Poradnik do RGSS (RPG Maker XP)

Wprowadzenie do języka ruby

:: Autor ::
Dawid „Pajper” Pieper

:: email
dawidpieper@o2.pl

www.rpgmaker.pl

Poradnik do RGSS (RPG Maker XP)

Wprowadzenie do języka ruby

Ruby jest dynamicznym językiem programowania zorientowanym obiektowo. To, że jest dynamicznym oznacza, że jest interpretowany w czasie rzeczywistym - czyli nie zmienia się w plik exe, a zmieniany jest w tak zwany kod pośredni. Dzięki temu zmiany mogą być nanoszone natychmiastowo, jednak gry w nim napisane działają wolniej od tych w np. C++. To, że jest zorientowany obiektowo oznacza, że wszystko jest obiektem, a tak zwane klasy są grupami obiektowymi.

To właśnie jest podstawowym założeniem tego języka. Kod pośredni jest to plik, który jest zmieniany w instrukcje wysyłane do procesora w czasie pracy aplikacji i dlatego zajmuje ona więcej pamięci RAM. RGSS, o którym głównie będę pisał, jest to język oparty na ruby gdzie firma Enterbrain postanowiła dołączyć go jako rozszerzenie dla swojego programu RPG Maker. Od wersji RPG Maker XP, RGSS stało się podstawowym narzędziem tego programu.

Jego pełna nazwa to ruby game scripting system, co oznacza ryby'owy system skryptowania gier. Dzięki niemu szybko napiszemy okno, odtworzymy dźwięk czy wyświetlimy bitmapę. Wykorzystuje on język ruby jako podstawę oraz bibliotekę winapi napisaną dla C++ do wyświetlania okien itp. Dzięki temu możemy używać jej poleceń. Język jest bardzo podobny do pythona.

Niestety oznacza to, że czasami wcięcia powodują problemy podczas wykonywania skryptu, należy na to uważać. Kurs możesz swobodnie publikować na innych stronach pod następującymi warunkami:

- Nie wprowadzisz żadnych zmień, dokument ten musi zostać w nienaruszonym stanie
- Poradnik RGSS (RPG Maker XP) jest darmowy, dlatego nie możesz za niego wymagać jakiejkolwiek kwoty(!). Jeżeli go udostępniasz – to za darmo.
- Napiszesz, źródło pochodzenia, skąd ci się udało ten kurs pobrać (domyślnie tylko Twierdza RPG Makera ma do niego prawo). Przy każdym odnośniku do tego poradnika, ważne jest by o tym poinformować(!)

Poradnik RGSS (RPG Maker XP)

Został napisany przez Dawida „Pajper” Pieper, dla Twierdzy RPG Makera (www.rpgmaker.pl) i tam został też opublikowany. Tutaj także podziękowania dla Reptile (reptile@o2.pl), który wszystkie moje części kursu posklejał w jedną całość, obrobił tekst, a gdzie trzeba było to poprawił błędy. Mam nadzieję, że przez ten poradnik nauczę was podstaw, tak aby język ruby wam dobrze służył.

Życzę miłej lektury ☺

Dawid Pieper.
dawidpieper@o2.pl
www.rpgmaker.pl

~ SPIS TREŚCI ~

~ ROZDZIAŁ 1 ~
Podstawy języka

~ ROZDZIAŁ 2 ~
Pliki i nowe instrukcje

~ ROZDZIAŁ 3 ~
Zarządzanie modułami, systemem i programem

~ ROZDZIAŁ 4 ~
Biblioteki DLL, oraz treści zaawansowane

ROZDZIAŁ 1
Podstawy języka

Ruby to język programowania. Najczęściej powstające w nim aplikacje to gry, ale pozwala również na tworzenie aplikacji jak kalkulator czy edytor tekstu, a nawet grafik. Jest bardzo podobny do javy, więc jeśli znasz ten język RUBY będzie dla ciebie "dziecinnie proste". W tym poradniku będę głównie pisał o RUBY w rpg makerze, xp czyli korzystającym z bibliotek RGSS.100, 101, 102, 103 i 104 (w trakcie instalacji doinstalowywana jest odpowiednia biblioteka).

Możesz jednak też pisać dzięki niemu zwykłe programy. Jeśli polecenie jest dostępne tylko w ruby rgss napiszę o tym.

W tym rozdziale poruszymy takie zagadnienia jak:

- klasy
- zmienne
- definicje
- klawiaturę
- kompilację
- usypianie aplikacji na dwa sposoby
- używanie grafiki
- używanie muzyki i innych dźwięków
- wyświetlanie okienek dialogowych
- wyświetlanie okienek wewnątrz-aplikacyjnych
- pętlę loop do
- instrukcje warunkowe
- polecenie return

Kompilacja i początek

Ruby nie wymaga kompilacji. Naszym zadaniem jest po prostu napisać kod, zapisać go w odpowiednim katalogu i wpisać ścieżkę do niego. Jeśli korzystasz z RPG Makera uruchamiasz projekt i uruchamiasz edytor skryptów (klawiszem F11), a jeśli nie to pobierz program JRUBY firmy Oracle (licencja darmowa). W RPG Makerze skrypty są domyślnie zapisane w data/script.rxdata. Jak skończysz tworzenie gry możesz zmienić ścieżkę, ale stracisz możliwość edycji skryptu, więc opiszę to na samym końcu poradnika.

Charakterystyka języka

Klasy - najważniejszym podziałem kodu są klasy. Każda klasa odpowiada za wywoływanie tych i nie innych poleceń. Możemy napisać kod programu bez klas, ale w wielkich projektach - jak gry - jest to praktycznie niemożliwe. Jeśli chcemy, by dana akcja wywoływała się wiele razy zamiast ciągle pisać ten sam kod tworzymy klasę.

W edytorze rpgmaker znajduje się lista dzieląca skrypt na kategorie. zwykle w każdej kategorii znajduje się jedna klasa, ale kategorie nie są rozróżniane przez system poza tym, że na początku każdej kategorii musimy pisać do jakiej klasy należy.
Jak to zrobić?

```
class nazwa_klasy
def main
(polecenia klasy)
end
end
```

Nazwa zwykle składa się z nazwy `Scene_nazwa_sceny_po_angielsku`. Nie używaj spacji, zastąp ją podkreślnikiem (_).Wypisaną wyżej `nazwa_klasy` zastąp nazwą klasy. Zaraz, zaraz.... Co oznacza `def main` i dodatkowe `end`? O tym napiszę w jednej z następnych lekcji. A jak zmienić aktualnie uruchomioną klasę?

```
$scene = nazwa_klasy.definicja
```

Nazwa_klasy zastąp nazwą uruchamianej klasy definicja - zastąp definicją (domyślnie: `new`)
Przykład:

```
class Scene_Test
def main
$scene = Scene_Examm.new
end
end
class Scene_Exam
def main
$scene = Scene_Test.new
```

```
end
end
```

Jest to dość złośliwy skrypt, który bez przerwy uruchamia na przemian dwie klasy aż ktoś go wyłączy. Jeśli w jakieś kategorii nie wpisze się klasy (brak polecenia `class nazwa_klasy`), a zamiast niego wpiszę się `begin`, uruchomi się on automatycznie.
Przykład:

```
begin
end
```

Zadaniem tego programu jest po prostu włączenie się i wyłączenie. Nie zmieni on nic w naszym systemie. A jak wyłączyć program podczas przetwarzania klas?

```
$scene = nil
```

Zmienne

Nie możliwe jest stworzenie gry bez zmiennych. Zmienna to znak lub wyraz, pod którym zapisana jest w pamięci ram jakaś liczba lub jakieś zdanie. W rpgmakerze zmienne ze względu na zasięg działania dzielimy na cztery rodzaje:

- globalne
  - zaczynają się od znaku $
  - można ich używać w całej aplikacji i są identyczne
  - jedna opóźnia ładowanie się programu na kilka milisekund, ale więcej może znacznie je opóźnić
- instancji
  - zaczynają się od znaku @
  - można ich użyć w całej funkcji
  - nieznacznie (wiele) opóźnia ładowanie się funkcji.
- klasowe
  - zaczynają się od znaków @@
  - można ich użyć w całej klasie
  - nieznacznie (wiele) opóźnia ładowanie się klasy.
- proste
  - nie zaczynają się ani od $ ani od @
  - można ich używać w jednej definicji (patrz definicje)
  - nieznacznie opóźniają uruchamianie się definicji

Jak nadać wartość zmiennej?

- jeśli zmienna ma zawierać dane liczbowe:
  `nazwa_zmiennej = liczba`

- jeśli zmienna ma mieć wartość true lub false:
  `nazwa_zmiennej = true`

lub
```
nazwa_zmiennej = false
```
lub
```
nazwa_zmiennej = true/false
```

powyższy kod odwróci wartość zmiennej

- jeśli zmienna ma zawierać tekst:
```
nazwa_zmiennej = "tekst"
```

  zwróć uwagę na cudzysłów

- jeśli zmienna nie ma zawierać żadnych danych:
```
nazwa_zmiennej = nil
```

- jeśli zmienna ma zawierać nazwę klasy (tylko dla `$scene`)
```
$scene = nazwa_klasy.definicja
nazwa_zmiennej = nazwa_klasy.definicja
```

- jeśli zmienna ma zawierać okienko (patrz okna):
```
$window = nazwa_okna.definicja
nazwa_zmiennej = nazwa_okna.definicja
```

Jak wynika z rodzajów zmiennej zmiana klasy odbywa się dzięki zmiennej. `$scene` jest więc automatycznie generowaną zmienną globalną. A jak sprawić, by jakaś zmienna zmieniła swoją wartość dzięki działaniom matematycznym (dotyczy tylko zmiennych z liczbami)?

```
nazwa_zmiennej = nazwa_zmiennej + 1
```
(doda do zmiennej liczbę jeden)

```
nazwa_zmiennej = nazwa_zmiennej - 1
```
(odejmie od zmiennej liczbę jeden)

```
nazwa_zmiennej = nazwa_zmiennej * 1
```
(przemnoży zmienną przez jeden)

```
nazwa_zmiennej = nazwa_zmiennej / 1
```
(podzieli zmienną przez jeden)

Wyświetlanie tekstu w naszej aplikacji

Dotychczas wszystkie operacje, które wykonywaliśmy nie pokazywały się na ekranie. Teraz spróbujemy wypisać na ekranie tekst lub wartość zmiennej. Niech komputer wyświetli okienko dialogowe z tekstem i przyciskiem ok. Zaczynamy!

```
print ("komunikat pierwszy")
print ("komunikat drugi)
print ("komunikat trzeci")
```

Jak za pewne się domyślasz skrypt ten pokaże po sobie trzy okienka z informacjami: tekst pierwszy, tekst drugi i tekst trzeci. A co zrobić żeby w okienku pojawiła się zmienna?

```
print(nazwa_zmiennej)
```

Teraz nareszcie jakieś zadanie praktyczne. Stwórz nowy projekt w rpgmakerze lub JRUBY. Jeśli korzystasz z rpgmakera zjedź do main i napis
`$scene = Scene_Title.new` zmień na `$scene = Scene_Test.new`
Stwórz nową kategorię (nazwa dowolna) i wpisz w niej:

```
class Scene_Test
def main
$zmienna = 2
$zmienna = $zmienna+3
print ($zmienna)
print ("mam dosyć działania")
$scene = nil
end
end
```

Jeśli korzystasz z Jruby wpisz ten sam kod zamieniając `class Scene_Test` na `Begin`. Najpierw zgadnij co się stanie, a następnie włącz ten program. Powinny się pojawić dwa okienka. Jeśli przewidziałeś, co się stanie, to pojmujesz dotychczasowe lekcje. Właśnie posiadłeś najbardziej podstawowe z najbardziej podstawowych umiejętności pisania w RUBY!

Okna

Uwaga LEKCJA TYLKO DLA RPG MAKERA XP!

Dotychczas wyświetlany przez nas tekst nie pojawiał się w grze, a w osobnym okienku dialogowym. Ale przecież nie powinno się zrobić projektu bez tekstu w samej grze! Jak więc zrobić okno z tekstem, które nie znika po wciśnięciu ok i nie jest osobnym oknem dialogowym?
Tak:

Uwaga! Skrypt jest trudny, więc radzę go przenieść metodą kopiuj-wklej!

```
class nazwa_okna < Window_Base
def initialize
super(0, 0, szerokość_okna, wysokość_okna)
self.contents = Bitmap.new(width - 32, height - 32)
self.contents.font.name = nazwa_czcionki
self.contents.font.size = rozmiar_czcionki
self.back_opacity = 160
refresh
```

```
end
def refresh
self.contents.clear
self.contents.font.color = kolor_czcionki
self.contents.draw_text(0, 0,szerokość_tekstu,wysokość_tekstu,
treść_okna)
end
end
```

Wyjaśnienie:

- nazwa_okna
  nazwa okna w grze (nie pojawia się na samym oknie)

- szerokość_okna
  szerokość okna w pikselach

- wysokość_okna
  wysokość okna w pikselach

- nazwa_czcionki
  czcionka, którą będzie zapisany tekst (domyślnie $defaultfontname)

- rozmiar_czcionki
  rozmiar_czcionki w pikselach (domyślny to $defaultfontsize)

- kolor_czcionki
  kolor czcionki według kodów html (domyślny to normal_color)

- szerokość_tekstu
  szerokość pola z tekstem podana w pikselach

- wysokość_tekstu
  wysokość pola z tekstem podana w pikselach

- treść_okna
  to tekst wyświetlany w oknie (w cudzysłowie) lub zmienna, z której ów tekst pochodzi

Oczywiście w tym skrypcie konfigurujemy również inne rzeczy jak opacity, ale napiszę o nich dużo później. To co napisałem powinno Wam na chwilę obecną wystarczyć.

Warunki

Na prawdę trudno jest stworzyć grę bez warunków. Bo w jaki sposób powiedzieć: "po wciśnięciu strzałki w górę rusz się o krok w górę"? Warunki możemy podzielić na trzy główne rodzaje:

- liczbowe
- tekstowe
- potwierdzające

Oczywiście nic nie stoi na przeszkodzie by jeden warunek był wszystkich trzech typów. Przykłady:

```
if $zmienna == 2
print ("zmienna $zmienna jest równa dwa")
end
```

Jak za pewne się domyślacie, jeśli $zmienna jest równa dwa pojawi się o tym informacja. A oto budowa warunku:

zapytanie
akcja w razie potwierdzenia
else
akcje w razie niepotwierdzenia
elsif
akcje dla innego potwierdzenia
end

Oczywiście funkcje `else` oraz `elsif` są opcjonalne, po `elsif` piszemy nowy warunek. Wszystko kończymy poleceniem `end`. Za pewne zadajecie sobie pytanie: "Zaraz, zaraz. Dlaczego autor tego kursu w przykładzie użył dwóch znaków =?". Odpowiedź jest prosta: dla ruby w warunkach liczbowych dwa znaki równości oznaczają równość.

Oto znaki, których używamy w warunkach liczbowych

- `=` - różne
- `==` - równe
- `<` - mniejsze niż
- `>` - większe niż
- `<=` - mniejsze lub równe
- `>=` - większe lub równe

A jak to jest ze zmiennymi tekstowymi? Przykład:

```
if $zmienna == "zmienna"
print ("wszystko się zgadza")
else
```

```
print ("coś się nie zgadza")
end
```

A teraz czas na operatory. Co to są operatory? Operatory to dodatkowe parametry warunku. Przykładowo operatory pozwalają na stworzenie takiego warunku:: "jeśli $a jest równe 3, a $b nie jest równe 3".

OPERATORY

- and
  oraz, i - pozwala na stworzenie dwóch warunków jednocześnie.

- or
  lub, bądź też - pozwala na wykonanie się warunku, gdy chociaż na jedno z zapytań padła odpowiedź: "prawda"

- xor
  albo - warunek jest przetwarzany, gdy na tylko jedno z zapytań padła odpowiedź: "prawda"

Definicje

W kilku poprzednich lekcjach wspominałem o definicjach, ale co to jest? Osobiście moje pierwsze skojarzenie z definicją brzmi: "geometria", ale jest to trochę mylące, gdyż w ruby definicjo oznaczają coś jakby podklasy. Zanim zaczniemy mała, drobna uwaga:
"definicje mogą być przypisywane tylko i wyłącznie klasą, a nie na przykład funkcji begin"

Dlaczego o tym piszę? Wielokrotnie popełniałem błąd próbując przypisać definicję `main` do `begin`u w skutek czego kompilacja się nie powiodła. Czas na jakiś przykład (jak zwykle na początek banalny):

```
begin
$scene = Scene_NASZASCENAPIERWSZA.new
end
class Scene_NASZASCENAPIERWSZA
def main
continue
end
def continue
$scene = Scene_NASZASCENADRUGA.new
end
end
class Scene_NASZASCENADRUGA
def main
dalej
```

```
end
def dalej
print ("do zobaczenia")
end
end
```

Jak mam nadzieję zrozumieliście każda klasa ma dwie definicje. Przypisałem je poleceniem `def nazwa_definicji`. Najpierw uruchomił się program i otworzył `begin`. Następnie otworzyła się domyślna definicja sceny `Scene_NASZASCENAPIERWSZA`, ale owa domyślna definicja uruchomiła kolejną definicję, czyli `continue`.

Definicja `continue` uruchomiła klasę `Scene_NASZASCENADRUGA`, a właściwie jej domyślną definicję, czyli `main`. Definicja `main` uruchomiła definicję dalej. Wtedy pojawiło się okienko pożegnalne i program się wyłączył.
Więc czym są definicje?

Są to poszczegulne elementy klas.
polecenie `$scene = Nazwa.new` uruchamia definicję `main`, ponieważ tak domyślnie ustawiony jest nasz program. Możemy to zmienić.

Poszukajmy skryptu `main`. Jest tam coś takiego.
```
while $scene = nil
$scene.main
```

Kiedy zamiast `$scene.main` napiszemy na przykład `$scene.start` domyślną definicją stanie się `start`. Kolejność definicji:

- pierwszą definicją, która zawsze się uruchamia jest brak definicji (`def nil`). Kiedy tutaj coś jest napisane program odczytuje taką klasę przed innymi
- drugą definicją, którą odczytuje program jest `def initialize`. Jak za pewne zauważyliście definicja ta występowała przy okienku. Jej zadaniem jest zdefiniowanie na przykład zmiennych. Nie wykonuje ona widocznych operacji.
- trzecią definicją, która się uruchamia jest zmienna domyślna klas (zwykle `def main`).
- następne definicje uruchamiają się po wywołaniu

Czy możemy zmienić tą kolejność? Oczywiście!
Jak napisałem w dziale o zmiennych klasę zmieniamy wpisując `$zmienna = klasa.definicja`. Oznacza to, że jeśli mamy klasę `Scene_Class` i w niej definicję def cos, którą chcemy uruchomić jako pierwszą piszemy `$scene = Scene_Class.cos`. W ten sposób zamiast uruchomienia definicji domyślnej uruchomiliśmy definicję `cos`. Przez definicje możemy przesyłać również argumenty, ale o tym kiedy indziej.

Klawiatura

Uwaga LEKCJA TYLKO DLA RPG MAKERA XP!

Podczas gry potrzebne są klawisze. Dzięki rpgmakerowi, a dokładniej bibliotece rpgxp możemy użyć następujących klawiszy (chyba się nie pomylę):

- shift oraz z
- escape oraz x
- enter oraz c
- a
- s
- d
- q
- w
- f3-11

Dlaczego czasem pisałem coś oraz coś? Niektóre klawisze działają jednocześnie. Na przykład programowi jest obojętne czy naciśniemy c czy enter. Zadziała tak samo. Jak nie wierzycie pobierzcie dowolną darmową grę zrobioną w rpgxp i zobaczcie. Oczywiście błąd ten można pominąć dzięki biblioteką ruby_dll, ale o tym później. W jaki sposób stworzyć warunek klawiszowy?

```
if Input.trigger?(Input::KODKLAWISZA)
```

Gdzie kod klawisza zastępujemy odpowiadającą mu literą. Lista kodów:
a: shift oraz z
b: escape oraz x
c: enter oraz c
l: q
r: w
x: a
y: s
z: d

Zaraz, zaraz. A co z pozostałymi klawiszami, jak v, b, n, m itd.?
Domyślna biblioteka rgss ich nie obsługuje.

A jak użyć klawiszy Fx?

```
if Input.press?(Input::Fx)
```

x zastąp numerem F.

Dlaczego skrypt nie działa?
Ponieważ by zadziałał należy użyć pętli. Chwilowo by zadziałał należy mieć wciśnięty klawisz akurat w chwili, gdy program sprawdza, czy go naciskamy. O pętlach powiem w jednej z następnych lekcji.

Muzyka

Uwaga LEKCJA TYLKO DLA RPG MAKERA XP!

Możliwe co prawda jest stworzenie programu bez dźwięków, ale gra bez takowych wydaje się słaba i nieciekawa. Dlatego dodajmy do gry jakieś dźwięki! Domyślnie biblioteka rgss daje nam cztery rodzaje audio:

- BGM - background music - muzyka tła
- BGS - background sound - dźwięki tła
- ME - music effect - efekty muzyczne
- SE - sound effect - efekty dźwiękowe

Kiedy tworzymy grę z poziomu edytora zdarzeń dźwięki SE musimy mieć akurat w folderze Audio/SE/, gdyż inaczej program ich nie zobaczy. Podobnie jest z ME, BGM i BGS. W edytorze skryptów jest jednak inaczej. Jako SE możemy uruchomić plik zapisany w folderze ME, a nawet plik zapisany w folderz graphic czy c:\windows\waves.
Jak to zrobić?

Domyślnie rpgmaker dostarcza nam polecenie, które pozwala mu przetwarzać dźwięk. Powoduje to jednak spowolnienie aktywacji o około 1-5 milisekund. Jest to mało, więc podam jeszcze jedną wadę skorzystania z tego polecenia. Może ono przetwarzać tylko jeden dźwięk jednocześnie (przetwarzać - nie odtwarzać). Dlatego skupie się na czystym i wbrew pozorom łatwiejszym w obsłudze poleceniu dostarczanym bezpośrednio przez bibliotekę rgss.
Jak to zrobić?

```
Audio.typ.play("ścieżka_do_pliku", głośność, głośność)
```

typ zastąp skrótem typu audio na przykład se. Pamiętaj napisać jednak ten skrót z małej litery. Pamiętaj o cudzysłowach.
A jak zatrzymać audio?

```
Audio.typ_stop
```

Uwaga! Polecenie te zatrzyma wszystkie dźwięki tego typu

Czy można dodać własny typ audio? Oczywiście, ale nie wchodzi to w zakres tej lekcji.

Pętla loop do

Co to są pętle? Pętle występują - podobnie jak zmienne - chyba w każdym języku programowania. Są bardzo przydatne na przykład przy tworzeniu warunków z udziałem klawiatury, czy przesuwającym się tekstem. Krótko rzecz ujmując pętla służy do powtarzania danej akcji aż do momentu jej złamania. Jak użyć takich pętli? Przyda nam się definicja, było program się nie zawieszał

```
class Scene_Cos
def main
$zmienna = 0
llop do
Graphics.update
Input.update
update
if $scene != self
break
end
end
end
def update
$zmienna = $zmienna+1
print ("to jest komunikat.") if $zmienna == 10000
print ("do zobaczenia")
$scene = nil
end
end
end
```

I jak to ma działać? Poco te polecenia, jak Input.update czy break? Pozwalają one na odświerzenie stanu aplikacji.

- Input update
  zwróć uwagę na wielkość liter odświerza klawisze. Jak tego nie zrobimy klawiatura zacznie "wariować" - chyba nie działa w jRuby

- Graphics.update
  zwróć uwagę na wielkość liter - odświerza ekran, by obrazki nie nakładały się na siebie. W jednej z następnych lekcji opiszę wyświetlanie grafiky.
- break
  łamie pętle. Do czego to służy napiszę za chwilę.

- loop do
  logiczne otwiera pętle, którą kończymy poleceniem end.

No więc co to ten break?
Skrypt ten łamie pętle i przechodzi do dalszej części klasy.
Ja napisałem coś takiego:

```
if $scene != self
break
end
```

Skrypt ten powodował, że gdy zmienimy klasę, na przykład ze `Scene_Cos` na `Scene_Cos_Jeszcze` pętla przestanie działać. Inaczej gra zacznie strasznie wolno działać, gdyż w końcu będzie przetwarzać dziesiątki pętli jednocześnie. Kolejnym zastosowaniem

tego skryptu jest usuwanie problemów, które mogą, ale nie muszą pojawiać się podczas reaktywacji klasy.

Czy można polecenia break użyć do czegoś jeszcze?

```
class Scene_Cos
def main
print ("przypisuję zmiennej @zmienna wartość 0") @zmienna = 0
print ("rozpoczynam wykonywanie pętli.")
loop do
Input.update
Graphics.update
update
if $scene != self
break
end
end
print ("pętla się skończyła, do widzenia!") $scene = nil end
def update
@zmienna = @zmienna+1 print("pętla przechodzi kolejny raz.")
if @zmienna == 50
print ("nareszcie mogę przerwać pętlę!")
break
end
end
end
```

Program ten po przetworzeniu pętli się wyłączył, czyli zadziałał podobnie do pierwszego przykładu, ale jest pewna różnica. Program wyłączył się po skończeniu przetwarzania pętli. był skrypt `loop do`, `end`, a dopiero potem skrypt wyłączający pętlę. Zatem polecenie `break` pozwala - jak już napisałem - opuścić pętle i kontynuować skrypt.
A jak stworzyć obsługę klawiszy?

Napiszę prosty program, którego celem jest zamknięcie go klawiszem escape.

```
class Scene_Program_Keyboard_Shortcut
def main
loop do
Input.update
Graphics.update
update
if $scene != self
break
end
end
end
def update
if Input.trigger?(Input::B)
print ("Cześć!") $scene = nil
end
end
```

```
end
```

Skrypt jest bardzo prosty, więc nie będę go komentował ☺

Pierwsze okienko z kursorem

Uwaga LEKCJA TYLKO DLA RPG MAKERA XP!

Ten skrypt będzie o wiele trudniejszy, więc nie radzę tego czytać zanim nie zrozumiesz treści poprzednich lekcji. Na ten skrypt składają się warunki, pętle, definicje, klasy, klawisze, okienka i wiele więcej! Co to jest kursor? To oczywiście pozycja naszego zaznaczenia. Krótko mówiąc napiszę tutaj jak zrobić okno z możliwością wyboru kilku opcji. Wyjątkowo nie napiszę skryptu ręczne, a skopiuję z rpg makera i przekształcę by uniknąć błędów.

```
#==============================================================
# ■ Scene_End #-----------------------------------------------
#  ゲーム終了画面の処理を行うクラスです。
#==============================================================

class Scene_Cos
def main
# コマンドウィンドウを作成
s1 = "pokaż informacje o programie"
s2 = "Pokaż informacje o przeznaczeniu programu"
s3 = "Wyjście z programu"
@command_window = Window_Command.new(192, [s1, s2, s3])
@command_window.x = 320 - @command_window.width / 2
@command_window.y = 240 - @command_window.height / 2
Graphics.transition
loop do
Graphics.update
Input.update
update
if $scene != self
break
end
end
@command_window.dispose
end
def update
@command_window.update
```

```
if Input.trigger?(Input::B)
$scene = nil
end
if Input.trigger?(Input::C)
case @command_window.index
when 0
print ("Program test, aktualna klasa: Scene_Cos. Autor Dawid
Pieper, język ruby.")
when 1
print ("Program ten został stworzony jako przykład dla kursu
ruby mojego autorstwa.")
when 2 # やめる
$scene = nil
end
return
end
end
end
```

Jak za pewne się domyślacie skrypt ten korzysta z klasy `Window_Command`, która została stworzona automatycznie, dodawanie opcji do wyboru. U góry skryptu widniał napis:
`s1 = ; s2 = ; s3 =`

Zmienna `s1` odpowiada za tekst pierwszej opcji, `s2` za tekst drugiej opcji i.t.d. Na przykład gdy napiszemy s1 = "coś tam" pierwsza opcja będzie się nazywała coś tam. Następnie linijkę pod s znajduje się napis:
`@command_window = Window_Command.new(192, [s1, s2, s3])`

tam gdzie znajduje się lista zmiennych `s1, s2, s3` gdy dodajemy nową opcję trzeba ją tu wpisać. Natomiast w definicji `update` było coś takiego: `when 0, when 1, when 2`.

- when 0 - odpowiada za to, co się stanie po wyborze opcji s1
- when 1 - odpowiada za to, co się stanie po wciśnięciu opcji s2
- i tak dalej

Oczywiście nazwy zmiennych `s1, s2 i s3` można zmienić. Jak to zrobić?

Najpierw definiujemy ich tekst, a potem piszemy coś takiego:
```
@command_window = Window_Command.new(192,
[jakaszmiennapierwsza, jakaszmiennadruga, jakaszmiennatrzecia,
jakaszmiennaczwarta, itakdalej])
```

W pętli dodano również funkcję, która zamknie program po wciśnięciu escape.
Można dodawać więcej takich funkcji na przykład na shift.

Polecenie: sleep

Dzisiaj dla odpoczynku jakaś bardzo, bardzo prosta lekcja. Mowa o bardzo ważnym, ale i łatwym poleceniu sleep. Jak tego użyć i jak to działa?

```
sleep(czas)
```

Jak widać polecenie jest bardzo proste. Po jego wpisaniu program nie będzie wykonywał nowych akcji przez określony czas. Może być to na przykład przydatne gdy chcemy, by po skończeniu się muzyki otworzyło się jakieś menu lub pojawił się jakiś tekst.
W jakich jednostkach podajemy czas?

W ramkach. Ramki to może nieoficjalna nazwa, gdyż zwykle posługuje się angielską nazwą: frames. W bardzo, ale to bardzo dużym uproszczeniu jedna ramka to 0,1 sekundy, ale to nie do końca prawda. 900 ramek to minuta, więc 600 ramek to 40 sekund.

To też jest uproszczenie. Ramki są bowiem liczbą określającą ile razy program przeczyta dany skrypt. Dlatego na szybkich komputerach jedna ramka może trwać na przykład 24 nanosekundy krócej. Ja osobiście korzystam z takiego sposobu: jedna minuta to 400 ramek, więc jedna sekunda to 1,(1) ramki. Nawias oczywiście oznacza okres. W przypadku użycia niektórych bibliotek można napisać na przykład: `sleep(5-minutes)`, ale nie zawsze to działa.

Polecenie: return

Polecenie return jest za pewne jeszcze łatwiejsze od polecenia sleep, więc lekcja będzie krótka.
Jak go użyć?

```
$scene = klasa
return
```

Do czego ten skrypt służy?
Polecenie return informuje klasę o tym, że ma się załadować według poprzednich ustawień. Na przykład gdy zamykamy menu gry i wracamy na mapę wykonywane jest polecenie return, by postać pojawiła się tam, gdzie ostatnio, a nie na przykład na początku mapy. Zwykle parametr ten jest opcjonalny, ale bezpiecznie jest go użyć dla samej pewności.

Trochę grafiki, czyli sprite

Klasyczna gra potrzebuje grafikę, prawda? W tej lekcji nareszcie pokażę wam jak użyć obrazków w grze. Spodziewam się, że długo na to czekaliście. Tutaj opiszę wyświetlanie grafiki metodą sprite. Jest to najłatwiejsza z metod i zalecam na razie na niej się skupić.

```
@sprite = Sprite.new
@sprite.bitmap = TYP::Ciasteczka.folder("obrazek")
Graphics.transition(120)
Graphics.update
Graphics.freeze
@sprite.bitmap.dispose
@sprite.dispose
Graphics.transition(40)
Graphics.freeze
```

Skrypt ten uruchomił grafikę, czyli obrazek, podtrzymał go przez 120 ramek, zaktualizował, odczekał 40 ramek i ukrył. Za pewne jedyna linijka, która zasługuje na szerszy komentarz to:

```
@sprite.bitmap = TYP::Ciasteczka.folder("obrazek")
```

Więc tak:

- zamiast TYP jak tworzysz w rpgmakerze zawsze pisz tu RPG
- ciasteczka - w rpg makerze pis tutaj Cache
- folder - w folderze graphic znajduje się kilka folderów. Wpisz tu nazwę któregoś z nich. Uwaga! Nie zakładaj własnych folderów.
- obrazek - plik z obrazkiem łącznie z rozszerzeniem.

Metoda ta przyjmuje jakąś zmienną jako obrazek. Zmiennej można zmienić nazwę, ale układ powinien być zawsze taki:

```
zmienna = sprite.new
zmienna.bitmap = TYP::Ciasteczka.folder("obrazek")
```

Jak używasz grafiki pamiętaj zawsze odświeżać obrazki poleceniami:

- Graphics.update
- Graphics.freeze

Pamiętaj, że polecenie `Graphics.transition` działa podobnie do `sleep`. Różnica polega na tym, że w przypadku `sleep` program przestaje reagować i odpowiadać, a w przypadku `Graphics.transition` ignoruje akcje użytkownika, ale odpowiada.

Do zobaczenia w następnym rozdziale!

ROZDZIAŁ 2
Pliki i nowe instrukcje

To jest druga część kursu ruby mojego autorstwa. Nie zabieraj się za naukę ruby na tym poziomie, do póki nie zrozumiesz poprzedniej części. Kurs ten dostępny jest częściowo dla jRuby, ale dedykowany jest dla użytkowników programu rpg maker xp. By uniknąć problemów z rozpoznaniem co jest kodem programu (skryptem), a co dalszym tekstem kody zaznaczę kursywą.

W tym rozdziale poruszymy takie zagadnienia jak:

- otwieranie plików
- zapis i odczyt z/do plików metodą marshal
- komentarze
- warunki istnieniowe
- funkcję ładowania *require*
- zmianę profilu procesu gry
- klasę wsteczną each
- kopiowanie i przenoszenie plików
- zakładanie folderów
- Zmianę zmiennej klasowej
- pętlę while
- siatki tekstowe i liczbowe
- operacje dźwiękowe (wyciszanie)

Komentarze

Komentarze to przydatna rzecz, o której zapomniałem napisać w poprzedniej części tego kursu. Służą one do zaznaczania jakiś miejsc lub do napisania instrukcji dla użytkownika naszego skryptu.
Jak tego użyć?

Kiedy na początku linii postawimy znak `#`, to cała zawartość tej linii zostanie zignorowana nawet, gdy linia ta zawiera polecenia.
Przykład:

```
class Scene_Comment
def main
print ("jakiś tekst")
#print("xxxxxxxxxx")
end
end
```

Tekst: "jakiś tekst" zostanie wyświetlony, ale tekst: "xxxxxxxxxx" nie, bo został zawarty w komentarzu. Dzięki tej funkcji mogę w następnych skryptach mogę dodawać instrukcję w samym skrypcie, a nie poza nim jak do tej pory.

Potwierdzanie istnienia plików metodą kodowania marshal

Dzięki temu poleceniu możesz sprawdzić, czy dany plik istnieje. Jak tego użyć?

```
if FileTest.exist?("nazwapliku")
end
```

Jest to warunek, więc można dodać polecenie `else`. Logiczne jest zatem, że kiedy plik nazwa-pliku istnieje warunek zostanie wykonany. W przeciwnym wypadku zostanie wykonane `else` lub warunek zostanie zignorowany.

Do czego ten warunek między innymi może się przydać?

- sprawdzanie istnienia zapisu
- sprawdzanie pierwszego uruchomienia gry na przykład w celu powitania
- zapisywanie ustawień
- finalizacja instalacji

Zapis do pliku i odczyt z pliku

Oczywiście! Zapis i odczyt jest niezbędny do zrobienia ustawień, czy zapisu gry.
Jak zapisać coś do pliku?

```
zmienna-plikowa = File.open('plik', 'typ')
Marshal.dump("tekst", zmienna_plikowa)
zmienna_plikowa.close
```

- zmienna_plikowa - zmienna, która będzie używana do zapamiętania, że plik był otwarty, może być dowolną zmienną nieużywaną do innych celów.
- plik - nazwa pliku, na którym będziemy wykonywali operacje
- typ otwarcia pliku (więcej poniżej)
- tekst - tekst, który zostanie zapisany do pliku

A jak odczytać coś z pliku?

```
zmienne_plikowa = File.open('plik', 'typ') zmienna_tekstowa =
Marshal.load(zmienna_plikowa) zmienna_plikowa.close
```

- zmienna_plikowa - tak, jak poprzednio
- plik - tak jak poprzednio
- typ - tak, jak poprzednio
- zmienna-tekstowa - zmienna, do której zostanie zapisany tekst z pliku

Ale co to jest ten typ?
Podobnie jak w PHP w ruby możemy otwierać pliki używając różnych typów.
Można jednak zauważyć pewne różnice!

- R - plik tylko do odczytu, czytany od pierwszej linijki
- RB - plik do odczytu i zapisu czytany od pierwszej linijki
- W - plik do zapisu, jest wpierw czyszczony
- WB - plik do zapisu i odczytu, wpierw jest czyszczony.
- A - plik do zapisu

Na przykład gdy wybierzemy typ **R**, nie możemy w tym pliku nic zapisywać. Jeśli chcesz zapisać lub odczytać dwie linijki, zapisz odpowiednie polecenia dwa razy, w razie odczytu używając innej `zmiennej_tekstowej`.

Jeśli zapisujesz liczby i nie zapiszesz ich w cudzysłowie zostaną zapisane w systemie szyfru ruby.

Napisy w ruby są kodowane wewnętrznym kodem, bez którego znajomości nie można edytować takich plików w notatniku. Kod ten można zmienić, ale teraz o tym nie napiszę.

Uwaga! Polecenie odczytu użyte po poleceniu zapisu odczyta następną linijkę, a nie tą zapisaną poprzednio. Według kodu tego programu linijki w rzeczywistości są oddzielane specjalnym znakiem, w rzeczywistości (chyba, że nie zmieniamy kodu) nową linią.

Zaraz, zaraz. A co oznacza to: "metodą marshal"?
Przy każdym poleceniu zapisu i odczytu pisałem w jakimś miejscu marshal na przykład `marshal.dump.`

Marshal zapisuje tekst formatem binarnym, dzięki czemu tak trudno edytować jest tekst w notatniku. Jest jednak dobrym sposobem na początek. Żeby zapisać tekst w ten sposób, by wyglądał on jak zwykły tekst .txt trzeba użyć metody yaml, ale o tym pewnie napiszę kiedy indziej.

Uruchamianie innych plików

Wiadomo, że są operacje, które łatwiej wywołać w innych językach. Zdarzają się też przypadki, gdy dana biblioteka nie działa w ruby'm albo gdy chcemy wyświetlić plik readme.txt. Funkcja ta po prostu uruchamia wybrany przez nas plik.

Zanim przejdziemy do pisania kodu kilka uwag:

- pamiętaj, że gdy wyświetlane jest inne okno gra przestaje działać (czeka aż użytkownik do niej powróci). Można to użyć by na przykład po wyświetleniu instrukcji gra nie liczyła czasu, ale jeśli tworzysz jakąś procedurę na przykład w C++ pamiętaj uruchamiać ją w tle.
- wyjątek dla powyższej reguły występuje gdy gra jest uruchamiana w priorytecie czasu rzeczywistego. Może kiedyś napiszę jak to zrobić.

No to przejdźmy do kodu:

Dla rpg makera
```
system('start plik')
```

Dla jRuby lub RUBY:
```
require 'thread'
thread.new { system('start plik') }
```

Jak widać kod w rpg makerze dzięki bibliotece RGSS jest odrobinę krótszy.

- plik - nazwa uruchamianego pliku

Jakie pliki możemy uruchomić?
Prawie wszystkie począwszy od plików tekstowych, poprzez muzykę i obrazy, a kończąc na innych aplikacjach. Pamiętaj jednak, że gdy na przykład używając tej metody otworzysz jakiś plik mp3 otworzy się również okno z programem go odtwarzającym.

Dlatego zalecam napisanie własnego odtwarzacza na przykład w ruby. Już powinieneś wiedzieć jak to zrobić.

Dlaczego polecenie wygląda tak?
Polecenie `system` odnosi się do wewnętrznej konsoli dostępnej po wpisaniu w polu uruchom CMD.

Są jednak widoczne pewne różnice, na przykład piszemy `start plik`, a nie `start=plik`. Podobnie nie możemy wpisać po prostu nazwy pliku jako polecenie. Używając tej metody nie powinno się uruchamiać nieskojarzonych z systemem plików, gdyż w takim przypadku dojdzie do sytuacji, gdy użytkownik zostanie poproszony o wybranie programu, którym chce otworzyć dany plik. W nazwach plików dla bezpieczeństwa nie używaj spacji i polskich znaków!

Funkcja require (dla ładowania modułów)

Lekcja tylko dla ruby i jRuby, gdyż rpg maker rgss wykonuje ją automatycznie i nie jest ona tam potrzebna, chodź można jej użyć. Funkcja require służy do ładowania modułów. Moduły dodają do naszego programu dodatkowych poleceń.

Nie mylcie tego z bibliotekami, gdyż ładuje je się w podobny sposób, ale trzeba je skopiować do folderu z naszą aplikacją lub też zarejestrować ją w systemie. Kiedy zapiszemy nasz projekt moduł zostanie automatycznie do niej dodany.

Jak tego użyć?

```
require 'moduł'
```

- moduł - nazwa modułu

Przykładem modułu jest moduł YAML, o którym pewnie kiedyś napiszę.

Na przykład
```
require 'YAML'
```

Wysoki profil systemowy

Wysoki profil systemowy powoduje, że dla systemu nasza gra jest ważniejsza od innych aplikacji. Dzięki temu nasz projekt będzie działał szybciej i w razie problemu system spróbuje je naprawić. Napiszę zaraz jak ustawić wysoki profil w systemie.

Lista profilów od najniższego:

1. niski
2. poniżej normalnego
3. normalny (domyślny)
4. Powyżej normalnego
5. wysoki
6. czasu rzeczywistego

Profil czasu systemowego pozwala działać grze nawet, gdy jej okno jest zamknięte, ale teraz nie napiszę jak go ustawić. Ustawianie wysokiego profilu systemowego:

```
@GetCurrentProcess=Win32API.new('kernel32','GetCurrentProcess',[],'i').call
@SetPriorityClass=Win32API.new('kernel32','SetPriorityClass',['p', 'i'],'i')
@SetPriorityClass.call(@GetCurrentProcess, 0x00000080)
```

A może teraz to wyjaśnię.
Podaję skrypt z komentarzami.

```
#Uruchomienie biblioteki systemowego zarządzania procesami i ustawienie procesu naszej gry do edycji.
@GetCurrentProcess=Win32API.new('kernel32','GetCurrentProcess',[],'i').call

#Ustawienie klasy profilowej w zarządzaniu procesami i stworzenie wzorca profillu wysokiego klasy pierwszej, zarejestrowanie klasy aktualnego profilu i przygotowanie procesu do zmiany, zapisanie głuwnego profilu, przygotowanie edytora.
@SetPriorityClass=Win32API.new('kernel32','SetPriorityClass',['p', 'i'],'i')

#Zmiana profilu dzięki kodowi instrukcyjnemu 0x00000080.
@SetPriorityClass.call(@GetCurrentProcess, 0x00000080)
```

Skrypcik ten wykonuje kilka kroków.

- Przygotowanie API systemowego.
- zapisanie nazwy procesu do zmiennej
- zapisanie kodu profilu do zmiennej.
- przygotowanie klasy profilowej
- Zmiana profilu

Aby ustawić inny profil zmieniamy kod 0x00000080 na kod innego profilu.
Poniżej przedstawiam ich liczbę.

- niski: 0x00000040
- Poniżej normalnego 0x00000050
- normalny 0x00000060

- Powyżej normalnego: 0x00000070
- wysoki: 0x00000080
- czasu rzeczywistego 0x00000100

Jak widać czym wyższy profil tym wyższy numer.
Aby stopniować czas rzeczywisty zamieniamy 1 w 0x00000100 na wyższą liczbę (1-5).

Skrypt ten korzysta z pewnej biblioteki, nie musimy jej jednak osobno pobierać, gdyż jest to biblioteka dołączana od ręki do systemu windows.

To tyle, mam nadzieję, że lekcja było zrozumiała.

Klasa each

Wyrwijmy się od tego zarządzania plikami i procesami i zajmijmy się czymś łatwym i innym. Mowa o klasie each.

```
class each
main
def main
end
end
```

Klasa each jest uruchamiana jeszcze przed poleceniem begin. Nie używa jej się do wykonywania jakichkolwiek akcji widzialnych. Używana jest ona do rejestrowania na przykład bibliotek i modułów. Podobnie jak inne klasy używa definicji. Możemy jednak w niej pisać poza definicjami:

```
class each
$rgss = "RGSS103J.dll"
main
def main
$zmienna = 0
end
end
```

Jak widać w obu przykładach aby wywołać w tej klasie definicję `main` trzeba to napisać. Spowodowane jest to tym, że ustawienie głównej definicji dla klas odbywa się dopiero podczas przetwarzania `begin`.

W klasie `each` **nigdy** nie odwołujemy się do innych klas na przykład `$scene = Scene_Title.new`. Po przejściu klasa automatycznie uruchomi `begin`. W klasie `each` nie stosuj warunków ani pętli.

Klasa ta służy wyłącznie do przygotowywania startu programu.
Tylko startu!

- inicjacji bibliotek
- initializacji
- dołączania modułów
- definiowania tytułu
- aktywacji interpretera

Za pewne nauczę was tego kiedy indziej.

Pętla while

Dotychczas poznałeś jeden rodzaj pętli: pętlę `loop do`. Czas na coś nowego. Zanim użyjesz tej pętli dobrze się zastanów, gdyż jej użycie może spowolnić grę, ale jest często niezbędne. Na przykład by ustalić działanie `$scene`.

Ruby w domyślnie powstałych skryptach dzięki tej pętli ustawia, że `$scene` jest zmienną klasy, a nie na przykład `$jakaszmiennaktoraniejestzmienna`. Pętla ta to tak zwana pętla warunkowa. Działa ona przez cały czas sprawdzając czy jakiś warunek się spełnił, a gdy się spełnił się wykonuje.

Jak jej użyć?

```
while warunek
#gdy warunek zostanie spełniony
else
#gdy warunek nie zostanie spełniony
end
```

Warunek to zwykły warunek jak po poleceniu `if` na przykład `while $zmienna == 0`

Wyjątkiem jest to, że w zwykłym warunku piszemy na przeczenie po prostu `=` , a w pętli `while !=`.

Jeśli chcemy, by pętla przestała działać oczywiście korzystamy z polecenia `break`.

Zmiana zmiennej klasy

W tej lekcji pokażę jak sprawić, by zmienną klasy była inna zmienna niż `$scene`. Najpierw jednak pamiętaj, że w chwili, kiedy to zrobisz wszystkie dotychczas napisane skrypty korzystające ze zmiennej `$scene` przestaną działać.

Możesz też sprawić, by dwie zmienne były zmiennymi klasy, na przykład `$scene` i `$class`. Pamiętaj, że zmienną klasy musi być scena globalna zaczynająca się od znaku `$`, a nie zmienna klasowa (`@@`) ani definicyjna.

No to przejdźmy do dzieła.

W skrypcie main znajdujemy:

```
while $scene != nil
$scene.main
end
```

Usuwamy to by napisać nasz własny skrypt.
Skorzystamy tutaj z pętli `while`

```
while $zmienna != nil
$zmienna.main
end
```

Jak widać przy pomocy tego skryptu można też zmienić domyślną definicję na przykład zamiast `main` pisząc: `definicjadomyslna`.

Siatki liczbowe

Co to jest? Siatka liczbowa to zmienna wymiarowa, czyli taka zmienna, do której można w nawiasie dodać wartość, na przykład:

```
$zmienna[1] = true
$zmienna[2] = false
```

Nie tworząc siatki nie można wywołać tego efektu. Aby założyć siatkę liczbową piszemy coś takiego:

```
siatka = []
```

Siatka może być za równo zmienną klasową, definicyjną jak i globalną. Kiedy karzemy wypisać wartość jednego obiektu siatki, na przykład `print ($siatka[5])` pojawi się jego wartość, ale kiedy spróbujemy zaprintować całą siatkę `print ($siatka)` pojawią się jej wszystkie wartości.

Przykładem siatki liczbowej, która jest używana w programie rpg maker jest siatka `$game_switches`, czy `$game_variablse`.

Siatki tekstowe

Siatka tekstowa lub też obiektowa, jak czasem może być nazywana, to rodzaj siatki, w którym zamiast liczb w nawiasach podaje się wyrazy po kropce.
Na przykład:

```
$game_temp.common_event_id = 0
```

Tworzenie takiej siatki jest trudniejsze, gdyż trzeba założyć do tego osobną klasę i na dodatek trzeba przewidzieć wszystkie wartości. No ale zamiast obsypywać teorią zabierzmy się do rzeczy ☺

```
class siatka attr_accessor :obiekt def initialize @obiekt =
wartosc end end
```

- siatka - nazwa siatki bez znaku początkowego, jak $. Dotyczy również nazwy klasy.
- nazwa obiektu pisana po kropce, po nazwie siatki.
- wartosc – wartość wstępna obiektu.

Jak widać by stworzyć tą siatkę trzeba przewidzieć wszystkie możliwe obiekty i ich początkowe wartości. Zauważ, że część polecenia siatki znajduje się poza definicjami. Klasa zawierająca siatkę może jednocześnie pełnić inne funkcje, na przykład wyświetlanie okienka.

Jeśli dodamy więcej wartości dodajemy ją u góry jak wartość obiekt, a następnie w definicji initializacji dodajemy jej wartość jak w przykładzie.

Pamiętaj, że dla klasy, która zawiera siatkę jej obiekty są zmiennymi instancji, jednakże nie każda zmienna instancji jest obiektem.

To wszystko, aby zmienić wartość obiektu siatki piszemy `$siatka.obiekt = "cos"`

Ładowanie siatek

Ładowanie siatek odbywa się za pomocą polecenia:

```
$zmienna = siatka.new
```

- zmienna - zmienna, której przypisujemy siatkę
- siatka - nazwa klasy siatki

Akcje wyciszające dźwięk

Nazwa chyba niezbyt trafna. Napiszę jak zrobić, by dźwięk stopniowo się ściszał. Mowa o poleceniu `fade`.

```
Audio.typ_fade(czas)
```

- typ: BGM, BGS, ME lub SE
- czas podany w dziwnej jednostce, 8000 to mniej więcej 2,5 sekundy.

Operacja spowoduje, że w trakcie trwania gry lub innej sceny dany rodzaj dźwięku stopniowo będzie coraz cichszy. Kiedy ucichnie się oczywiście skończy. Może to być używane podczas uruchamiania się gry.

Wartości czasowe powyżej 20000 są bardzo rzadko używane. Na przykład, gdy wybierzemy w grze stworzonej w rpg makerze dłuższy dźwięk potwierdzenia po wciśnięciu opcji wyjście zacznie on się szybko wyciszać `Audio.se_fade(800)`

To tyle, była to bardzo krótka lekcja.

Zakładanie folderów

I wróćmy do operacji na plikach. Mowa o zakładaniu folderów. Znowu korzystamy z polecenia `system`

```
system('md lokalizacja\folder')
```

- lokalizacja - ścieżka na przykład c:\, dla folderu z grą pozostaw to miejsce puste i usuń znak \ na końcu.
- folder - nazwa folderu

Czasem chcemy coś skopiować do na przykład folderu program files, możemy skorzystać z odnośników zaznaczonych. Oto one:

- %programfiles% - program files
- %appdata% - folder ustawień użytkownika
- %userprofile% - folder użytkownika. Dla pulpitu %userprofile%\desktop, a dla
- menu start "%userprofile%\menu start".

Kopiowanie i przenoszenie plików

Pamiętacie jak mówiłem, że polecenie *system* nie działa dokładnie tak, jak wiersz poleceń? Przykładem na to może być fakt, że nie działa tu polecenie copy ani move. Aby ich użyć tworzymy nowy plik na przykład copy.bat. Pamiętaj o końcówce .bat, nie .txt. Kliknij na nowy plik prawym przyciskiem myszy i wybierz: "edytuj".
Teraz napisz:

ten skrypt nie jest napisany w języku ruby, ale jest wykonywany przez ruby! *@echo off exit*

Pamiętaj, że jeśli korzystasz z tych funkcji powinieneś ustawić programowi profil czasu rzeczywistego! Pomiędzy poleceniami *@echo off*, a *exit* możesz pisać swoje skrypty.

Nie będę tutaj opisywał całego programowania wsadowego, gdyż nie tego dotyczy ten kurs. Nie będę tłumaczył więc działania pętli programu, napiszę tylko jak kopiować i przenosić pliki.

Kopiowanie plików:
Ten skrypt nie jest napisany w języku ruby, ale jest uruchamiany przez ten język *copy lokalizacja\plik lokalizacja\plik*

Przenoszenie plików:
Ten skrypt nie jest napisany w języku ruby, ale jest przez niego uruchamiany *move lokalizacja\plik lokalizacja\plik*

Domyślną lokalizacją jest ta z uruchamianym plikiem.
Aby skopiować wszystkie pliki napisz *.*, aby skopiować wszystkie pliki o danym rozszerzeniu napisz *.rozszerzenie (np. *.txt), aby skopiować wszystkie pliki o danej nazwie napisz nazwa.*, aby skopiować pliki zaczynającą się od danego słowa napisz słowo*.* (a*.txt) .

****3 grosze Reptile w tym temacie:**
Gwiazdka (*) tutaj symbolizuje „wszystko”. Szybki przykład: wyobraźmy sobie, że w folderze mamy następujące pliki:
Anna.txt || Adam.txt || Aneta.txt || Ace.txt || Ace.bat || Bartek.txt

Jak podamy polecenie copy *.txt, to program skopiuje nam wszystkie pliki zawierające rozszerzenie TXT (czyli: Anna.txt || Adam.txt || Aneta.txt || Ace.txt || Bartek.txt)

Jeżeli podamy polecenie copy a*.*, to program skopiuje nam wszystkie pliki zaczynające się na literę A oraz o obojętnie jakim rozszerzeniu (czyli skopiuje: Anna.txt || Adam.txt || Aneta.txt || Ace.txt || Ace.bat)

ROZDZIAŁ 3
Zarządzanie modułami, systemem i programem

Zajmiemy się tutaj zaawansowanymi rzeczami, jak moduły, biblioteki, czy klasyfikacje obiektowe. Jeżeli dobrze nie zrozumiałeś poprzednich części nie zabieraj się za tą. Skrypty zaznaczam kursywą.

W tym rozdziale poruszymy takie zagadnienia jak:

- jak obsługiwać moduły zewnętrzne i wewnętrzne
- jak działa polecenie puts i raise
- jak tworzyć procedury
- jak przekazywać zmienne przez definicje
- Jak używać pętli: until oraz for
- jak utworzyć klasę dziedziczną
- jak obsługiwać wyjątki
- jak konwertować zmienne

Pisanie po ekranie

Wcześniej poznaliśmy funkcje: `super` piszącą w okienku oraz `print` wyświetlającą osobne okno dialogowe. Funkcja `puts` wyświetla tekst poza okienkiem w pierwszym wolnym miejscu ekranu.

Podczas stosowania należy się upewnić czy tekst nie zasłania obrazków:

```
puts "tekst"
```

Jest to bardzo prosty skrypt i dlatego właśnie jego wybrałem na rozgrzewkę.

Zamiast "tekst" możemy wpisać zmienną, na przykład `puts $jakasstraszniedlugazmiennapokazujacazefunkcjaputsrozpoznajezmiennegdyichdlugoscjestnawettaduzajakdlugosc tejjednejzmiennnej`

Zmienna ta powinna być przypisana jakieś wartości, by na ekranie nie ukazał się napis: `nil`.

Polecenie: raise

Kiedy piszemy dłuższe skrypty i coś nie działa lub chcemy stworzyć diagnostykę używamy polecenia *raise*. Kiedy wystąpi błąd polecenie raise wskaże dział i linijkę oraz dokładny opis błędu.

```
raise ("Wystąpił nieistniejący błąd")
```

Kiedy program natrafi na tą instrukcje na przykład w dziale main w linijce 222 pojawi się następujący komunikat: wystąpił nieistniejący błąd, function raise in main, on line 222.

Po kliknięciu OK gra się wyłączy.

Funkcja ta przydaje się na przykład podczas wydawania wersji beta, by beta testerzy wiedzieli jaki błąd wystąpił i gdzie. wpisanie samego `raise` wskaże położenie błędu bez jego opisu. Podczas tworzenia gry zwłaszcza przydaje się zmiana w `main print` informujący o braku pliku na `raise` gdzie błąd wystąpił. Pamiętaj jednak później powrócić do dawnej wersji, by nie zamęczać gracza jakimiś napisami.

Moduły zewnętrzne

Moduły są chyba najbardziej złożoną i skomplikowaną rzeczą w ruby.
Moduły dzielimy na:

- zewnętrzne
- wewnętrzne

Chwilowo z tym drugim "damy sobie spokój" i zajmiemy się modułami zewnętrznymi. Moduł zewnętrzny nazywany również biblioteką modułową jest zwykle danymi, które pozwalają na dodawanie do aplikacji nowe polecenia, jak choćby pobieranie danych z internetu czy obsługa całej klawiatury.

Moduły mogą jednak też być klasami zapisanymi w osobnym pliku Tak na prawdę biblioteki modułowe to tylko fragmenty kodu zapisane w osobnych plikach. Możemy na przykład w module zapisać jakąś klasę, a po załadowaniu modułu będzie ona widoczna w grze. Ale dość teorii. Jak załadować moduł?

Moduł musi być skopiowany do folderu z grą. Pamiętaj, że wiele modułów posiada więcej niż jeden plik. Moduły możesz pobrać z różnych miejsc. Zastanawiałem się, czy na twierdzy nie dodać działu właśnie z modułami. A teraz do roboty, gdyż tej teorii dla wielu osób może być zbyt dużo.

```
Ustawienie folderu danego:
$LOAD_PATH << "."
#Załadowanie modułu
require 'moduł'
```

- moduł - nazwa modułu nie pliku lub początek nazwy pliku bez rozszerzenia

A teraz wyjaśnienie.
Zmienna `$LOAD_PATH` jest zmienną jedyną w swoim rodzaju i trudno wyjaśnić jak działa.
Znak `<<` nadał jej wartość binarną, nie tekst ani liczbę.
Nie można tego zrobić w stosunku do innych zmiennych, chyba, że również są zmiennymi binarnymi.
Używając znaków: `"."` nadaliśmy jej wartość folderu z grą.
Kiedy wpiszemy na przykład
`"C:\"` zmienna będzie wyszukiwać modułów w głównym folderze dysku C.

Edytując tą zmienną możemy wszystkie moduły umieścić w jednym folderze na przykład modules:

```
$LOAD_PATH << "./modules"
```

Tworzenie modułów zewnętrznych

Aby stworzyć moduł zewnętrzny należy po prostu założyć plik w notatniku o nazwie dwuczłonowej:

- nazwamodułu
- .rb

Czyli jeśli zakładasz moduł o nazwie test, nazwa pliku musi brzmieć: "test.rb"
Następnie importujesz moduł funkcją `require`

Moduły wewnętrzne

Moduły wewnętrzne są czymś zupełnie innym niż zewnętrzne, chociaż mają elementy wspólne. Moduł wewnętrzny wygląda tak, jak zewnętrzny z tą różnicą, że jest osadzony w samym skrypcie.

Moduł ten działa podobnie do klasy, ale nie może dziedziczyć (patrz. dziedziczenie), ale może być częściowo wykorzystywany w innych klasach.

```
#tworzenie modułu
module nazwa
#zamknięcie modułu
end
#Otwarcie modułu
include nazwa
```

- nazwa - nazwa modułu

Jak widać moduły otwieramy funkcją `include`, a tworzymy `module`

Funkcja rescue

Funkcja rescue to funkcja obsługi wyjątków. Wyjątki mówią programowi co zrobić, gdy akcja się nie powiedzie.

```
class Scene_Test
def main
file = file.open('plik', 'r')
rescue
print("plik nie został znaleziony")
end
file.close
end
end
```

Dzięki temu zamiast angielskich i dziwnych komunikatów pokaże się jasny i polski komunikat, po czym gra zostanie zamknięta. Podczas przetwarzania funkcji `rescue` nie możemy robić nic poza poniżej wypisanymi poleceniami i możliwościami:

- używanie wewnętrznych podfunkcji `rescue`
- *print*
- *raise*
- zmiana wartości zmiennych
- warunki
- pętla *while*

A co to są podfunkcje `require`? Są to dodatkowe polecenia, na potrzeby tego kursu wybrałem:

- retry - próbuje ponownie przetworzyć klasę, jeśli zmienna $scene zostanie zmioniona przetwarza inną klasę

Przekazywanie zmiennych poprzez definicje

```
class cos
def main
pierwsza
end
def pierwsza
druga("jakiestraszniedlogiezdaniektorepokazesiewdefinicjimimoi
nnegoprzekazuzmiennejnadodatekbezodstepowchociazmoznaichuzyc")
end
def druga(x)
print (x) end
end
```

Przekazaliśmy zmienną definicyjną z innej definicji?
Zrobiliśmy to podając zmienną w nawiasie.
A można w ten sposób przekazać więcej zmiennych?

```
class cos
def main
pierwsza
end
def pierwsza
druga(1, 2, 3, 4, 5, 6, 7, 8, 9)
end
def druga(a, b, c, d, e, f, g, h, i, j)
print (a, b, c, d)
print(e, f, g, h)
print(i, j) end
end
```

W tym przypadku przekazaliśmy aż dziesięć zmiennych.
Możemy też przekazywać inne rodzaje zmiennych, ale poza instancjami nie widzę sensu ich przekazywania, gdyż równie dobrze mogę zapisać:

```
$a = 1
$b = 2
$c = 3
#i.t.d.
druga
```

Konwersja zmiennych

Konwersja zmiennych jest strasznie prosta. Nie omawiałem jednak jej wcześniej, gdyż jeszcze nigdy mi się nie przydała i nie do końca wiem komu mogłaby się przydać w programie, którego nazwa nie brzmi na przykład: "kalkulator". No to o konwersji.

```
$a = 111
$b = "444"
$a.to_S
$b.to_U
print ($a * 2, $b * 2)
```

Aż dziwne!
Zmienna, która była tekstową zmieniła się w liczbową i mogliśmy na niej wykonywać operacje matematyczne. Natomiast druga zmienna stała się z liczbowej tekstową i nie można było jej mnożyć.

- zmienna.to_S - zmienia zmienne liczbowe i potwierdzające na tekstowe
- zmienna.to_U - zmienia zmienne tekstowe z liczbami na zmienne liczbowe

I to już wszystko, łatwe, prawda?

Dziedziczenie klas i ich definicji

Dziedziczenie klasy pozwala klasie niższej przejmowanie definicji z klasy wyższej. No to może jakiś przykład?

```
class Wyzsza
def main
$scene = nizsza.new
end
public
def powrot
$scene = Wyzsza.new
end
end
class Nizsza < Wyzsza
def main
powrot
end
end
```

Jak widać klasa `Nizsza` skorzystała z definicji klasy `Wyzsza` przez to, że ustaliliśmy dziedziczenie znakiem `<`

Polecenie `public` oznacza, że wszystkie definicje poniżej będą mogły być dziedziczone. Nie zapisaliśmy go przed definicją główną, by nie połączyła się z definicjami o tej samej nazwie w następnej klasie. Czy po za `public` istnieje więcej takich rodzajów?
Tak.

- *public* - publiczna definicja może być użyta w każdej klasie połączonej z daną, zarówno podrzędnej, jak i nadrzędnej
- *protected - może być użyta w klasie podrzędnej*
- *private - może być użyta tylko w danej klasie*

Pętla for

A oto kolejna pętla: `for`. Pętla ta po prostu liczy ile razy ma przejść. Używana jest, gdy na przykład dana akcja ma być wykonana, mhm, 1000000 razy.

```
for identyfikator in min..max
#polecenia pętli
end
```

- identyfikator pętli (domyślnie i) to unikalny identyfikator (zwykle litera) używany tylko raz w definicji.
- miin - wartość lub zmienna, do której pętla będzie odliczana (zalecane 0)
- max - wartość liczbowa lub zmienna, od której pętla będzie liczona.

Na przykład, gdy min=0, a max=4 pętla przejdzie pięć razy, gdyż cztery razy były wpisane w pozycji max i jeden, by wartość pętli była mniejsza niż zero.

Pętla until

Pętla `until` jest też czasem nazywana pętlą odwróconego warunku. Jest ona odwrotnością wcześniej poznanej przez ciebie (mam nadzieję) pętli `while`. Ma ona tą samo konstrukcje, ale zamiast `while` piszemy `until` i działa trochę inaczej. Pętla ta bowiem sprawdza, czy warunek nie jest spełniony:

```
$zmienna = 25 while $zmienna != 0 $zmienna = $zmienna - 1 end
until $zmienna == 0 $zmienna = $zmienna - 1 end
```

Pętle te działają identycznie, ale dlaczego?
Przecież warunki są odwrotne!

Pętla `until` w przeciwieństwie do `while` sprawdza, czy warunek **nie** jest spełniony i wykonuje się tylko wtedy.

Tak sobie myślę, że powinienem ją opisać w poprzednim rozdziale kursu po pętli while, ale po prostu zapomniałem, więc zostanie ona już tutaj.

Procedury

Procedura to coś w stylu klasy, ale po jej wykonaniu akcja trwa dalej (**przykład tylko dla rpg makera xp).**

```
procedura = proc do
print ("procedura się wykonuje")
end
class procedura
def main
print("aby wykonać procedurę wciśnij enter, a by zamknąć ten
program escape.")
loop do
Input.update
Graphics.update
update
if $scene != self
break
end
end
end
def update
if Input.trigger?(Input::B) $scene = nil
end
if Input.trigger?(Input::C)
procedura.call #=> procedura
end
end
end
```

- procedura - unikalna nazwa procedury

Jak widać procedury tworzymy poleceniem nazwaprocedury = proc do, a kończymy blok standardowo poleceniem end. Natomiast wywołujemy je pisząc nazwaprocedury.call #=> nazwa procedury. Polecenie proc możemy zamienić poleceniem lambda, zmieniając troszkę sposób zapisu procedury:

```
procedura = lambda
{
#polecenia procedury
}
```

Błędy

Tutaj opiszę błędy, które mogą wystąpić po interpretacji (lub podczas) skryptu.

Script is changing:
Skrypt się zbyt długo wykonuje. Jest to najbardziej denerwujący błąd.
Można go próbować usunąć poleceniem Graphics.update, ale nie zawsze się to udaje.

syntags error
Błąd skryptu, na przykład literówka

... undefined for nil, nil class
Tej akcji nie możesz wywołać poza klasą.

No such file or directory
Brakuje jakiegoś pliku

Tych błędów może być więcej, ale te trzy najczęściej występują, a jak ktoś zna angielski bez trudu będzie mógł przetłumaczyć resztę, a ja teraz szczerze mówiąc nie chcę się bawić w pisanie błędnych skryptów, by obejrzeć możliwe błędy ☺

ROZDZIAŁ 4
Biblioteki DLL, oraz treści zaawansowane

Nie planowałem napisania czwartego rozdziału tego kursu, ale jakoś tak się złożyło. Nie zdążyłem bowiem napisać o niektórych zagadnieniach, jak np:

- bibliotekach dll
- obsłudze daty
- obsłudze połączeń internetowych
- Zwróceniach wartości definicji
- i innych

Ale za to uda nam się poruszyć dość inne ciekawe zagadnienia jak:

- obsługa pełnej klawiatury
- obsługa myszki
- pełny ekran
- zamrażanie obiektów

Zatem nie tracąc więcej czasu, przejdźmy do roboty ☺

Przetwarzanie czasu

W większości kursów ten temat jest jednym z pierwszym, w tym przypadku jest inaczej. Do przetwarzania daty służy o trudnej do przetłumaczenia nazwie: `date` ☺ .

Zobaczmy jak ona działa:

```
date.at(czas)
```

- czas - liczba sekund od dnia 01 I 1970

Kiedy na przykład jako czas wpiszemy "0" i zaprintujemy datę zobaczymy napis:
"01 JAN 1970, 00:00:00"

A jak sprawdzić jaka teraz jest data i godzina? Jest kilka sposobów, ja pokażę ten najprostszy. Opiera się on na pliku tymczasowym.

```
temporary = File.open("temp", "w") $data = temporary.mtime
temporary.close system('del temporary') system('exit')
```

W tym przykładzie stworzyliśmy plik tymczasowy i sprawdziliśmy jego datę modyfikacji poprzesz funkcję `mtime.$date`.

Biblioteki DLL

Czas na ten poruszany chyba od pierwszej części kursu temat. Czy pamiętacie, jak przy okazji modułów mówiłem, że są chyba najbardziej złożoną funkcją ruby? Jednak się myliłem, gdyż zapomniałem o bibliotekach.

Pozwalają one grom stworzonym w programie RPG MAKER XP osiągnąć poziom największych, firmowych produkcji, o ile oczywiście się postaramy. Biblioteki DLL to zbiór poleceń napisanych w innych językach programowania - zwykle w C++ -, które pozwalają na niedostępne w inny sposób korzystanie na przykład z myszy. Teraz napiszę jak użyć biblioteki DLL.

```
win32api.new("rodzaj wywołania", "wywołanie", "parametr
górny", "parametr dolny")
e. >
```

- rodzaj wywołania - którego typu systemowego używamy, więcej niżej. Najczęściej używane to `user32` i `kernel32`
- wywołanie - nazwa wywoływanej funkcji

- parametr górny - pierwszy parametr
- parametr dolny - drugi parametr

I tu zaczyna się problem. O ile z rodzajem wywołania i wywołaniem zwykle nie ma problemu, to jaki wybrać parametr górny i dolny? Czasami parametry te są nieważne i możemy tu wpisać cokolwiek, ale nie zawsze. Rodzaj wywołania to sposób w jaki wywołamy bibliotekę. Najczęstsze to:

- user32 - pobranie danych poprzez domyślną bibliotekę systemowa
- kernel32 - wywołanie zewnętrznej biblioteki

No to może jakiś przykład?
Wyobraźmy sobie, że mamy bibliotekę o nazwie "biblioteka", która ma funkcję "set" służącą do ustawienia kolorów ekranu: tła i tekstu. Kolor tła to pierwszy parametr, natomiast tekstu drugi, a kolory mamy wpisać po angielsku.

```
win32api.new("kernel32", "biblioteka", "white", "black")
```

Gdzie powinna się znajdować biblioteka?
Może być w dwóch miejscach: w folderze z grą i zarejestrowana w systemie. Rejestracji w systemie może dokonać na przykład instalator. Na rejestracje jeśli się nie mylę są trzy sposoby.

1. rejestracja bezpośrednio z poleceń ruby:
   system('regsvr32.exe "plik" /s /q /n /i /u')

   system('exit')
2. 2. rejestracja ręczna:
   Otwórz menu start \ programy \ akcesoria \ wiersz poleceń i wpisz powyższy kod pomijając system(' oraz ').
3. 3. Poprzez plik wsadowy:
   Stwórz nowy plik o dowolnej nazwie o końcówce .bat.
   I wpisz do niego powyższe polecenie pomijając napisy system(' oraz ')

Napisy */s /q /n /i /u* wykluczają się nawzajem i wpisujemy tylko te, które chcemy.
W poleceniu zawsze musi wystąpić polecenie /i lub /u:
/i - instalacja
/u - deinstalacja

Pozostałe trzy są opcjonalne. Jedyne, które ci się przyda to /q , które nie informuje o przebiegu instalacji okienkową wiadomością.

Jeśli chcesz poznać działanie pozostałych, w wierszu poleceń wpisz po prostu regsvr32.exe
Ale długi był ten rozdział ☺, ale był bardzo ważny i do zawartej w nim wiedzy będę się często odnosił w przyszłości.

Obsługa całej klawiatury

To czas wykonać jakąś operację na bibliotece DLL. Podobnie, jak w przypadku profili użyjemy tu domyślnej biblioteki systemowej, więc nie musimy nic pobierać.
No to może pokażę jak to zrobić?

```
GetKeyState =
Win32API.new("user32","GetAsyncKeyState",['i'],'i')
if GetKeyState.call(identyfikator) & 0x01 == 1
end
```

- identyfikator - specjalny kod klawisza.

I zaczynają się kolejne problemy, gdyż każdy klawisz ma swój liczbowy identyfikator, który nie dość, że jest. Często Nic nie mówiący, to jeszcze zapisany w systemie szesnastkowym, jak większość instrukcji przekazywanych do bibliotek.
No więc jakie są te kody?

Podam najważniejsze:
- enter - 0x0
- shift - 0x10
- lewy control - 0xA2
- prawy control - 0xA3
- lewy alt - 0xA4
- prawy alt - 0xA5
- backspace - 0x08
- tab - 0x09
- dowolny control - 0x11
- dowolny alt - 0x12
- escape - 0x1B
- spacja - 0x20

Jeśli chodzi o litery i cyfry, to odpowiadają im liczby z kodu ASCII. Oznacza to, że możesz to sprawdzić nawet w wordzie.
- 0x41 - a
- 0x42 - b
- 0x43 - c
- 0x44 - d
- i.t.d.

Pamiętaj jednak, że kody są w zapisie szesnastkowym, więc po liczbie 39 nie następuje 40, a 4A.
- 0 - 0
- 1 - 1
- 2 - 2
- 3 - 3
- 4 - 4
- 5 - 5

- 6 - 6
- 7 - 7
- 8 - 8
- 9 - 9
- 10 - A
- 11 - B
- 12 - C
- 13 - D
- 14 - E
- 15 - F
- 17 - 10
- 18 - 11
- 19 - 12
- 20 - 13

I tak dalej.

Jeśli chodzi o cyfry, to zaczynają się one od 30, więc przykładowo ósemka ma kod 0x38. Kodów strzałek nie podaję, gdyż w RGSS są dostępne już procedury ich obsługi. Pełną listę znaków można znaleźć w wielu miejscach, więc jej nie przepisuję. Została jeszcze tylko jedna kwestia: Do momentu wciśnięcia następnego klawisza zmienna `GetKeyState.call` będzie przechowywała stary klawisz. W tym celu radzę ją zmazać poleceniem `GetKeyState = nil`.
A oto mały przykład:

```
GetKeyState =
Win32API.new("user32","GetAsyncKeyState",['i'],'i')
if GetKeyState.call(0x1B) & 0x01 == 1
print("Nacisnąłeś escape, czyli klawisz o identyfikatorze
0x1B. Z tego powodu się wyłączam.")
$scene = nil
end
```

Obsługa myszy

Czas na ten łatwo przyswajalny, chodź trudny do wykorzystania temat. Mowa o obsłudze myszy. Dane które powinieneś znać to identyfikatory przycisków myszy. Ich wciśnięcie sprawdza się opisanym wcześniej sposobem.

- lewy przycisk myszy - 0x01
- środkowy przycisk myszy - 0x04
- Prawy przycisk myszy - 0x02

Istnieje też coś takiego, jak czwarty i piąty przycisk myszy.
Szczerze mówiąc nie wiem co to jest, ale powiem, że ich identyfikatory to 0x05 i 0x06

Można sprawdzić, ale teraz ważniejsze jest pisanie kursu tym bardziej, że nie widziałem jeszcze pięcio-przyciskowej myszy, chodź może któregoś nie zauważyłem. A teraz sprawdźmy jak otrzymać położenie wskaźnika.

```
$getCursorPos = Win32API.new("user32", "GetCursorPos", ['P'],
'V')
```

Oczywiście jak to ma w zwyczaju biblioteka winapi dane są zapisane w sposób binarny i trzeba je rozszyfrować.

```
$mysz_x = get_pos('x')
$myszka_y = get_pos('y')
```

A teraz przepiszę zmodyfikowany fragment pewnego skryptu by rozróżnić, czy mowa o danej x, czy y.

```
lpPoint = " " * 8 # store two LONGs
$getCursorPos.Call(lpPoint)
x,y = lpPoint.unpack("LL") # get the actual values
if coord_type == 'x'
return x
elsif coord_type == 'y'
return y
end
```

To wszystko powinno się zamknąć w jednej klasie w różnych definicjach. To chwilowo tyle, dodam tylko, że na pełnym ekranie jedno pole równa się ósmemu ilorazowi z właściwej współrzędnej.

Pełny erkan

```
$showm = Win32API.new 'user32', 'keybd_event', %w(l l l l), ''
$showm.call(18,0,0,0)
$showm.call(13,0,0,0)
$showm.call(13,0,2,0)
$showm.call(18,0,2,0)
```

Skrypt jest bardzo prosty. Po prostu zwiększa okno w zależności od rozdzielczości (zmienna `$shown.call`). Teraz jeszcze jedno. Kiedy powrócimy do klasy, gdzie powiększył się ekran, to ponownie się on zmniejszy, a tego nie chcemy.

Przykładowo gdyby dochodziło do powiększenia się ekranu w menu, ten by się pomniejszał po powrocie do niego, a gdybyśmy go powiększyli w begin'ie dochodziło by do zmniejszenia po wciśnięciu F12 Dlatego w swoim projekcie - grze audio z obsługą myszy - napisałem coś takiego:

```
if $shown == nil
$showm = Win32API.new 'user32', 'keybd_event', %w(l l l l), ''
$showm.call(18,0,0,0)
$showm.call(13,0,0,0)
$showm.call(13,0,2,0)
$showm.call(18,0,2,0)
end
```

Myślę, że skryptu nie muszę omawiać, więc to tyle na ten temat.

Wszystko o poleceniu return

Czy pamiętacie jeszcze polecenie *return*? Na końcu poprzedniej części zalecałem wam się pobawić jego kosztem ☺.Jak ktoś jeszcze nie zrozumiał jak ono działa, to wyjaśniam.
Zobaczmy ten przykład:

```
def definicja(liczba, znak, wyraz)
print("przypiszmy wartości i skończmy z tą definicją")
return(1232655345656, "h", "kurs")
end
def ain
definicja
print(definicja)
end
```

Zatem polecenie return zwraca wartości dla zmiennych przypisanych definicji i pozwala na ich dalszy odczyt.

Definicja powróciła do definicji *main*
liczba = 1232655345656
znak = "h"
wyraz = "kurs"

Prawda, że często przyspiesza wiele operacji?
Dobrze, to tyle na ten temat.

Odtwarzanie filmów – pliki *.avi

Było o dźwiękach i grafice, więc czas na filmy. Oczywiści skorzystamy z biblioteki systemowej zawsze chętnej w udzielaniu programistom wszelkiej pomocy mimo wielu błędów w skryptach i ciekawych zjawisk, jak bluescreenów.

```
@wnd = Win32API.new('user32','FindWindowEx','%w(l,l,p,p)','L')
@temp = @wnd.call(0,0,nil,tytul).to_s
movie = Win32API.new('winmm','mciSendString','%w(p,p,l,l)','V')
movie.call("open \""+film+"\" alias FILE style 1073741824 parent " + @temp.to_s,0,0,0)
@message = Win32API.new('user32','SendMessage','%w(l,l,l,l)','V')
@detector = Win32API.new('user32','GetSystemMetrics','%w(l)','L')
```

- tytul - tytuł okna z filmem
- film - nazwa pliku z filmem

A teraz jakieś wyjaśnienia?
Programiści z C++ już spotkali się z parametrem HWND, w ruby to tylko WND. Jest to tak zwany uchwyt naszego okna z filmem. Ładujemy okno, wyświetlamy film i dzięki detektorowi sprawdzamy poprawność danych. Ktoś napisał skrypt pozwalający na odtwarzanie filmów i polecam jego użycie zamiast ciągłego pisania kodu ręcznie.

Prawda, że proste? I kto by pomyślał, że w rpg makerze xp można odtwarzać filmy?

Instrukcja case

Jest to zamiennik dla instrukcji `if`.
Te dwa skrypty działają identycznie:

skrypt pierwszy

```
if $zzmienna 00 9
Audio.se_play("Audio/SE/0.mp3", 100, 100)
end
if $zmienna == 1
print("wystąpił jakiś błąd")
end
if $zmienna == 3
$scene = nil
end
```

skrypt drugi

```
case $zmienna
when 0
Audio.se_play("Audio/SE/0.mpc", 100, 100)
when 1
print("Wystąpił błąd")
when 2
$scene = nil
end
```

Czy już wiesz jak działa instrukcja `case`? To skrócony warunek. Więcej objaśnień chyba nie potrzeba poza tym, że po `when` można też używać ciągów znaków w cudzysłowie i takich zwrotów, jak na przykład `nil`.

Z instrukcją `case` już się spotkałeś w części pierwszej przy okienkach z kursorem.

Aliasowanie

Co to jest alias? Jest to jakiś skrót.
Przykładowo kiedy mamy adres mailowy dawidpieper@o2.pl możemy użyć aliasów, by po wysłaniu wiadomości na adres a@b.c doszła ona do nas. Podobnie jest z adresami IP, stronami internetowymi. Tutaj jednak nazwa brzmi inaczej: domeny.

Przykładowo adres strony twierdzy jest inny, ale dzięki domenie można na nią wejść pisząc po prostu: rpgmaker.pl

W ruby'm zastosowanie aliasów jest troszeczkę inne. Przydają się one gdy piszemy własny skrypt. Może się zdarzyć, że chcemy dopisać coś do definicji w jakieś klasie. Czy to oznacza, że całą klasę trzeba przepisać na nowo? Oczywiście nie.

A może definicję? Jak wiemy na czym polega aliasowanie i to nie jest potrzebne.
Może jakiś przykład?

```
class Scene_Test
def main
updat
end
def update
loop do
if Input.trigger?(Input::C)
```

```
$scene = nil
end
end
end
end
#teraz coś co piszemy w naszym skrypcie:
class Scene_Test
alias pajper update
def update
pajper
if Input.trigger?(Input::B)
print("Wcisnąłeś escape lub x.")
end
end
end
```

Aliasy dodają coś do wcześniej zapisanych definicji.
Teraz po uruchomieniu klasy `Scene_Test` obsługiwany będzie również klawisz escape. Napis pajper był unikalną nazwą aliasu, a update nazwą definicji, na którą alias wskazuje. Jeśli w definicji nazwę aliasu wstawimy na początku nasz skrypt będzie dodany na dole, jeśli u góry nasz skrypt pojawi się nad definicją.

W ten sposób możemy coś dodać zarówno na dole, jak i u góry definicji.
To tyle na ten temat.

Zamrażanie obiektów

Zamrożenie obiektu sprawia, ze nie można go edytować. Kiedy zabezpieczymy w ten sposób jakaś definicje, aliasy nie będą na nią działać. Jeśli zamrozimy całą klasę, stanie się to z nią. Możemy również zabezpieczyć warunek, bibliotekę i inne obiekty.
Robimy to poleceniem `freeze`

```
class Scene_Test
def main
definicja
end
def definicja
main.freeze
$scene = Scene_Ultimate.new
end
end
class Scene_ultimate
def main
$scene = nil
end
end
```

Aliasowanie sceny test już nie jest możliwe. Aby zamrozić całą scenę w scenie ultimate napisz:

```
Scene_Test.freeze
```

A jak zamrozić zmienną, by nikt jej nam nie zmienił?

```
$zmienna = 0
$zmienna.freeze
end
```

Zamrożoną zmienną nazywamy stałą, gdyż zmiana jej wartości jest niemożliwa. Ustalanie stałych często może pomóc w wyszukiwaniu błędów.

A jak zamrozić warunek?
Jest to rzadko robione, jednak niesie to pewne korzyści.

```
$zmiennaprzetrzymujacawarunek = if $zmienna == 0
print("cos")
end
```

Teraz nikt nie zmodyfikuje następnymi skryptami naszego warunku, a zmienna `$zmiennaprzetrzymajacawarunek` wciąż na niego wskazuje.

Uwaga! Zamrożenie obiektu jest czynnością nieodwracalną.

Warunki jednoinstrukcyjne

Warunki jednoinstrukcyjne to takie warunki, które mają jedną instrukcje. Zamiast zapisywania warunku w trzech linijkach można to zrobić w jednej.

```
if($zmienna == 0)
print "wiadomość";
```

W ruby'm jednak to nie zadziała. ten skrypt w ruby napiszemy tak,
uwaga! w jednej linijce:

```
print ("wiadomość") if $zmienna == 0
```

Lista komend

Przedstawiam tutaj listę ciekawych komend, których możemy używać dzięki poleceniu *system*

Pamiętaj wszystko kończyć poleceniem *system('exit')*

- copy plikzrodlowy plik docelowy
  kopiuje plik., dopisz po copy /q, by pliki były zamieniane

- move plikzrodlowy plik docelowy
  przenosi plik- dopisz /q, aby pliki były automatycznie nadpisywane.

- del /q plik
  usuwa plik

- pause
  przerywa działanie aż do wciśnięcia dowolnego klawisza

- ren stara nowa
  zmienia nazwę pliku.

- if
  instrukcja warunkowa, bez end, polecenia pisane w nawiasach.

- md nazwa
  zakłada katalog

- cd ścieżka
  przejdź do innego katalogu.

- cmd
  wyświetl okienko z konsolą.

- start plik
  uruchom plik.

- print plik
  otwiera menu wyboru drukarki i drukuje dany plik. Przydatne, gdy wyświetlamy na przykład bestiariusz i chcemy dodać możliwość jego wydruku.

- shutdown
  wyłącza komputer

Oczywiście tych poleceń jest wiele więcej, te są najczęściej używane.

Podsumowanie

To koniec podstawowego kursu RGSS, chodź taki podstawowy to on nie jest. Posiadłeś tutaj wiedzę, którą wiele nowszych skrypterów nie może się pochwalić. Chciałem pisać następne części, ale jest to już na prawdę *zaawansowana* wiedza, jak wyświetlanie rozbudowanych okien dialogowych, czy dodawanie paska menu. Dlatego tym się teraz nie zajmę. Może napiszę następny kurs o nazwie: "*zaawansowany kurs języka RGSS*"?
Tym czasem zajmę się pewnie rpg makerem vx.

Na chwilę obecną podsumuję ten cały kurs: cztery rozdziały. Najpierw poznaliśmy podstawowe założenia języka i jego sposób wykonania. Potem napisałem coś o naszym środowisku pracy i napisaliśmy pierwszy program. Każdy program składa się z części głównej, zapisanej pomiędzy ramami *begin* i *end* Po za tym istnieją klasy, znajdujące się pomiędzy ramami *class nazwa* oraz *end*
Klasę zmieniamy zmienną $scene zapisując $scene = klasa.definicja. Definicja jest to grupa poleceń podklasy. Domyślną definicją wywołaną dla polecenia klasa.new jest definicja main. Potem napisałem o zmiennych. Dzielimy je na zmienne klasy, definicji, globalne i instancyjne. Różnią się one znakiem i zasięgiem działania.

Na przykład zmienne globalne zaczynane od znaku $ działają w całej aplikacji, a zmienne klasy działają w klasie i zaczynają się od znaków @@. Zmienne instancji (@) działają w całym obiekcie, w którym zostały utworzone (w klasie, definicji, warunku itd.), a klasy definicyjne nie zaczynają się od żadnego znaku i działają w całej definicji.

Później napisałem o wyświetlaniu tekstu. Dokonujemy tego pisząc *print ("Tekst")* Wyświetli się wtedy osobne okienko dialogowe z wiadomością. Potem napisałem o tworzeniu okienek, obsłudze klawiatury, kursorze, dźwięku i okienkach z kursorem.
Na tym zakończyliśmy pierwszy rozdział

W drugiej części zajęliśmy się plikami. Napisałem o otwieraniu, przenoszeniu, kopiowaniu, edycji plików. Poznałeś też pętle while, drugą, gdyż w pierwszej części poznałeś pętlę loop do. Potem pokazałem jak obsługiwać siatki, czyli grupy zmiennych połączone tak zwanym wskaźnikiem, na przykład $siatka[1] $siatka[2] $siatka[3] $siatka[4] itd.

Potem opowiedziałem również o siatkach tekstowych i zmianę profilu. Na koniec powiedziałem coś o wyciszaniu muzyki. Na tym koniec rozdziału drugiego.

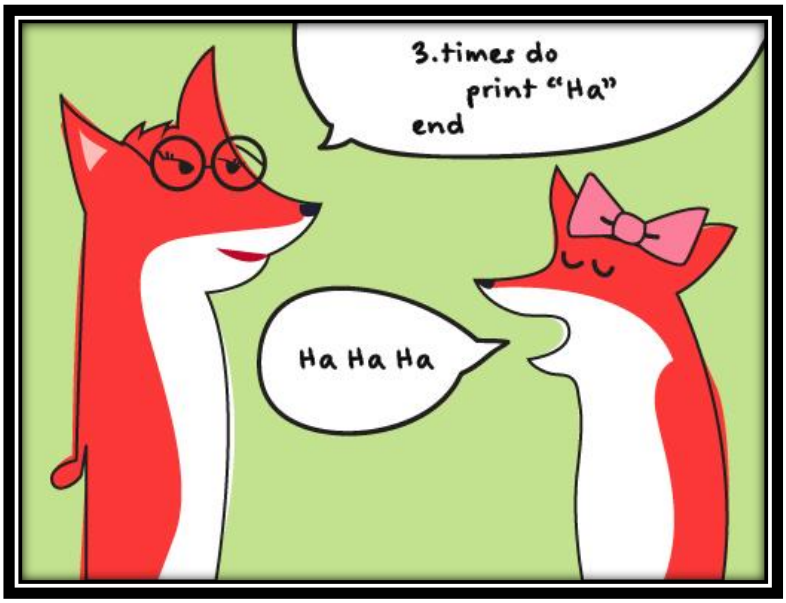
W części trzeciej skupiłem się na modułach, poświęcając im trzy lekcje. Moduły dzielimy na zewnętrzne - osadzone w osobnych plikach - i wewnętrzne - zapisane w samej aplikacji. Opisałem też takie rzeczy, jak konwersja zmiennych, obsługę wyjątków i błędów. Potem skupiłem się na procedurach - czyli często powtarzanych czynnościach definiowanych raz

i używanych wielokrotnie dzięki jednej linijce kodu. Na koniec poznałeś dwie nowe pętle: until oraz for. W tej części kursu opisałem głównie biblioteki DLL.

Skupiłem się na używaniu ich i takich czynnościach, jak obsłudze klawiatury, myszy, danych przekazywanych definicjami, wyświetlaniu filmów, obsłudze pełnego ekranu... Potem opisałem coś jeszcze o siatkach i o zamrażaniu obiektów i ich aliasowaniu.
Na tym skończył się właściwie kurs języka ruby game scripting system.

To już koniec podstawowego kursu RGSS z mojej strony. Mam nadzieję, że dzięki temu Poradnikowi, udało mi się chociaż odrobinę przybliżyć Tobie jezyk ruby w RPG Makerze XP, a edytowanie skryptów, czy pisanie swoich (jakiś prostych) nie sprawi Ci problemów ☺
Do zobaczenia, do następnego razu!

Nie zapominaj odwiedzać nas na stronie Twierdzy RPG Makera, gdzie znajdziesz wiele materiałów, porad dotyczącą świetnego programu jakim jest RPG Maker ☺



Poradnik RGSS (RPG Maker XP) – wprowadzenie do języka ruby
Autor: Dawid „Pajper" Pieper (dawidpieper@o2.pl)
Korekta, poprawki: Reptile (reptile@o2.pl)

www.rpgmaker.pl