TKOM - Dokumentacja wstępna

Dawid Kaszyński

Wymagania funkcjonalne

- Typowanie silne statyczne
- Dostępne proste typy danych:
  - int
  - float
  - string
- Dostępne kolekcje
  - słownik - obsługujący jako klucze i wartości typy proste
- Zmienna każdego z typów typu może mieć konkretną wartość albo wartość *null*
- Istnieje operator *?*, który sprawia, że jeśli przy odwołaniu do klucza słownika dany klucz nie istnieje to zamiast wyjątku *NullPointerException* powodującego zakończenie wywołania programu zostania przypisana wartośc *null*.
- Pętla *while*
- Pętla *foreach* umożliwiająca iterowanie po kolekcji (kluczach słownika)
- Instrukcje warunkowe *if*, *else*
- Definiowane funkcji, również z możliwością użycia *void*
- Wbudowana funkcja *print* służąca do wypisania tekstu na standardowe wyjście.
- Wbudowana funkcja input wczytająca tekst od użytkownika ze standardowego wejścia
- Wbudowana funkcja *exit* powodująca zakończenie programu z kodem podanym jako argument
- Obsługa komentarzy, które są oznaczane znakiem #
- Wykonanie programu rozpoczyna się od funkcji *void main* (w większości z poniższych przykładów została ona pominięta dla czytelności)
- Argumenty do funkcji są przekazywane przez wartość dla typów prostych i przez referencję dla kolekcji (słownika)
- Zmienne widoczne wewnątrz bloku na którego poziomie zostały zadeklarowane

Przykłady użycia

Deklaracje do zmiennych oraz przypisanie

```
int x = 3 * 3 + 2 - 1;
int y = 4;

int z;
z = x + y;

int a = z as float + 2.0;
```

```
string b = "Ala" + " " + "ma" + " " + "kota" + a as string;
```

Użycie słownika z przekazaniem go do funkcji przez referencję

```
void modifyDict(dict<string, int> mp, string key, int val) {
    mp[key] = val;
}

void main() {
    dict<string, int> mp = {"a": 1, "b": 2, "c": 3};

    modifyDict(mp, "a", 3);
    modifyDict(mp, "d", 4);

    int x = mp["a"];

    foreach (string key : mp) {
        print(mp[key]);  # 3 2 3 4
    }
}
```

Obsłużenie wyjątku *NullPointerException* przy braku klucza w słowniku

```
dict<int, int> mp = {};
    int y = mp["z"]?;  # Obsłużone NullPointerException, y = null
    if (y == null) {
        print("Brak klucza z");
    }
```

Odwołanie do nieistniejącego klucza słownika bez operatora *?*

```
dict<int, int> mp = {};
int x = mp[1]  # NullPointerException
```

Proste typy przekazywane do funkcji przez wartość

```
void notWorkingModifyInt(int val) {
    val = val + 1;
}

int x = 1;
notWorkingModifyInt(x);
print(x);  # 1
```

Instrukcja *if* z wieloma warunkami i deklaracja funkcji

```
float getFloat() {
    return 3.5;
}

void printWithPrefix(string s) {
    print("Result: " + s);
}

int a = 1;
int b = 3;

if (a == b || (a < b && b <= getFloat() as int)) {
      printWithPrefix("a");  # Ten warunek zostanie wykonany
} else if ( 1 == 1) {
     printWithPrefix("b");
} else {
     printWithPrefix("c");
}
```

Deklaracja i wywołanie funkcji, użycie instrukcji *if*, *else* oraz *while*

```
int fib(int n) {
      int a = 0;
      int b = 1;

      if (n == 0) {
            return a;
```

```
      } else if (n == 1) {
            return b;
      }

      int i = 2;
      while (i <= n) {
            int tmp = b;
            b = a + b;
            a = tmp;
      }

      return b;
}

void main() {
    int x = fib(10);
    print(x as string);
}
```

Wbudowane funkcje input i exit

```
print("Podaj hasło");

string password = input();

if (password != "secret") {
    print("Złe hasło");
    exit(1);
}
```

Operacja na zmiennej z wartością oraz na zmiennej o wartości null

```
int x = 3;
int y = null;
int z = x + y;  # Wykonanie programu zostanie przerwane z błędem
                # NullPointerException
```

Zapis gramatyki

```
program = {variableDeclaration | functionDefinition};
```

```
statement = ifStatement |
            whileStatement |
            forEachStatement |
            variableDeclaration |
            assignment |
            functionCall |
            returnStatement;

functionDefinition = functionReturnType identifier parameterList
statementBlock;

parameterList = "(" [type identifier, {"," type identifier}] ")";

statementBlock = "{" {statement} "}";

ifStatement = "if" "(" expression ")" statementBlock ["else"
statementBlock];

whileStatement = "while" "(" expression ")" statementBlock;

forEachStatement = "foreach" "(" simpleType identifier ":" identifier
")" statementBlock;

variableDeclaration = type identifier ["=" expression] ";";

assignment = (identifier "=" expression ";") |
             (identifier "[" expression "]" "=" expression ";");

functionCall = functionCallAsExpression ";";

argumentList = "(" [expression, {"," expression}] ")";

returnStatement = "return" expression ";";

expression = andExpression {orOperator andExpression};

andExpression = relationExpression {andOperator relationExpression};

relationExpression = additiveExpression {relationOperator
additiveExpression};

additiveExpression = multiplicativeExpression {additiveOperator
multiplicativeExpression};

multiplicativeExpression = negatedSingleExpression
{multiplicativeOperator negatedSingleExpression};
```

```
negatedSingleExpression = ["!" | "-"] castedExpression;

castedExpression = singleExpression ["as" simpleType];

singleExpression = identifier |
                   literal |
                   "(" expression ")" |
                   functionCallAsExpression |
                   identifier "[" expression "]" ["?"];

functionCallAsExpression = identifier argumentList;

functionReturnType = type | "void";

type = simpleType | parametrizedType;

parametrizedType = collectionType "<" simpleType "," simpleType ">";

collectionType = "dict";

simpleType = "int" |
             "float" |
             "string";

relationOperator = "<" |
                   ">" |
                   "<=" |
                   ">=" |
                   "==" |
                   "!=";

orOperator = "||";

andOperator = "&&";

additiveOperator = "+" |
                   "-";

multiplicativeOperator = "*" |
                         "/"
                         "%";

identifier = (letter | "_") {letter | digit | "_"};

literal = stringLiteral |
```

```
          numberLiteral |
          dictionaryLiteral |
          "null";

letter = "a".."z" | "A".."Z";

digit = "0" | nonZeroDigit;

nonZeroDigit = "1" .. "9";

stringLiteral = "\"" .. "\"";

numberLiteral = nonZeroDigit {digit} ["." digit {digit}];

dictionaryLiteral = "{" [expression ":" expression {"," expression ":"
expression }] "}";
```

Realizacja techniczna

- Implementacja w Javie
- Budowanie projektu z gradle
- Testy jednostkowe i integracyjne JUnit