TKOM - etap 2

Hubert Truszewski

Temat: 4 - język opisu brył i ich właściwości. Wersja ze statycznym i silnym typowaniem.

Opis funkcjonalności

W ramach języka będą wspierane różne typy brył i możliwe będzie ich wyświetlenie na ekranie. Dostępne bryły:

- prostopadłościan
- ostrosłup prawidłowy trójkątny
- stożek
- walec
- kula

Dla prostopadłościanu, ostrosłupa, stożka oraz walca dostępne są metody do obliczenia:

- objętości
- pola podstawy
- pola powierzchni bocznej
- pola powierzchni całkowitej

Dla kuli dostępne bedą metody do obliczenia średnicy, objętości oraz pola powierchni całkowitej.

Każdy z typów brył udostępnia dostęp do wartości atrybutów przez następujące pola:

- prostopadłościan - a, b, c - długości krawędzi
- ostrosłup prawidłowy trójkątny - a - długość krawędzi podstawy, H - wysokość ostrosłupa
- stożek - r - długość promienia podstawy, l - długość tworzącej stożka
- walec - r - długość promienia podstawy, H - wysokość walca
- kula - R - promień kuli.

Dostępne typy primitywne: int, double, bool, string.

Do przechowywania kolekcji dostępny jest generyczny typ `List<?>`. Aby dodać bryłę do kolekcji należy wywoałać metodę `add()` na liście. W celu dostępu do przechowywanych wartości należy wywołać metodę `next()` na liście, która zwróci kolejną wartość.

Język wymaga, aby w kodzie była funkcja o nazwie `main`, która jest punktem wejścia do wykonywania kodu.

Dostępne jest instrukcja warunkowa `if` oraz pętla `while`, które sprawdzają prawdziwość podanego warunku na podstawie jego rzutowania na typ bool.

Argumenty przekazywane do funkcji są przekazywane przez wartość, a zmienne są niemutowalne. Zakres widoczności zmiennej to obręb bloku, w którym została zadeklarowana.

Do wyświetlenia kolekcji służy obiekt typu `Screen`. Aby to zrobić należy utworzyć obiekt tego typu oraz wywołać na nim metodę `show()`, która jako argument przyjmuje listę brył.

Przykładowy kod

```
void showFigures(List<SolidFigure> figuresList) {
    Screen screen = Screen;
    int counter = 0;
    while (counter < figuresList.length) {
        screen.add(figuresList.next());
    }
    screen.show();
}

// Przykład funkcji, zwraca sumę objętości wszystkich brył w kolekcji
int totalVolumes(List<SolidFigure> figuresList) {
    // Deklaracja zmiennej i jednoczesna jej inicjalizacja
    int volumes = 0;
    // Przykład pętli while
    while (counter < figuresList.length) {
        counter = counter + figuresList.next().volume();
    }
    // Zwracanie obliczonej wartości
    return volumes;
}

// Funkcja main, która stanowi punkt wejścia do programu
void main() {
    // deklaracja listy dla brył
    List<SolidFigure> figureList;
    // Tworzenie nowej bryły
    Sphere s = Sphere(5);
    // Dodanie bryły do listy
    figureList.add(s);
    Cone c = Cone(2, 5);
    figureList.add(c);
    // Wyświetlenie brył na ekranie
    showFigures(figureList);
}
```

Operatory

Precedencja

1 = najwyższy priorytet

Priorytet	Operator	Łączność
1	Dostęp do obiektu (.)	lewostronny
2	Negacja (!)	prawostronny
3	Rzutowanie (as)	lewostronny
4	Mnożenie oraz dzielenie (*, /)	lewostronny

Priorytet	Operator	Łączność
5	Dodawanie oraz odejmowanie (+, -)	lewostronny
6	Porównanie (==, !=, >, <, >=, <=>)	lewostronny
7	operacja AND (&&)	lewostronny
8	operacja OR (||)	lewostronny

Opis działania

Operatory +, -, *, / działają w przypadku działań na typach primitywnych. Operatory porównania działają zarówno dla typów prymitywnych jak i dla brył, dla których wykorzystują wartość objętości dla porównań, ponieważ jest to jedyna wspólna cecha wszystkich typów.

Komentarze

- jednolinikowe np. `// jakis tekst`

Obsługa błędów

Jeżeli w czasie działania zostanie napotkany błąd, np. niedomnknięty nawias zostanie rzucony wyjątek z odpowiednim komunikatem. Kod:

```
    if (a {
        int a = 5;
    }
```

Przykład:

```
Error in line 4, position 10: missing right parenthesis
```

Podział na komponenty:

- aplikacja - odpowiada za otworzenie aplikacji oraz załadowanie pozostałych komponentów i ich zależności
- input - wejście kodu - z pliku lub ze standardowego wejścia
- lekser - odpowiada za analizę leksykalną, generuje tokeny
- parser - odpowiada za analizę składniową
- interpreter - wykonuje faktyczny kod programu, odpowiada za przechwycenie zgłaszanych wyjątków

Testowanie

Testować zamierzam głównie pisząc testy jednostkowe przy użyciu `JUnit`. Każdy z komponentów będzie posiadał osobny zestaw testów.

Składnia w formacie EBNF:

```
program                   = {function_declaration};
function_declaration      = type, identifier, "(", {type, identifier},
")", code_block;
code_block                = "{", statement, "}";
statement                 = conditional_statement
                          | variable_declaration
                          | variable_assignment
                          | variable_decl_init
                          | return_statement
                          | function_call;
conditional_statement     = if_statement | while_loop;
if_statement              = "if (", expression, ")", code_block, ["else",
code_block];
while_loop                = "while (", expression, ")", code_block;
variable_declaration      = type, identifier;
variable_decl_init        = type, variable_assignment;
variable_assignment       = identifier, "=", expression, ";";
return_statement          = "return", [expression], ";";
expression                = or_expression;
or_expression             = and_expression, {or_operator, and_expression};
and_expression            = coparison_expression, {and_operator,
coparison_expression};
coparison_expression      = addition_expression, {comparison_opearator,
addition_expression};
addition_expression       = multiplication_expression, {addition_operator,
multiplication_expression};
multiplication_expression = casting_expression, {multiplication_operator,
casting_expression};
casting_expression        = negation_expression, [casting_operator, type];
negation_expression       = [negation_operator], access_expression;
access_expression         = simple_expression, {access_operator,
identifier, ["(", [simple_expression | identifier], ")"]};
simple_expression         = number | string_literal | function_call | "(",
expression, ")";
function_call             = identifier, "(", {expression}, ")", ";", ;
type                      = "int"
                          | "string"
                          | "double"
                          | "bool"
                          | "void"
                          | "Cone"
                          | "Cylinder"
                          | "Sphere"
                          | "Cuboid"
                          | "Pyramid"
                          | "Screen";
identifier                = letter, {letter | digit};
string_literal            = '"', {string_element}, '"';
string_element            = letter | escape_character | digit |
special_character;
letter                    = "A" | "B" | ... | "Z" | "a" | "b" | ... | "z";
escape_character          = "\";
```

```
special_character          = " " | "." | "," | "@" | "!" | "#" | ...
number                     = ["-"], (int_number | float_number);
float_number               = int_number, ".", int_number;
int_number                 = "0" | (digit_non_zero, {digit});
digit                      = "0" | digit;
digit_non_zero             = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
| "9" |;

addition_operator          = "+" | "-";
multiplication_operator    = "*" | "/";
negation_operator          = "!";
access_operator            = ".";
comparison_operator        = "==", "!=", ">", "<", ">=", "<=";
and_operator               = "&&";
or_operator                = "||";
casting_operator           = "as";
```