Funkcje

#function #arguments #parameters #return

Author: Piotr Niemczuk
Nickname: Pyoneru

Teoria

Twój biznes z tworzeniem origami się rozwija!
Postanawiasz zatrudnić pomocnika. Niestety okazał się on trochę nie ogarnięty.

Jego zadaniem jest skupowanie papieru do tworzenia orgiami, niestety za każdym razem musisz mu tłumaczyć co ma zrobić.

1. Idź do papierni.
2. Odbierz zamówienie na nasz sklep
3. Zapłać
4. Wróć z papierem

Postanawiasz dla niego stworzyć instrukcje kroków, w której zawierasz wszystko co za każdym razem mu powtarzasz (nie, nie zwalaniasz go).

```
Instrukcja odebrania zaopatrzenia

Krok 1.
Weź pieniadze od właściciela (ode mnie!). Nie wydawaj je na głupoty!

Krok 2.
Idź do papierni.

Krok 3.
Odbierz zamówienie na nasz sklep.
(kiedy zapytają o pieniądze, patrz krok 4)

Krok 4.
Zapłać. Pieniędzmi którymi ode mnie dostałeś!
```

```
Krok 5.
Wróć z papierem do sklepu
```

Od tej pory za każdym razem kiedy wysyłasz go po zaopatrzenie to przekazujesz mu tą instrukcję, oszczędzając sobie czasu.

Funkcja to skondensowanie intrukcji które często wykonujesz do wywołania jednej. Zamiast tłumaczyć pomocnikowi za każym razem wszystkie kroki, przekazujesz mu już wcześniej gotową instrukcję.

Praktyka

Przykładowa funkcja

```java
public class Main{

        // Defenicja funkcji
        public static int add(int a, int b){
                return a + b;
        }

        public static void main(String[] args){
                // Wywołanie funkcji, wewnątrz funkcji main.
                int result = add(5, 3);
        }
}
```

> ⓘ **Funkcje tworzymy separatycznie!**
>
> Nie tworzymy funkcji, wewnątrz innej funkcji.
> Możemy **wywołać** jedną funkcje, wewnątrz drugiej ale nie możemy jej **zdefinować**!
> Funkcja main to również funkcja, jak sama nazwa wskazuje :)

Anatomia funkcji

```
public static RESULT_TYPE NAME(PARAMETERS){
        // body
        return RESULT_TYPE;
}
```

> ⚠ **Nie zawsze public static**
>
> Funkcja nie zawsze będzie zaczynała się od `public static`, ale na ten moment przyjmij że tak jest. Omówimy to w następnych materiałach.

RESULT_TYPE

Funkcja może nam przekazać informację zwrotną. Wówczas mówimy że funkcja zwraca jakąś wartość.

> *Powracając do naszego pracownika.*
> *Pracownikowi wręczamy pieniądzę - są to parametry.*
> *Następnie wykonuje dla nas jakieś działanie - ciało funkcji.*
> *A na koniec przynosi nam nowy papier - zwraca wartość.*

```
public static int something(){ // funkcja zwraca typ int
        return 5; // wiec musimy zwrócić typ int
}

public static String something(){ // funkcja zwraca typ String
        return "Ala ma kota, a kot ma Ale"; // wiec musi zwrócić typ String
}

public static void something(){ // funkcja zwraca typ void
        // funkcja nie musi niczego zwracać
}
```

`void` - pustka, czyli typ pusty. `void` nie przechowuje żadnej wartości.

PARAMETERS | Optional

Funkcja może posiadać parametry. Czyli możemy jej przekazać argumenty aby zrobiła coś na ich podstawie.

Parametry definiujemy w postaci `typ` i `nazwa` po przecinku.

```
public static void something(String param1, int param2){} // Funckja przyjmuje 2 argumenty, pierwszy typu String, drugi typu int
```

> ⓘ **Parametry vs Argumenty**
>
> Parametrami nazywamy zmienne w definicji funkcji. Natomiast kiedy wywołujemy funkcje i przekazujemy jej faktyczne wartości, to nazywamy je już argumentami. Nie przejmuj się jeżeli Ci się to myli. Sam sprawdzałem co jest które zanim to napisałem :D
>
> ```
> public static void something(int a, int b){} // parametry
> something(2, 3); // argumenty
> ```

return | Optional

Słowa kluczowego `return` używamy kiedy chcemy zwrócić jakąś wartość z funkcji.

```
public static int add(int a, int b){
        int result = a + b;
        return result; // Zwróć wartość zmiennej `result`
}
```

Funkcja może mieć więcej niż jedno `return`, a czasem jest to nawet wymagane.

```
public static isOdd(int number){
        if(number % 2 == 1){
                return true; // true jezeli liczba jest nie parzysta
        }else{
                return false; // false jezeli liczba jest parzysta
        }
}
```

Jeżeli funkcja korzysta z instrukcji warunkowych to musisz rozpatrzeć wszystkie możliwości kiedy funkcja zwraca jakąś wartość.

Wyobraź sobie że piszesz grę, gracz może spotkać jednego z 3 rodzajów potworów

- Goblin

- Szczur
- Złośnik
  Każdy z tych potworów ma różne statystyki i różną liczbe punktów życia. Potrzebujesz funkcji która zwróci Ci punkty życia dla danego potwora.

Funkcja przyjemuje jako argument, nazwe potwora - typ `String`, a zwraca jego punkty życia - typ `int`.

```
public int getMonsterHP(String name){
        if(name.equals("goblin")){
                return 250;
        }
        if(name.equals("szczur")){
                return 50;
        }
        if(name.equals("złośnik")){
                return 156; // true value
        }
}
```

Na pierwszy rzut oka wszytko wygląda ok, jednak program nam się nie skompiluje ponieważ dla języka Java istnieje ryzyko że wartość zmiennej `name` będzie jeszcze inna i wówczas niewiadomo co funkcja ma zwrócić.

> ⓘ **return przerywa działanie funkcji**
>
> Kiedy funkcja napotka słowo kluczowe `return` to kończy swoje działanie. Dalsze instrukcje nie zostaną wykonane.

Aby obsłużyć tą sytuacje potrzebujemy wartości domyślnej. Czyli coś co ostatecznie funkcja zwróci jeżeli żaden warunek nie zostanie spełniony.

```
public static int getMonsterHP(String name){
        if(name.equals("goblin")){
                return 250;
        }
        if(name.equals("szczur")){
                return 50;
        }
        if(name.equals("złośnik")){
```

```
                return 156; // true value
        }
        return -1;
}
```

Wówczas możemy ustalić że gdy funkcja zwróci wartość ujemną, oznaczać to będzie że nie odnaleziono takiego potwora.

Przykład użycia funkcji i tego co zwróci

```
int hpMonster = getMonsterHP("goblin"); // 250;
hpMonster = getMonsterHP("szczur"); // 50
hpMonster = getMonsterHP("złośnik"); // 156;
hpMonster = getMonsterHP("ent"); // -1, nie odnaleziono potwora.
hpMonster = getMonsterHP("Złośnik"); // -1, metoda equals odróżnia wielkość
liter więc `Złośnik`, to nie to samo co 'złośnik' :)
```

Słowo kluczowe `return` nie zawsze musi coś zwracać. Jeżeli funkcja zwraca typ `void` czyli nic, to możemy użyć `return` po prostu do przerwania działania funkcji. Coś jak break dla pętli.

```
public static void main(String[] args){
 // kod programu
 String command = in.nextLine();
 if(command.equals("end")){
         return;
 }
 // dalszy kod programu
}
```

Wystarczy użyć słowa kluczowego `return` bez żadnej wartości. Pamiętaj o średniku!

Przeładowanie nazw funkcji

Czy możemy mieć dwie lub więcej funkcji o tej samej nazwie?
Tak, ale są pewne warunki.

Kiedy wywołamy funkcje, język java musi mieć możliwość dedukcji, którą należy wykonać.

Zobaczmy prosty przykład.

```
public static int add(int a, int b){
        return a + b;
}

public static double add(double a, double b){
        return a + b;
}
```

Mamy dwie funkcje o nazwie `add`. Obie funkcje przyjmują dwa parametry - `a` i `b`. Jednakże, jedna funkcja działa na typie `int`, druga na typie `double`.

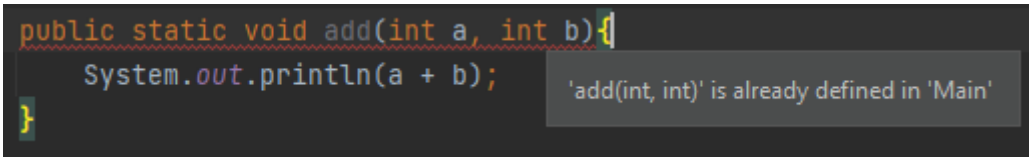
Dzięki temu, język Java jest w stanie wydedukować którą funkcje należy użyć.

```
int iResult = add(5, 3);
double dResult = add(5.3, 3.14);
```

Jednak nie jesteśmy w stanie stworzyć jeszcze takiej funkcji.

```
public static void int(int a, int b){
        System.out.println(a + b);
}
```

Dostaniemy taki komunikat błędu.



Definicja funkcji musi się różnić tym co funkcja przyjmuje, czyli parametrami - ilość, typ i kolejność ich występowania (nazwy nie mają znaczenia), a nie tym co funkcja zwraca.

```
public static int add(int a, int b) // dobrze
public static int add(int a, int b, int c) // dobrze, rożna liczba parametrów
public static int add(int x, int y) // źle, to samo co pierwsza definicja
public static int add(double a, double b) // dobrze, różny typ danych
public static int add(double a, int b) // dobrze, różny typ danych
public static int add(int a, double b) // dobrze, rożny typ danych i różna
kolejność
public static void add(int a, int b) // źle, to samo co pierwsza definicja. Typ
```

```
zwracany jest nie istotny.
```

Podsumowanie

Dzięki funkcją możemy uporządkować nasz kod, fragmeny kodu możemy wyodrębnić do osobnych funkcji.

`getValueFromUser` - może wypisać komunikat do użytkownika i pobrać od niego wartość, a następnie nam ją zwrócić.

`hello` - komunikat powitalny dla naszego programu

`isOdd` - sprawdź czy liczba jest nie parzysta

i tak dalej.

Eksperymentuj! Dzięki funkcją nasz kod programu staje się uporządkowany, a co za tym idzie, bardziej czytelny. Jesteśmy w stanie szybko modyfikować odpowiednie fragmenty kodu. A sama logika głównej aplikacji staje się przejrzystsza!