Plezuro

Wydanie 1.0

Piotr Sroczkowski

2015-12-13

Plezuro to język skryptowy

Główna wersja jest kompilowana do Javascripta.

```
$f = {this + first * 2};
$y = f(2, f(5, 9));
[y, y]
```

ROZDZIAŁ 1

Przewodnik

To jest przewodnik po języku programowania Plezuro.

Dla kogo to jest?

Jeśli jesteś programistą, który zna już inny język programowania (np. C, Java, Python...), ten przewodnik przyniesie ci wiedzę potrzebną do tworzenia aplikacji w Plezuro. Jednakże w technologiach programistycznych najważniejsze jest doświadczenie. Zatem po prostu próbuj pisać jak najwięcej w Plezuro.

Jeśli programowanie jest czymś nowym dla ciebie, ten poradnik powinien być zrozumiały dla ciebie, ponieważ wyjaśnia on, jak wszystko działa. Jeśli jednak nie rozumiesz tego, być może najpierw powinieneś nauczyć się podstaw Javascripta (ten język jest najbardziej zbliżony do Plezuro) lub ewentualnie nauczyć się jeszcze innego języka programowania.

Inne przydatne materiały

Jeśli chcesz, nauczyć się więcej, powinieneś przeczytać specyfikację Plezuro. Jest ona napisana w niezbyt skomplikowany sposób. To po prostu spojrzenie z innej strony. Możesz przeglądać skrypty Plezuro na Githubie i próbować napisać coś podobnego.

1.1 O Plezuro

1.1.1 Pochodzenie

Skąd się wzięło Plezuro? Otóż jego autor nie był w pełni usatysfakcjonowany żadnym z dotychczas istniejących języków programowania. Zatem pomyślał o stworzeniu nowego. Nazwa pochodzi z Esperanto i znaczy ‘przyjemność’.

Przydatne materiały

Na Githubie

https://github.com/oprogramador/plezuro_js

Oficjalna strona internetowa:

https://plezuro.herokuapp.com

1.1.2 Podstawowe zasady

Plezuro to imperatywny, obiektowy, funkcyjny, proceduralny i refleksyjny język programowania

Główne założenia to:

1. Kod powinien być możliwie krótki.
2. Bardzo prosta składnia.
3. Kod powinien być łatwy do przeczytania dla początkującego programisty.
4. Potęga języka powinna być oparta na standardowej bibliotece, a nie na jego składni.
5. Jawne jest zawsze lepsze niż domniemane.
6. Wszystko jest zmienną.
7. Wszystko jest obiektem.
8. Wielokrotne dziedziczenie.
9. Nie ma różnicy pomiędzy modułem, klasą i przestrzenią nazw.
10. Brak adnotacji i innych dodatkowych składni - wszystko jest oparte na podstawowej składki języka.

1.1.3 Co i jak

Obecnie główna wersja Plezuro jest kompilowana do Javascripta. W przyszłości jest planowane kompilowanie do innych języków (prawdopodobnie C, C♯, Java, Python, Ruby i PHP, ewentualnie Lisp i Fortran). Istnieje również wersja interpretowana zaimplementowana w C♯, która jednak nie jest dłużej rozwijana. Tamta wersja nie jest w pełni zgodna z główną wersją Plezuro. To coś na kształt prototypu końcowego produktu.

Plezuro.js

Używając pliku wykonywalnego plezuro.jar (pobierz https://plezuro.herokuapp.com/downloads/plezuro.jar), możesz skompilować skrypt z Plezuro do Javascripta. Działa to dla całych plików, ale nie dla kodu zagnieżdżonego w HTML. Można używać Plezuro w przeglądarce, po stronie serwera (używając Node.js), w tworzeniu aplikacji mobilnych (używając pakietu Cordova) lub w jakiejkolwiek innej technologii używającej Javascript. Inną ważną sprawą jest dołączenie biblioteki plezuro.js (pobierz z https://plezuro.herokuapp.com/downloads/plezuro.js). Możesz to zrobić w HTML w aplikacji na przeglądarkę lub Cordovie bądź przy użyciu `require` w Node.js). Istnieje automatyczne wsparcie dla wszystkich bibliotek Javascriptowych, bo można w Plezuro używać wszystkich zmiennych z Javascript. Zmienna nie może zawierać znaku dolara w jej nazwie w przeciwieństwie do Javascripta, więc aby użyć jQuery, zamiast znaku dolara należy użyć zmiennej jQuery lub ewentualnie `eval('$')`.

Podstawowe użycie kompilatora: `plezuro.jar input.plez output.plez.js`

1.2 Podstawy

1.2.1 Hello world

'Hello world' (Witaj świecie) w programowaniu oznacza najprostszą aplikację w konkretnym języku programowania. Pokazuje to jak ogólnie wygląda dany język.

Istnieje kilka sposobów w Plezuro na napisanie czegoś takiego. Jest to język skryptowy, więc nie jest wymagana żadna główna funkcja ani klasa. Najprostszy skrypt po prostu zwraca wartość. Można również użyć metody 'dumpl' do zapisu do bufora (najczęściej jest to konsola).

```
'Hello world!'
```

```
'Hello world!'.dumpl
```

1.2.2 Zmienne

Prawdopodobnie najbardziej podstawową możliwością języka programowania są zmienne. Co to jest? Zmienna to blok pamięci, który możesz zmieniać w trakcie wykonywania programu. Możesz przypisać taki blok do symbolu, a następnie to działa jak wyrażenia matematyczne. Jedyna różnica jest taka, że zmienna może zmieniać swoją wartość. Np. `$x = 2 + 5; $y = x * 2`.

W Plezuro tak jak w większości języków nazwa zmiennej może zaczynać się literą lub podkreśleniem _ a następne znaki to litera, podkreślenie lub cyfra. Wielkość liter nie ma znaczenia (tak jak w C, Javie, Pythonie i tak dalej, przeciwnie do SQL i HTML). Można używać jedynie znaków ASCII. Jest tak dla czytelności kodu (byłoby zupełnie dziwne jeśli ktoś użyłby liter arabskich, chińskich czy jakiejś mieszaniny tego i tego). Zalecany styl nazywania zmiennych (w tym funkcji i modułów) to camelcase (np. `aVeryInterestingVariable`).

Deklaracja

Zanim możemy używać jakąkolwiek zmienną, musimy najpierw ją zadeklarować. To bardzo proste, wystarczy napisać nazwę zmiennej, poprzedzając ją bezpośrednio znakiem dolara `$`. Przy następnych wystąpieniach zmiennej należy jej nazwę pisać już bez dolara.

Zasięg

Zasięg zmiennej jest ograniczony w obrębie klamer (`{`, `}`), które to są używane do tworzenia funkcji (nawet instrukcja warunkowa czy pętla to wewnętrzna funkcja). Zatem jeśli chcesz używać zmiennej w obrębie wielu funkcji, musisz zadeklarować ją w odpowiednim miejscu.

Czym może być zmienna?

W Plezuro występuje typowanie dynamiczne tak jak w innych dynamicznych językach takich jak Python, Ruby czy Javascript. Również wszystko jest zmienną (uwzględniając funkcje i moduły). Zatem zmienna może zmieniać swój typ w trakcie wykonywania programu. Np. na początku jest ona liczbą, następnie listą, a w końcu funkcją.

```
// number
$a = 34;

// number - scientific notation
$a1 = 1e34;
$a2 = 2.34e-89;

// number - binary
$a3 = 0b10111;

// number - octal
$a4 = 052;

// number - hexadecimal
$a5 = 0xae;

// string
$b = 'abc';

// list
$c = [a, b];

// associative array
$d = %('a': a, 'b': b)
```

1.2.3 Komentarze

Komentarze wyglądają tak jak w C++, Javie czy C♯. Możesz zakomentować jedną linię, używając `//` lub wiele linii przy użyciu `/*` oraz `*/`.

```
$x = 2 + // This is a comment
21 * 3;
/* Another
  comment
*/
x++;
x
```

1.2.4 Funkcje

Aby napisać funkcję, należy użyć klamer (`{`, `}`). Każdy kod wewnątrz klamry jest funkcją. To dotyczy również instrukcji warunkowych i pętel. Zerowy argument jest dostępny poprzez słowo kluczowe 'this' a następne 'first', 'second', 'third'. Jest również dostęp do tablicy wszystkich argumentów poprzez słowo kluczowe 'args'.

Funkcja zwraca wartość wartość ostatniego wyrażenia (tak jak w Ruby). Nie istnieje natomiast słowo kluczowe `return`.

```
$f = {
  this * 2 + first + second / third
};
f(1, 2, 3, 4)
```

Aby zliczyć czas wykonywania funkcji, użyj metody ‘time’. Argumenty przekazane to tej oto metody są następnie przekazywane do funkcji (pierwszy argument staje się zerowym, drugi pierwszym itd.).

Zwraca to obiekt z metodami:

- `result` - wartość zwrócona przez funkcję
- `time` - czas wykonywania w sekundach

```
$f = {
  this * 2 + first
};
$res = f.time(9, 3);
[res.time, res.result]
```

1.2.5 Wbudowane metody

W dalszych miejscach poradnika zdobędziesz pewną wiedzę o modułach i metodach w Plezuro. Jednak już w tym momencie powinieneś wiedzieć jak działają metody. Prosto mówiąc, to jest jak wywołanie funkcji. Zerowy argument (zwany this - tym obiektem) jest dostępny poprzez słowo kluczowe 'this'. Tak jak w przypadku funkcji, możesz używać słów kluczowych takich jak 'first', 'second', 'third', 'args'. Wywołując metodę, najpierw piszesz zerowy argument (this), następnie kropkę `.`, nazwę funkcji, otwarcie nawiasu ('('), argumenty oddzielone przecinkiem `,` i w końcu zamknięcie nawiasu `)`. Jednakże przy braku argumentów nawias nie jest konieczny.

```
$x = 15;
x = x.sin.cos.tan;
x += [x, x, x].length;
x
```

1.2.6 Kolekcje

Jedną z nieodzownych możliwości języka programowania są kolekcje. Oczywiście, Plezuro zapewnia niektóre z nich.

Lista (List)

Podstawową z nich jest lista. Aby ją stworzyć, powinieneś użyć nawiasów kwadratowych (`[, ]`) oraz elementów oddzielonych przecinkiem `,`. Lista może zawierać obiekty różnych typów (tak jak lista w Pythonie, Ruby, Javascripcie, PHP jak również List<Object> w Javie czy List<object> w C♯). Z tego wynika, że lista może zawierać nawet inne listy. Ponadto lista może mieścić w sobie samo-referencję (to znaczy że jeden z jej elementów jest wskaźnikiem na nią samą), ponieważ listy są przekazywane przez referencję do funkcji oraz kolekcji.

```
$x = 4;
$a = [x, x];
$b = [x, a, 1, ['abc', 56]];
b[0] = b;
b
```

Zbiór (Set)

Aby utworzyć zbiór, możesz użyć znaku dolara `$` oraz nawiasów. Ta kolekcja zachowuje się jak zbiór matematyczny. Każda wartość może wystąpić co najwyżej jeden raz.

```
$a = $(3, 4, [2, 9], $('ab', 90));
a.len
```

Słownik (Dictionary)

W prostym ujęciu to zbiór par klucz-wartość. To tak jak dictionary w Pythonie, hash w Ruby, Dictionary<object, object> w C♯ czy Map<Object, Object> w Javie. Jednakże istnieje istotna różnica pomiędzy słownikiem Plezuro a tablicą asocjacyjną w PHP czy obiektem w Javascripcie gdyż w słowniku nie ma znaczenia kolejność elementów i ogólnie jest przechowywany na drzewie binarnym. Możesz zapisać słownik przy użyciu krzyżyka # oraz nawiasów.

```
$x = 'abc';
$dict = #(
  x: [4, 5],
  'def': 49,
  'ghj': 'ooo'
);
dict.get(x)
```

Tablica asocjacyjna

To jest jak tablica asocjacyjna w PHP, czy też obiekt w Javascripcie. W wersji Plezuro kompilowanej do Javascripta główne zastosowanie tego typu kolekcji to przekazywanie obiektów do metod z bibliotek javascriptowych. Możesz to zapisać jako procent `%` z nawiasami.

```
$a = %(
  'a': 12,
  'b': 90
);
a['a']
```

1.2.7 Operatory

Jedną z ważnych możliwości języka programowania jest użycie operatorów. Technicznie byłoby możliwe stworzyć język bez operatorów. Jednak uproszczają one bardzo składnię. Na przykład w wyrażeniu `1+3` mamy operator `+`, który wykonuje dodawanie. W porównaniu z innymi językami, Plezuro posiada pewne specjalne operatory takie jak przecinek `,` czy średnik `;`.

Ważną sprawą w Plezuro jest to, że nie można mieć żadnego operatora bezpośrednio (nie licząc białych znaków i komentarzy) przed zamknięciem jakiegokolwiek nawiasu (‘‘)‘‘, `]`, `}`). Zatem po ostatnim elemencie listy nie można postawić przecinka a także po ostatnim wyrażeniu funkcji nie można postawić średnika. W takim przypadku doszłoby do błędu składniowego.

Akcja operatora zależy od typu zerowego argumentu. Działanie jest identyczne jak w przypadku wywołania metody. Jest możliwe zmienić akcję operatorów w trakcie wykonywania programu (możesz nawet spowodować, że wyrażenie *2+2* zwróci wynik inny niż 4, więc nie jest zalecane używać tego nadmiernie)

Ogólnie możemy wyróżnić pewne główne akcje operatorów (np. dla liczb i stringów).

Typ lewego argumentu	Operator	Akcja	Przykład	Wynik
Liczba (Number)	`+`	dodawanie	`1+4`	`5`
Liczba (Number)	`-`	odejmowanie	`4-7`	`-3`
Liczba (Number)	`*`	mnożenie	`8.5*2`	`17`
Liczba (Number)	`/`	dzielenie	`1/4`	`0.25`
Liczba (Number)	`^`	potęga	`4^3`	`64`
Liczba (Number)	`%`	reszta z dzielenia	`7%3`	`1`
String	`+`	konkatenacja	`'a'+'b'`	`'ab'`
String	`*`	mnożenie	`'a'\*3`	`'aaa'`
Lista (List)	`+`	konkatenacja	`[2,3]+[1]`	`[2,3,1]`
Lista (List)	`*`	mnożenie	`[2,3]*2`	`[2,3,2,3]`

```
$a = 21;
$b = [a, a+90];
$c = a * a + b[0];
[a, b, c]
```

Istnieją również operatory złożone, które stanowią pewne udogodnienie (syntactic sugar) np. `a += b` oznacza `a = a + b`.

Pełna lista operatorów złożonych:

- `+=`
- `-=`
- `*=`
- `/=`
- `^=`
- `&=`
- `|=`
- `%=`
- `.=`

Pełna lista operatorów:

- jednoargumentowe operatory (operator po lewej, argument po prawej)
  - `!`
  - `&&`
  - `**`
  - `#`
  - `~`
  - `=>`
- jednoargumentowe operatory (operator po prawej stronie)
  - `++`
  - `--`
- dwuargumentowe operatory (zerowy argument, operator, pierwszy argument)
  - `;`
  - `,`
  - `:=`
  - `=`
  - `+=`
  - `-=`
  - `*=`
  - `=`
  - `^=`
  - `&=`
  - `|=`
  - `%=`

- .=
- ~~
- <->
- <<
- >>
- ?
- |
- &
- <=>
- >=
- >
- <=
- <
- !=
- ==
- !==
- ===
- =~
- !~
- +
- -
- %
- *
- /
- ^
- ^^
- .
- ..
- :

1.2.8 Instrukcje warunkowe

Na początku powinieneś napisać warunek (funkcję zwracającą wartość boolowską). Następnie '.if' oraz w nawiasach i klamrach wyrażenie, które będzie wykonywane, jeśli początkowy warunek jest prawdziwy. To po prostu metoda typu Funkcja. Następnie możesz użyć metod 'elif' oraz 'else', które to nie są wymagane.

```
$x = 21;
{x > 0}.if({
  x++
}).elif({x < 9}, {
  x--
}).else({
  x *= 2
});
x
```

Istnieje poza tym inna możliwość. Można użyć dwóch operatów: `:`, który to tworzy parę oraz `?`, który zależnie od wartości boolowskiej po lewej stronie zwraca klucz lub wartość pary po prawej stronie. W innych językach istnieje tylko jeden operator `? :`, jednak w Plezuro nie ma operatorów złożonych z dwóch osobnych części.

```
$x = 9;
x = (x < 3) ? 'a' : 'b';       // 'b'
x + ((x < 'e') ? 'a' : 'b')    // 'a'
                               // finally x = 'ba'
```

1.2.9 Pętle

Pętle wyglądają podobnie jak wyrażenie warunkowe. Jest używana metoda z typu Function.

while

Na początku należy napisać funkcję (zerowy argument), od której zwróconej wartości zależy, czy funkcja będąca pierwszym argumentem zostanie wykonana. Po pierwszej iteracji funkcja warunkowa jest wykonywana kolejny raz. Późniejsze wykonywanie funkcji będącej pierwszym argumentem znów zależy od wartości zwróconej przez funkcję warunkową i tak dalej.

```
$n = 1;
{n < 1000}.while({
  n = 2*n+1
});
n
```

do

Na początku pisze się funkcję warunkową. Wykonanie pętli zależy od wartości zwróconej przez tę funkcję. Zatem iteracje przebiegają tak długo, dopóki funkcja zwraca wartość true.

```
$i = 1;
{i *= 2; i < 1000}.do;
i
```

each

Jest używana do iterowania listy. W wewnętrznej funkcji zerowy argument to indeks obecnego elementu, natomiast pierwszy argument to ten właśnie element.

```
$a = [
  23,
  200,
  20,
  3
];
$s = 0;
$s2 = [];
a.each({
  s += first;
  s2 << this;
});
[s, s2]
```

foreach

Zerowym argumentem jest funkcja, a następnymi listy (ilość argumentów jest nieograniczona). W każdej iteracji argumenty w funkcji wewnętrznej są zbudowane z indeksu oraz elementów z tychże list na pozycji odpowiadającej indeksowi. Wykonywanie pętli trwa dopóki dojdziemy do ostatniego elementu najdłuższej listy. Jeśli któraś z list się skończyła, otrzymujemy wartość `null` w odpowiednim miejscu.

```
$s = [];
{
  s << first + second
}.forEach([9, 2, 4], [10, 20, 1]);
$s2 = [];
{
  s2 << args
}.forEach([2, 9], [90, 11, 30], [0], [], [4, 2]);
[s, s2]
```

times

Jest to używane w przypadku potrzeby wykonania pewnej procedury określoną liczbę razy. To prostsze od pozostałych pętli. Zerowy argument to liczba (razy), zaś pierwszy to funkcja, która to przyjmuje licznik iteracji startujący od zera jako swój zerowy argument.

```
$ar = [];
20.times({
  ar << this
});
ar
```

1.2.10 Przedział (range)

Do prostej iteracji w ciągu arytmetycznym, Plezuro daje nam użyteczne narzędzia, którym jest przedział. Aby go stworzyć, należy użyć operatora `..`

Istnieje metoda `each`, która iteruje przedział. W wewnętrznej funkcji zerowy argument to indeks (liczba naturalna licząc od 0) a pierwszy argument to obecna liczba z tego przedziału.

```
$a = [];
(1..7..30).each({
  a << args
});
$a2 = [];
(1..4).each({
  a2 << args
});
[a, a2]
```

1.2.11 Import

Jest zalecane dzielenie projektów programistycznych na wiele plików źródłowych. Można to zrobić, wykorzystując metodę ‘import’ z typu String. Można również przekazywać parametry do importowanego skryptu, to działa w taki sposób jak w przypadku funkcji.

```
$a = './bin/js/doc/loop.plez.js'.import;
a
```

1.2.12 Losowanie (random)

W programowaniu jest użyteczne w wielu przypadkach generowanie liczb oraz tekstów losowych. Np. kiedy tworzysz gracza komputerowego w jakiejś grze, czy generujesz linki rejestracyjne w aplikacji webowej i tak dalej.

W Plezuro to bardzo proste. W przypadku losowej liczby, nie musisz wywoływać żadnej funkcji czy metody (jednak później jest to kompilowane do kodu z wywołaniem metody). Po prostu wystarczy napisać słowo kluczowe *rand* i otrzymasz liczbę z równomiernego rozkładu między 0 a 1.

Aby wygenerować losowy ciąg znaków, użyj `String.rand`. Pierwszy argument określa długość wynikowego ciągu, domyślna wartość to 32. Wynik jest zbudowany z liter, cyfr i podkreślenia.

```
$a = rand;
$b = rand * a + rand;
$f = {
  rand * this
};
$c = String.rand;
$d = String.rand(200);
[a, b, c, d, f(20), rand]
```

1.2.13 Regex

Poza programowaniem imperatywnym (tak jak to jest w Plezuro, istnieją również inne paradygmaty programowania. To oznacza, że możesz wykonać tę samą czynność na przeróżne sposoby. Może wydaje się to nieco skomplikowane, ale to często sprawia, że trudne problemy stają się łatwymi.

Jednym z takich paradygmatów jest regex (wyrażenie regularne). Wygląda to jak normalny tekst, jednak są tutaj umieszczone pewne specjalne znaki, które mogą oznaczać jeden z wielu znaków, powtórkę ciągu znaków itd. Bardzo przydatne jest to w walidacji formularzy internetowych (np. kod pocztowy, imię, data).

Zależnie od silnika, wyrażenia regularne zachowują się na różne sposoby. Obecnie Plezuro używa już istniejących silników, zatem w wersji kompilowanej do Javascripta zasady są identyczne jak w Javascripcie. Możesz się nauczyć tego z http://www.w3schools.com/jsref/jsref_obj_regexp.asp .

Składnia jest następująca: `r` oraz tekst pomiędzy apostrofami lub cudzysłowem. W przypadku apostrofów umieszczonymi pomiędzy apostrofami napisz je dwukrotnie, to samo w przypadku cudzysłowu pomiędzy cudzysłowami (np. `r"ab""cd"`, `r'ef''gh'`).

Żeby przetestować regex (sprawdzić czy dany ciąg znaków do niego pasuje), użyj operatora `=~`. (Kolejność nie ma znaczenia, albo string - operator - regex, albo regex - operator - string.) Istnieje również operator, który neguje powyższe `!~`.

```
$x = r'[a-z]';
$b = x =~ 'abc';
$b1 = 'dfg' =~ x;
$b2 = 'dfg' !~ x;
$b3 = x !~ 'xxx';
$res = [x, b, b1, b2, b3];
res << 'gh-kk' =~ r'[a-z]'                // true
    << 'gh-kk' =~ r'^[a-z]$'              // false
    << 'gh-kk' =~ r'^[a-z]+-[a-z]+$'      // true
    << 'gh-kk' =~ r'^[a-z]{2}-[a-z]{2}$'  // true
    << 'gh-kko' =~ r'^[a-z]{2}-[a-z]{2}$' // false
    << '\\' =~ r'^\\$'                    // true
    << '\\' =~ r'\\'                      // true
    << '\\' =~ r"\\"                      // true
    << '0+9' =~ r"^[0-9]\+[0-9]$"         // true
    << '0+97' =~ r"^[0-9]\+[0-9]$"        // false
    << 'a"k' =~ r"^a""k"                  // true
    << 'ak' =~ r"^a""k"                   // false
    << 'a"k' =~ r'^a""k'                  // true
    << 'a"k' =~ r'^a"k'                   // true
    << "a'k" =~ r'^a''k'                  // true
    << "ak" =~ r'^a''k'                    // false
    << "a'k" =~ r"^a''k"                  // true
    << "a'k" =~ r"^a'k"                   // true
;
res
```

1.2.14 Magiczne stałe

Podobnie do PHP i Ruby, Plezuro posiada magiczne stałe. Co to jest i dlaczego tego używać? To jest coś poniekąd jak zmienna a trochę jak stała. Wartość zależy od miejsca, gdzie to jest użyte.

Zatem mamy następujące magiczne stałe:

- __pos__ - poziome położenie w obrębie linii (licząc od 0, to pozycja gdzie słowo kluczowe zaczyna się)
- __line__ - numer linii w pliku źródłowym (licząc od 0)
- __file__ - nazwa pliku źródłowego
- __dir__ - pełna nazwa folderu pliku źródłowego

```
$x = %(
    'pos': __pos__,
    'line': __line__,
    'file': __file__,
    'dir': __dir__
);
x
```

1.2.15 Wyjątki

Jedną z potężnych możliwości języka programowania jest przechwytywanie wyjątków. To jest możliwe również w Plezuro. Jednakże składnia jest nieco inna, ponieważ w tym języku prawie wszystko jest oparte na metodach (nawet instrukcje warunkowe i pętle).

```
$x;
$y;
{
    x = wfwfwfe
}.try({
    x = this['message']
});
{
    Error.new('an error').throw
}.try({
    y = this['message']
});
[x, y]
```

1.3 Programowanie obiektowe

Plezuro jest językiem obiektowym, występuje tutaj wielokrotne dziedziczenie. Jest bardziej dynamiczny niż Java i .NET ponieważ możesz dodawać, zmieniać i usuwać metody w trakcie wykonywania programu.

1.3.1 Moduły

Moduły są jednocześnie klasami i przestrzeniami nazw. Można użyć modułu do stworzenia obiektu, można go użyć w sposób statyczny (tak jak pola i metody statyczne w Javie), a także przypisać inne moduły jako pola statyczne, tworząc w ten sposób przestrzenie nazw. Oczywiście da się tworzyć wiele modułów w tym samym pliku źródłowym, wszakże zalecane jest pisać dokładnie jeden moduł w jednym pliku.

Aby stworzyć moduł, należy użyć składni `Module.create` i przekazać do tej metody dokładnie jeden parametr, który to jest tablicą asocjacyjną zawierającą nazwę modułu (pole 'name'), przestrzeń nazw modułu (pole 'namespace'), metody (pole 'methods'), a także rodziców (pole 'parents') - czyli moduły z których dziedziczymy. Istnieją również inne pola, które można przekazać.

Aby stworzyć obiekt danego modułu, należy użyć metody 'new'.

```
$Person = Module.create(%(
    'name': 'Person',
    'methods': %(
        'do' : {
            @a * 2 + first
        }
    )
));
$Student = Module.create(%(
    'name': 'Student',
    'parents': [Person]
));
$s = Student.new(%('a': 14));
s.do(5)
```

1.3.2 Obiekty i metody

Głównym celem tworzenia modułu jest jego użycie jako szablonu do budowania obiektów i wywoływania metod z tychże obiektów. Kiedy wywołujesz metodę, obiekt jest dostępny poprzez słowo kluczowe 'this' i tak jak w przypadku wywołania funkcji, można używać słów kluczowych `first`, `second`, `third` oraz `args`.

Pola obiektów

Aby uzyskać dostęp do pola obiektu z metody wywołanej na tym obiekcie, należy użyć znaku małpa `@` a bezpośrednio po nim podać nazwę pola. Pola mogą być tworzone w konstruktorze, kiedy obiekt jest tworzony lub później w jakiejkolwiek metodzie.

Wywołanie metody z innej metody

Aby wywołać metodę z innej metody, należy napisać słowo kluczowe `this` oraz kropkę.

Pamiętaj, w przypadku funkcji wewnętrznej (np. pętle), słowo kluczowe `this` oznacza co innego. Wtedy jest to zerowy argument funkcji wewnętrznej. W ten sam sposób działają pola obiektu.

```
$Person = Module.create(%(
    'name': 'Person',
    'methods': %(
        'say' : {
            'I am '+@name+', '+@age+' years old'
        },
        'long_say': {
            'Good morning, '+this.say
        },
        'older': {
            @age++
        }
    )
));
$adam = Person.new(%('name': 'Adam', 'age': 20));
adam.older;
[adam.say, adam.long_say]
```

1.3.3 Konstruktor

Technicznie byłoby możliwe stworzyć obiekt bez konstruktora. Możesz zainicjalizować wszystkie pola jedną lub wieloma metodami (używając wzorca builder). Ewentualnie po tym można by zamrozić cały obiekt lub tylko niektóre z jego pól.

Aczkolwiek dla czytelności kodu źródłowego dobrym pomysłem jest użycie konstruktorów. Czym jest dokładnie konstruktor? To metoda wywoływana bezpośrednio po zaalokowaniu pamięci dla obiektu. Jest to bardzo jawne w Objective C, gdzie najpierw w kodzie jest alokowana pamięć, a następnie przebiega inicjalizacja (w większości przypadków jest to w tej samej linii).

W Plezuro konstruktor to metoda z nazwą 'init'. Taka metoda jest wywoływana automatycznie po stworzeniu obiektu.

Konstruktor domyślny

Wszakże kiedy brakuje w kodzie konstruktora, istnieje pewien konstruktor, który jest wywoływany. Zwany jest konstruktorem domyślnym. Co to dokładnie robi? Wywołuje on konstruktory wszystkich modułów macierzystych. Kiedy moduł nie ma żadnych jawnych rodziców, dziedziczy on po BasicModule, więc w takiej sytuacji jest wywoływany konstruktor BasicModule, który to tworzy pola obiektu z tablicy asocjacyjnej przekazanej jako pierwszy argument.

```
$Person = Module.create(%(
  'name': 'Person',
  'methods': %(
      'init': {
          @age = 0;
          @name = first
      },
      'say': {
          'I am '+@name+', '+@age+' years old'
      }
  )
));
$adam = Person.new('Adam');
adam.say
```

1.3.4 Operatory

Podobnie do definiowania modułu, można zdefiniować akcje, które zostaną wykonane kiedy na obiekt będzie działać odpowiedni operator.

```
$Vector = Module.create(%(
    'name': 'Vector',
    'methods': %(
        'list': {
            @list
        },
        '+': {
            $result = [];
            $i = 0;
            $a = this;
            $b = first;
            {
                result.push(a.list[i] + b.list[i]);
                i++;
                i < a.list.len
            }.do;
            Vector.new(%('list': result))
        }
    )
));
$a = Vector.new(%('list': [3, 4, 9]));
$b = Vector.new(%('list': [6, 0, 9]));
a+b
```

1.3.5 Dziedziczenie

Żeby pisać kod obiektowy, poza klasami (w przypadku Plezuro modułami) jest niezbędne użycie dziedziczenia i polimorfizmu. W Plezuro występuje wielokrotne dziedziczenie tak jak w Pythonie, jest także nieco podobne do wielokrotnego dziedziczenia w C++ przy użyciu wirtualnego wiązania metod. Dlaczego wielokrotne dziedziczenie? To po prostu bardziej zbliżone do prawdziwego życia. Np. pies jest w tym samym czasie zwierzęciem domowym i drapieżnikiem

Jak to działa? To bardzo proste, kiedy wywołujesz metodę, na obiekcie, kiedy istnieje taka metoda bezpośrednio w module danego obiektu, metoda ta jest wywołana. W przeciwnym razie jest to metoda z jednego z rodziców modułu (kolejność znajdowania metody jest taki sam jak kolejność zadeklarowanych rodziców przy tworzeniu modułu). Algorytm znajdowania metody jest rekurencyjny. Jeśli żadna metoda nie została znaleziona, jest wyrzucany wyjątek. W taki sposób możesz to zaobserwować. Natomiast w rzeczywistości w momencie stworzenia modułu wszystkie metody są wiązane (bezpośrednie i dziedziczone), tak jest ze względu na wydajność.

A co na temat polimorfizmu? W Plezuro tak jak w innych dynamicznych językach występuje duck typing. Zatem można wywołać metodę z konkretną nazwą na danym obiekcie, nie znając modułu tego obiektu. Można także używać wielokrotnego dziedziczenia.

Wywołanie metody rodzica

Ważną sprawą jest możliwość wywołania metody rodzica. Składnia jest podobna do tej w Pythonie, ponieważ z powodu wielokrotnego dziedziczenia, trzeba określić, z którego rodzica metoda ma być wywołana. Jest również możliwe wywołać metodę z dalszego przodka a nawet z jakiegokolwiek innego modułu z powodu duck typing. Składnia jest następująca: najpierw nazwa modułu, następnie podwójny dwukropek *::*, nazwa metody oraz słowo `this` jako zerowy argument.

```
$Animals = Module.create(%(
    'name': 'Animals'
));
$Animal = Module.create(%(
    'name': 'Animal',
    'namespace': Animals,
    'staticFields': %(
        'nr': 0
    ),
    'methods': %(
        'init': {
            Animal::nr++;
            @age = 0
        },
        'say': {
            'I am '+@age
        },
        'older': {
            @age++
        }
    )
));
$Dog = Module.create(%(
    'name': 'Dog',
    'parents': [Animal],
    'namespace': Animals,
    'methods': %(
        'types': {
            ['a', 'b', 'c']
        },
        'say': {
            Animal::say(this)+', dog'
        }
    )
));
$dog = Dog.new;
[Animals::Dog['className'], Animals::Animal['className'], Animal::nr, dog.types[0], dog.say]
```

1.3.6 Pola i metody statyczne

Poza użyciem modułu jako szablon do tworzenia obiektów, możesz użyć go w statyczny sposób za pomocą pól i metod statycznych. Powinieneś pamiętać, że statyczne pola i metody nie są dziedziczone. Każde pole statyczne i każda metoda statyczna są związane dokładnie z jednym modułem.

```
$Machine = Module.create(%(
    'name': 'Machine',
    'staticFields': %(
        'state': 0
    ),
    'staticMethods': %(
        'change': {
            @state = @state.sin + @state.cos
        },
        'doubleChange': {
            this.change;
            this.change
        }
    )
));
Machine.doubleChange;
Machine.doubleChange;
Machine::state
```

1.3.7 Dynamiczna zmiana modułu

Dodanie/usuwanie metody

Poza tworzeniem metody w czasie powstawania modułu, można również dynamicznie dodawać nowe metody później. Można by powiedzieć 'w trakcje wykonywania programu' ale to określenie nie jest odpowiednie w przypadku Plezuro ponieważ w tymże języku wszystko dzieje się w trakcie wykonywania programu. Nowa metoda jest widoczna w bezpośrednich obiektach danego modułu (stworzonych nawet przed dodaniem nowej metody), jak również w obiektach modułów dziedziczących. Możesz zrealizować to, używając metodę 'addMethod'.

Usuwanie metody

Jest nawet możliwe usunąć metodę. Mimo wszystko nie rób tego bez powodu. Aby to zrobić, użyj metody 'removeMethod'.

```
$Person = Module.create(%(
    'name': 'Person',
    'methods': %(
        'do' : {
            @a * 2 + first
        }
    )
));
$Student = Module.create(%(
    'name': 'Student',
    'parents': [Person]
));
student = Student.new(%('a': 90));
$x = student.do(1, 3, 4);
Person.addMethod('say', {
    'person a='+this['fields']['a']
});
$y = student.say;
Student.addMethod('say', {
    'student a='+this['fields']['a']
});
$z = student.say;
Person.addMethod('say', {
    'person2 a='+this['fields']['a']
});
$z1 = student.say;
Student.removeMethod('say');
$z2 = student.say;
Person.removeMethod('say');
$msg;
{
    student.say
}.try({
    msg = this['message']
});
[x, y, z, z1, z2, msg]
```

1.4 Programowanie na przeglądarkę internetową

Jak już wiesz, Plezuro jest kompilowany do Javascripta. W związku z tym możemy pisać aplikacje wykonywane w przeglądarce internetowej (aplikacje webowe wykonywane po stronie klienta).

Co należy znać najpierw?

Aby móc tworzyć aplikacje webowe w Plezuro, powinieneś znać HTML i CSS (przynajmniej podstawy). Możesz się nauczyć tego z następujących przewodników:

- http://www.w3schools.com/html/default.asp
- http://www.w3schools.com/css/default.asp

1.4.1 Co i jak

Przeciwnie do Javascripta, Plezuro nie może być zagnieżdżony w kodzie HTML. To nie brakująca możliwość ale raczej sposób na utrzymanie czytelnego kodu. Zatem trzeba dołączyć w pliku HTML Javascript wygenerowany ze źródła w Plezuro. Ponadto należy dołączyć bibliotekę plezuro.js.

basic/main.plez

```
window.alert('blabla')
```

1.4.2 Wyjście

Można używać standardowych sposobów Javascripta do wyświetlania na wyjściu:

- zapis do alertu, używając `window.alert()`
- zapis na wyjście HTML, używając `document.write()`
- zapis do elementu HTML, używając `element.innerHTML=`
- zapis do konsoli przeglądarki, używając `console.log()`

Jednak zalecanym sposobem w Plezuro jest użycie metody `dumpl`.

Wygląda to jak Javascript, ale wiedz, że `alert` bez `window` nie zadziała, dopóki nie przekaże mu się zmiennej `window` jako zerowy argument. Jest tak, ponieważ zerowy argument w Plezuro jest traktowany jako dany obiekt.

```
window.alert('aa');
document.write('bb');
alert(window, 'cc');
document['body']['innerHTML'] = 'dd';
console.log('ee');
'ff'.dump1
```

1.4.3 Elementy HTML

W programowaniu HTML (nie każde programowanie na przeglądarkę internetową to programowanie HTML, ponieważ istnieją takie technologie jak aplety Java, Adobe Flash czy Silverlight, jednakże są one przestarzałe) prawie wszystko, co widzisz w przeglądarce to elementy HTML. Wyjątkami są alerty i różne rodzaje powiadomień z przeglądarki.

To potężne narzędzie. W programowaniu aplikacji desktopowych w większości frameworków nie ma takiego narzędzie i to czyni programowanie GUI znacznie trudniejszym.

HTML DOM

Ten skrót oznacza Document Object Model (dokumentowo-obiektowy model). Przedstawia drzewo obiektów.

Czym jest element HTML?

Może być najmniejszą niepodzielną częścią widoku, który widzisz w przeglądarce (np. input, button, span), jak również tym, czego nie możesz zobaczyć (ponieważ jest ukryty, ma rozmiar 0 lub nie zawiera żadnego tekstu) albo ewentualnie drzewem innych elementów.

Czym jest drzewo?

Możemy sobie wyobrazić drzewo obiektów na kształt prawdziwego drzewa. Każdy element (węzeł) to miejsce, gdzie jedna gałąź rozdziela się na wiele gałęzi (w szczególnym przypadku nadal jedno), liść (gdzie drzewo się kończy) lub początek drzewa (gdzie wyrasta z ziemi).

Relacje w drzewie

Jeden element dla innego może być:

- dzieckiem - child (w szczególnym przypadku pierwszym dzieckiem - first child)
- rodzicem - parent
- rodzeństwem - sibling (w szczególnym przypadku następnym rodzeństwem - next sibling)
- przodkiem - ancestor
- potomkiem - offspring
- niczym z powyższych

Zauważmy, że dziecko jest jednocześnie potomkiem, jak również rodzic jest jednocześnie przodkiem.

Skoro już znasz podstawy HTML, powinieneś wiedzieć, że zewnętrznym elementem jest HTML. Tak na prawdę to dziecko elementu document, który to jest najbardziej zewnętrznym elementem w całym HTML DOM.

Dlaczego jest to tak ważne?

Znając element, możesz operować na jego dzieciach, rodzeństwie, rodzicu. Możesz zmieniać kolejność dzieci, kopiować poddrzewo i tak dalej.

Co Plezuro może zrobić?

- zmieniać elementy HTML (np. tekst)
- dodawać elementy HTML
- usuwać elementy HTML
- klonować elementy HTML
- zmieniać atrybuty HTML
- dodawać atrybuty HTML

Listing 1.1: basic/index.html

```
<html>
<body>
<script src="../../../plezuro.js"></script>
<script src="main.plez.js"></script>
</body>
</html>
```

- usuwać atrybuty HTML
- zmieniać style CSS
- tworzyć nowe zdarzenia HTML

Z tego wynikają dodatkowe możliwości. Np. klonując, usuwając i dodając element, możesz zmieniać jego położenie.

1.4.4 Metody i zdarzenia HTML DOM

Po pierwszych podrozdziałach, jesteś w stanie stworzyć statycznie dokument HTML przy użyciu `document.write`. Jednak prawdopodobnie chciałbyś stworzyć interaktywną aplikację. Np. po kliknięciu na przycisk, input zmieni swoją wartość.

Jak dostać element?

Możesz użyć standardowych metod Javascript w Plezuro. Ewentualnie da się użyć metod z zewnętrznych bibliotek takich jak jQuery.

document.getElementById

Zwraca element z pasującym atrybutem 'id'.

document.getElementsByTagName

Zwraca kolekcję wszystkich elementów z nazwą tagu określoną jako pierwszy parametr. Np. `document.getElementsByTagName('div')` zwróci kolekcję wszystkich divów w dokumencie.

document.getElementsByName

Kolekcja wszystkich elementów z wartością atrybutu 'name' określoną jako pierwszy argument.

document.getElementsByClassName

Kolekcja wszystkich elementów z wartością atrybutu 'class' określoną jako pierwszy argument.

document.getElementsByTagNameNS

Podobnie do getElementsByTagName ale pierwszy argument określa nazwę przestrzeni nazw a drugi nazwę tagu. Jest to bardziej używane w XML DOM.

document.querySelector

Zwraca pierwszy element, do którego pasuje selektor CSS w pierwszym argumencie. Np. `document.querySelector('table button')` zwróci pierwszy przycisk (button) w pierwszej tabeli, która zawiera jakikolwiek przycisk.

document.querySelectorAll

Zwraca kolekcję wszystkich elementów, dla których pasuje podany selektor CSS.

Jak działać na elementach?

innerHTML

To najprostszy sposób, aby zmienić zawartość elementu. Po prostu wstawia jakikolwiek HTML do wnętrza elementu. Np. `element['innerHTML'] = '<button>OK</button>'`.

setAttribute

Np. `element.setAttribute('name', 'wiek')`

styl

Możesz zmienić styl CSS. Można użyć nazwy CSS z myślnikiem – jak również camelcase. Np. `element['style']['backgroundColor'] = 'red'`

document.createElement

Możesz stworzyć element. Pierwszy argument określa nazwę tagu. Np. aby stworzyć div, należy napisać `document.createElement('div')`.

document.appendChild

Skoro już stworzyłeś element, możesz dodać go do innego elementu za równo istniejącego w HTML DOM lub nie.

document.removeChild

Możesz usunąć element przekazany jako pierwszy argument.

document.replaceChild

Możesz zastąpić element przekazany jako pierwszy argument na element przekazany jako drugi argument.

document.createTextNode

Możesz stworzyć węzeł tekstowy z wartością określoną jako pierwszy argument, a następnie dodać taki węzeł do elementu.

Jak nasłuchiwać zdarzenia?

Aby zapewnić interakcję, przydatne jest nasłuchiwane zdarzeń. Możemy wykryć m.in.: kliknięcie myszą, wciśnięcie klawisza, załadowanie dokumentu, zmianę dokumentu, zmianę obiektu, zmianę rozmiaru okna przeglądarki.

Możesz zobaczyć pełną listę zdarzeń dla Firefox na https://developer.mozilla.org/en-US/docs/Web/Events, a dla dowolnej przeglądarki na http://www.w3schools.com/tags/ref_eventattributes.asp.

Wymieńmy najważniejsze z nich.

onclick

Jest wywoływany po kliknięciu myszą (wciśnięcie i puszczenie przycisku myszy).Np. `element['onclick'] = { this['parentNode'].removeChild(this); null }`

onkeydown

Jest wywoływany po wciśnięciu klawisza.

onkeypress

Jest wywoływany, kiedy klawisz jest wciśnięty (nawet wiele razy).

onkeyup

Jest wywoływany po puszczeniu klawisza.

dom_methods/main.plez

```
document.getElementById('mainButton')['onclick'] = {
  $a = Number.parseFloat(document.getElementById('a')['value']);
  $b = Number.parseFloat(document.getElementById('b')['value']);
  $c = a + b;
  document.getElementById('c')['value'] = c;
  this['style']['backgroundColor'] = 'rgb('+(rand*256).floor+', '+(rand*256).floor+', '+(rand*256).floor+')';
  null
}
```

1.4.5 Pamięć

W programowaniu prawie cały czas potrzeba zapisywać jakieś dane. Część nich jest bardziej trwała, a inna część mniej.

W programowaniu na przeglądarkę internetową, dostępne są następujące rodzaje pamięci:

- Trwałe dopóki strona jest przeładowana lub zamknięta:
  - HTML DOM
  - pamięć Javascript
- Bardziej trwałe:
  - localStorage
  - sessionStorage
  - indexedDB
  - cookies

Naturalnie, w Plezuro jest pełny dostęp do wszystkich tych rodzajów. Ponadto jest wymiana pamięci pomiędzy Plezuro a Javascript poprzez zmienne globalne lub pola obiektu (publiczne jak również prywatne).

storage/main.plez

```
document.getElementById('mainButton')['onclick'] = {
  $counter = localStorage['counter'];
  {Object.isUndefined(counter)}.if({
    counter = '-1'
  });
  counter = Number.parseFloat(counter);
  counter++;
  localStorage['counter'] = counter;
  document.getElementById('counter')['value'] = counter;
  null
}
```

1.4.6 jQuery

jQuery to potężna biblioteka Javascript, która daje ci wszystkie możliwości działania w przeglądarce, których brakuje w czystym Javascript i która czyni kod krótszym i bardziej czytelnym.

Główne możliwości to:

- selekcja elementów HTML
- manipulacja DOM
- zdarzenia
- AJAX
- kontrola przetwarzania asynchronicznego
- rozszerzalność
- wsparcie w wielu przeglądarkach

Teoretycznie jQuery działa w taki sam sposób we wszystkich przeglądarkach. W rzeczywistości mogą się zdarzyć pewne różnice, np. selektor używa document.querySelectorAll, który zachowuje się nieco różnie w zależności od przeglądarki.

Aby używać tę bibliotekę, powinieneś być świadomy:

- Najpopularniejsze użycie w Javascript jest oparte na zmiennej `$`. W Plezuro jest to sprzeczne z zasadami nazewnictwa zmiennych, zatem aby uzyskać dostęp do jQuery, powinieneś użyć zmiennej `jQuery` lub ewentualnie `eval('$')`.
- Musisz przekazać odpowiednią zmienną jako pierwszy argument funkcji jQuery, a nie jako zerowy argument.
- W Plezuro wszystko jest obiektem, a selektor jQuery musi być prawdziwym stringiem z Javascripta, zatem musisz wywołać metodę `toString`.

jQuery/main.plez

Listing 1.2: dom_methods/index.html

```
<html>
<body>
<label for="a">a=</label><input id="a"/>
<label for="b">b=</label><input id="b"/>
<label for="c">c=</label><input id="c" disabled/>
<button id="mainButton">OK</button>


<script src="../../../plezuro.js"></script>
<script src="main.plez.js"></script>

</body>
</html>
```

Listing 1.3: storage/index.html

```
<html>
<body>
<label for="counter">counter=</label><input id="counter" disabled/>
<button id="mainButton">OK</button>

<script src="../../../plezuro.js"></script>
<script src="main.plez.js"></script>

</body>
</html>
```

Listing 1.4: jQuery/index.html

```
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
</head>
<body>
<label for="a">a=</label><input id="a"/>
<label for="b">b=</label><input id="b"/>
<label for="c">c=</label><input id="c" disabled/>
<button id="mainButton">OK</button>

<script src="../../../plezuro.js"></script>
<script src="main.plez.js"></script>

</body>
</html>
```

```
jQuery(null, '#mainButton'.toString).click({
  $a = Number.parseFloat(jQuery(null, '#a'.toString).val);
  $b = Number.parseFloat(jQuery(null, '#b'.toString).val);
  $c = a + b;
  jQuery(null, '#c'.toString).val(c);
  jQuery(null, this).css('backgroundColor', 'rgb('+(rand*256).floor+', '+(rand*256).floor+','+
(rand*256).floor+')');
})
```

Na szczęście, istnieje prostszy sposób na wywołanie jQuery w Plezuro przy użyciu specjalnego połączenia. Wystarczy użyć funkcji `_jQ` i wtedy kod wygląda w następujący sposób:

jQuery_binding/main.plez

```
_jQ('#mainButton').click({
  $a = Number.parseFloat(_jQ('#a').val);
  $b = Number.parseFloat(_jQ('#b').val);
  $c = a + b;
  _jQ('#c').val(c);
  _jQ(this).css('backgroundColor', 'rgb('+(rand*256).floor+', '+(rand*256).floor+','+(rand*256).floor+')');
})
```

1.4.7 AJAX

AJAX to skrót od Asynchronous Javascript and XML (Asynchroniczny Javascript i XML). Jest to zbiór praktyk programistycznych do tworzenia asynchronicznych aplikacji webowych. Chociaż sama nazwa mówi o XML, może być użyty jakikolwiek rodzaj danych (JSON, CSV, HTML, CSS, obrazki, Javascript). AJAX jest prawie zawsze powiązany z programowaniem po stronie serwera (odpowiedź serwera zależna od żądania). Technicznie to zawsze prawda, bo wysyłasz żądanie, a serwer w zależności od niego zwraca odpowiedź. Tymczasem nie zawsze się to wiąże ze skryptem w języku takim jak PHP, Python czy Ruby. Jest tak, ponieważ serwer może też zwrócić statyczne dane (np. obrazki) w zależności od ścieżki. Najlepszym przykładem jest serwer Apache 2.

Z punktu widzenia programisty front-end, AJAX umożliwia komunikację z serwerem bez przeładowania strony. Np. chciałbyś przechować jakieś dane na serwerze z powodów bezpieczeństwa, sprawdzić czy dane wprowadzone przez użytkownika pasują do tych na serwerze czy też ściągnąć jakieś zasoby dynamicznie (HTML, CSS, obrazki, inne Javascripty), w przypadku których byłyby wielkie rozmiary, jeśli pobierałoby się je w całości.

Prawdopodobnie najlepszy sposób do wysłania zapytania AJAX jest użycie jQuery z powodów kompatybilności między przeglądarkami i czytelności kodu.

ajax/main.plez

```
_jQ('#mainButton').click({
  _jQ('#container').load('part.html'.toString);
  null;
  null
})
```

1.4.8 AngularJS

AngularJS to framework używany głównie w aplikacjach na jedną podstronę (single page application). Stanowi on część pakietu MEAN:

- MongoDB - baza danych
- Express.js - framework back-end
- AngularJS - framework front-end
- Node.js - platforma back-end

Wszystkie spośród tychże czterech technologii są oparte na Javascripcie, więc można je używać w Plezuro.

Przykłady poniżej zostały wzięte z https://angularjs.org/, a następnie zaadoptowane do Plezuro.

angular/todo.plez

Listing 1.5: ajax/index.html

```
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
</head>
<body>
<button id="mainButton">OK</button>
<div id="container"></div>

<script src="../../../plezuro.js"></script>
<script src="main.plez.js"></script>

</body>
</html>
```

Listing 1.6: angular/index.html

```
<!doctype html>
<html ng-app="todoApp">
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.min.js"></script>
    <script src="../../../plezuro.js"></script>
    <script src="todo.plez.js"></script>
    <link rel="stylesheet" href="todo.css">
  </head>
  <body>
    <h2>Todo</h2>
    <div ng-controller="TodoListController as todoList">
      <span>{{todoList.remaining()}} of {{todoList.todos.length}} remaining</span>
      [ <a href="" ng-click="todoList.archive()">archive</a> ]
      <ul class="unstyled">
        <li ng-repeat="todo in todoList.todos">
          <input type="checkbox" ng-model="todo.done">
          <span class="done-{{todo.done}}">{{todo.text}}</span>
        </li>
      </ul>
      <form ng-submit="todoList.addTodo()">
        <input type="text" ng-model="todoList.todoText"  size="30"
               placeholder="add new todo here">
        <input class="btn-primary" type="submit" value="add">
      </form>
    </div>
  </body>
</html>
```

Listing 1.7: angular/todo.css

```
.done-true {
  text-decoration: line-through;
  color: grey;
}
```

```
angular.module('todoApp', [])
  .controller('TodoListController'.toString, {
    $todoList = this;
    todoList['todos'] = [
      %('text': 'learn angular', 'done': true),
      %('text': 'build an angular app', 'done': false)
    ];

    todoList['addTodo'] = {
      todoList['todos'].push(%('text': todoList['todoText'], 'done':false));
      todoList['todoText'] = ''
    };

    todoList['remaining'] = {
      $count = 0;
      angular.forEach(todoList['todos'], {
        $todo = first;
        {!todo['done']}.if({ count++ })
      });
      count
    };

    todoList['archive'] = {
      $oldTodos = todoList['todos'];
      todoList['todos'] = [];
      angular.forEach(oldTodos, {
        $todo = first;
        {!todo['done']}.if({
          todoList['todos'].push(todo)
        })
      })
    }
  })
```

Specyfikacja

To jest formalna specyfikacja języka programowania Plezuro. Celem jest zdefiniowanie zasad, które muszą być zaimplementowane w każdej wersji tego języka.

Dla kogo to jest?

Dla programistów, którzy chcieliby zaimplementować inną wersję Plezuro, bibliotekę, rozszerzenie jak również dla standardowych użytkowników, którzy pragną poszerzyć swoją wiedzę. Przewodnik jest bardziej dla nauki i prostego zrozumienia, zaś specyfikacja bardziej dla dokładnej wiedzy jak aplikacje w Plezuro powinny się zachowywać i co za tym idzie, jak kompilatory i interpretatory powinny działać.

Jednak to kod jest najlepszym sposobem na dokładne zdefiniowanie zachowania. Zatem każda wersja Plezuro musi przejść pozytywnie wszystkie testy automatyczne napisane dla głównej wersji.

2.1 Typy tokenów

Pierwszym etapem kompilacji jest tokenizacja.

Plezuro posiada następujące typy tokenów (kolejność ma znaczenie, kiedy regex lub warunek wielu tokenów pasuje, pierwszy z nich wygrywa):

Typ	Regex lub warunek	Przykład
komentarz jednoliniowy	\/\/.*	//a comment
komentarz wieloliniowy	\/*.*	/*a comment*/
regex	r((‘([^’]|(‘’))*’)|(“([^”]|(“”))*”))	r’[a-k]’
nawias	\[|\]|\(|\)|\{|\}|(#\()|(\$\()|(%\(	{this + first}
liczba	‘^(0x[0-9a-f]+)|(0b[01]+)|(0[0-7]+)|([0-9]+(.[0-9]+)?(e[+-]?[0-9]+)?)$’	23.45e56
deklaracja	\$[A-Za-z_]+[A-Za-z_0-9]*	$aVariable
pole klasy	[A-Za-z_]+[A-Za-z_0-9]*::[A-Za-z_]+[A-Za-z_0-9]*	Person::totalNr
pole obiektu	@[A-Za-z_]+[A-Za-z_0-9]*	@name
biały znak	[\t]+	
operator	jeden ze zdefiniowanych ciągów znaków	+
symbol	[A-Za-z_]+[A-Za-z_0-9]*	aVariable
string	(‘’)|(‘.*?([^\\]|(\\\\))’)|(“”)|(”.*?([^\\]|(\\\\))”)	‘a text’

2.2 Białe znaki i komentarze

Komentarze zachowują się tak, jak by były zastąpione pojedynczą spacją. Zatem kod wewnątrz nie jest wykonywany. Tak samo białe znaki nie są wykonywane. Chociaż mają minimalny wpływ na kod: nie można wstawić komentarza (ani białego znaku wewnątrz nazwy symbolu (zmiennej); jedyna możliwość to wstawiać komentarze pomiędzy różnymi tokenami.

2.3 Nawiasy

Istnieją następujące typy nawiasów:

Otwarcie	Domknięcie	Użycie
(	)	porządek operatorów
[	]	tworzenie listy, dostęp do elementu
{	}	funkcja
$(	)	tworzenie zbioru
#(	)	tworzenie słownika
%(	)	tworzenie tablicy asocjacyjnej

Ogólne zasady pomiędzy nawiasami:

1. Każdy nawias musi być domknięty.
2. Domknięcie nawiasu bez wcześniejszego jego otwarcia wyrzuca wyjątek.
3. Zasada stosu: każde otwarcie nawiasu jest wrzucane na stos, domknięcie nawiasu powoduje ściągnięcie elementu ze stosu, kiedy ściągnięty element nie pasuje do domknięcia nawiasu, wyrzucany jest wyjątek.

2.4 Stałe tokeny

Stały token oznacza token, którego wartość jest na twardo wpisana w kodzie.

Typy stałych tokenów:

- Number
- String

2.4.1 Number

Każda liczba (Number) is zmiennoprzecinkowa. Ogólnie odzwierciedla to matematyczne liczby rzeczywiste.

Istnieją następujące notacje:

Nazwa	Regex	Przykład
Dziesiętny (w tym notacja naukowa)	[0-9]+(\.[0-9]+)?(e[\+\-]?[0-9]+)?	1.2e45
Binarny	0b[01]+	0b1101
Ósemkowy	0[0-7]+	072
Szesnastkowy	0x[0-9a-f]+	0xa4f

2.4.2 String

To zbiór znaków unicode dowolnej długości (jedyny limit to pamięć zarezerwowana dla aplikacji). Znakiem ograniczającym string jest albo pojedynczy apostrof `'` albo cudzysłów `"`. Znaki specjalne wewnątrz stringa muszą być zapisane za pomocą backslasha `\`.

Lista znaków specjalnych

- \t - tabulator
- \n - nowa linia
- \\ - backslash

2.5 Symbole

Symbol oznacza nazwę nadaną przez programistę do czegokolwiek w programie (zależnie od języka to może być zmienna, funkcja, klasa, struktura, unia, trait, interfejs itd.). W Plezuro wszystko jest zmienną, więc każdy symbol oznacza zmienną.

The rules of the variable naming:

1. Pierwszy znak musi być literą z kodu ASCII, lub podkreśleniem _.
2. Każdy z następnych znaków musi być literą z kodu ASCII, cyfrą lub podkreśleniem _.

2.5.1 Deklaracja

Każdy symbol musi być zadeklarowany przy pierwszym użyciu w pliku źródłowym. Deklaracja zawiera znak dolara `$` a następnie nazwę symbolu.

2.5.2 Pola klasy

Pole klasy (inaczej pole modułu) zawiera w swej składni nazwę modułu, podwójny dwukropek `::` oraz nazwę pola (tutaj te same zasady jak w przypadku symbolu).

2.5.3 Pola obiektu

Pole klasy (inaczej pole modułu) zawiera w swej składni nazwę modułu, podwójny dwukropek `::` oraz nazwę pola (tutaj te same zasady jak w przypadku symbolu).

2.6 Operatory

Reguła znajdowania operatora:

- Jednoargumentowy operator (operator po lewej, argument po prawej) jest jednym z:
  - "!"
  - "&&"
  - "**"
  - "#"

  - “~”
  - “=>”
- Jednoargumentowy operator (operator po lewej, argument po prawej) jest jednym z:
  - “++”
  - “–“
- Dwuaargumentowy operator (zerowy argument, operator, pierwszy argument, kolejność od tych wykonywanych na samym końcu do tych samym początku) jest jednym z:
  - ”;”
  - ”,”
  - ”:=”
  - “=”
  - “+=”
  - “-=”
  - “*=”
  - “/=”
  - “^=”
  - “&=”
  - “|=”
  - “%=”
  - ”.=”
  - “~~”
  - “<->”
  - “<<”
  - “>>”
  - ”?”
  - “|”
  - “&”
  - “<=>”
  - “>=”
  - “>”
  - “<=”
  - “<”
  - ”!=”
  - “==”
  - ”!==”
  - “===”

- “=~”
- ”!~”
- “+”
- “-“
- “%”
- “*”
- “/”
- “^”
- “^^”
- ”.”
- ”..”
- ”:”

2.7 Błędy składniowe

Z powodu bardzo prostej składni jest bardzo mało typów błędów składniowych. Nawet instrukcje warunkowe i pętle są tworzone jako wywołania metod, więc błędy składniowe mogą wystąpić jedynie, gdy tokeny nie są ułożone w odpowiedni sposób.

2.7.1 Jak wykryć błędy składniowe?

W Plezuro nie ma fatalnych błędów. Każdy błąd to wyjątek. Kiedy skrypt zawierający błąd składniowy jest importowany, wyjątek powinien zostać wyrzucony. Kompilator powinien stworzyć plik wynikowy (lub funkcję w pliku wynikowym, gdzie wszystkie skrypty są łączone razem), który wyrzuci wyjątek.

2.7.2 Pełna lista błędów składniowych

Nazwa	Wystąpienie	Przykłady
BracketStackException	nawias nie jest domknięty, zamknięcie nieodpowiedniego typu nawiasu, nadmierne domknięcie nawiasu	(2 + 3 (3 + 1 + [)] 4 + 5)
NonExistentTokenException	token nieistniejącego typu	$$\alpha\beta\gamma$$ = 21
OperatorAfterBracketCloseException	po zamknięciu nawiasu coś innego niż operator lub następne domknięcie nawiasu	$x=[2] “oo”
OperatorAfterBracketOpenException	nieodpowiedni operator po otwarciu nawiasu lub początku skryptu	* 43
OperatorAfterOperatorException	operator po operatorze (chociaż są wyjątki od tego)	2 + * 5
OperatorBeforeBracketCloseException	operator przed domknięciem nawiasu	(2 + 3 -)
ValueAfterValueException	stały token po innym stałym tokenie	3 + *

2.8 Zmienne

Wszystko można przypisać do zmiennej. Każda zmienna musi być zadeklarowana.

2.8.1 Zasięg

Deklaracja określa zasięg zmiennej. Jest możliwe ukryć zmienną poprzez inną o tej samej nazwie.

```
$x = 21;
$y = 30;
$f = {
  $x = this;
  x + 2
};
$a = f(x); // should return 23
$b = f(y); // should return 32
[a, b]
```

2.9 Wyrażenia

Wyrażenie to pojedyncza komenda oddzielona średnikiem `;`. Chociaż może być wiele instrukcji (np. `x = y = 2 + 4`). Co to znaczy dokładnie? Z punktu widzenia Plezuro, wyrażenie jest najmniejszą, niepodzielną instrukcją, ale z punktu widzenia CPU (lub nawet czysto matematycznego CPU) jedno wyrażenie może zawierać wiele instrukcji.

2.10 Funkcje

Funkcja to zbiór operacji, które mogą być wykonywane w dowolnym momencie w trakcie wykonywania programu. Funkcja bierze dowolną ilość argumentów, które to są dostępne poprzez następujące słowa kluczowe:

- this - zerowy argument
- first - pierwszy argument
- second - drugi argument
- third - trzeci argument
- args - lista argumentów

W przypadku funkcji wewnętrznej, argumenty zewnętrznej funkcji są ukryte argumentami wewnętrznej.

Każda funkcja zwraca jakąś wartość. Typ nie nie jest określony. Są dwa sposoby na zwrócenie wartości:

- jawnie - za pomocą słowa kluczowego `return`
- sposób domniemany - ostatnie wyrażenie jest zwracane

2.10.1 Metody

Metoda działa w ten sam sposób co funkcja. Jedyna różnica to to, że obiekt (zmienna przed kropką) jest przekazany jako zerowy argument, a następne argumenty są numerowane od 1. W wywołaniu metody, kiedy nie ma argumentów (za wyjątkiem obiektu this), nawias po nazwie metody nie jest wymagany.

2.10.2 Skrypty

Nawet skrypty zachowują się w ten sam sposób co funkcja. Biorą argumenty (dostęp w ten sam sposób) i zwracają wartość.

2.11 Magiczne stałe

Każdy kompilator (a także intepretator) Plezuro, musi obsłużyć następujące magiczne stałe:

- __pos__ - pozioma pozycja w linii (licząc od 0, to pozycja gdzie zaczyna się słowo kluczowe)
- __line__ - numer linii w pliku źródłowym (licząc od 0)
- __file__ - nazwa pliku źródłowego
- __dir__ - pełny folder pliku źródłowego

Jest zalecane zastąpić każde z powyższych słów kluczowych w kodzie wynikowym poprzez stringa. Nie ma zalecenia co do kodu pośredniego. __dir__ nie musi zwracać ścieżki w formie kanonicznej (wszystkie podfoldery od najwyższego do najniższego (np. '/home/user/programming/plezuro/my_plezuro_project') ale musi zawierać prawidłową pełną ścieżkę unixową (np. '/home/user/programming/something/project/../../plezuro/my_plezuro_project').

Nawet dla systemów nie unixowych ścieżka musi być w notacji unixowej (ze slashami).

2.12 Moduły

Moduł jest jednocześnie klasą, przestrzenią nazw i statycznym modułem. Moduł określa typ zmiennej. Każda zmienna jest związana z jakimś modułem (wbudowanym lub stworzonym przez programistę Plezuro). Zatem moduł określa akcję przy wywołaniu metody lub operatora.

2.12.1 Konstruktor

Zawsze po stworzeniu obiektu konstruktor jest wywołany. Jego nazwa to 'init'.

2.12.2 Dziedziczenie

Moduł może dziedziczyć po wielu innych modułach. Relacja dziedziczenia jest przechodnia, nierówna. To znaczy, że moduł nie może dziedziczyć po samym sobie, a także przodek modułu nie może być jednocześnie jego potomkiem. Dziedziczenie dotyczy metod i operatorów (bez pól i metod statycznych). Można nadpisywać to, co jest odziedziczone. Kiedy moduł nie dziedziczy jawnie, dziedziczy po Module.BasicModule, który to jest przodkiem wszystkich modułów.

2.12.3 Pola i metody statyczne

Pola i metody statyczne są związane dokładnie z jednym modułem. Nie są dziedziczone.

2.12.4 Duck typing

Wszędzie występuje duck typing. To znaczy, że da się wywołać metodę z daną nazwą z obiektu, jeśli taka metoda istnieje w jego module (bezpośrednio lub jest odziedziczona). Ponadto obiekty mogą być przekazywane jako zerowy argument do metod z zupełnie niezależnych modułów.

ROZDZIAŁ 3

Implementacja

To jest dokumentacja implementacji głównej wersji Plezuro.

Dla kogo to jest?

Dla programistów zamierzających pisać rozszerzenia do języka oraz do tych, którzy chcą stworzyć ich własne implementacje (dla otrzymania dokładnej wiedzy jak to działa i jak napisać kompilator w prosty sposób)

3.1 Narzędzia

Podstawowym narzędziem jest Java 1.8. Aczkolwiek w przyszłości jest zaplanowany kompilator zaimplementowany w Plezuro.

Aby w pełni wykorzystać źródło Plezuro, potrzebujesz:

1. JRE 1.8
2. JDK 1.8
3. Unix-podobny system operacyjny.
4. Realpath
5. Node.js
6. Nesh

W systemach nie-unixowych rozwijanie Plezuro jest nadal możliwe, chociaż trzeba będzie wtedy przepisać skrypty automatyzujące.

3.2 Główne części

Kod źródłowy jest podzielony na następujące części:

- node_modules - zewnętrzne biblioteki dające funkcjonalności większe niż standardowy Javascript
- scripts - automatyzujące skrypty do kompilowania, uruchamiania i testowania Plezuro
- src/java - kod źródłowy kompilatora w Javie:
- src/js - biblioteki Javascript dające możliwości Plezuro do Javascripta

3.3 Ogólny algorytm

Możemy określić główne części algorytmu:

1. Kompilacja
   (a) Wczytanie skryptu.
   (b) Sprawdzenie czy skrypt nie jest pusty (w przeciwnym razie zwraca null).
   (c) Podział kodu na linie.
   (d) Tokenizacja:
      - linie są dzielone na tokeny
      - z wyjątkiem tokenów wieloliniowych, które mogą się rozciągać na wiele linii
   (e) Ewentualna zmiana typów tokenów (np. z podstawowych do ich subtokenów).
   (f) Walidacja - wykrycie błędów składniowych, badając kolejność tokenów.
   (g) Konwersja tokenów Plezuro to tokenów języka wynikowego.
   (h) Zapis wyjścia do pliku.
2. Wiązanie bibliotek z języka wynikowego (obecnie Javascript).

3.4 Skrypty

W folderze ‘scripts’ są narzędzia do kompilacji, uruchamiania i testowania Plezuro

Nazwa	Akcja	Przykład
build.sh	kompilowanie kodu Javy do plików .class	scripts/build.sh
clear.sh	usuwanie wszystkich wygenerowanych plików	scripts/clear.sh
deploy_js.sh	kompilowanie pojedynczego skryptu (do folderu ‘bin’) dla plików Plezuro, kopiowanie dla pozostałych plików, rekurencyjne działanie dla folderów	scripts/deploy_js.sh src/plezuro/tests/1.plez
export_jar.sh	eksport plików .class to pojedynczego .jar	scripts/export_jar.sh
make.sh	build.sh, export_jar.sh, kopiowanie pliku .jar do ‘/usr/bin’ make_js.sh	scripts/make.sh
make_js.sh	make_js_lib.sh, kompilowanie skryptów Plezuro z ‘src/plezuro’ do ‘bin/js’	scripts/make_js.sh
make_js_lib.sh	łączenie wszystkich bibliotek Javascript do bin/js/plezuro.js	scripts/make_js_lib.sh
test.sh	testowanie skryptów Plezuro	scripts/test.sh

Jak widzisz, niektóre ze skryptów wywołują inne. Zatem aby zrobić wszystko, czego potrzebujesz, po prostu uruchom trzy z nich:

scripts/clear.sh

scripts/make.sh

scripts/test.sh

Te trzy nie są połączone w jeden, aby ich wyjście się nie pomieszało.

3.5 Kompilator

Cały kompilator jest obecnie zaimplementowany w Javie.

Jest podzielony na następujące pakiety:

- mondo.engine - jądra kompilatora
- mondo.invalidToken - klasy wyjątków związanych z błędami składniowymi i ich wiązanie z językiem wynikowym
- mondo.main - główna klasa i funkcja
- mondo.token - translacja konkretnych tokenów do wynikowego języka

mondo.main i mondo.engine są niezależne od języka wynikowego, natomiast mondo.token oraz mondo.invalidToken są zależne. Zatem jeśli chcesz napisać kompilator Plezuro to innego języka niż Javascript, powinieneś zmienić jedynie pakiety mondo.token oraz mondo.invalidToken.

3.5.1 Pakiet engine

To jądro kompilatora. Jest niezależne od języka wynikowego.

Klasy:

- Engine - obsługuje argumenty kompilatora. Dla każdej pary plik źródłowy - plik wynikowy wywołuje Parser.
- Parser - czyta wejście, wychwytuje sytuację z pustym plikiem, wywołuje Tokenizer, wywołuje podstawowe operacje na każdym tokenie i ostatecznie zapisuje na wyjście.
- Tokenizer- dzieli kod na tokeny według warunków danych w klasach tokenów.
- Validator - wykrywa wszystkie możliwe błędy składniowe.

3.5.2 Pakiet invalidToken

To zbiór wszystkich klas wyjątków. Zawiera ich mapowanie do Javascripta. Wszystkie pozostałe klasy dziedziczą po InvalidTokenException.

3.5.3 Pakiet token

To zbiór wszystkich tokenów wraz z subtokenami występujących w Plezuro oraz ich translacja do Javascripta.