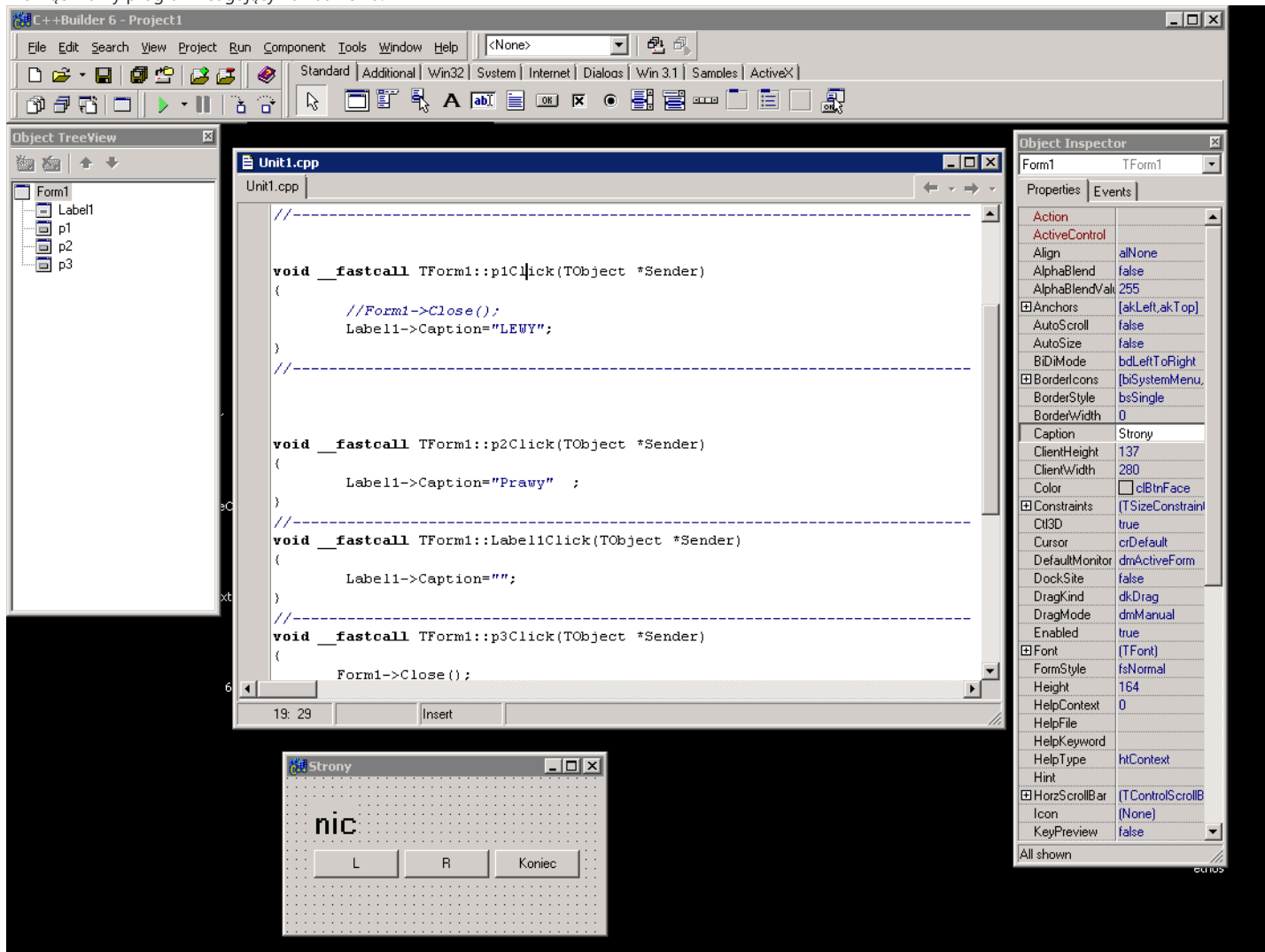
w AI2 zarówno procedura jak i procedura z rezultatem może przyjmować parametry (argumenty), które zostaną przekazane do jej wnętrza celem dalszej obsługi. Więcej informacji o procedurach znajduje się w module 2 w sekcji *Elementy programowania w AI2*.
Poniżej przykład procedury w systemie ApplInventor:



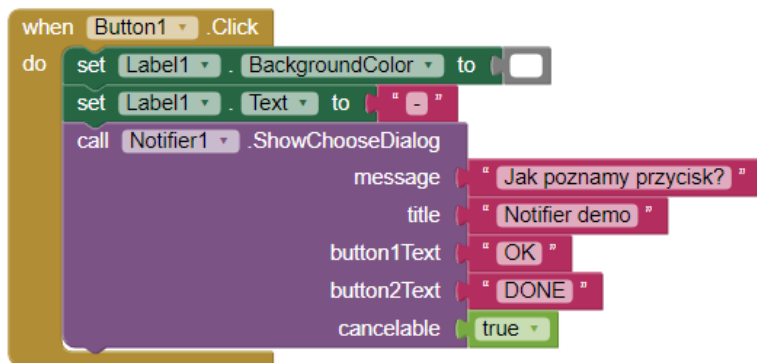
15 Obiektowość, właściwości, zdarzenia

Języki programowania można podzielić na obiektowe i nie-obiektowe. W nie obiektowych tworzy się procedury i funkcje obsługi danych i pewnych zdarzeń czy akcji w oderwaniu od danych. W językach obiektowych najpierw tworzy się obiekty odwzorowujące czy opisujące rzeczywistość w jakiej ma pracować oprogramowanie. Obiekty takie zawierają dokładny opis tego czego dotyczą czyli posiadają właściwości (zmienne, stałe) raz metody obsługi tych zmiennych czyli co można z obiektem zrobić. Standardowym przykładem jest punkt na ekranie. Punkt może mieć parametry: współrzędne X i Y oraz metody np: pokaz, ukryj, pobierz_wspolrzedne, ustaw_na_xy itd. Im bardziej złożony obiekt tym więcej może mieć parametrów i metod.
Dodatkowo w programowaniu obiektowym mamy coś takiego jak obsługa zdarzeń. Przykładowo mamy przycisk na ekranie (ang. button) który jest tak oprogramowany, że kiedy wykryje kliknięcie myszką czy inny sposób jego naciśnięcia wywołwana jest pewna metoda obsługi takiego zdarzenia wykonująca to co programista ustali. Np przycisk "Zakończ program" będzie tak oprogramowany, że po jego naciśnięciu zostanie (za pomocą odpowiedniej metody) wywołane zamknięcie aplikacji.

Wspomniany już kilka razy ApplInventor jest obiektowym językiem czy raczej systemem tworzenia aplikacji. Mamy tam sekcje budowy interfejsu i sekcje oprogramowania tego interfejsu. Ustalamy w niej parametry elementów takich jak przyciski, pola edycyjne kolory itp. Ustalamy też zachowanie się aplikacji poprzez oprogramowanie zdarzeń. Przykładowo naciśnięcie przycisku "Koniec" zakończy pracę aplikacji. Poniżej przykładowy widok programu w C++ gdzie mamy kilka obiektów takich jak przyciski czy napis. W zależności jaki przycisk zostanie naciśnięty nad przyciskami pojawi się napis lewy lub prawy. Inaczej mówiąc mamy program reagujący na zdarzenia.



Poniżej widok obsługi zdarzenia typu naciśnięcie przycisku (ang. Button) w ApplInventor:



Kiedy naciśnięty zostanie przycisk opisany jako Button1 aplikacja wykona szereg czynności jakie zaplanował programista. Można powiedzieć, że wykona zdefiniowany blok kodu. Inaczej mówiąc w programie sterowanym zdarzeniami tworzymy (programujemy) akcje ich obsługi zgodnie z zadana logika działania programu.

16 Komentarze, opisy, dokumentacja

Niezależnie od tego w jakim języki i systemie tworzymy naszą aplikację zawsze stosujemy komentarze i opisy tego co tworzymy. Często już po kilku dniach kiedy pomysł i koncepcja wylecą z głowy nie jesteśmy w stanie dokładnie odtworzyć w pamięci co i jak z tym fragmentem programu. W tym pomogą nam komentarze robione w kodzie oraz dokumentacja do poszczególnych elementów tworzonego oprogramowania. Zawsze należy tworzyć ją równolegle z programem spisując aktualne kroki i koncepcje a także pomysły na przyszłość. Tworzone procedury i funkcje koniecznie należy dobrze opisać ponieważ będą wykorzystywane wielokrotnie w przyszłości. Poniżej widok komentarza w AI2:

